Dynamic Faceted Search for Discovery-driven Analysis

Debabrata Dash[1] Jun Rao[2] Nimrod Megiddo[2] Anastasia Ailamaki[1] Guy Lohman[2]
ddash@cs.cmu.edu junrao@us.ibm.com megiddo@us.ibm.com natassa@cmu.edu lohman@us.ibm.com
[1]Computer Science Dept., CMU [2]IBM Almaden Research Center

ABSTRACT

We propose a dynamic faceted search system for discovery-driven analysis on data with both textual content and structured attributes. From a keyword query, we want to dynamically select a small set of "interesting" attributes and present aggregates on them to a user. Similar to work in OLAP exploration, we define "interestingness" as how surprising an aggregated value is, based on a given expectation. We make two new contributions by proposing a novel "navigational" expectation that's particularly useful in the context of faceted search, and a novel interestingness measure through judicious application of p-values. Through a user survey, we find the new expectation and interestingness metric quite effective. We develop an efficient dynamic faceted search system by improving a popular open source engine, *Solr*. Our system exploits compressed bitmaps for caching the posting lists in an inverted index, and a novel directory structure called a *bitset tree* for fast bitset intersection. We conduct a comprehensive experimental study on large real data sets and show that our engine performs 2 to 3 times faster than Solr.

1. Introduction

An increasing amount of information consists of a combination of both structured and unstructured data. For example, patent documents contain structured properties such as inventors, assignees, class codes, and filing date, as well as a body of unstructured text. Helpdesk tickets store not only structured data such as the ticket's originator, responsible party, and status, but also text describing the problem and its origin. Increasingly, enterprises want to run analytics [5, 27] on text to extract valuable structured information such as chemical compounds used in a patent and products mentioned in a helpdesk ticket. As a result, the number of unique structured properties in those data sets can be fairly large (from mid to high tens or even hundreds). Performing discovery-driven analysis on such data sets becomes challenging since a user may not know which properties to focus on. Ideally, a user would like to just type in some keywords into a system which would then guide him to areas of interest.

A promising query interface for such mixed data is *Faceted search* [28], which is widely used by e-commerce sites such as amazon.com and shopping.com for querying their catalogs. For example, a user might enter "digital camera" in the keyword window of shopping.com. There are potentially thousands of matches, but only a few popular ones can be displayed on the screen. To assist navigation, the system also shows in a separate panel summaries of search results, such as a count of digital cameras in each range of price and resolution (we refer to properties such as price and resolution as *facets*). When the user selects a particular price range such as "\$200–\$300", the system adds a structured constraint on price to the original query, and refreshes the top matches and the summaries with results from the new query. The navigation process continues until the user finds the desired camera. Faceted search offers several advantages. First, it smoothly integrates free text search with structured querying. Second, the counts on selected facets serve as context for further navigation. For example, a user might choose to focus his search in the price range that has the most cameras.

Today's faceted search systems are designed for browsing catalog data and are not directly suitable for discovery-driven exploration. First, to preserve browsing consistency, facets selected for navigation tend to be "static", i.e., they often don't change with different keywords. A typical heuristic rule to select facets is to favor those with more counts [25]. For example, consider a keyword search for "XML" on a repository of software patents. A traditional faceted search system is likely to present for navigation an assignee facet with values such as IBM and Microsoft, since they have more patents on "XML" in terms of the absolute counts. While such a result may be useful for certain people, others may find a startup with only five patents, but all on "XML", to be more interesting. Second, when browsing online catalogs, the navigational facets are single-dimensional only. An important aspect of discovery is to identify interesting correlations, and thus the ability to present facets in pairs, triples, etc. is critical.

We propose an enhanced faceted search system for the kind of discovery-driven analysis that is often performed in On-Line Analytical Processing (*OLAP*) systems. From a potentially large search result, we want to automatically and dynamically discover a small set of facets and values that are deemed most "interesting" to a user. Using this information, the user can quickly understand important patterns in the query result and can use these patterns to refine his search.

Following earlier work in structured OLAP [21, 22], we define "interestingness" as how surprising or unexpected a summary is, according to some expectation. We make the following new contributions. First, since interestingness is subjective, we allow users to set expectations of their own. In particular, we propose a novel "navigational" method of setting a user's expectation that naturally fits how he navigates in a faceted search system. Second, we propose a novel method of measuring the degree of surprise through judicious use of p-values. Our method is unbiased to domain size and makes intuitive sense. Finally, we validate the relevance of our dynamically selected facets by conducting a user survey based on a real data set. The survey result is positive.

For better performance, many traditional *OLAP* systems pre-compute a data cube [11], and then perform subsequent analysis on the cube itself rather than on the base data. This is impossible when structured data is mixed with text, since maintaining a cube including all possible keywords in the text is prohibitive. Similar to existing faceted search systems, we

build a runtime engine on top of an inverted index and dynamically compute aggregations over results returned by the index. Our dynamic faceted search engine is computationally intensive because it not only considers single-dimensional facets, but also facet combinations. We improve the performance of existing systems by using two new ideas. First, we cache facet data in a compressed bitmap for better space utilization as well as faster set intersections. Second, we develop a novel directory structure called a *bitset tree* on top of the inverted index to reduce the overhead of unnecessary bitset intersections.

The rest of the paper is organized as follows. We define the terminology and formally state our problem in Section 2. We describe different expectations that a user can set and our "interestingness" measure in Section 3. Section 4 revisits existing faceted search implementations and describes the design of our improved implementation. In Section 5, we present results from the user survey and the performance evaluation using two real data sets. We survey the related work in Section 6 and conclude in Section 7.

2. Terminology and Problem Statement

In this section, we introduce the terminology used in the rest of the paper, and define the problem that we want to solve.

Definition 1. A *repository* D is a collection of documents, each of which is composed of some free text and one or more <*facet*: *value*> pairs. For simplicity, we assume that both the facet and the value are strings, although in general the values can be typed. Given a facet F and a value f in F, we call $<F : f>$ an instance of facet F. All unique values associated with a facet F form the *domain* of F. We allow each document to have any number of instances of a particular facet. For example, a publication can have two facet instances, <author : X> and <author : Y>.

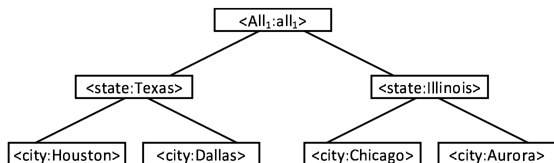


Figure 1: Facet Hierarchy 1

Definition 2. Often, multiple facets represent the same concept at different granularities. We can organize the domain of these facets into a facet *hierarchy*. Each node in the hierarchy stores a <facet : value> pair. A node $<F_1 : f_1>$ is the parent of another node $<F_2 : f_2>$ if for each document, $F_2 = f_2$ implies $F_1 = f_1$. For example, in the facet hierarchy shown in Figure 1, node <state : Texas> is the parent of node <city : Houston>. We also add to the hierarchy a unique dummy root node of the form $<All_i : all_i>$ and assume that the equality $All_i = all_i$ is always true. A facet may be present in more than one hierarchy.

Definition 3. For simplicity, we assume that a *query* q on the repository has the form "*keywords* && $F_1 = f_1$ && $F_2 = f_2 \ldots$". The result of q is denoted by D_q and it includes the set of documents having the specified keywords and satisfying all constraints on selected facets. A typical user session starts with a query with just the keywords, followed by the addition or the removal of constraints on certain facets to the original query.

Definition 4. Given a query q, we define a *facet summary* for a facet set $F_1, \ldots, F_m$ as a list of tuples $<f_1, \ldots, f_m, A(f_1, \ldots, f_m)>$ over D_q, where f_i is an instance of facet F_i and $A(f_1, \ldots, f_m)$ is an aggregate of documents in D_q that contain all these facet instances. In this paper, we focus only on aggregates that count the number of documents.

Problem Definition: Given a repository of documents with n facets, and two integers K_1 and K_2, we want to select K_1 facet sets and a facet summary for each with up to K_2 tuples that are the most "interesting" to a user, i.e., they are the most unexpected or surprising to a user based on his expectation. For easy reference, we include important symbols in the following table.

Symbol	Explanation
D	a repository
F	a facet
f	a facet value
q	a query
D_q	documents in D matching query q

3. Measure of "Interestingness"

The concept of "interestingness" has been studied in the context of analyzing OLAP cubes [21, 22, 28]. In all that work, "interestingness" is defined by how surprising it is that a cell value in the cube is different from an expected one. There are different ways of setting the expectation. For example, in [22, 28], the expected value of a cell is derived from other sibling cells at the same level or from cells that are one or more levels higher in the dimensional hierarchy. The expectation is often set by the system, although [21] allows a user to specify a list of cells as "known" and uses them to set the expectation for other cells.

Following this work, we also measure "interestingness" as how surprising an actual aggregated value is, given a certain expectation. Different users may have different expectations and we try to make the process of setting the expectation easy for the users. In Section 3.1, we describe three useful methods of setting the expectation. The "navigational" method is particularly suitable in the context of faceted search and is the best method found in our survey. Previous work didn't pay too much attention to make sure that the computed degree of interestingness is comparable within and across domains. For example, the KL-divergence [4] used in [21] is very sensitive to domain sizes. In Section 3.2, we describe a novel interestingness measure that addresses this problem and also makes intuitive sense.

3.1 Setting the Expectation

For a given set of facet values $f_1, f_2, \ldots, f_m$ from facets $F_1, F_2, \ldots, F_m$, we define $C_D(f_1, \ldots, f_m)$ and $C_q(f_1, \ldots, f_m)$ as the count of the number of documents with all those facet values in D and D_q, respectively. We use $\mathbf{E}[C_q(f_1, \ldots, f_m)]$ to denote an "expected" value for $C_q(f_1, \ldots, f_m)$. We identify three different methods of setting a user's expectation.

Natural: Suppose that a user knows very little about the data in the repository. Absent any specific knowledge, the user very likely assumes some natural distribution in the data set. For example, the user may think that documents in the repository are uniformly distributed along each facet, and facets are independent of each other. Based on this, we define a *natural* method of setting the expectation. Given a query q, the expected counts are set as follows. For an individual facet instance $<F : f>$, $\mathbf{E}[C_q(f)] = |D_q|/$ (number of unique values in F in D_q). For an instance $f_1, \ldots, f_m$ of a facet set,

we estimate the expected count as

$$\mathbf{E}[C_q(f_1, \ldots, f_m)] = |D_q| \cdot \prod_{i=1}^{m} (C_q(f_i)/|D_q|) \ .$$

As we can see, the former is based on the uniformity assumption while the latter is based on the independence assumption. In this case, interesting facets tend to be those that are skewed or correlated in the query result. For example, in the patent repository, assignee IBM is likely to be interesting since IBM owns many more patents than others. Such information could be useful for people who don't know much about the patent literature.

Navigational: If a user is already somewhat familiar with the repository, natural expectation may no longer be appropriate. For instance, most researchers won't be surprised by the fact that IBM owns a lot of patents. This motivates the second method, *navigational*, which sets the expectation based on how the user navigates the results. When a user types the first keyword query q_1, we set the counts proportionally based on the data distribution in the complete repository. Specifically, we have $\mathbf{E}[C_{q_1}(f_1, \ldots, f_m)] = |D_{q_1}| \cdot (C_D(f_1, \ldots, f_m)/|D|)$. If a user issues a second query q2 by drilling into a facet instance, we reset the expectation using the result from the previous query, i.e., $\mathbf{E}[C_{q_2}(f_1, \ldots, f_m)] = |D_{q_2}| \cdot (C_{q_1}(f_1, \ldots, f_m)/|D_{q_1}|)$. Using such an expectation, IBM may no longer be an interesting assignee just because of the large absolute patent count. Instead, the startup that we mentioned in the introduction could be more interesting. Although it only owns 5 patents on "XML", that number is significant given the low expectation based on its overall patent record.

Ad hoc: If a user is not satisfied with the previous two methods, the user can use an *ad hoc* one by telling the system to set expectation based on an arbitrary query q of the user's choice. Similar to the navigational method, we set the count for each facet value proportionally based on the distribution of the result of q. A good example is for analyzing facets with an ordered domain. If a user is looking at a subset of patents in the current year, then it makes sense to set the expectation using a similar query, but constrained to the previous year instead of the current one.

3.2 Measuring Degree of Interestingness

In this section, we first discuss the interestingness of a single facet instance, followed by that of the whole facet.

Single facet instance: We need to quantify the degree of interestingness based on an actual and an expected count. Let us assume for now that the expectation is set from the distribution of all data in the repository. We must be able to decide, for example, which of the following hypothetical findings is more interesting given a query q on a patent repository: (i) 45 patents out of 500 in D_q were issued in year 2000, whereas out of the repository of 100,000 patents, 5% of them were issued in that year; or (ii) 10 of the 100 patents in D_q were filed by IBM, whereas IBM owns 2% of all patents. It seems that there are many distance metrics that we could use here. However, the fact that the facets *year* and *assignee* have different domain size makes the selection particularly challenging, since it's not clear how one should normalize the distance values cross domains to ensure that they are comparable.

As a guiding principle, we calculate a level of interestingness of a facet instance by evaluating it with respect to a scenario in which its associated count is generated by random sampling. The presumption is that the smaller the probability of observing the count under random sampling, the more interesting the facet instance. Of course such a measure of interestingness ignores any knowledge about the facet instance except the frequency.

Specifically, suppose that a certain facet value occurs in r out of R documents in the repository and in q out of Q documents in the output of a certain query. Also suppose $\frac{q}{Q} \geq \frac{r}{R}$. Then, the interestingness of that facet value vis-a-vis the query could be evaluated by the probability that in a random sample of size Q there will be at least q documents with that facet value. Such a probability is often called in the statistical hypothesis testing the p-value, which gives the probability of obtaining a result at least as extreme as a given data point, under the null hypothesis. That probability is derived from the hypergeometric distribution and is equal to

$$\sum_{k=0}^{q} \frac{\binom{r}{k}\binom{R-r}{Q-k}}{\binom{R}{Q}} \ .$$

A similar hypothesis can be derived when $\frac{q}{Q} < \frac{r}{R}$. The hypergeometric distribution is based on sampling without replacement, which is different from sampling with replacement in our scenario. However, such a difference is often negligible since we expect the number of matching documents in a query to be a small fraction of the whole repository (typically 5% or less). Note that these p-values serve only as an intuitive measure of interestingness; the precise accuracy is not crucial. In many cases, such p-values can be approximated using the normal distribution or the Poisson distribution, and we exploit such an approximation in our implementation.

Suppose that in the above example, we calculate p-values of about 0.000032 and 0.005 for year 2000 and assignee IBM, respectively. We can decide that the facet instance year 2000 is more interesting, because statistically, it is more unlikely that the associated actual count is produced by chance. We may also conclude that the entire facet year is more interesting because at least one of its instances appears to be more correlated with the query.

The use of p-values is one way to introduce a "common denominator" for comparing interestingness across different facets of varying domain size. We choose p-values for the following reasons. First, it makes intuitive sense. This is because p-values essentially measure how extreme an event is, which agrees with the definition of a surprise. Second, independent of facet type, p-values always produce a normalized degree of interestingness between 0 and 1. The interestingness of a facet can be easily computed by aggregating the p-values on its instances. Finally, when experimenting different distance metrics, we find the results produced by p-values to be more relevant. The feedback from the user survey (see Section 5.2) strongly supports this claim.

For a facet with a large number of instances, the above-mentioned probability measure may not be appropriate because, even under random sampling, it is to be expected that some values will have small p-values[1]. For example, suppose that the facet of inventor in a repository of 100,000 patents has 2,000 distinct values (for simplicity, assume that each patent has only a single inventor). Thus, on average there

[1]In fact, the p-values observed in random sampling are distributed uniformly, so the expected value of the least p-value in a sample with replacement of size Q is $\frac{1}{Q+1}$.

are 50 patents per inventor. For the sake of simplicity, let's assume that each inventor has precisely 50 patents. Suppose that the output of a certain query contains 1,000 documents, and a certain inventor appears in four patents in the query output. In a random sample, the expected number of patents with the same inventor is only 0.5. This particular inventor may seem interesting because in a random sample of 1,000 the probability that this inventor will appear in at least four patents is approximately 0.0017. However, there are 2,000 inventors in the repository, and the probability that at least one of them will have at least four patents in the query output is approximately 0.973. Therefore, it is not interesting at all to see that there exists some inventor like this in the query output. On the other hand, if there exists an inventor that appears in 10 out of 1,000 patents in the query output, then it is interesting because the probability of such an event in a random sample of size 1000 is less than 10^{-6}. In our implementation, for a facet with a large domain, we ensure that we only use a p-value if its product with the domain size is still small enough.

The whole facet: Presenting individual facet instances from many facets to the user could be confusing. Instead, we want to organize the instances by their facets and select only a small number of interesting facets. Given the interestingness scores of the individual facet instance, we wish to associate an aggregate score of interestingness with the whole facet. One way of computing the interestingness of an entire facet is to evaluate the probability that in a random sample of the same size as the query output, the distribution of all facet instances will be as far from the distribution in the entire repository as is the distribution observed in the query output. One possibility is to use the Chi^2 distribution to evaluate the sum-of-squares distance $\sum_{i=1}^{n} \frac{(q_i - Qr_i/R)^2}{Qr_i/R}$, where q_i and r_i are the count of the ith facet instance in the query and the repository, respectively. However, there are a couple of problems with the above approach. First, a human user often is presented with and capable of digest a small number of values. Using all values in a facet to compute an aggregated degree of interestingness for the facet may not reflect what a user actually observes. Second, an interesting value in a facet may easily be "diluted" when aggregated with a large number of less interesting ones. We employ a simpler, but more practical way of ranking the interestingness at the facet level. For each facet f, we consider the p-values of only the k most interesting values in f. If the p-values of those values are $p_1 \leq \cdots \leq p_k$, then we first replace them by $s_i = -\log p_i$ $(i = 1, \ldots, k)$. Thus, $S_1 \geq \cdots \geq S_k$. The final measure of interestingness of facet f is computed as $\sum_{i=1}^{k} W_i \cdot S_i$, where W_i is a weight. The larger the aggregated value, the more interesting the facet is. We experiment with three different schemes of setting the weights, *MaxWeight*, *AvgWeight* and *HybridWeight*. *MaxWeight* assigns 1 to W_1 and 0 to the rest of weights. It favors a facet with at least one very interesting value. *AvgWeight* assigns each W_i an equal weight. The last one, *HybridWeight*, averages the interestingness computed by *MaxWeight* and *AvgWeight*. In Section 5.2, we evaluate the effectiveness of those three schemes in our user survey.

To summarize, through judicious application of p-values, we quantitatively compute the interestingness of individual facet instances as well as the whole facet. We then present to the user the top K_1 most interesting facet sets and within each, the top K_2 most interesting facet instances. For each facet instance, in addition to the actual count, we also show an expected count and an associated degree of interestingness from our calculation. The discussion in this section assumes that the expectation is set using all data in the repository. The same analysis can be applied to other kinds of expectation. The only difference is that the repository is no longer the original one, but a contrived one that gives those expected values.

4. Implementing Dynamic Faceted Search

In this section, we describe the design of a dynamic faceted search system that we built. We review existing faceted search implementations based on inverted indexes in Section 4.1. Section 4.2 describes techniques that significantly improve the computation of facet summaries. We cover other implementation details in Section 4.3.

4.1 Existing Approaches

All of today's faceted search engines are built on top of inverted (or text) indexes. In addition to supporting text search well, such an implementation offers several flexibilities: (1) no schema is needed a priori since every document is self-describing; (2) missing values are not indexed, ideal for sparse data sets; (3) indexing documents with multiple values in the same facet is easy.

A typical inverted index maintains an ordered list of *terms*. Each term points to a *posting list* that includes an ordered list of the *ID*s of documents containing that term. A directory structure built on top of the term list is used for quick term lookups. To perform a search, the text index first locates the terms matching the list of keywords, and then merges the posting lists of those terms to compute the matching document set. Modern text indexes are extremely efficient in merging posting lists (using any combination of union and intersection), through zigzag-style joins [15]. A inverted index can optionally save certain terms of each document in a *store*, indexed by document *ID*. Given a document, one can fetch any stored term quickly.

To support faceted search over an inverted index, one simply needs to map every token in the free text and every $<facet, value>$ pair in the input documents to an index term. Given a faceted query q, the system (1) identifies all matching documents D_q; (2) computes the count on each facet value (singletons only) over D_q; and (3) selects a subset of facet values and the corresponding counts to present to the user. The first step is simply traditional IR. A significant fraction of time is spent in the second step. We are aware of two possible implementations of that step.

A simple implementation is to keep <facet,value> pairs for each document in the store. At runtime, for each matching document d, fetch all facet instances of d by probing the store, and then construct a hash table to compute the count on all facet values. A second implementation is used in Solr [23], an open source engine built on top of Lucene [16]. Solr is currently used by cnet.com to power its faceted search. A user provides to Solr a configuration file that includes all facets on which counts are to be computed. Solr has a unique design that indexes these facets without storing them. To compute facet counts for a query q, Solr enumerates every facet instance $<F, f>$ from the index and intersects its posting list with D_q. From the intersected set, it derives the count on facet value f. To speed up performance, Solr caches each posting list to a bitset and represent the bitset in one of two ways. If the bitset is dense, it is represented as a bitmap.

Otherwise, it is represented as a hash map of document *IDs*.

Compared with the simple implementation, Solr has several advantages. First, the bitset representation of facets is more compact since facet values are not duplicated. Second, because facets are stored separately in different bitsets, it avoids fetching facets that are not needed for counting. There are several reasons why some facets don't have to be counted: (1) Facets are often organized in hierarchies, and users usually only care about higher level of facets when issuing keyword queries; (2) When a user drills into a specific value of facet F, for the new query, it is not necessary to compute counts for F or any of its ancestors since they are all bound to a single value in the new result set. Last but not least, Solr does not need to construct a hash table for computing the counts since they are calculated one group at a time.

4.2 Improving Solr

We adopt the general Solr approach in our implementation, but make some important improvements. One limitation in Solr is that it has to choose a threshold that decides the representation of the bitset. The bitmap representation is faster when performing document intersections, but sometimes consumes more space. It is not obvious how to choose a threshold that optimally balances space and time. Such a tradeoff has been extensively studied in relational databases. Our first improvement is to always represent a bitset as a compressed bitmap using Word-Aligned Hybrid (WAH) code [29]. WAH code is a run-length encoding of a bitmap. In WAH, there are two types of words: literal words and fill words. The former is a verbatim representation of 31 bits and the latter encodes the length of a list of all 0's or 1's in 30 bits. A bitmap is broken into groups of 31 bits first and then converted into a sequence of literal and fill words. Operations on bitmaps such as intersection can be performed on WAH code directly without decoding. As shown in [29], compared to other bitmap compression methods, WAH offers a good balance between space and performance.

A second limitation in Solr is that it has to intersect the matching document set D_q with the bitset of every facet instance. For facets with a large domain, such computation can be quite expensive. Our second improvement is to reduce the number of intersections by building a novel directory structure called *bitset tree* on top of the bitsets of a facet.

Building and Using a Bitset Tree: We create one bitset tree per facet F. A bitset tree is a balanced multi-way tree, in which each node has up to s <bitset, node pointer> entries (s is a fanout parameter). We build a bitset tree bottom-up, level by level. Starting with the leaf nodes, for each bitset b corresponding to facet instance $<F : f>$, we create an entry $<b, null>$. We then divide all entries into groups of size s (the last group may be smaller than s). Precisely how entries are divided will be explained later in this section. For each group, we generate a leaf node holding all entries in that group. We then build the next level of nodes. For each node e in the previous level, we create a new entry $<b', e>$, where b' is computed by bitwise "oring" the bitsets in e. After that, we again divide the newly created entries into groups of size s, and generate a new node to hold all the entries in each group. We continue building the next higher level of nodes until there is only a single new node created. We refer to the last node created as the *root*.

Given D_q as a bitset, we can use the bitset tree on a facet F to guide us to the bitsets on which an intersection with D_q is indeed needed. We begin at the root of the tree and intersect D_q with each bitset at the root node. If the result of an intersection returns an *empty* bitset (all bits are zero), we can prune the corresponding branch since no document in D_q has any facet values in that branch. Otherwise, we follow the corresponding node pointer to check bitsets in the new node. Note that, in general, we may have to follow multiple pointers (resembling the traditional R-Tree). We continue this process until we reach the leaf nodes. We intersect D_q with each bitset there, and return the intersected result if it is not empty. From each returned bitset, we can count the number of documents on a certain facet value.

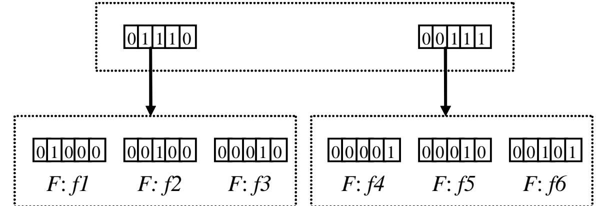


Figure 2: A bitset tree on F with a fanout of 3

To illustrate how a bitset tree works, consider a small repository with five documents. Assume a facet F has only six values. An example bitset tree on F of fanout 3 is shown in Figure 2. Each node is designated by a dashed box. The six bitsets in F are grouped into two leaf nodes. The root node has two entries, each pointing to a leaf node. Suppose D_q is a bitset of 10000. We start at the root node and intersect D_q with the two bitsets there. Both intersected results are empty. We can stop right here and return. We save four bitset intersections in this case (Solr intersects D_q with all 6 bitsets in the leaf nodes). If D_q is a bitset of 01000, we again intersect it with both bitsets in the root node. Since only the result from the left bitset is non-empty, we just need to visit the left leaf node and intersect D_q with the three bitsets there. We perform a total of five bitset intersections, instead of six. Note that our saving is much more significant with more facet values. Occasionally, we pay overhead when using a bitset tree. For instance, if D_q is a bitset of 00100, we have to intersect D_q with all eight bitsets in three nodes and pay two extra bitset operations. However, we try to avoid those cases through an analysis explained later in the section.

Notice that we only have to perform the full bitset intersection at the leaf nodes. At any internal node, as soon as we know that the result of an intersection is not empty, we can stop the current intersection and continue to visit nodes in the next level. We can exploit bitset trees for computing aggregates on combinations of facet values as well. To compute counts on a pair of facets $\{F_1, F_2\}$, we first use D_q to probe the bitset tree on facet F_1. We obtain an intermediate bitset I_{f_1} for each value f_1 in F_1(we exclude empty I_{f_1}). Next, we use each I_{f_1} to probe the bitset tree on facet F_2. From the returned bitsets, we can compute the actual count $C_q(f_1, f_2)$ for every f_2 in F_2.

Analysis of a Bitset Tree: In general, given b bitsets, a bitset tree with fanout s has $log_s(b)$ levels. When using a bitset tree is beneficial, only a small number of branches is actually followed. Thus, we expect to perform $h \cdot s \cdot log_s(b)$ bitset intersections, where h is a small constant. This number is minimized when $s/ln(s)$ is minimized over the natural numbers, i.e., for $s = 3$.

The saving from using a bitset tree depends on how bitsets are grouped into nodes. Ideally, we want to group bitsets in

such a way that shared bits are common within groups, but rare across groups. Similarly to R-Trees [24], constructing a bitset tree with optimal performance is NP-hard. We solve the problem heuristically by picking the first bitset for a node e at random and then continuing to add the next available bitset that shares the most bits with all bitsets in e. Although such a process is quadratic to the number of bitsets, it is not a big concern since bitset trees are built only once.

Given a facet tree T with n leaf nodes, and m documents in D_q, what is the expected fraction of nodes in T (call it V_T) that we have to visit to perform bitset intersection with D_q? If we assume that documents in D_q are selected at random, we expect that fraction at the leaf level to be:

$$\left(1 - \left(1 - \frac{1}{n}\right)^m\right) \quad (1)$$

Applying the same analysis on every tree level, we obtain V_T as:

$$\frac{\sum_{k=0}^{log\ n}\left(1-\left(1-\frac{s^k}{n}\right)^m\right)\frac{n}{s^k}}{\sum_{k=0}^{log\ n}\frac{n}{s^k}} \quad (2)$$

Since most of the nodes are in the leaves, we can use formula (1) to approximate V_T. For n=100,000, the estimate is about 0.5% for m=500, and 5% for $m = 5,000$. For $n = 1,000$, the estimates are 40% and 99% for $m = 500$ and $m = 5,000$, respectively. Obviously, bitset trees are beneficial when the facet domain is relatively large. Less obviously, they are also very useful when we compute counts on facet combinations. Remember that to compute counts on a pair of facets $\{F_1, F_2\}$, we first intersect D_q with bitsets in facet F_1. Each intermediate bitset I_{f_1} tends to be sparse because bits in D_q are spread over all facet instances in F_1. As a result, when intersecting I_{f_1} with bitsets in facet F_2, the bitset tree on F_2 is extremely helpful in reducing the number of intersections. Based on the above estimates, our system dynamically decides at runtime whether to use the bitset tree or to visit all leaf nodes for bitset intersection.

4.3 Other Implementation Details

Aggregates on Facet Combinations: When considering facet combinations, we avoid choosing facets within the same facet hierarchy. Those facets are defined to have functional dependencies and are less likely to be useful when presented together. However, the total number of facet combinations can still be large. From the analysis in Section 3, we observe that the larger the number of unique value combinations in a facet set, the less interesting the facet set tends to be. Therefore, we heuristically prune a facet set if the number of accumulated facet values exceeds a threshold, set proportional to $|D_q|$. Note that we enumerate facet sets in increasing set size, i.e., single facets first, then pairs, then triples, and so on. This way, if a facet set is pruned, it is easy to remove all its supersets from further enumeration.

Supporting Hierarchies: Solr does not directly support facet hierarchies. In our implementation, for a given facet F in hierarchy h, each value f in F is encoded as a term by a full path from root to $<F, f>$ in h. This is very similar to techniques used for indexing XML documents [10]. If a query includes a constraint $F = f$, we can easily identify terms corresponding to descendants of $<F, f>$ in a hierarchy by checking a prefix in each term.

Incremental Support: One potential problem with any bitmap implementation is that it is expensive to update. Fortunately, this is not a big concern for us. An inverted index typically maintains multiple index segments, each responsible for a non-overlapping set of documents. New documents are first buffered in memory and are not immediately searchable. Over time, a new index segment is created using the new documents. Periodically, smaller segments are merged into bigger ones to reduce search overhead and to remove deleted documents. The compressed bitmaps and the bitset trees used in our implementation are maintained per segment, and are constructed each time that a new segment is created.

Hierarchy	Facet	Description
1	id	Patent ID
2	asn	Assignee name
3 (level 1)	cntry	Assignee country
3 (level 2)	state	Assignee state
4	asn_code	Assignee code
5	inv	Inventor name
6 (level 1)	inv_cntry	Inventor country
6 (level 2)	inv_state	Inventor state
6 (level 3)	inv_city	Inventor city
7 (level 1)	cat	Patent category
7 (level 2)	sub_cat	Patent sub-category
8	app_year	Application year
9 (level 1)	g_year	Grant year
9 (level 2)	g_month	Grant month
9 (level 3)	g_day	Grant date
10	nclass	Patent class

Table 1: Facet Hierarchy of Patent dataset

Dataset	# of Docs	# of Facets	Index Size
DBLP	13K	14	11 MB
PAT-1	123K	236	4 GB
PAT-2	494K	475	16 GB
PAT-3	773K	591	32 GB
PAT-4	1,790K	815	52 GB

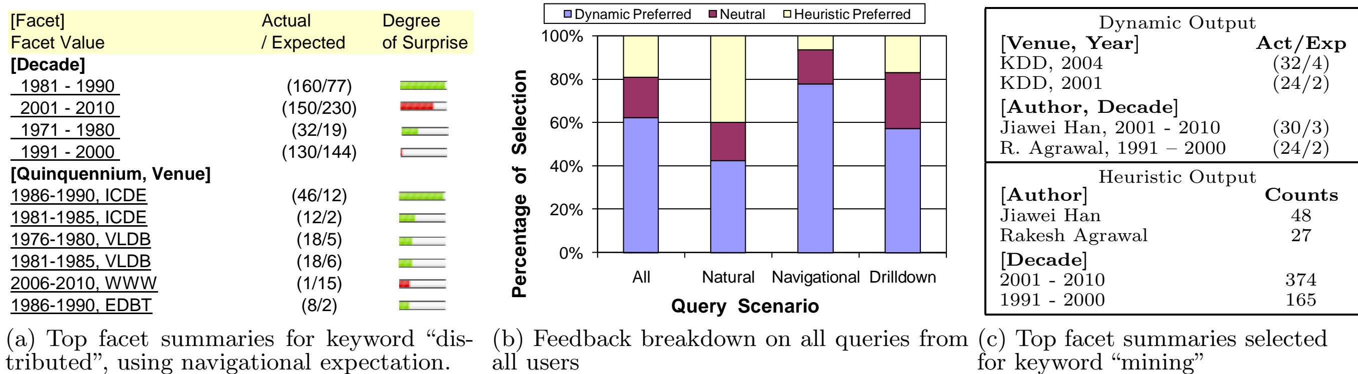
Table 2: Summary of datasets

5. Evaluation

In this section, we conduct a comprehensive evaluation of the dynamic faceted search system that we built, on both relevance and performance using two real data sets. We introduce the experimental setup in Section 5.1. In Section 5.2, we summarize the findings of a user survey on relevance. We describe the performance results in Section 5.3.

5.1 Setup

We select two types of data with a mixture of structured and unstructured information for our tests, DBLP [6] and Patent [18]. The DBLP data contains about 13,000 papers published in 26 venues (e.g., SIGMOD, VLDB, TODS, etc) in the past 30 years. It has 14 facets organized in 6 hierarchies, including author, venue, time (e.g., decade, year), location (e.g., country, city), number of authors per paper, and number of citations per paper. We use the title of each paper as text for keyword searches. The Patent data has about 1.8 million U.S. patents from the past 30 years. We use the full description of a patent as text. There are 16 facets organized into 10 hierarchies. The facet names and the hierarchy levels are given in Table 1. The DBLP data has been manually cleaned and we use it to conduct the user survey. The Patent data is



[Facet] Facet Value	Actual / Expected	Degree of Surprise
[Decade]		
1981 - 1990	(160/77)	
2001 - 2010	(150/230)	
1971 - 1980	(32/19)	
1991 - 2000	(130/144)	
[Quinquennium, Venue]		
1986-1990, ICDE	(46/12)	
1981-1985, ICDE	(12/2)	
1976-1980, VLDB	(18/5)	
1981-1985, VLDB	(18/6)	
2006-2010, WWW	(1/15)	
1986-1990, EDBT	(8/2)	

Dynamic Output	
[Venue, Year]	**Act/Exp**
KDD, 2004	(32/4)
KDD, 2001	(24/2)
[Author, Decade]	
Jiawei Han, 2001 - 2010	(30/3)
R. Agrawal, 1991 – 2000	(24/2)
Heuristic Output	
[Author]	**Counts**
Jiawei Han	48
Rakesh Agrawal	27
[Decade]	
2001 - 2010	374
1991 - 2000	165

(a) Top facet summaries for keyword "distributed", using navigational expectation. (b) Feedback breakdown on all queries from all users (c) Top facet summaries selected for keyword "mining"

Figure 3: Results from the survey

much larger in size and we use it for performance evaluation.

We index all facets in both datasets using Lucene [16], an open source inverted index. Lucene organizes the index structure in segments, each containing a subset of documents. Since the DBLP dataset is relatively small, all documents are indexed into one Lucene index segment. We create four Patent datasets PAT-1 to PAT-4, each including a varying number of patents. Each Patent dataset is indexed by Lucene in segments of approximately 4GB each. In all datasets, if the domain of a facet is larger than 1000, we create extra levels of facets in a hierarchy by grouping domain values alphabetically. Table 2 summarizes the sizing parameters of all the datasets. Our dynamic faceted search engine currently only considers facet sets up to size 2, since most important correlations are between a pair of facet values.

5.2 Results from a User Survey

To understand the usefulness of the dynamic faceted search approach in general, and the effectiveness of our proposed expectations and p-value based measure in particular, we conducted a user survey using a web interface that we built on our engine. Through our interface, a user can choose one of the two types of expectations that we currently support, *natural* and *navigational*, and one of the three weighting methods of aggregating interestingness (as described in Section 3.2). When a user types a keyword, we display a page with three sections. The middle section contains a list of the top matching documents returned by the inverted index. The left section shows the facet summaries selected dynamically by our approach. An example output is shown in Figure 3(a). For each facet instance, in addition to the actual count, we also show the expected count, and a bar indicating the degree of interestingness. The bar is colored green if the expected count is less than the actual one, and red otherwise. Although not shown in the figure, we also provide a one-line explanation to help the user better understand how we set the expectation. For example, for the navigational expectation, we show the query used for computing the expectation. For comparison, we mimic how today's faceted search systems select facets and present in the right section facet summaries selected using a heuristic rule that orders facets just by decreasing counts. We interviewed a total of 15 computer science graduate students, 9 from CMU and 6 from EPFL. We conducted the survey in two parts.

In the first part, we fixed the weighting method to *HybridWeight* and focused on the effectiveness of different expectations and the measure. For each user, we performed tests on three keyword queries. Two of the keywords were provided by us: "*distributed*" and "*mining*", each having about 400 matches. We let the user pick the third keyword, with the only requirement that the number of matching documents be sufficiently large. For each query, the user saw three outputs selected by our dynamic approach, one based on the *natural* expectation, and the other two based on the *navigational* expectation. For the latter two, one used the complete repository to set the expectation, and the other was a drill-in query and used the previous query to set the expectation. For each of the three dynamic outputs and the output selected heuristically, we asked the user for a score between 1 and 10 indicating the perceived relevance of the facet summaries. A score of 1 means not useful at all, and a score of 10 means very relevant. We compare the score of each dynamic output with the heuristic one for each query. There were a total of $15 \times 3 \times 3 = 135$ comparisons. We categorize each comparison as positive, neutral, or negative, depending on whether the dynamic score was higher, equal, or lower than the heuristic one. The breakdown of the results is shown in Figure 3(b).

Overall, about 60% of the answers were positive while only 20% were negative. In particular, users overwhelmingly preferred the outputs from *navigational*. We first summarize why the users liked the navigational approach. The top table in Figure 3(c) shows dynamically selected facet summaries for the keyword "mining". Our users found it useful to know that 2004 and 2001 are important years for KDD since they had more shares of the mining papers than expected. They also found it interesting that two famous authors were more productive in the mining area in 2000s and 1990s, respectively, and had moved on to other topics since then. All users except one liked the fact that our system can show facets in pairs, since they exposed interesting correlations. In comparison, users found the heuristic output given in the bottom table in Figure 3(c) less informative. Similarly, for the keyword "distributed" (the navigational output is given by Figure 3(a)), users liked the fact that facet "Decade" was selected as the top one. The ordering of the facet instances in "Decade", although not according to decreasing counts, clearly demonstrates trend changes in the area of distributed systems. For one of the user selected keywords–"relational"–the most interesting facet instance is dynamically selected to be author "E. F. Codd", since most of his publications are on "relational" with very few in other areas. However, "E. F. Codd" was not selected by the heuristic approach, since many other authors have published more "relational" papers than Codd. When the user drilled into "E. F. Codd", our system used the pre-

vious query to set the expectation and dynamically selected "Citations Range" as the top facet. It becomes immediately clear that papers by Codd are more frequently cited than other papers containing the keyword "relational". All these results convinced our users that the navigational way of setting the expectation and our p-value based measure are useful in discovering interesting or unexpected patterns. Finally, most users expressed interest in the "Ad Hoc" method of setting the expectation, which is not currently supported by our system.

Our dynamic approach also received some negative feedback. Certain facets such as *city* and *type* were seldom wanted by certain users no matter how statistically interesting they were. Similarly, other facets such as *author* were always preferred by some users, independent of the keywords. In the future, we plan to improve our interface by allowing a user to prune uninteresting facets and to "pin" interesting ones interactively. Our system will then dynamically select the remaining facets. Overall, the feedback for the *natural* expectation is neutral. An important reason is that when our users picked the third keyword, they mostly selected keywords from their own areas of research. Therefore, the facet summaries selected using the *natural* expectation were not very surprising or informative to them. Our conjecture is that, had they selected keywords from an unfamiliar area, their reaction might have been different.

In the second part of the survey, we evaluated the trade-offs among different ways of aggregating the degree of interestingness. Each user was asked to score the facet summaries that were selected dynamically for keyword "xml" using each of the three weighting methods. The number of people who preferred *HybridWeight*, *MinWeight* and *AverageWeight* were 7, 6 and 2, respectively. This shows that the interestingness of a facet is strongly influenced by a few of its most interesting instances.

5.3 Performance Results

In this section, we study the performance aspect of the dynamic faceted search, using the Patent dataset. We implemented a total of five different versions in Java. A *simple* version, based on the simple approach described in Section 4.1. We favor the *simple* version by keeping only facets, not text, in the store. A *Solr* version implements the approach used in Solr as described in Section 4.1, but with extension to support hierarchies as described in Section 4.3. The third implementation, called *compressed*, improves *Solr* by representing all bitsets in WAH code. The fourth implementation, called *tree*, improves *Solr* by using *bitset trees* (as described in Section 4.2) to reduce the number of bitset intersections, but keeping the bitset representation used in Solr. The last implementation, called *compressed-tree*, applies both WAH code and bitset-trees on *Solr*. We used Apache Commons-Maths Library [30] to compute the degree of interestingness based on the description in Section 3. We performed all experiments on a 3GHz P4 desktop machine with 1GB of memory and a single disk drive, running Linux.

In all tests, we set the expectation to be navigational (time taken for other kinds of expectations is comparable), and we consider the top level facets in all hierarchies. We break the total elapsed time of a query into pure search time and summary computation time. The former is the time that Lucene takes to compute the matching documents and rank them. Ideally, we want the two times to be of comparable length. In order to have better control of the query size, we sometimes simulate queries by selecting a subset of documents at random. Unless specified otherwise, all queries used in this section are simulated.

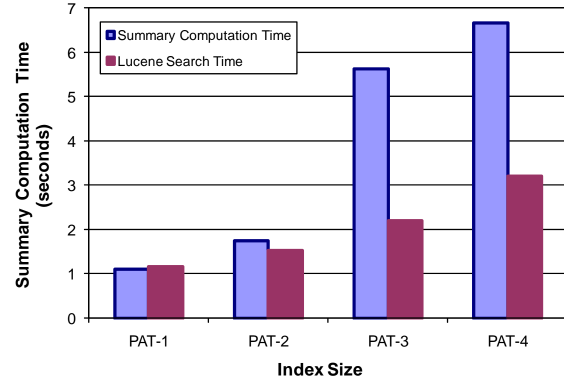


Figure 4: Time breakdown of *Tree* on different index sizes, for a query matching 25,000 documents.

Scaling with Data Size.

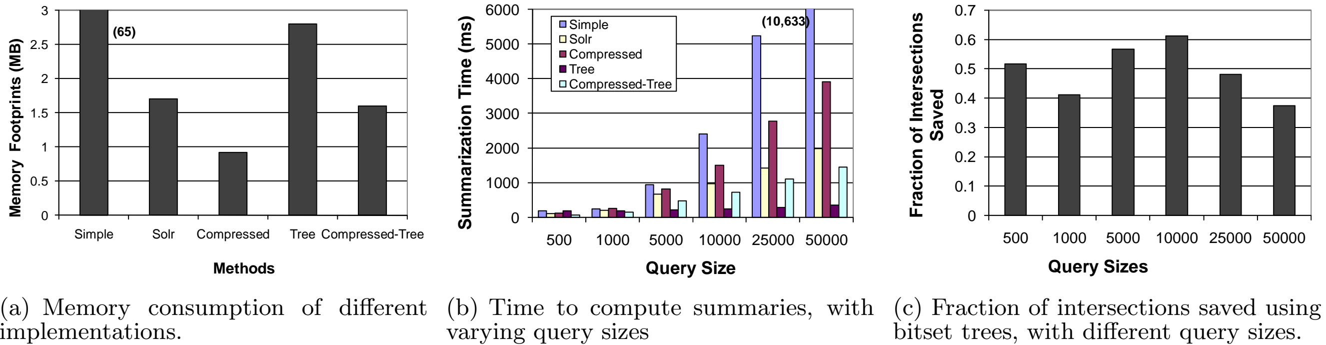
To understand where the time goes in our system, we first run a query that matches about 25,000 documents using the *tree* implementation, and break the total time into search time and summary computation time. The result is shown in Figure 4. When the data size is small, the two times are comparable. As the data size gets larger, the summary computation time starts to dominate. In the rest of the section, we will be focusing mainly on the time to compute facet summaries.

Effect of WAH Compression and Bitset Trees.

In this section, we compare the performance of the five different implementations of dynamic faceted search. In the first experiment, we study the pure CPU overhead of each of the implementations. To achieve this, we consider the smallest dataset, PAT-1, and cache in memory the index terms of top level facets. For the *simple* implementation, we further keep the index store in memory. We also keep the bitset trees in memory for implementations that require them. We present the total amount of memory consumption for each implementation in Figure 5(a). The memory consumption of *simple* is about 65MB, while all other implementations require less than 3MB memory footprint. WAH encoding helps reduce the memory consumption by almost 50%. Bitset-trees consume some extra space because of the directory structure.

Figure 5(b) shows, for each implementation, the summary computation time using randomly selected document sets of varying sizes. As we can see, even when the entire dataset fits in memory, the *simple* implementation is still much slower compared to others. This *simple* implementation operates on one document at a time and has the overhead of maintaining a hash table for computing the counts for all facet instances. All other implementations avoid such overhead since they aggregate counts one group at a time.

Although WAH encoding saves space, it does not improve performance. When performing intersection on WAH-encoded bitsets, we need to maintain an extra Java object to store the run-length information in the current position. Such overhead offsets the benefit from a smaller footprint in memory. Another reason is because the number of documents in PAT-1 is relatively small, about 110,000. When encoding bitsets on such a dataset, many 31-bit chunks have at least a 1-bit



(a) Memory consumption of different implementations.

(b) Time to compute summaries, with varying query sizes

(c) Fraction of intersections saved using bitset trees, with different query sizes.

Figure 5: Performance comparisons for PAT-1 dataset (memory resident)

and have to be represented as a literal word. Therefore, the compression that WAH can achieve is limited. We'd like to investigate in the future whether using byte-aligned encoding performs better for small datasets.

Exploiting bitset-trees improves performance significantly. Although there is no facet with a very large domain in this dataset, bitset-trees are very useful when computing counts on facet pairs. After intersecting the query bitset with the bitsets on the first facet, the intermediate bitsets become sparse. When using such intermediate bitsets to probe the second facet, the bitset tree on the second facet can prune many unnecessary bitset intersections. In Figure 5(c), we show the fraction of total bitset intersections that are saved because of the bitset-trees. On average, the bitset-trees give us about a 40% saving. We note that these savings may not translate proportionally to savings in time since the cost of bitset intersections also depends on the density of the bitsets.

To summarize the in-memory test, the *tree* implementation gives the best performance for most of the time, and scales well as query sizes approach the total number of documents in the index. WAH compression is mainly useful when the query is much smaller than the number of documents in the index.

Next we study the performance of each of the methods on the large data set PAT-4. For a fair comparison, we allocate a 50MB buffer pool to each method. Data that can't be cached in memory has to be paged out and read back when accessed again. The *simple* implementation uses the buffer pool to cache documents fetched from the store. The rest of the implementation caches bitsets and/or bitset-trees in the buffer pool. We measure the total elapsed time on summary computation for each implementation, with varying query sizes.

Figure 6 shows the results in seconds. As expected, *simple* performs the worst across the board. Most of its overhead comes from the additional random I/Os when fetching facets from the document store. The document buffer pool is not very effective since temporal locality is low. On this much larger data set, compression becomes very effective. With WAH encoding, the *compressed* implementation runs about twice as fast as *Solr*, while consuming slightly less memory. Also, the benefit from using bitset-trees is almost as large as compression. Because we use formula (1) in Section 4.2 to dynamically decide whether to use a bitset-tree for probing or not, the *tree* implementation always performs better than *Solr*. Combining compression and bitset-trees together gives the best performance. However, the benefits are not additive. This is because using WAH code, the bitset intersection on internal nodes in a bitset-tree becomes more expensive since they tend to be denser than the leaves. Nevertheless, the combined implementation in general runs at least two to three times as fast as *Solr* and orders of magnitude faster than *simple*.

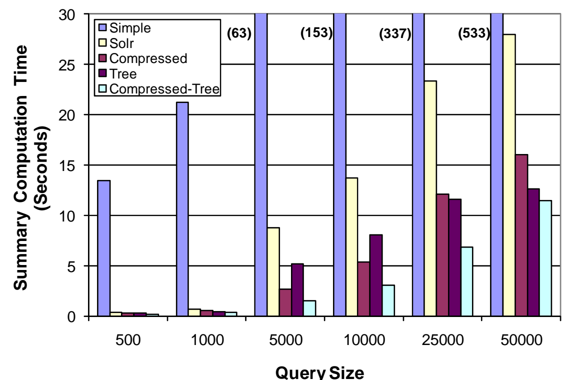


Figure 6: Time to compute summaries on PAT-4 dataset, with different query sizes

From the results of the experiments we conclude that the *simple* method does not give acceptable performance in any scenario. We found that our techniques, including WAH encoding and bitset-trees, provide significant performance improvement over existing solutions. On a single computer, our best implementation of the dynamic faceted search can keep a user session interactive for queries matching 5000 documents or less. In the future we intend to make the process interactive for even larger queries by computing the interestingness on multiple servers in parallel.

6. Related Work

There exist several commercial implementations of faceted search. Both Endeca [8] and IBM's Websphere content discovery server (formally iPhrase) [25] are mainly intended for managing product catalogs in e-commerce sites and thus often do not have a large repository to deal with. Google Base [2] provides a faceted search interface. After a user types in a keyword, certain facets are presented to the user for further navigation. However, based on our knowledge, the facet selection in those systems is primitive, and none of them automatically and dynamically selects interesting facets on a per query basis as we do. The Flamenco system [9] also implements a faceted search interface, but mostly addresses the user interface issues.

Instead of returning a long list of matching documents, search sites such as Clusty [3] group similar documents in

the result together and create a faceted-like display on the fly. Groups are dynamically generated through a taxonomy on the text of the result set. In comparison, our work discovers useful information from pre-identified facets. Also, Clusty does not consider a user's prior knowledge when generating the groups.

There exists work [1, 12] on extending relational databases to support IR-style queries. The focus on retrieving the top K matching tuples. More recently, [28] studies how to integrate faceted search into OLAP analysis. A keyword query is first converted to joins of dimensional and fact tables. The most interesting dimensional attributes and their values are discovered from the join results, by comparing aggregates of a measure at different levels. In comparison, our work provides discovery-driven analysis in the context of faceted search.

Text analytics [5][27] tries to automatically extract structured information from text by using a variety of technologies including statistical and rule-based natural language processing, information retrieval, machine learning, ontologies, and automated reasoning. It can derive not only basic entities such as persons, locations, and organizations, but also relationships between those entities. Such extracted information allows more precise queries. If we model each type of extracted information as a new facet, the number of facets associated with a repository becomes significantly larger. Therefore, automatically selecting interesting facets for a given query becomes much more important.

There exists work in the data mining area on discovering interesting information. For example, [20] describes how to discover interesting association rules by pruning uninteresting ones. Uninteresting rules are those whose removal results in the elimination of other rules the most. The work in [14] deals with identifying interesting missing association rules. There has been research on identifying interesting patterns through time series analysis and a good survey can be found at [19]. Finally, [13] discovers correlated attribute pairs in a relational database. It applies chi-squared analysis for identifying correlations and employs random sampling for efficiency. However, those analyses are not driven by user queries.

7. Conclusion and Future Work

We develop a novel dynamic faceted search system to support OLAP-style discovery-driven analysis on a large set of structured and unstructured data. We propose an intuitive and effective way of measuring "interestingness" and a novel navigational method of setting a user's expectation. The feedback from a user survey validates that our approach is promising. By exploiting WAH codes and bitset trees, we built an efficient runtime engine on top of an inverted index. We want to pursue a couple of directions in the future. First, as mentioned in Section 5.2, we'd like to improve the facet selection by incorporating user feedback. Second, we'd like to explore how to extend the aggregates to functions other than count, such as sum or average on some numerical measures. Finally, for even larger data sets, we want to investigate how to support dynamic faceted search in a distributed environment.

8. References

[1] Sanjay Agrawal, et al: DBXplorer: A System for Keyword-Based Search over Relational Databases. In ICDE 2002: 5-16

[2] http://base.google.com/

[3] http://clusty.com/

[4] Thomas M. Cover and Joy a. Thomas. Elements of Information Theory. 1992.

[5] W. Dakka, et al: Automatic discovery of useful facet terms. In SIGIR Faceted Search Workshop, 2006

[6] DBLP dataset: http://dblp.uni-trier.de/xml/

[7] Bradley Efron and Robert J. Tibshirani: An introduction to the bootstrap. Chapman & Hall, 1993

[8] http://endeca.com/

[9] The Flamenco Search Interface Project. http://flamenco.berkeley.edu/

[10] Roy Goldman and Jennifer Widom, DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases, In VLDB 1997

[11] Jim Gray, et al: Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. ICDE 1996: 152-159

[12] Vagelis Hristidis, Yannis Papakonstantinou: DISCOVER: Keyword Search in Relational Databases. VLDB 2002

[13] Ihab F. Ilyas, et al: CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies. In SIGMOD 2004

[14] Bing Liu, et al: Identifying Interesting Missing Patterns. In PAKDD, 1997

[15] Xiaohui Long et al: Optimized Query Execution in Large Search Engines with Global Page Ordering, VLDB 2003

[16] http://lucene.apache.org/

[17] Tom M. Mitchell: Machine learning. McGraw-Hill,1997

[18] Patent dataset: http://www.nber.org/patents

[19] John Roddick, et al: A Survey of Temporal Knowledge Discovery Paradigms and Methods. In IEEE TKDE, 2002

[20] Sigal Sahar: Interesting Via What is not Interesting, In KDD 1999.

[21] Sunita Sarawagi: User-Adaptive Exploration of Multidimensional Data. VLDB 2000: 307-316

[22] Sunita Sarawagi, et al: Discovery-Driven Exploration of OLAP Data Cubes. In EDBT 1998

[23] http://incubator.apache.org/solr/

[24] Jayme Luiz Szwarcfiter: Optimal multiway search trees for variable size keys, In Acta Informatica, Vol, 21, No. 1, 1984

[25] IBM Websphere content discovery server (formally iPhrase), http://www.ibm.com/software/data/discovery/content/

[26] Witten, I.H., et al: Managing Gigabytes: Compressing and Indexing Documents and Images. 1994

[27] http://www.research.ibm.com/UIMA/

[28] Ping Wu, et al: From Keyword-based Retrieval to Keyword-driven Analytical Processing: A Multi-faceted Approach. SIGMOD 2007

[29] Kesheng Wu, Ekow J. Otoo, Arie Shoshani: Optimizing bitmap indices with efficient compression. ACM Trans. Database Syst. 31(1): 1-38 (2006)

[30] http://commons.apache.org/math/