Meet Charles, big data query advisor

Thibault Sellam
CWI
thibault.sellam@cwi.nl

Martin Kersten
CWI
martin.kersten@cwi.nl

ABSTRACT

In scientific data management and business analytics, the most informative queries are a holy grail. Data collection becomes increasingly simpler, yet data exploration gets significantly harder. Exploratory querying is likely to return an empty or an overwhelming result set. On the other hand, data mining algorithms require extensive preparation, ample time and do not scale well.

In this paper, we address this challenge at its core, i.e., *how to query the query space* associated with a given database. The space considered is formed by conjunctive predicates. To express them, we introduce the Segmentation Description Language (SDL). The user provides a query. Charles, our query advisory system, breaks its extent into meaningful segments and returns the subsequent SDL descriptions. This provides insight into the set described and offers the user directions for further exploration.

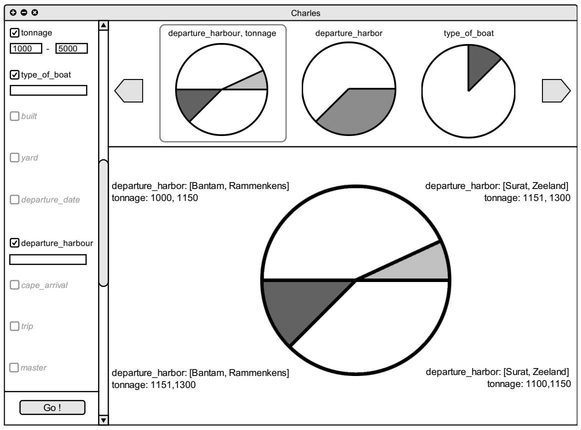
We introduce a novel algorithm to generate SDL answers. We evaluate them using four orthogonal criteria: homogeneity, simplicity, breadth, and entropy. A prototype implementation has been constructed and the landscape of follow-up research is sketched.

1. INTRODUCTION

The data deluge in science and business challenges the way data is turned into useful knowledge. A scientist or data analyst willing to grind the data of e.g., seismology, astronomy, life sciences or weblogs, must choose between running queries or rely on data mining algorithms.

Querying a multi-terabytes database directly is the Spartan method. It is done with SQL, or Map-Reduce tools such as Hive [18]. Often, it is necessary to run the results through post-processing tools (e.g., R [1]). These tools are often implemented as domain specific web applications (e.g., Orfeus [19] for seismology).

Data mining is an alternative to raw queries. Its vocation is to extract "knowledge" automatically from data sets. For instance, these methods aim at discovering frequent patterns, clusters or classifiers. Many different technologies have been proposed, based on a single machine (Weka [10]) or Map-Reduce clusters (Pig Latin [14]). However, scaling these algorithms to multiple gigabytes, let alone terabytes, requires lots of preparation, time and skills.



☑ tonnage
1000 - 5000
☑ type_of_boat
□ built
□ yard
□ departure_date
☑ departure_harbour
□ cape_arrival
□ trip
□ master

departure_harbor: [Bantam, Rammenkens]
tonnage: 1000, 1150

departure_harbor: [Surat, Zeeland]
tonnage: 1151, 1300

departure_harbor: [Bantam, Rammenkens]
tonnage: 1151,1300

departure_harbor: [Surat, Zeeland]
tonnage: 1100,1150

Figure 1: The interface of Charles

The key observation is that these methods are based on the same underlying assumption: *the user knows what he wants*. He has a precise objective for data analysis, although he may not know how to reach it. An SQL query sent to a scientific database is already a model, a hypothesis to be proofed by the database content. Once the data is cleaned, analysis tools require precise queries, yet users may not know what they want or how they want it. In order to break this Gordian knot, we propose a new kind of exploration scheme. Namely, *for any non-trivially sized database, we query the query space*. The idea is to infer queries from data, instead of the opposite. This data may be the whole database, but also the result of a previous query. Thus, we answer queries by queries. This output serves two purposes. First, it gives options for further exploration. Second, it provides a compact summary of the data.

We introduce our prototype implementation, Charles. Charles proposes and ranks several ways to query a set of tuples provided by a user. This is illustrated by Figure 1. The left panel shows the user's query, the *context*. The top panel summarizes Charles' ranked answer list. Each pie-chart represents a set of queries, cutting the database into disjoint pieces. We call them *segmentations*. The largest panel displays the current selection.

Our aim is to isolate and describe large chunks of data in interaction time. We present our algorithm, HB-cuts (Hierarchical-Binary cuts). It splits the dataset into possibly equally wide pieces. These splits are applied successively along dependent attributes. Thus, each segmentation reveals one aspect of the data.

6th Biennial Conference on Innovative Data Systems Research (CIDR '13)
January 6-9, 2013, Asilomar, California, USA.

Charles is based on the following contributions:

- We introduce the Segmentation Description Language (SDL) as a summarization language for database result sets.
- We present metrics to evaluate and rank segmentations.
- We develop a computationally tractable algorithm to generate segmentations, HB-cuts.

Charles can be positioned as a new member of emerging schemes aimed at database summarization and interactive exploration. It is implemented as a front-end for SQL systems. This simplifies experimentation and portability of the code. Its development is undertaken in the context of the MonetDB system.

The remainder of the paper is organized as follows. In Section 2, we define SDL and expose the assumptions underlying our work. Section 3 introduces the evaluation criteria for SDL query sets: homogeneity, simplicity, breadth and entropy. We describe in Section 4 how to generate segmentations. The current state of the prototype and future research directions are discussed in Section 5.

2. CONCEPTS AND METHOD

Our first attempt starts with two restrictions. First, the dataset is contained in one relation. Second, the input and output of our system is based on a proprietary query language, called SDL. SDL can only express conjunctions of predicates. The remainder of this section presents the syntax and semantics of SDL. Let T be the relation and $Attr$ be one column. The literals $a_0, a_1 ..., a_K$ are values from its value domain.

Definition 1. An SDL predicate C expresses either:

- a range constraint, denoted by $Attr : [a_0, a_1]$
- a set constraint, denoted by $Attr : \{a_0, a_1, ..., a_K\}$
- no constraint, denoted by $Attr :$

Definition 2. An SDL query $Q = (C_0, C_1, ..., C_N)$ expresses a conjunction of predicates over table T.

For instance, the query below is based on all three types of constraints:

$$(date : [1550, 1650], tonnage :, type : \{'jacht', 'fluit'\})$$

The result set of a query Q is denoted by $R(Q)$, and $|R(Q)|$ is its cardinality. $C(Q)$ is the cover $|R(Q)|/|T|$.

Our aim is to infer *segmentations* from the dataset D ($\subset T$).

Definition 3. A segmentation is a set of queries $S = \{Q_j\}_{j \in [0,M]}$ wich defines a partition of D:

- for any $(j, k) \in [0, M]^2$, $j \neq k$: $R(Q_j) \cap R(Q_k) = \emptyset$
- $\bigcup_{j \in [0,M]} R(Q_j) = D$

A few examples of segmentations are displayed in Figure 1. We call the constituent queries *segments*.

We aim to explore the data interactively. First, the user specifies a population he is interested in with a SDL statement (potentially the whole database). The system then generates several segmentations and presents them in a ranked list. Each answer describes one aspect of the data. The sets can be presented in a graphical way (e.g. with TreeMaps or pie charts). The user can then select one SDL query, and submit it for further exploration.

By convention, we choose to restrict the exploration to the columns mentioned by the user. For instance, if a user submits the query $(boat\_type :, date :, cape\_arrival :)$, we will be oblivious to all the other attributes in the database. Also, the answers generated by the system are not necessarily based on all three columns.

3. QUALITY METRICS

Obviously, there are many ways to segment the search context. The question is what constitutes a "good" segmentation. As we wish to create informative vistas on the database, we propose four criteria: homogeneity, simplicity, breadth, and entropy.

HOMOGENEITY Charles considers all attributes in the context to be equally important. However, the segments it creates are based on a few columns only. Among those, all items described by a query should be "similar". Furthermore, distinct queries should separate "different" items. Assigning a quantitative measure to this property is still an open research challenge. The clustering literature proposes many measures, depending on the type of data. For instance, metrics based on intra- and extra-cluster distance [9] are well suited for low dimension quantitative data. Alternatively, information theory-based measurements provide good assessments of similarity for categorical values. Other approaches may use the spatial density in the neighbourhood of each point [8]. Each kind of data distribution calls for a different evaluation strategy, and there is to our knowledge no "universal" measure of clustering quality. Besides, the calculations are data and compute intensive. They do not scale well to the datawarehouses we have in mind.

As stated above, our aim is not to create an accurate cluster analysis of the data, but to summarize it by proposing queries. We wish to explore the query space, not the data space. Therefore, *we purposely neglect to quantify homogeneity*. However, the segmentations should still be meaningful. In the next section, we present a heuristic that creates "good enough" groups.

SIMPLICITY Charles presents results to the user as shown in Figure 1. It describes the SDL queries with text, and highlights visually how they cover the search context. As it should return legible results, we keep the sets as *simple* as possible, i.e., each individual SDL query should contain as few predicates as possible. We measure the complexity of a segmentation with the maximum number of constraints among all of its queries. We label this criterion as $P(S_i)$ for it can be calculated readily.

Principle 1. Simple segmentations are more legible.

BREADTH Our goal to aid the user in his understanding of the database content implies that the segmentation should convey the broadest semantics possible. We are more interested in a segmentation that takes several attributes into account than a segmentation on just one attribute. Therefore, we maximize the number of distinct columns across the queries of our segmentations, which we call *SDL breadth*:

Principle 2. Broad segmentations are more informative.

This principle is not directly dependent on the previous one, simplicity. It is possible to improve either one without degrading the other. However, they act as safeguards against one another. The first criterion tends to favor SDL queries with few constraints, if any. The second principle imposes that the query carries more semantic information.

ENTROPY Suppose we take a random point in the database. The point may end up in the segment described by Q_0, Q_1, or any other. The entropy of the segmentation describes the *uncertainty* of this random variable. Its value is 0 when the set consists of one piece, and it reaches $\log M$ when there are M perfectly balanced segments.

Definition 4. The entropy of segmentation $S = \{Q_j\}_{j \in [0,M]}$ is

$$E(S) = -\sum_{j=0}^{M} C(Q_j) \log C(Q_j)$$

$E(S)$ describes two aspects of the segmentations. First, it grows with the the *depth* of the set, the number of SDL queries. Second, if two sets contain the same number of queries, the entropy favours the most *balanced* one. In fact, we assume that the user is primarily interested in the most significant parts of the data, that is, the queries with the highest covers. The most particular subpopulations should appear later in the exploration process, as the user refines his search. Therefore a perfect split would have segments of the same size.

Principle 3. High entropy segmentations are deeper and more balanced.

The principles create a 3-dimensional space to navigate or rank segmentations. They hint at what the perfect segmentation looks like. It has many queries, based on one attribute each, and the partitions have the same size.

4. WHERE ARE THE BEST QUERIES ?

In this section, we propose a heuristic to generate segmentations in a short amount of time: HB-cuts (Hierarchical Binary cuts). The heuristic is a direct application of the principles we presented. The basic idea is to split the data in two equal parts recursively. In order to generate meaningful segmentations, we apply the splits on dependent attributes only. Consider for instance the case illustrated by Figure 1. We may split the dataset in two on the attribute *type_of_boat*. Then, each of the pieces may themselves be split in two on the attribute *tonnage*, assuming that there is a dependency between *tonnage* and *type_of_boat*. The process is repeated until a threshold is reached. This creates semantically coherent segmentations with satisfying scores on the metrics exposed previously.

4.1 Primitives

In this section, we define three basic operations to derive segmentations from a context query. They are all illustrated in Figure 2.

CUT This is the most important primitive of our algorithm. Intuitively, it splits a query in two equal pieces, along a predefined attribute. We define it formally as follows:

Definition 5. Consider the query $Q = (C_0, \dots, C_k, \dots, C_N)$. min_k, max_k, and med_k are respecively the minimum, maximum and median point of the values covered by C_k. We *cut* Q on the attribute $attr_k$ as follows:

$$CUT_{attr_k}(Q) = \{(Q, att_k : [min_k, med_k[), (Q, att_k : [med_k, max_k])\}$$

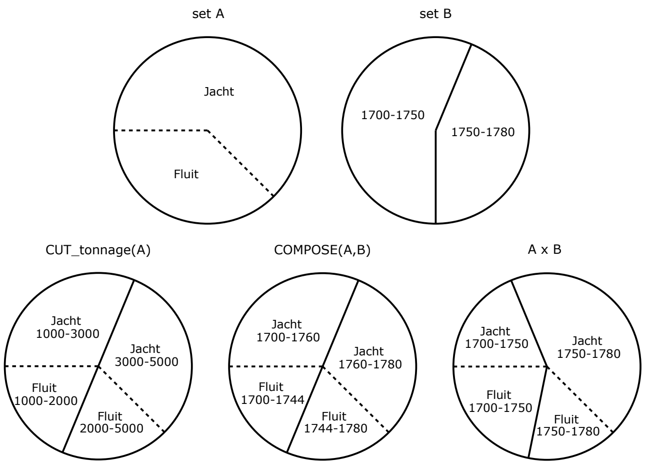


Figure 2: Cut, composition and product of segmentations

How the median point is calculated depends on the data type. For integers, reals, or dates, we use the arithmetic median. For nominal values, we have to make more arbitrary choices. We choose to sort the values by order of occurrence for columns with low cardinality, and alphabetically otherwise. Then, we set med_k at the value for which the accumulated frequency is the closest to 50%. In general, the two resulting pieces may have different sizes. This will be taken into account when we have to rank the segmentations later on.

We extend the CUT operation to segmentations: we cut each of the queries in the set. This doubles the total number of partitions.

Definition 6. Consider the segmentation $S = \{Q_0, \dots, Q_L\}$. We define the CUT operator as follows:

$$CUT_{att_k}(S) = CUT_{att_k}(Q_0) \cup \dots \cup CUT_{att_k}(Q_L)$$

COMPOSITION This operation is used to combine two segmentations. Intuitively, it cuts the queries of one segmentation on the attributes of the other.

Definition 7. Consider the segmentations S_1 and S_2. Suppose that the queries of S_2 are all based on the same set of attributes att_1, att_2,..., att_N. Then we have:

$$COMPOSE(S_1, S_2) = CUT_{att_1}\Big(CUT_{att_2}\big(\dots CUT_{att_N}(S_1)\dots\big)\Big)$$

PRODUCT This operator is another way to combine two segmentations. Consider the example in Figure 2. The sets A and B are both based on two queries. We create a new segmentation $A \times B$ by intersecting each piece of the first with each piece of the latter, thus creating 4 queries.

Definition 8. Consider $S_1 = \{Q_1^0, \dots, Q_1^K\}$ and $S_2 = \{Q_2^0, \dots, Q_2^L\}$. We define the *SDL product* operator $\times$:

$$S_1 \times S_2 = \{(Q_1^i, Q_2^j) \mid (i,j) \in [0,K] \times [0,L]\}$$

The SDL product has a notable feature: it can give a hint about the dependency between two variables. If the product of two balanced segmentations is also balanced, then there is no dependency

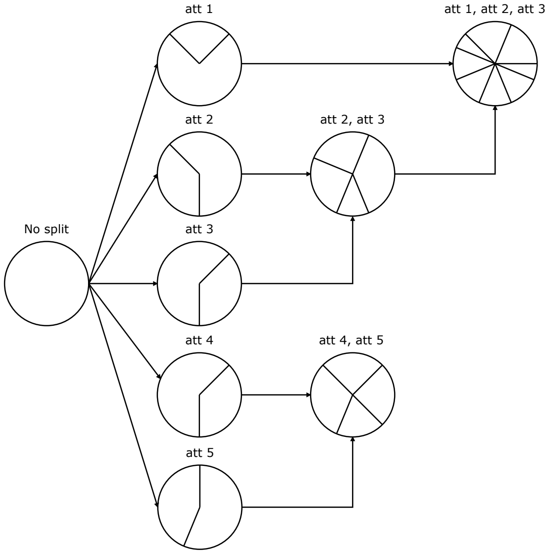


Figure 3: Example execution of HB-cuts

between their variables. The example of Figure 2 shows a dependence between the type of boat and the departure date. This can be quantified with our entropy criterion:

Proposition 1. Consider two segmentations $S_1 = \{Q_1^0, \dots, Q_1^K\}$ and $S_2 = \{Q_2^0, \dots, Q_2^L\}$. We pick one random tuple in the database. X_1 (resp. X_2) is the random variable which describes its partition in S_1 (resp. S_2). X_1 and X_2 are independent iff

$$E(S_1 \times S_2) = E(S_1) + E(S_2)$$

The quotient $E(S_1 \times S_2)/(E(S_1) + E(S_2))$ decreases with the degree of dependence between X_1 and X_2. We label it $INDEP(S_1, S_2)$.

4.2 Heuristics

The algorithm is initialized by cutting the context query on each of its attributes. For N columns, this will create a candidate list of N binary segmentations. Among these candidates, we detect the most dependent couple of segmentations and compose them. The procedure is then repeated until a stopping condition is met. All intermediate results we encounter are stored and returned by order of entropy once the algorithm is done. Figure 3 shows an example of execution for a query with 5 attributes. The procedure generates and returns 8 segmentations.

To retrieve the most dependent couple, we enumerate every possible set $\{S_1, S_2\}$ from the candidate list. For each of these, we evaluate the product $S_1 \times S_2$ and calculate $INDEP(S_1, S_2)$. As stated previously, this value decreases with the dependence of S_1 and S_2. If S_1^* and S_2^* yield the smallest value, we replace them by $COMPOSE(S_1^*, S_2^*)$ in the candidate list, then repeat.

As the algorithm is executed, the level of dependence between candidates degrades and the segmentations contain more and more SDL queries. Therefore, we propose to set two stopping criteria. First, we should not group independent attributes in the same queries. Therefore a maximal value should be set on the fraction $INDEP(S_1, S_2)$, possibly through statistical hypothesis testing. In our case, a threshold of 0.99 gave statisfying results with most data sets. Second, there is a limit to the amount of information that a user can grasp: we consider that a pie chart with more than a dozen slices is hard to read. Therefore, we set an upper bound on the number of queries in a segmentation. Figure 4 shows the pseudo-code for our algorithm.

```
function HB CUTS(query, maxIndep, maxDepth)
    $cand \leftarrow \{\}$
    for $i \leftarrow 0, nbAttributes(query)$ do
        $cand \leftarrow cand \cup \{CUT_{attr_i}(query)\}$
    end for

    $newSeg \leftarrow \{\}$
    $ind \leftarrow 0, dep \leftarrow 0$
    $output \leftarrow \{\}$
    while $true$ do
        $\{S_1^*, S_2^*\} \leftarrow argmin_{S_1, S_1 \in cand}\ INDEP(S_1, S_2)$
        $newSeg \leftarrow COMPOSE(S_1^*, S_2^*)$
        $ind \leftarrow INDEP(S_1^*, S_2^*)$
        $dep \leftarrow depth(newSeg)$
        if $ind \geq maxIndep \parallel dep \geq maxDepth$ then
            break
        else
            $cand \leftarrow cand \cup \{newSeg\}$
            $cand \leftarrow cand - \{S_1^*, S_2^*\}$
            $output \leftarrow output \cup \{S_1^*, S_2^*\}$
        end if
    end while
    $output \leftarrow output \cup cand$

    return $sort(output)$
end function
```

Figure 4: Segmentation algorithm

5. CURRENT AND FUTURE DIRECTIONS

We implemented our query space exploration scheme in a proof-of-concept system, named Charles. In this section, we highlight some of the challenges encountered along the way. Next, we charter the development and research road-map towards a full-blown and scalable version.

5.1 Charles' Implementation

Charles has been implemented as a C application on top of MonetDB. The GUI shown in Figure 1 has been implemented as a standalone program in Python. It can be turned into a fancy web-application readily. Each SDL set is represented by a pie-chart where each slice is represented by an SDL query. The search context is presented in the leftmost panel. It enumerates the columns of interest and any a priori defined value constraints. The top panel presents a ranked list of candidate sets suggested by Charles. Selecting one opens it for detailed inspection and interaction in the main panel.

The actual algorithm poses two challenges: the explosion of the search space and the performance of the database operations. The search space explosion is directly related to the number of attributes considered in the search context (horizontal scalability). The performance of database operations depends on the number of tuples (vertical scalability).

Horizontal scalability is the hardest to achieve because the search space grows exponentially. However, some optimizations are possible. For instance, the calculations of SDL products and entropy can be reused from one iteration to the next. A safety line comes from the visualization. A pie-chart with more than a dozen slices is difficult to read. Also, a large number of candidates is overwhelming. This sets an upper bound on the size of the search space.

Vertical scalability depends a lot on the back-end choices. Two types of operations are performed: median calculations and counts over predicates. The former is not commonly supported by systems. Overall, we favor OLAP technologies. As we cannot know which columns will be queried, creating indices a priori is not possible. Therefore, column-based systems such as MonetDB [13] are well for suited Charles' workloads.

5.2 Broader perspectives

Our procedure is based on four steps: initial candidates generation, derivation of the segmentations, ranking and visualization. Each of these steps could be generalized.

First, we only consider median cuts. This is a serious limitation. Assume we split the domain of an attribute *size*. This attribute follows a Gaussian distribution. With the current state of the system, there is no way to obtain a pie-chart displaying the second third of the population. However, this subset is very dense and may be very interesting for a user. We have to develop support for other quantiles, along with selection rules.

Also, our heuristic relies on a heavy restriction: all queries in an segmentation are based on the same attributes. It would be interesting to consider other options. For instance, we could cut each piece of a segmentation on a potentially different attribute. The main issue with this approach is the explosion of the search space. This may be tackled with randomized algorithms. Another research direction is to design a generation engine for *lazy* segmentation generation. Currently, Charles generates all possible answers to a user query in one go, then returns them. It may be beneficial to spread the computation time: the system would only generate a small set of queries, and create more upon request.

The overall evaluation and ranking process can be greatly improved with other types of knowledge. We do not use any notion of "interestingness" or "surprise". Also, we did not consider the clustering quality measures offered by the data mining community. We may gain from incorporating such notions. It is however unclear yet which measure can be used, and how to include them into our current evaluation scheme.

Two improvements for the visualization scheme present themselves readily. First, the display could be clarified with hierarchical visualizations, such as tree-maps or multi-level pies. Second, the only information that Charles gives about the segments is their counts. It may be interesting to display more. For instance, the distribution of some attributes could be plotted, with uni- or multivariate representations. However, determining which attributes to display is almost as difficult as choosing the queries to illustrate.

Finally, the implementation of Charles could benefit from the incorporation of sampling strategies. The calculation of medians is a major bottleneck. However, not all tuples are necessary to give good results.

6. RELATED WORK

We identify four ways to generate interesting queries from a database in the literature: database summarization, faceted search, query recommandation and subspace clustering.

6.1 Database summarization

Database summarization appears as a contender for Charles. For instance, Yang et al.[21] introduce a method to return reduced versions of relational data. The result is less precise, but gives hints about the actual content of the database. It can be used as input for the knowledge discovery processes. In the data mining community, Mampaey et al. [12] propose to summarize transaction data with a few significant item-sets, chosen with information theoretical principles. The objective of these methods is similar to ours. However, our approach is different. These methods reduce the number of items, but do not give any hint for the analysis. Our approach tries to separate and rank different views over the data. Each segmentation highlights one aspect of the data. Also, we have strong requirements on the response time, and Charles should not require any preparation. This is not the priority for these works.

6.2 Faceted search

The information retrieval community has been proposing several interfaces for user-friendly querying of large repositories. Among those, *faceted search* is the closest to ours. It has been adopted by many popular e-commerce websites. One of the earliest engines for SQL databases is Flamenco [7]. The user enters a query, usually keyword-based. The system does not return all the results. Instead, it selects a few significant items and outputs them. In a separate window, it displays several attributes such as *price* or *brand*. They are the *facets*. For each of these attributes several value ranges are proposed. The user refines his selection by choosing one of these categories.

Faceted search is an active research field, and several authors have been looking into extending it for OLAP analysis. Wu et al. [20] embed it into a wider framework called KDAP (Keyword-Driven Analytical Processing). This framework generates many queries from a keyword, such as joins between facts and dimension tables. The results of these queries are then aggregated and organized in facets. Recently, Ben-Yitzhak et al. [3] have been working on extending the initial model. Usually, the facets come with counts for each subcategory. This gives an idea of the content of the database to the user. This could be replaced by more complex aggregations, such as selections, sums or averages.

Somehow, faceted search also aims at producing queries from tuples. However, in all the aformentioned cases, the user still has to build the queries. The facets are merely building blocks.

One of the main challenges of faceted search is the *facet selection* problem. While the data can have many attributes, only a few facets can be presented to the user. Which are the most interesting ones ? Dash et al. propose to return the most *surprising* ones [6]. They reveal data for which the distribution differs significantly from an expected value. This variant of faceted search is very close to our work: the system helps the user by identifying interesting subgroups. However, our notion of interest is very different. We try to summarize the general case, while Dash et al. focus on unexpected patterns. DynaCet [16] is another attempt to guide the exploration. The system provides facets such that any tuple can be reached with a minimal number of queries. It can build a decision tree over the whole data set, or only a few levels each time the user drills in. The objective of this work is very different from ours. DynaCet helps the user identify precise tuples of interest: this is a search scenario. We seek to generate summaries of the data. Also, as in most faceted search applications, all the facets are based on one attribute only. Our system is the opposite as it maximizes the number of columns involed in the segmentations.

6.3 Query recommendation

Several authors have proposed query recommendation algorithms. A common approach is to use the query log of the database. For instance, Chatzopoulou et al. [5] assume that if two users have a similar querying behavior, we can use the log of one to recommend queries to the other. They draw inspiration from collaborative filtering systems proposed on the Web. SnipSuggest [11] uses the same approach to generate context-aware recommendations: the user starts typing a query, then the system proposes several completions. The more a user writes, the more accurate the suggestions will be. Our work does not use the query log, it is based on statistical properties of the data. Therefore, we can drop the assumption that several experts have been independently using the same database for the exact same purpose.

The work of Sarawagi et al. [17] is closer to ours. It recommends interesting regions of the data based on statistics. However, there are two main differences. First, it is based on exceptional behaviours. An anticipated value of each observation can be calculated by creating a statistical model of the neighbourhood. Their system compares this expectated value to the observed one, and reports the abnormal deviations. As stated in the previous section, we are interested in the general case, not the exceptions. Second, the method of Sarawagi et al. targets data cubes exclusively.

6.4 Subspace clustering

Traditional clustering considers that all the dimensions are equally important. However, this assumption does not hold with many real life datasets. Also, a high number of dimensions tends to make items equidistant [4]. Subspace clustering addresses both of these problems. These methods detect clusters in various sub-spaces of the data. Therefore, each cluster is returned along with the subset of dimensions on which its tuples are similar. A review of these algorithms was written by Parsons et al. [15].

The first algorithm of this field is CLIQUE [2]. It splits each dimension in bins and detects the densest. Then, it explores all the possible combinations of bins. This creates cells of higher dimension, that can also be combined. The results can be expressed as a set of hyper-rectangles written in Disjunctive Normal Form such as $((30 < age < 50) \wedge (5 < salary < 8)) \vee ((40 < age < 60) \wedge (2 < salary < 6))$. This is similar to SDL partitions. Both approaches aim at partitioning the data in interesting subsets and express the results as queries. Also both approaches start with single attribute splits then combine them to generate new partitions. However, the objective is different. CLIQUE aims at discovering high density sub-spaces. We generate instant and general hints about the content of the data.

7. SUMMARY AND OUTLOOK

Enabling fast and intuitive data exploration is the Grand Challenge of the Big Data era. Brute-force parallel processing techniques such as Map-Reduce on Hadoop clusters push the wall, but do not break it. A paradigm shift is needed.

To bring out the knowledge hidden in the data, we propose to explore the barren landscape of database queries. Our proof-of-concept system, Charles, probes a query space delimited by the user and returns the most promising candidates. We believe this opens up ample opportunities for innovative theoretical, algorithmic, and systems research.

8. ACKNOWLEDGMENTS

This work was supported by the Dutch national program COMMIT.

9. REFERENCES

[1] *The R Project for Statistical Computing*. http://www.r-project.org/index.html.

[2] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. *SIGMOD Rec.*, 1998.

[3] O. Ben-Yitzhak, N. Golbandi, N. Har'El, R. Lempel, A. Neumann, S. Ofek-Koifman, D. Sheinwald, E. Shekita, B. Sznajder, and S. Yogev. Beyond basic faceted search. In *Proc. WSDM*, 2008.

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

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

[6] D. Dash, J. Rao, N. Megiddo, A. Ailamaki, and G. Lohman. Dynamic faceted search for discovery-driven analysis. In *CIKM*, 2008.

[7] J. English, M. Hearst, R. Sinha, K. Swearingen, and K. Lee. Flexible search and navigation using faceted metadata. Technical report, University of Berkeley, 2002.

[8] M. Ester, H. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *SIGKDD*, 1996.

[9] G. Gan, C. Ma, and J. Wu. *Data clustering*. SIAM, 2007.

[10] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. Witten. The weka data mining software: an update. *SIGKDD Expl.*, 2009.

[11] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu. Snipsuggest: context-aware autocompletion for sql. *Proc. VLDB endow.*, 2010.

[12] M. Mampaey, N. Tatti, and J. Vreeken. Tell me what i need to know: succinctly summarizing data with itemsets. In *SIGKDD*, 2011.

[13] S. Manegold, M. Kersten, and P. Boncz. Database architecture evolution: mammals flourished long before dinosaurs became extinct. *Proc. VLDB endow.*, 2009.

[14] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proc. SIGMOD*, 2008.

[15] L. Parsons, E. Haque, and H. Liu. Subspace clustering for high dimensional data: a review. *SIGKDD Expl.*, 2004.

[16] S. Roy, H. Wang, U. Nambiar, G. Das, and M. Mohania. Dynacet: Building dynamic faceted search systems over databases. In *ICDE*, 2009.

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

[18] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB endow.*, 2009.

[19] T. van Eck. The orfeus seismological software library. *Seismological Research Letters*, 1997.

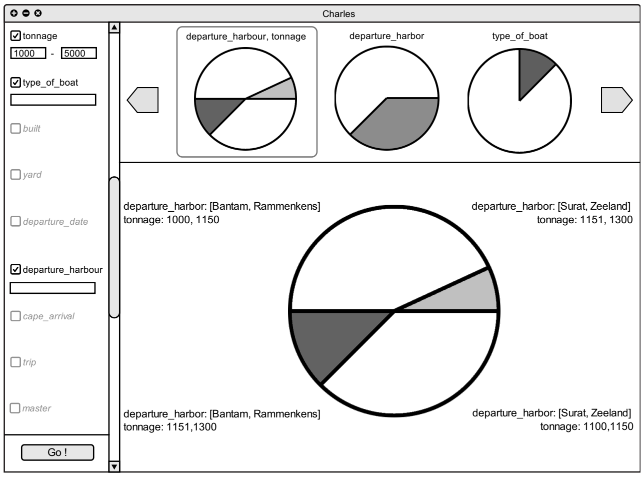
[20] Y. Wu P., Sismanis and B. Reinwald. Towards keyword-driven analytical processing. In *Proc. SIGMOD*, 2007.

[21] X. Yang, C. Procopiuc, and D. Srivastava. Summarizing relational databases. *Proc. VLDB endow.*, 2009.

Demonstration proposal
Meet Charles, big data query advisor

Thibault Sellam
CWI
thibault.sellam@cwi.nl

Martin Kersten
CWI
martin.kersten@cwi.nl



☑ tonnage
1000 - 5000
☑ type_of_boat
□ built
□ yard
□ departure_date
☑ departure_harbour
□ cape_arrival
□ trip
□ master

departure_harbor: [Bantam, Rammenkens]
tonnage: 1000, 1150

departure_harbor: [Surat, Zeeland]
tonnage: 1151, 1300

departure_harbor: [Bantam, Rammenkens]
tonnage: 1151,1300

departure_harbor: [Surat, Zeeland]
tonnage: 1100,1150

Fast and intuitive exploration of terabytes large databases is the Holy Grail of the Big Data era. As business and scientific data warehouses enclose more and more data, discovering facts and patterns gets harder. Navigating through millions of tuples with SQL or Map Reduce jobs still requires too much time and hardware.

The user targeted by our system has a large amount of data, he is vaguely familiar with statistics but was never really interested in SQL. He guesses that his database contains valuable knowledge, but he does not know where to look. The dataset can be stored in a commodity hard disk, but it is much too large for Excel. It is managed with any SQL-based DBMS, preferably optimized for OLAP operations.

What does the data look like ? What should our user look for ? We introduce Charles, query advisor. Charles answers a query with more queries. He will guide the user through the query space.

Charles is a tool for database summarization and exploration. It is developed as front-end for SQL systems. Its actual implementation and performance evaluation is undertaken in the context of the MonetDB system. Its aim is to generate queries from data regions described the user.

We introduce a data description language called SDL (Segmentation Description Language). An SDL query describes a subset of the data with conjunctive predicates. Charles' task is to break down such a query into smaller queries. These queries describe disjoint, preferably equally sized and potentially interesting pieces of the data. They are displayed with interactive visualizations. This gives an overview of the data to the user, he can select one of the pieces and submit it for further exploration.

In this demonstration, we will use a few domain-specific databases, covering topics such as history and astronomy. Our ambition is to show how Charles can strip down their content in a few minutes, and bring the audience from a state of total ignorance to topic familiarity within a few minutes.

Charles is at the state of research prototype. We believe it charters an exciting landscape for both theoretical and systems research.

Acknowledgments

This publication was supported by the Dutch national program COMMIT.