Index Structures for Querying the Deep Web

Jian Qiu Feng Shao Misha Zatsman[1] Jayavel Shanmugasundaram

Department of Computer Science
Cornell University
{jianq,fshao,jai}@cs.cornell.edu mz36@cornell.edu

1. INTRODUCTION

Most current web search engines can only crawl, index, and query over static web pages, also referred to as the "surface web". A large fraction of the Internet data, however, is stored in Internet-attached databases or the "deep web". For example, the data about auctions in ebay.com is stored in an Internet-attached database, but is not visible to current web search engines. Some studies estimate that the size of the deep web is 400-500 times the size of the surface web [1]. Consequently, current web search engine technology can be used to query only a small fraction of all the data available off the Internet.

As part of the Deep Glue project at Cornell University, we are building a query engine for deep web data sources. A key component of the Deep Glue system, and the focus of this paper, is the **indexer module**. Given a user query, the indexer module uses index structures to identify a *superset* of the deep web data sources that are relevant to the user query. The query module (not described here) then evaluates the user query by contacting the potentially relevant data sources identified by the indexer module.

Index structures for the deep web differ in two fundamental respects from traditional inverted list index structures used by current web search engines. First, index structures for the deep web have to deal with *structured data* because the underlying database is typically richly structured and typed; this is in contrast to the mostly unstructured HTML data available off the surface web. Second, index structures for the deep web must deal with data volumes that are *orders of magnitude larger* than that for the surface web [1]. To address the above issues, we devise new index structures for the deep web. The index structures understand the structure/typing of the underlying data, and can thus be used to support both equality and range queries. The index structures can also be heavily compressed so that their space requirements are far less (up to an order of magnitude less) than the size of the original index.

Aggressive compression of our index structures is possible because we allow the indexing module to return a *superset* of the data sources that are relevant to a user query. By returning a superset of the relevant data sources, no relevant data sources are missed. However, the query module has the extra overhead of contacting some data sources that are not relevant to the query. To evaluate this tradeoff, we present preliminary experimental results over 1000 synthetically generated deep web data sources. Our results are promising and indicate that if we compress the index size by a factor of 10, the number of extra data sources contacted is less than 10 (more details are presented in Section 4). Further, the compression factors in our index structures are tunable, and can be used to tradeoff index size for precision.

The rest of this paper is organized as follows. In Section 2, we describe our system and query model, and in Section 3, we describe various deep web index structures. In Section 4, we experimentally evaluate the performance of the index structures. In Section 5 we discuss related work, and in Section 6, we present our conclusions and outline directions for future research.

2. SYSTEM AND QUERY MODEL

Figure 1 shows the architecture of the Deep Glue system. Given a user query, the query engine contacts the indexer to determine a superset of the data sources that are relevant to the user query. The query engine then evaluates the user query by contacting the relevant data sources. The indexer constructs the deep web index structures offline; this can be done either by crawling deep web data sources [13] or by using some previously agreed upon protocol for transferring index data [4]. This paper focuses on the organization of the deep web index structures once all the indexing data is obtained from the deep web data sources.

For the purposes of this paper, we make the following assumptions. We assume that the deep web data sources are pre-classified into a set of domains such as online car dealers, online auctions, and online travel agents. We further assume that all the deep web data sources within the same domain export their data using the same logical database schema (the mapping from different physical database schemas to the same logical database schema could be done using mediators [16]). For simplicity, we also assume that the database schema conforms to the relational data model. In the relational schema, a subset of the attributes are referred to as the indexing attributes;

International Workshop on the Web and Databases (WebDB).
June 12-13, 2003, San Diego, California.

[1]Currently at Google

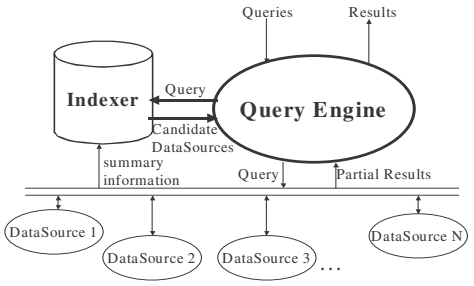

Figure 1: System Architecture

value	data sources
1	d6,d7,d8
2	d2,d3,d4,d6
3	d4
4	d1,d4,d5
5	d2,d3,d6
6	d1,d7,d8

Figure 2: Uncompressed Index

value	Cluster id
1	c1
2	c2
3	c3
4	c3
5	c2
6	c1

Cluster id	data sources
c1	d1,d6,d7,d8
c2	d2,d3,d4,d6
c3	d1,d4,d5

Figure 3: Value Clustered Index

these could be attributes such as price, date, make, model, isbn number, etc. The values of the indexing attributes are indexed by the Deep Glue system, and are used to direct queries to the relevant data sources.

In this paper, we focus on equality and range selection queries on a single indexing attribute. For example, in the domain of online car dealers with relational schema Car(Id, Make, Model, Year, Price), we can answer queries such as "find all year 2003 cars (Year = 2003)" or "find all cars that cost less than $1000 (Price < 1000)", where date and price are each indexing attributes. We do not consider multiple attribute selection queries or join queries in this paper, and this is a subject for future research.

3. INDEX STRUCTURES

We now turn to the main focus of this paper, which is devising space-efficient index structures for evaluating equality and range queries over deep web data sources. We first present a simple uncompressed index structure, and then present various techniques for compressing this index structure. Please note that all the compression techniques explored in this paper are complementary to existing database and inverted list compression techniques such as [2][5][10][11][14][15]. Such compression techniques can be applied on top of our indices to achieve further compression.

3.1 Uncompressed Index (UI)

An Uncompressed Index (UI) can be created as follows. For each distinct value *v* of the indexing attribute, we can store the list of data sources that contain at least one tuple that has the value *v* for the indexing attribute. A small example of such an index is shown in Figure 2. Data sources d6, d7, and d8 have at least one tuple that has value 1 for the indexing attribute, data sources d2, d3, d4, and d6 have at least one tuple that has value 2 for the indexing attribute, and so on. Building a B+-tree index on the value column will enable equality and range queries to be efficiently handled. For handling equality queries, we can simply lookup the list of databases that contain the desired value. For handling range queries, we can union the set of data sources for all the values in the desired range.

In realistic deep web scenarios, the number of distinct values of the indexing attribute and the number of data sources are both likely to be large. Since the index structure has to be built for *each* indexing attribute, the overall size of the index is also likely to be very large, and in the worst case, could approach the entire size of the indexed deep web. Consequently, we need to explore techniques for effectively compressing the index structure.

Note that we cannot simply use a compression utility such as gzip to compress each index structure. If we used gzip to compress the index, then the entire index would have to be uncompressed for *every query* over the indexing attribute, which would make the index lookup time very expensive.

We can, however, use database compression techniques that compress individual fields of data [2][5][11][15]. We use such techniques, specifically [11], for compressing the lists of data sources. We could also use inverted list compression techniques such as [10][14]. However, we go significantly beyond this and provide an *additional* compression factor of 10 in space while still efficiently supporting equality and range queries.

3.2 Value Clustered Index (VCI)

The key idea behind the Value Clustered Index (VCI) is the following. In UI, the lists of data sources associated with different values are often similar (although not necessarily the same). This captures the intuition that "closely related" values are often stored in "closely related" data sources. For example, if the indexing attribute is the ISBN number of a book, then the ISBN numbers of out-of-print books are likely to have a very similar list of data sources, consisting mostly of online retailers specializing in out-of-print books. In the small example in Figure 2, values 1 and 6 have a similar list of data sources, as do values 2 and 5, and values 3 and 4.

Given that different values can have a similar list of data sources, we can save space by "grouping" or "clustering" such values together, and storing the associated list of data sources just once for each cluster (instead of repeating the list for each value in the cluster as in UI). As an illustration, consider the UI in Figure 2. Assume that the values are divided into three clusters, with cluster c1 containing values 1 and 6, cluster c2 containing values 2 and 5, and cluster c3 containing values 3 and 4. The resulting VCI is shown in Figure 3. Note that each value has a cluster id associated with it. Each cluster id in turn is associated with

value	Cluster id
1	c1,c3
2	c1,c2
3	c2
4	c2,c3
5	c1
6	c3

Cluster id	data sources
c1	d2,d3,d6
c2	d4,d5
c3	d1,d7,d8

Figure 4: DataSource Clustered

value	Cluster id
1	c3
2	c1,c2
3	c2
4	c2
5	c1
6	c3

Cluster id	data sources
c1	d2,d3,d6
c2	d1,d4,d5
c3	d1,d6,d7,d8

Figure 5: Value-DataSource Clustered

value	Cluster id
1	c1
2	c2
3	c2
4	c2
5	c3
6	c3

Cluster id	data sources
c1	d6, d7,d8
c2	d1,d2,d3,d4, d5,d6
c3	d1,d2,d3,d6, d7,d8

Figure 6: Histogram Based

a list of data sources. Thus, the list of data sources associated with a value can be determined by first looking up the cluster id for the value, and then looking up the list of databases associated with the cluster id (in reality, cluster id can be a physical disk pointer making the look up for the list of databases very efficient). A B+-tree index on the value will again enable equality and range queries.

It is important to note that in VCI, the list of data sources associated with each cluster is the *union* of the list of data sources associated with each value in the cluster. For example, for cluster c1, the list of data sources (d1,d6,d7,d8) is the union of the list of data sources associated with value 1 (d6,d7,d8) and value 6 (d1,d7,d8). By using the union, we ensure that no relevant data sources are missed for a query over a given value. However, some data sources that are not relevant to a query can also be returned. For example, a query for value 1 will have to contact d1 even though d1 does not contain that index attribute value. VCI attempts to minimize such "false positives" by clustering together only values that have a similar list of data sources.

Another interesting feature to note about VCI is that it allows space to be traded off for precision. At one extreme, if each distinct value is mapped to a separate cluster, VCI is the same as UI, and there are no false positives. At the other extreme, all values are mapped to the same cluster resulting in a very small index, but every data source will have to be contacted for every query resulting in a large number of false positives. In practice, a space-precision tradeoff between the two extremes will likely be most desirable, and this can be tuned to the needs of the application.

Thus far, we have described VCI assuming that we have some way of clustering values with similar lists of data sources. Thus, a practical issue is determining an efficient way to do this clustering that (a) scales to large data sets, and (b) attempts to minimize the number of false positives in each cluster. To handle the scalability issue, we rely on existing scalable clustering techniques such as Birch [17]. Birch is a general algorithm for clustering that can handle any clustering domain, as long as the notions of a centroid (the "mid-point" of a cluster), radius (a measure of the quality of a cluster) and cluster distance are defined for that domain. We now define centroid, radius and distance for VCI so that it attempts to minimize the number of false positives for queries. We use *ds(v)* to denote the set of data sources associated with value *v*. For a cluster having the set of values V:

$$centroid(V) = \bigcup_{v \in V} ds(v)$$

$$radius(V) = \frac{\sum_{v \in V} |centroid(V) - ds(v)|}{|V|}$$

The centroid of a cluster is the union of the data sources associated with the values in the cluster, or simply the data source list associated with a cluster. The radius of a cluster is the sum of the number of false positives for each value in the cluster, normalized by the number of values in the cluster. Thus, as desired, a cluster with small radius is of high quality, while a cluster with large radius is of low quality. The distance between two clusters V1 and V2 is:

$$distance(V1, V2) = |V1| \times |centroid(V2) - centroid(V1)| + |V2| \times |centroid(V1) - centroid(V2)|$$

The distance between two clusters is the additional number of false positives that would occur if the two clusters were merged together. This is computed as the sum of the following two quantities (a) the number of values in the first cluster multiplied by the number of data sources that only occur in the second cluster, and (b) the number of values in the second cluster multiplied by the number of data sources that only occur in the first cluster.

Using the above definitions of centroid, radius, and distance, we used Birch to generate VCI clusters that attempt to minimize the number of false positives. Further, by setting a threshold on the maximum allowable radius, we can control the space-precision tradeoff by choosing high quality clusters with the associated space overhead, or choosing lower quality clusters and getting more space compression.

3.3 DataSource Clustered Index (DCI)

VCI achieves space compression by clustering values that have similar data source lists. An alternative means of compression is possible by clustering data sources that have similar values. The intuition here is that closely related data sources (such as amazon.com and barnesandnoble.com) often have closely related sets of values (such as book ISBN numbers). Thus, instead of storing these data sources separately in the data source list,

they can be clustered and a single cluster id can be stored in the data source list.

In our example in Figure 2, data sources 1, 7, and 8, have a similar (though not identical) set of values and can be clustered together. Similarly, data sources 2, 3 and 6 have a similar set of values, and so do data sources 4 and 5. The DCI index structure resulting from clustering these related data sources together is shown in Figure 4. Note that a value is associated with a cluster id if *at least* one data source in the cluster contains the value – this is to ensure that no relevant data sources are missed during querying. This also means that DCI could have false positives because if a value is associated with a cluster id, then not every data source in that cluster necessarily contains that value. For example, in Figure 4, the value 3 is associated with cluster c2, and cluster c2 contains data source d5 even though d5 does not contain the value 3.

DCI offers a space-precision tradeoff similar to VCI. If the number of clusters equals the number of data sources, the index structure is identical to UI (with no false positives and associated space overhead); if the number of clusters is 1, then there are many false positives but little space overhead; intermediate tradeoff points can be got by adjusting the number of clusters. Like in VCI, the Birch algorithm is used for clustering databases. The formulae for centroid, radius, and distance are symmetric to VCI (using values in place of data sources and vice versa), and are presented in the Appendix.

3.4 Value-DataSource Clustered Index (VDCI)

Consider a scenario of online book data sources, where there is a natural clustering of data sources based on the ISBN numbers of the books the data sources carry (the ISBN numbers are in the categories popular books, out of print books, collector's books, etc.). Now consider a data source that carries both popular books and out of print books. Under both VCI and DCI, the data source will have to be clustered along with *either* the popular books cluster, *or* the out of print books cluster. There is no way to specify that the data source is part of one cluster for popular books, and another cluster for out of print books. VDCI attempts to address this limitation of VCI and DCI. In VDCI, a cluster is a set of values V *and* a set of data sources D, and implies that the data sources D are related with respect to the values V (or equivalently, the values V are related with respect to the data sources D). Thus, VDCI generalizes both VCI and DCI.

As an illustration of VDCI, consider again the example in Figure 2. Consider the data sources d2, d3, and d6. These data sources are similar in that they all contain values 2 and 5. Thus, VDCI allows the cluster ({2,5},{d2,d3,d6}) to be formed. In addition, d6 can also be part of the cluster with d1, d7 and d8 with respect to the values 1, resulting in the cluster ({1,6},{d1,d6,d7,d8}). Let us also assume that the cluster ({2,3,4},{d1,d4,d5}) is created. Then the resulting VDCI index structure is shown in Figure 5.

Since VDCI clusters with respect to each value-data source combination, a cluster is defined as a set of (value, data source) pairs. The definition of the centroid, radius and distance is a combination of the formulae for VCI and DCI and is presented in the Appendix.

Although VDCI generalizes both VCI and DCI, it suffers from the drawback that it needs to cluster each (value, data source) pair separately in order to find the right clusters. This is in contrast to VCI/DCI, which only need to cluster each value/data source separately.

3.5 Histogram Based Index (HBI)

VCI and VDCI do not consider the ordering among values when constructing clusters. Consequently, two adjacent values in the value space are no more likely to be clustered together (based on the data source list) than distant values in the value space. However, since range queries select adjacent values, it may be beneficial to cluster adjacent values together. HBI is based on this intuition, and clusters adjacent values into the same cluster. This is similar to grouping attribute values into "buckets" in histograms in relational systems [12]. Figure 6 shows the HBI structure for the UI in Figure 2 where value 1 is in one cluster, values 2, 3 and 4 are in a second cluster, and values 5 and 6 are in a third cluster.

Since HBI is limited to clustering only adjacent values in the same cluster, it loses some of the flexibility of VCI and VDCI. To minimize the effects of this loss of flexibility, it is important to determine the cluster boundaries effectively. For this, we introduce a threshold parameter for constructing histograms. All the values for the indexing attribute are sorted first and inserted into the clusters (histogram buckets) in ascending order. Whenever the inclusion of a new value causes average number of false positives of the current cluster to exceed the threshold, we create a new cluster for the new value. Thus we can control the index size by specifying the proper threshold for clusters.

4. EXPERIMENTAL EVALUATION

We now experimentally evaluate our index structures. There are three metrics that we use in our evaluation. The first is *index creation time*. The second is *compression factor*, which is the ratio of the size of the uncompressed index (UI) to the size of the compressed index. The third is *false positives*, which is the average number of false positives for equality/range queries.

4.1 Experimental Setup

We use synthetic data for our evaluation. We chose to use synthetic data for the following reasons. First, we can vary parameters to illustrate the tradeoffs between the different approaches. Second, we were not able to easily obtain real data from hundreds/thousands of deep web data sources, although this is something we are currently pursuing.

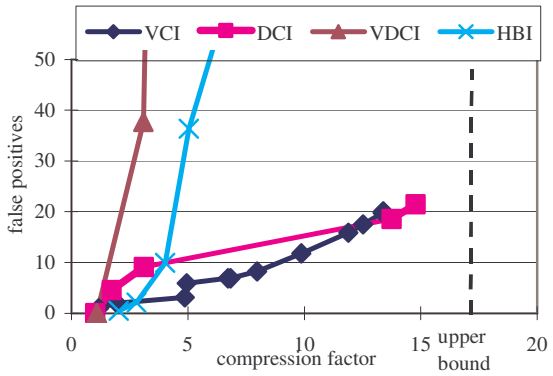

Figure 7 Equality Queries

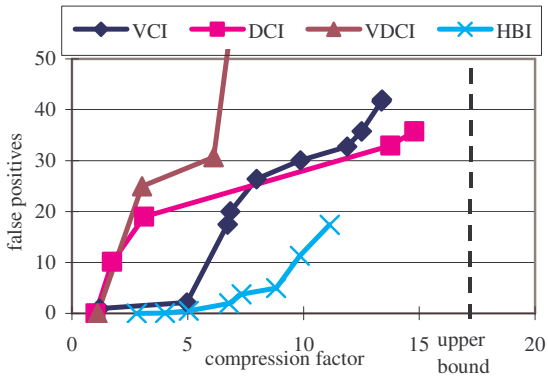

Figure 8 Range Queries

The synthetic data set was generated using the following parameters. The ***number of values*** is the number of distinct values for the indexing attributes (default value is 100,000). The ***number of data sources*** is the number of deep web data sources (default is 1000). The ***number of pairs*** is the number of unique (value, data source) pairs (default is 4,000,000). This is a measure of the data size, but is usually smaller than the number of tuples because duplicate values in the same data source are only counted once.

The ***number of groups*** is the number of logical groups among the data sources (default is 20). A logical group captures the notion that a set of data sources can have a similar distribution of values for the indexing attribute. Thus, the data values for each data source in a group are picked from the same Gaussian distribution characteristic of that group. The mean and standard deviation of the Gaussian distribution for a group is chosen so that there is at most a 5% overlap between the distributions of any two distinct groups. The ***group mode*** is the number of logical groups that each data source belongs to (default is 1). The group mode captures the intuition that a given data source can be similar to one set of data sources with respect to one set of values (such as popular books), and similar to another set of data sources with respect to another set of values (such as out-of-print books).

Finally, the ***value correlation*** is a measure of how the ordering in the value space maps to the ordering of values over which Gaussians are defined – this is especially important for range queries. The value correlation is defined in terms of the normalized number of permutations to go from the value space ordering to the Gaussian ordering. A value correlation of 1 implies full correlation, while a value correlation of 0 implies no correlation (default is 0.2). Using the default settings, the size of the uncompressed index (UI) is 8MB**.**

False positives for equality queries are measured by averaging the false positives for all distinct values. False positives for range queries are measured by averaging the false positives for all possible range queries of a given size. The ***size of range*** specifies the number of distinct values selected by the range query (default is 500).

We implemented all the index structures using C++, and our experiments were performed using a 2.8 GHz Pentium IV processor running Windows XP, with 1GB of main memory and 80GB of disk space.

4.2 Experimental Results

We now present some of our experimental results (more results are presented in the Appendix).

4.2.1 Index creation time

The index creation time for UI using default parameter values was 15 seconds. The index creation time for compressed index structures using the default parameter values and a compression factor of 10 was: VCI – 15 minutes, DCI – 3 minutes, VCDI – 3 hours, HBI – 2.5 minutes. VCDI has a much larger index creation time because it clusters (value, data source) pairs, as opposed to just the values (VCDI and HBI) or just the data sources (DCI). Since the number of (value, data source) pairs is much larger than the number of values or data sources, the clustering time for VCDI increases correspondingly. Although clustering is done offline, this overhead for VCDI could potentially become a bottleneck.

4.2.2 Compression factor and false positives

Figure 7 shows how the number of false positives varies with the compression factor for equality queries, assuming default parameter values. The standard deviation is less than 10, and is not shown. VCI performs the best, with only 10 false positives for a compression factor of 10. DCI performs slightly worse than VCI – this is because there are fewer data sources than there are values, and thus there is less flexibility in forming clusters in DCI. The false positives for HBI increase rapidly with compression because HBI only clusters adjacent values, even if they have very different data source lists.

The most surprising result is the bad performance of VDCI, even though it theoretically generalizes both VCI and DCI. VDCI performs badly for the following reasons. First, VDCI has to deal with a much larger number of data points, which increases the chance of creating bad clusters using scalable one or two pass clustering algorithms. Second, VDCI appears to have too many degrees of freedom with respect to choosing sets of values and data sources. We experimented with different distance functions for VDCI clustering, but could not overcome this fundamental problem. Although we are still exploring other ways to solve this issue, this coupled with the clustering time argues against using VDCI.

Note that false positives increase rapidly for all indices beyond a compression factor of 10. This increase is because the indices are approaching their theoretical upper bound for compression. This upper bound comes about because each index has to store the mapping from the value to the cluster id, for each value. The size of this mapping is thus the upper bound for compression, which is 480KB for the default settings.

Figure 8 shows the space-precision tradeoff for range queries. The results are similar to equality queries, except that HBI now outperforms even VCI. The good performance of HBI is attributable to the fact that it chooses clusters based on the value ordering. Since range queries go over the same value ordering, this reduces the number of false positives.

5. RELATED WORK

This work is related to the body of work on distributed databases [8] and information integration [3][7]. However, our focus is on *identifying relevant data sources* efficiently, while prior work has mostly focused on query processing *once the relevant data sources are identified*.

The GlOSS system [6] supports data source discovery for text documents. The Niagara system [9] uses an index structure that identifies a superset of data sources relevant to a user query. However, these systems do not address the issue of compression, and do not support structured queries such as range queries.

There has been work on field-level lossless compression in databases/indices [2][5][11][15] and inverted list [10][17]. Our work builds upon this work in compressing individual fields (see Section 3.1), but goes significantly beyond and achieves up to a factor of 10 further compression by trading off space for a little loss in precision. The relationship of our work to traditional compression techniques such as gzip is described in Section 3.1.

6. CONCLUSION AND FUTURE WORK

We have presented new space-efficient index structures for querying the deep web. The index structures support structured queries such as equality and range queries, and can be compressed by up to a factor of 10. However, this compression implies that extra data sources may have to be contacted during query processing. Our preliminary experimental results indicate that this overhead is minimal even for large compression ratios.

Our index structures also illustrate another tradeoff in handling equality vs. range queries. The VCI index based on clustering significantly outperforms the HBI index based on histograms for equality queries, and vice versa for range queries. We are thus exploring ways to combine the benefits of cluster-based and histogram-based indices so that they are effective for both types of queries.

Other avenues for future work include extensions to multiple attribute queries and joins, incremental index maintenance, using structured vocabularies in the index structure, incorporating the notion of ranking, and evaluating the index structures using real data sets.

7. REFERENCE

[1] M. Bergman, “The Deep Web: Surfacing Hidden Value”, Technical White Paper, http://www.brightplanet.com/deepcontent/tutorials/DeepWeb.

[2] Z. Chen, J. Gehrke, F. Korn, "Query Optimization In Compressed Database Systems", SIGMOD 2001.

[3] D. Florescu, et al., “Database Techniques for the World Wide Web: A Survey”, SIGMOD Record 27(3), 1998.

[4] Froogle: http://www.froogle.com

[5] J. Goldstein et al, "Compressing Relations and Indexes", ICDE 1998.

[6] L. Gravano, H. Garcia-Molina, A. Tomasic, "GlOSS: Text-Source Discovery over Internet", TODS 24(2), 1999.

[7] A. Gupta, V. Harinarayanan, A. Rajaraman, “Virtual Database Technology”, ICDE 1998.

[8] D. Kossmann, “The State of the Art in Distributed Query Processing”, ACM Computing Surveys 32(4), 2000.

[9] J. Naughton et al, “The Niagara Internet Query System”, IEEE Data Eng. Bulletin, 24(2), 2001.

[10] G. Navarro, et al., “Adding Compression to Block Addressing Inverted Indexes”, Information Retrieval, 3:49-77, 2000

[11] P. O’Neill, D. Quass, “Improved Query Performance with Variant Indices”, SIGMOD 1997.

[12] V. Poosala, et al., “Improved Histograms for Selectivity Estimation of Range Predicates”, SIGMOD 1996.

[13] S. Raghavan, H. Garcia-Molina, “Crawling the Hidden Web”, VLDB 2001.

[14] P. Weiss, Size Reduction of Inverted Files Using Data Compression and Data Structure Reorganization. PhD thesis, George Washington University, 1990.

[15] T. Westmann, et al., "Implementation and Performance of Compressed Databases", SIGMOD Record, 2000.

[16] G. Wiederhold, “Mediators in the Architecture of Future Information Systems”, IEEE Computer 25(3), 1992.

[17] T. Zhang, et al., “BIRCH: Efficient Data Clustering Method for Very Large Databases”, SIGMOD 1999.

APPENDIX A FORMULAE FOR DCI

We present the formulae for *centroid*, *radius*, and *distance* in DCI. Let *dv(d)* be the set of data values associated with a data source *d* and let a cluster be a set of data sources D, then

$$centroid(D) = \bigcup_{d \in D} dv(d)$$

$$radius(D) = \frac{\sum_{d \in D} |centroid(D) - dv(d)|}{|D|}$$

The centroid of a cluster is the union of the values associated with the data sources in the cluster. The radius of a cluster is the sum of the difference between the centroid and the values associated with each data source, normalized by the number of data sources in the cluster. The distance between two clusters D1 and D2 is:

$$distance(D1,D2) = |D1| \times |centroid(D2) - centroid(D1)| + |D2| \times |centroid(D1) - centroid(D2)|$$

The distance is computed as the sum of the following two quantities: (a) the number of data sources in the first cluster multiplied by the number of values that only occur in the second cluster, and (b) the number of data sources in the second cluster multiplied by the number of values that only occur in the first cluster.

APPENDIX B FORMULAE FOR VDCI

We present the formulae for *centroid*, *radius*, and *distance* in VDCI. A cluster C in VDCI is a set of (value, data source) pairs, i.e., C={(*v, d*)} where *v* is a value and *d* is a data source. Let

$$D = \bigcup_{(v,d) \in C} \{d\}, \quad V = \bigcup_{(v,d) \in C} \{v\}, \quad ds(v) = \bigcup_{(v,d) \in C} \{d\}$$

In other words, D is the set of data sources in C, V is the set of values in C and *ds(v)* is the set of data sources associated with value v in C. The *centroid* of C is,

$$centroid(C) = (V,D)$$

$$radius(C) = \frac{\sum_{v \in V} |D - ds(v)|}{|V|}$$

The centriod of a cluster is a pair in which the first component is the set of values in the cluster and the second component is the set of data sources in the cluster. The radius of a cluster is the sum of the number of false positives for each value in the cluster, normalized by the number of values in the cluster. The distance between two clusters C1 and C2 is:

$$distance(C1, C2) = |V1 - V2| \times |D2 - D1| + |V2 - V1| \times |D1 - D2|$$

where c*entroid(C1)=(V1,D1)* and *centroid(C2)=(V2,D2)*

The distance between two clusters is the additional number of false positives that would occur if the two clusters were merged together. This is computed as the sum of the following two quantities (a) the number of values that only occur in the first cluster multiplied by the number of data sources that only occur in the second cluster, and (b) the number of values that only occur in the second cluster multiplied by the number of data sources that only occur in the first cluster. We do not take the common values of the two clusters into account since the number of false positives associated with them is not an additional quantity.

APPENDIX C ADDITIONAL EXPERIMENTAL RESULTS

We now present additional experimental results obtained by varying the ***number of data sources***, ***number of pairs***, ***size of range*** and ***value correlation*** parameters. For each experiment, we varied one parameter and used default values for the rest. We set the compression factor to be 10 for these experiments. We do not show the performance numbers for VDCI in these experiments because its performance is consistently bad.

Figure 9 shows how the number of false positives varies with the number of data sources for equality queries. The number of false positives increases gradually with the number of data sources for VCI and DCI. The number of false positives for HBI, however, increases more rapidly because the data source lists of adjacent values become increasingly different as the number of data sources increases. Figure 10 shows how the number of false positives varies with the number of data sources for range queries. The trend is similar to Figure 9 for VCI and DCI. HBI, on the other hand, performs much better for range queries because adjacent values are clustered together.

Figure 11 shows the results when the number of pairs is varied for equality queries. For this graph, the index size was fixed to be 800KB. The graph shows that the false positives for all three approaches increase with the number of pairs. This is attributable to the fact that each value now occurs in a larger number of data sources. Figure 12 shows a similar graph for range queries. In this graph, however, the number of false positives decreases when the number of pairs increases. This can be explained by the fact that the data source lists for adjacent values become more similar because they each contain more data sources.

Figure 13 shows the results when the size of range is varied. For small range sizes, the number of false positives for HBI increases rapidly because its performance approaches that for equality queries (equality queries are simply range queries with a range size of 1). However, for relatively large range sizes, HBI performs well as expected. The false positives for VCI and DCI increases when the range size increases from 1 to 100; this is due to the fact that more than one cluster in VCI/DCI has to be contacted for a range query, and the error in each cluster adds to the overall error. The number of false positives decreases beyond a range size of 100 because the number of data sources containing values in the selected range increases faster than the number of newly selected clusters, thereby reducing the total number of false positives.

Figure 14 shows the results when the value correlation is varied for equality queries. VCI and DCI perform consistently well since they do not consider value ordering when constructing the clusters. In contrast, the number of false positives of HBI increases rapidly when the value correlation decreases – this is because HBI has to group adjacent values in the same cluster, and when the correlation decreases, adjacent values are likely to have increasingly dissimilar lists of data sources. We do not show the effects of varying value correlation for range queries.

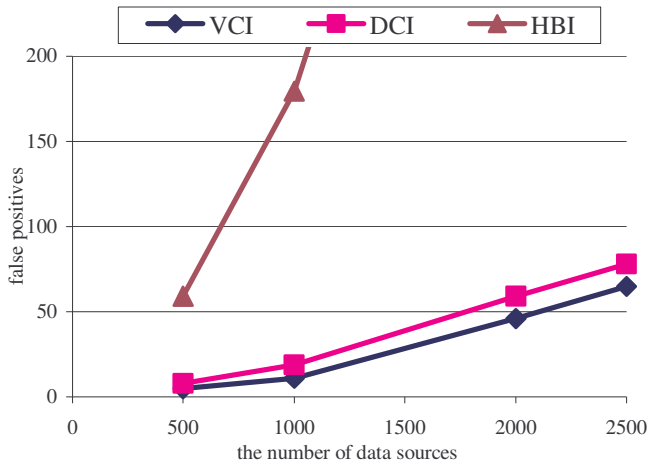

Figure 9 Effects of the number of data sources on equality queries

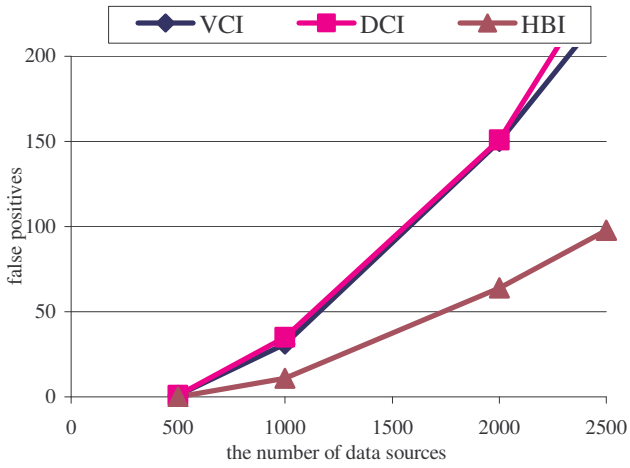

Figure 10 Effects of the number of data sources on range queries

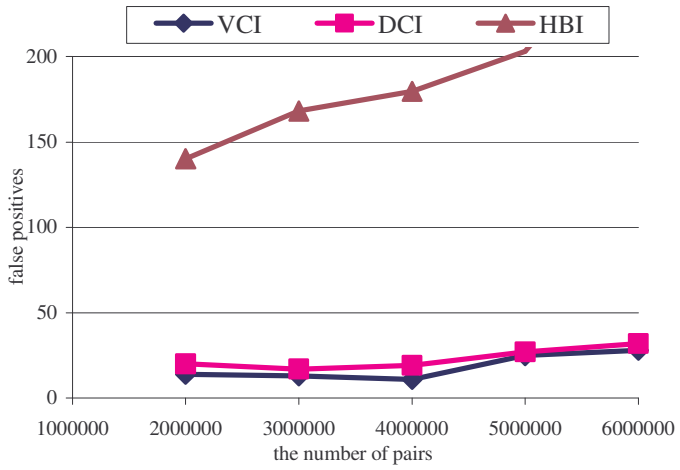

Figure 11 Effects of the number of pairs on equality queries

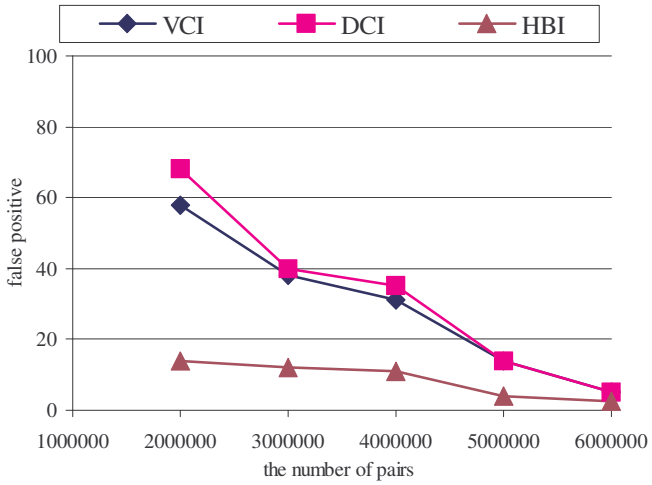

Figure 12 Effects of the number of pairs on range queries

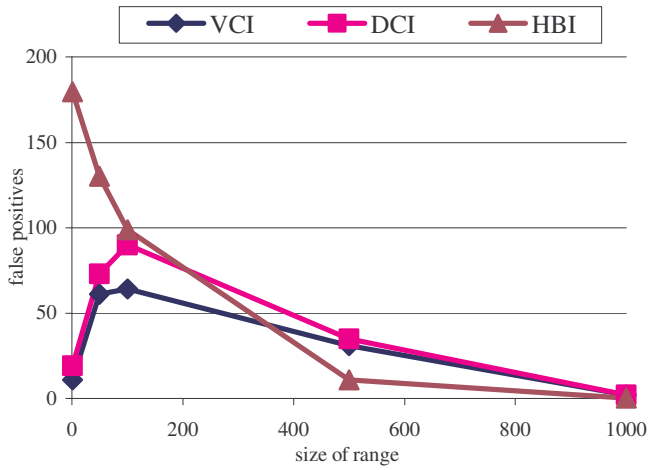

Figure 13 Effects of size of range on range queries

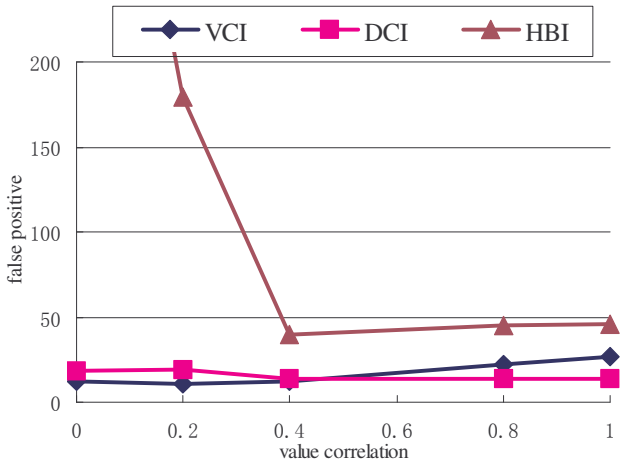

Figure 14 Effects of value correlation on equality queries