An Architecture for Cell-Centric Indexing of Datasets

Lixuan Qiu[1,2], Haiyan Jia[3], Brian D. Davison[2], and Jeff Heflin[2]

[1] Dept. of Computer Science, University of Southern California,
[2] Dept. of Computer Science and Engineering, Lehigh University
[3] Dept. of Journalism and Communication, Lehigh University
lixuanqi@usc.edu, {haj616,bdd3,jeh3}@lehigh.edu

Abstract. Increasingly, large collections of datasets are made available to the public via the Web, ranging from government-curated datasets like those of data.gov to communally-sourced datasets such as Wikipedia tables. It has become clear that traditional search techniques are insufficient for such sources, especially when the user is unfamiliar with the terminology used by the creators of the relevant datasets. We propose to address this problem by elevating the datum to a first-class object that is indexed, thereby making it less dependent on how a dataset is structured. In a data table, a cell contains a value for a particular row as described by a particular column. In our cell-centric indexing approach, we index the metadata of each cell, so that information about its dataset and column simply become metadata rather than constraining concepts. In this paper we define cell-centric indexing and present a system architecture that supports its use in exploring datasets. We describe how cell-centric indexing can be implemented using traditional information retrieval technology and evaluate the scalability of the architecture.

1 Introduction

The twenty-first century has experienced an information explosion; data is growing exponentially and users' information retrieval needs are becoming much more complicated [7]. Given people's increasing interests in datasets, there is a need for user-friendly search services for data journalists, scientists, decision makers, and the general public to locate datasets that can help them answer their questions.

Even though users, under many circumstances, are not experts in the domain in which they search, they should be able to easily use such an application; the query process should be responsive and efficient, the result should provide a general picture of what the dataset is about, and offer enough information for the users to know how likely the dataset will contain data that they need.

We present the novel concept of cell-centric indexing as the solution for data storage and querying. Traditional database management systems group data by

tables and then organize this data into rows and columns. Even column-oriented databases still assume this model, although data is stored by column instead of by row as in a relational database management system. These approaches are suitable when the users want to construct queries based on their knowledge of the schema, but what if they want to find out how specific data is stored? To enable flexible, schema-optional queries, we build inverted indices using individual cells as the fundamental unit.

These indices provide different fields that index both the content of the cell and its context. For our purposes, the context includes other cell values in the same row, the name of the column (if available), and metadata about the containing dataset. This approach allows us to refine our search by row descriptors, column descriptors or both at the same time. In essence we free the data from how it is structured, and schema information, when available, is merely one of the many ways to locate the data of interest. Thus, we take the view that fundamentally, users are searching for specific data (i.e., particular cells or collections thereof), and the tables are merely artifacts of how the data is stored.

We recognize that this approach also has downsides. In particular, an index of cells (and their contexts) will incur substantial storage overhead in comparison to an index of datasets. Moreover, if the desired search result is one or more datasets, at run-time there will be additional processing to assemble the cell-specific results to enable the retrieval and ranking at that level of granularity. However, our cell-centric approach gives us some additional flexibility and we believe that good system design, appropriate data structures, and efficient algorithms can ameliorate the costs.

The contributions of this paper are: (1) We propose cell-centric indexing as an innovative approach to an information retrieval system. A cell-centric index enables a user to find data without having to know the pre-existing structure of each table; (2) We introduce the mechanisms of our dataset search engine. We describe the structure and method of data storage and querying of our server; (3) We examine the efficiency of indexing and querying datasets.

The rest of the paper is organized as follows: we first discuss related work, briefly describe the idea of cell centric indexing and its advantages and disadvantages, introduce the structure of our server and the methodology involved in querying, and finally provide experimental results about building the indices and querying them.

2 Related Work

Scholars have investigated exploratory search to help searchers succeed in an unfamiliar area by proposing novel information retrieval algorithms and systems; some of them propose innovative user interfaces, while others try to predict the user's information need and to use the prediction to better facilitate the subsequent interaction.

Derthick et al. [3] describe a visual query language that dynamically links queries and visualizations. The system helps a user to locate information in

a multi-object database, illustrates the complicated relationship between attributes of multiple objects, and assists the user to clearly express their information retrieval needs in their queries. Similarly, Yogev et al. [13] demonstrate an exploratory search approach for entity-relationship data. The approach combines an expressive query language, exploratory search, and entity-relationship graph navigation. Their work enables people with little to no query language expertise to query rich entity-relationship data.

In the domain of web search, Koh et al. [8] devise a user interface that supports creativity in education and research. The model allows users to send their query to their desired commercial search engine or social platform in iterations. As the system goes through each iteration, it will combine the text and image results into a composition space. Addressing a similar problem, Bozzon et al. [1] design an interactive user interface that employs exploratory search and Yahoo! Query Language (YQL) to empower users to iteratively investigate results across multiple sources.

A tag cloud is a common and useful visualization of data that represents relative importance or frequency via size. Some researchers have adapted this idea to visualize query results. Fan et al. [5] focus on designing an interactive user interface with image clouds. The interface enables users to comprehend their latent query intentions and direct the system to form their personalized image recommendations. Dunaiski et al. [4] design and evaluate a search engine that incorporates exploratory search to ease researchers' scouting for academic publications and citation data. Its user interface unites concept lattices and tag clouds to present the query result in a readable composition to promote further exploratory search. On the other hand, Zhang et al. focus their work on graph data [14]. They combine faceted browsing with contextual tag clouds to create a system that allows users to rapidly explore knowledge graphs with billions of edges by visualizing conditional dependencies between selected classes and other data. Although they don't use tag clouds, Singh et al. [11] also display conditional dependencies in their data outline tool. For a given pivot attribute and set of automatically determined compare attributes, they show conditional values, grouped into clusters of interaction units.

In addition, scholars investigate query languages and models. Ianina et al. [7] concentrate on developing an exploratory search system that facilitates the user having a way to conduct long text queries, while minimizing the risk of returning empty results, since the iterative "query–browse–refine" process [12] may be time-consuming and require expertise. Meanwhile, Ferré and Hermann [6] focus more on the query language, LISQL, and they offer a search system that integrates LISQL and faceted search. The system helps users to build complex queries and enlightens users about their position in the data navigation process.

Other researchers try to think ahead of the user: i.e., predict the user's search intent so that they can better direct the search result. Peltonen et al. [9] utilize negative relevance feedback in an interactive intent model to direct the search. Negative relevance feedback predicts the most relevant keywords, which are later arranged in a radar graph where the center denotes the user, to represent the user's intent. Likewise, Ruotsalo et al. [10] propose a similar intent radar model

that predicts a user's next query in an interactive loop. The model uses reinforcement learning to control the exploration and exploitation of the results.

3 Cell-Centric Indexing

We define a table as $T = \langle l, H, V \rangle$ where l is the label of the table, $H = \langle h_1, h_2, ..., h_n \rangle$ is a list of the column headers, and V is an $m \times n$ matrix of the values contained in the table. $V_{i,j}$ refers to the value in the i-th row and the j-th column, which has the heading h_j.

Inverted indices are an important tool in information retrieval. An inverted index consists of a lexicon and a posting list for each term. The lexicon is simply an index of the unique terms contained in the document collection. The posting list for a term w is a sorted list of documents (in reality, document identifiers) that contain w.

Given a conjunctive query $Q = \{q_1, q_2, ...\}$, the set of matching documents can be found by simply intersecting the posting lists for each query term q_i. If the lexicon is sorted, then a binary search can find each posting list in $\mathcal{O}(\log n)$ time, and since the posting lists are sorted, a parallel walk through these lists can find the intersection in linear time. This allows for very fast response times, even when many documents are indexed.

Typically, an information retrieval system allows indexing in different *fields*. For example, the index for a collection of documents might have one field for title, another for subheadings, and another for body text. Among other things, fields can be used to differentiate the importance of a word based on where it appears: a word that appears in the title of a document is probably more meaningful than one that only appears once in its body.

A naïve approach to indexing a collection of datasets would be to simply treat each table as a document, and have separate fields for the label, column headings, and (possibly) values. When terms are used consistently and the user is familiar with the terminology, this will often work well. However, this approach has several weaknesses:

1. Any query on values has lost context of what column the value appears in and what identifying information might be present elsewhere in the same row. For example, a table that contains capitals like (Paris, France) and (Austin, Texas) is unlikely to be relevant to a query about "Paris Texas"
2. It is difficult to determine which new terms can be used to refine the query. Users would need to download some of the datasets and choose distinctive terms from the most relevant ones.
3. A user's constraint could be represented in different tables in very different ways. If the user is looking for "California Housing Prices", there may be a table with some variant of that name, there may be a "Real Estate Prices" table with rows specific to California, or there may be a "Housing Prices" table that has a column for each state, including California. A user should be able to explore the collection to see how the data is organized and what terminology is used.

We have proposed cell-centric indexing as a novel way to address the problems above. Rather than treating the table as the indexed object, each datum (cell in the table) is an indexed object. In its simplest form, we propose four fields: *content*: the value of the cell, *title*: the label of the dataset the cell appears in, *columnName*: the header of the column the cell appears in, and *rowContext*: the values in all cells in the same row as the indexed cell. Formally, a cell value $V_{i,j}$ from table $T = \langle l, H, V \rangle$ can be indexed with: $content = V_{i,j}$, $title = l$, $columnName = h_j$, and $rowcontext = \bigcup_{k=1}^{n} V_{i,k}$. This index would allow users to find all cells that have a column header token in common regardless of dataset, or all cells that appear in the same row as some identifying token, or look for the occurrence of specific values in specific columns.

However, in this form, users still need to know which keywords to use and which fields to use them in. A cell-centric index alone is not helpful to a user who is not already familiar with the collection of datasets. In order to support the user in exploring the data, we propose the abstraction *conditional frequency vectors* (CFVs). Let I be a set of items, D be a set of descriptors (e.g., tags that describe the items), and $F \subseteq I \times D$ be a set of item and descriptor pairs $\langle x_i, d_i \rangle$. Let Q be a query, where $Q(F) \subseteq F$ represents the pairs for only those items that match Q. Then a CFV for Q and F is a set of descriptor-frequency pairs where the frequency is the number of times that the corresponding descriptor occurs within $Q(F)$: $\{\langle d, f \rangle \mid f = \#\{\langle x, d \rangle | \ \langle x, d \rangle \in Q(F)\}\}$. For cell-centric indexing, the items I are the set of all cells regardless of source dataset, and F_i pairs cells with terms from the i-th field. Typically, we sort the CFV in terms of descending frequency.

Figure 1 shows the response of our prototype interface to the query with *title*="olympics" and *content*="athens". It displays a CFV for each field as a histogram; the longer the red bar, the more frequently that term co-occurs with the query. The CFV for the *title* field is $\langle\langle$*olympics*,32$\rangle$, $\langle$*medalists*,21$\rangle$,$\langle$*list*,14$\rangle$,...$\rangle$. As we can see, 15 datasets contain matches, most of which have "medalists" in the title and "games" in the column name. The user can refine this query and create new histograms by clicking on any terms in the result. For example, if the user clicks on "kenya" in the title histogram, the system will display a new set of histograms summarizing the datasets that have "athens" as a content field and both "olympics" and "kenya" in the title. The user explores the dataset collection by adding terms to and removing terms from the query.

4 System Architecture

The architecture of the system is depicted in Figure 2. At the core of our system is an Elasticsearch server. Elasticsearch is a scalable, distributed search engine that also supports complex analytics. Our system has two main functions: 1) parse collections of datasets, map them into the fields of a cell-centric index, and send indexing requests to Elasticsearch and 2) given a user query, issue a series of queries to Elasticsearch and construct histograms (CFVs) for each field.

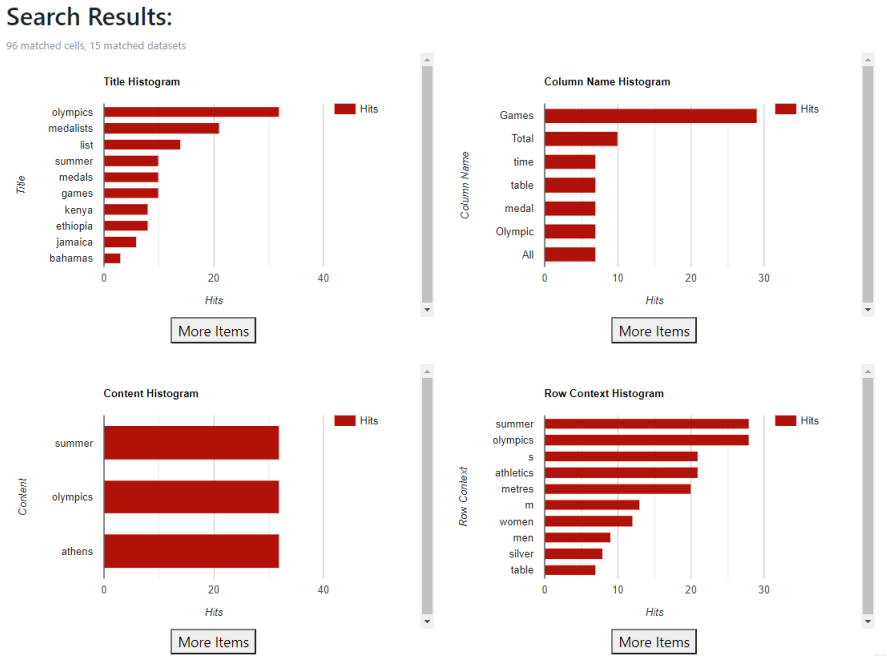

Fig. 1: Results for the Query {title:olympics,content:athens}

4.1 Index Definition

In Elasticsearch, a mapping defines how a document will be indexed: what fields will be used and how they will be processed. In cell-centric indexing **the cell is the document**, and our index must have fields that describe cells. Our mapping is summarized in Table 1. In addition to the four fields mentioned in Section 3, we have fields for the *fullTitle* (used to identify which specific datasets match the query) and metadata such as tags, notes, and organization. Note, that content is divided into two fields: *content* and *contentNumeric*, for reasons that will be described below. For each field, we give its type and, if applicable, the analyzer used to process text from the field.

We use three types of fields: *text*, *keyword*, and *double*. *Text* type fields are tokenized and processed by word analyzers, whereas *keyword* type fields are indexed as is (without tokenization or further processing). *Double* type fields are used to store 64-bit floating point numbers. Most of our fields are *text* fields, but contentNumeric is a *double* field, which allows it to store both integer and real numeric values, and fullTitle is a *keyword* field, since we want users to be able to view the complete name of the dataset in the result.

All *text* fields require an analyzer which determines how to tokenize the field and if any additional processing is required. The *stop* analyzer divides text at all non-letter characters and removes 33 stop words (such as “a”, “the”, ’“to”, etc.). For most fields, we use the *stop* analyzer, but we use the *wordDelimiter* analyzer for the *colunnName* field. In addition to dividing text at all non-letter

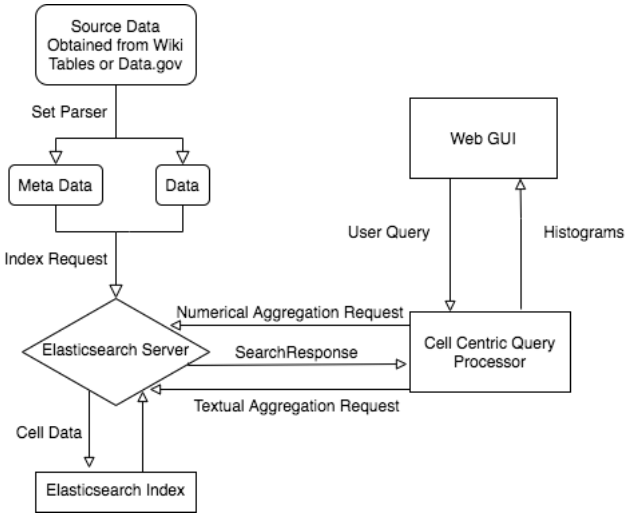

Fig. 2: Flowchart that shows the general work flow of the project

characters, it also divides text at letter case transitions (e.g., "birthDate" is tokenized to "birth" and "date"). This analyzer does not remove stop words.

Field	Type	Analyzer
columnName	text	wordDelimiter
content	text	stop
contentNumeric	double	N/A
rowContext	text	stop
title	text	stop
fullTitle	keyword	N/A
tags	text	stop
notes	text	stop
organization	text	stop

Tab. 1: Elasticsearch mappings used to implement cell-centric indexing

4.2 Indexing a Dateset

The system loads each dataset using the following process:

1. Read the metadata, which can include title, tags, notes and organization. If the original table is formatted as CSV, then this data might be contained in a separate file in the same directory, or as a row in a repository index file. If the table is formatted using JSON, the metadata may be specified along with the content, and there may be many datasets described in a single file.
2. Read the column headings of the data: $\langle h_1, h_2, ..., h_n \rangle$
3. For each row in the dataset:
   (a) Read the row values: $\langle v_1, v_2, ..., v_n \rangle$
   (b) Create *rowContext* by concatenating the values in the row. Note, to avoid creating different large context strings for each value in the row,

we create a single *rowContext*. This means that each value is also part of its own row context. This decision helps make the system more efficient. An additional efficiency consideration is that each value included in *rowContext* is truncated to the first 100 characters.

(c) Build an index request for each cell value v_i. If the content is numeric (integer or real), it will be indexed in the *contentNumeric* field; otherwise it is indexed in the *content* field. The *columnName* field will be indexed with the corresponding header h_i. The title is indexed twice, once as a tokenized field that can be used in queries, and again as keyword field that preserves the order of the title and can be used to precisely identify the dataset the cell originated from. All other metadata fields are indexed in a straight-forward way.

(d) For efficiency, index requests are batched and sent to the server in bulk. Synchronization is disabled during bulk loading to avoid excessive delays.

4.3 Querying the index

Algorithm 1 illustrates how a query is processed and how the results are composed. We can get a CFV for a particular field and query using Elasticsearch's *term aggregation* feature. This feature returns a list of terms that appear in the selected documents along with their frequency. Since we want users to be able to more accurately locate data that matches their interests, each query generates histograms for the *title*, *columnName*, *content*, *rowContext*, and *fullTitle* fields.

Unlike textual terms, numeric terms exhibit a greater variability. Histograms built using distinct numeric strings are unlikely to have significant value. For example, "135", "135.0" and "1.35E+2" are all equivalent, while some users might consider "135.0001" to be close enough. To address this, we create ranges over numeric values. Elasticsearch supports a *histogram aggregation* which can be used to build these distributions. The algorithm evaluated in this paper builds 10 buckets of equal size, although our team is currently experimenting with more sophisticated algorithms, including one in which we remove outliers, compute the mean and standard deviation over the remaining values, and then specify the buckets to have a width of one standard deviation with one bucket centered over the mean. Once the numeric content histogram is created, this is merged with the content histogram.

Algorithm 1: Process of Query Execution

```
Data: Query q
Issue query q requesting term aggregations for title, columnName, content,
  rowContext and fullTitle;
Build CFVs from the query response ;
if results contain contentNumeric data then
    Reissue q requesting histogram aggregation on contentNumeric data ;
    Merge results with content CFV ;
Return columnName, rowContext, title, content, and fullTitle CFVs ;
```

5 Experiments

In this paper, we evaluate whether the proposed system can be scaled to real-world collections of datasets while providing acceptable response times to user queries. All of our experiments are conducted using a Lehigh-hosted server. The server has one Intel(R) Xeon(R)Silver 4110 CPU @ 2.10GHz, with 8 cores (16 threads) and a total of 96GB RAM. It runs Java 11.0.7 and Elasticsearch 6.8.8.

5.1 Scalability of Loading Datasets

To evaluate the scalability of the system with respect to data loading, we load two different collections of tables. The WikiTables corpus[4] contains 1.6 million tables extracted from Wikipedia. We created five different subsets of the data, where each subsequent set contains all of the data in the previous set. The data.gov corpus [2] contains 7,485 tables downloaded from data.gov, the open data portal of the United States federal government. We created three subsets of this data, where each subsequent subset contained the prior subset. In both cases, we identify the subsets by size of the raw data files. Due to the variability of length in the values of different datasets, the number of cells is not necessarily proportional to the directory size. Also, the WikiTables corpus is stored in JSON, which leads to larger sizes than the CSV files of the data.gov corpus.

We loaded each subset and recorded the resulting index size in MB and the time to load the data in minutes. Table 2 displays the results for the WikiTables corpus. On average, the system can index 1 million table cells per minute and the resulting index is 100 bytes per cell. Table 3 shows the same experiment conducted on three different sizes of data.gov data. Here, the system only loads $\sim 650,000$ cells per minute and index size is more variable, although the largest set requires ~ 110 bytes per cell. We hypothesize that the slower throughput of data.gov is due to structural differences in the tables: data.gov tables are likely to have more columns and longer values in those columns, leading to significantly longer *rowContext*. However, due to the efficiencies of inverted indices, this cost is borne in indexing time (due to longer indexing messages), and the index size is not affected as much. Despite the difference in throughput, in both cases the load time scales linearly with the number of the cells, as depicted in Figure 3 and Figure 4.

We will note that the resulting Elasticsearch index is always larger than the original input data. For WikiTables, this is about 2.5x larger, while for data.gov it is 1.5x to 1.7x larger[5]. We have experimented with an alternative design that uses two indices: one for indexing cell-specific information and another for indexing table-specific information. The alternate design reduced index storage, but increased overall query time. Given that disk space is relatively cheap, and Elasticsearch makes it easy to add additional machines to scale up the size of a cluster, we chose to optimize query speed.

[4] `http://websail-fe.cs.northwestern.edu/TabEL/tables.json.gz`

[5] For the smallest size, the difference is 6.3x larger, but we believe this is due to mandatory overhead

Directory Size (MB)	Size of Index (MB)	Num Cells	Time (mins)
99.3	249.0	2,432,826	2.4
500.3	1,292.6	12,888,268	13.0
999.5	2,614.3	26,133,840	26.6
2,002.0	5,255.6	52,326,690	53.5
3,940.0	10,277.5	102,926,958	106.3

Tab. 2: Result of uploading various sizes of folders of JSON files from WikiTables

Directory Size (MB)	Size of Index (MB)	Num Cells	Time (mins)
26.0	163.3	1,835,238	2.9
466.3	694.0	5,398,841	8.7
756.4	1295.2	11,868,845	17.9

Tab. 3: Result of Uploading Various Sizes of Folders of CSV Files from DataGov

5.2 Extensive Query Simulation

With the purpose of testing the relationship between size of the index and query efficiency, we created three data collections from the WikiTables corpus. This results in data files of size 0.8GB, 2.4GB, and 4GB; where each smaller index's source is a subset of each larger index. Table 4 shows indexes' basic information.

Source Folder Size (GB)	Number of Cells in index	Index Size (GB)
0.8	20,381,595	2.1
2.4	61,602,873	6.2
4.0	102,926,958	10.3

Tab. 4: Statistics on WikiTable datasets used for query experiment

In order to evaluate the system's performance over a wide variety of possible queries, we designed a random query model. The first step of user query simulation is to create seed lists containing query terms that we can use as the "user's" starting filter. To do so, we build four CFVs using the empty query: title, columnName, content, and rowContext. Then we choose an appropriate range of frequency to perform our test; high frequency seeds are not desired because they do not sufficiently filter the results to be useful in exploration (in effect, they are analogous to stop words in traditional search); low frequency seeds are not suitable either, since we are simulating real users' choices and they are less likely to query for an uncommon word. Given these criteria, we chose to select terms

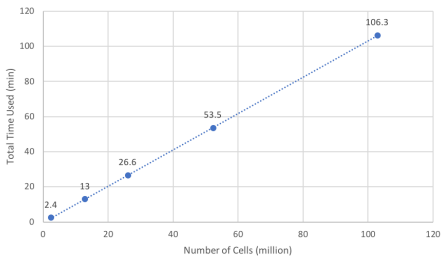

Fig. 3: Load time vs. number of cells for WikiTable corpus

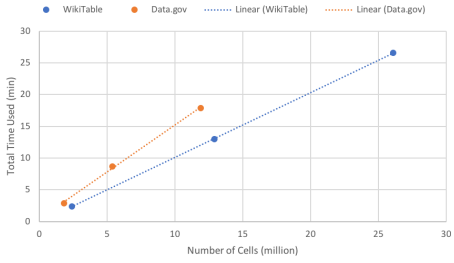

Fig. 4: Load time vs. number of cells for data.gov corpus. For comparison, the relevant range of WikiTables is also plotted.

with a frequency between 10,000 to 50,000. Then we create four lists, one each for seeds from title, columnName, rowContext, and content. Notably, content's seed list will only contain textual terms, since numerical terms in a dataset's content field are indexed separately by the contentNumeric field. We also generate a random-category seed list that has all the seeds from the lists above, so that when we arbitrarily select seeds from this list, each seed's category is randomized. We expect that users will rarely query on a specific number, and for many numbers, their frequency would be too small anyway, so only textual seeds are included in the seed list. Although some numbers like years might appear in many user queries, we feel this simplification keeps our set of test queries clean.

In order to simulate real-world testing conditions, we have designed a model that predicts the behavior of a normal user: we randomly choose a query from the seed list, execute the query, randomly select one of the resulting histograms, randomly select one of the top 20 *term*s from the resulting histogram, append the new *field:term* pair to the existing query and repeat. This process stops after the query has five terms, or a resulting histogram has two or fewer terms.

We then analyze the query performance by comparing different starting categories on the same index or same starting category on three different sizes (0.8GB, 2.4GB, 4.0GB) of indexes. In this process, timers are set at different points to measure the time consumed. We run one thousand iterations of the process, recording the queries and their times to a log file.

Since the final query of each iteration has a maximum length of five-terms, we can derive at most five different queries of different length from that original query. To illustrate this, suppose after an iteration, the final query looks like: $a\&b\&c\&d\&e$, where each letter represents a one-term query, eg. *title:football*. We can obtain five different queries: a, $a\&b$, $a\&b\&c$, $a\&b\&c\&d$, and $a\&b\&c\&d\&e$.

Five thousand query processes have been performed on each index, one thousand for each category (title, rowContext, content, columnName, and random). Generally, one thousand one-term queries are guaranteed, but as the query becomes longer, the number of queries with corresponding length will be smaller. Queries with more terms are more restrictive, leading to fewer results. Thus, the likelihood of satisfying the stopping criteria of too few results becomes greater. Table 5 displays the final number of queries based on the number of terms and the starting category on the 4.0 GB index.

	column name	**content**	**title**	**row context**	**random**	**total**
1-term	1,000	1,000	1,000	1,000	1,000	5,000
2-term	994	999	994	1,000	999	4,986
3-term	911	975	930	976	953	4,745
4-term	706	857	743	835	786	3,927
5-term	508	662	499	607	597	2,873
total	4,119	4,493	4,166	4,418	4,335	21,531

Tab. 5: Total number of queries of each length for each starting category on the 4.0 GB index

Figure 5 summarizes the performance of the system, assuming the first query term comes from the title field. Here we show the average query time (in milliseconds) for each index size for each length of query. As the queries grow in length,

execution time decreases. The gaps between 1-term and 2-term, 2-term and 3-term are the largest, since it is much more challenging to further improve the efficiency when the time usage is at around 50 ms level. It can also be observed that, although queries on larger indexes take more time, as the size of the index increases linearly, the time consumed grows sublinearly. For example, one-term queries on the largest index are only 1.7x slower than those on the smallest, even though this index is 5x larger. This trend holds regardless of the length of queries or the choice of the query's starting category. Figure 6 illustrates this by showing a similar pattern when the starting term is selected from the four categories at random. Together, these results imply that the system can scale to much larger indices with only a small degradation in performance.

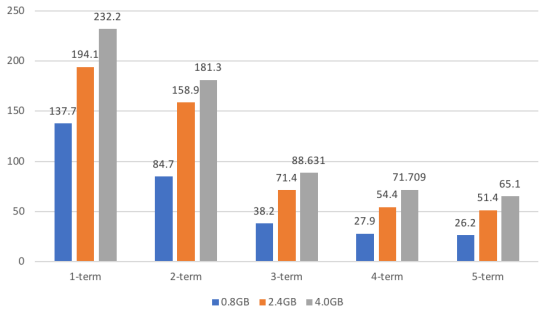

Fig. 5: Queries with first term from title field: average time (ms) grouped by query length and index size

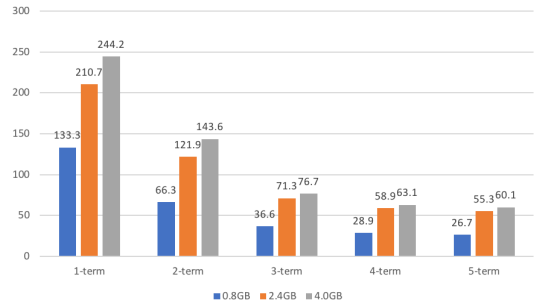

Fig. 6: Queries with first term from random field: average time (ms) grouped by query length and index size

Next, we investigate whether the type of the starting field has any significant impact on query performance. Table 6 shows the average and standard deviation of time consumed by the five different categories of one-term queries when issued against the 4.0GB collection. For the last row, i.e. random category, the query fields are randomized to give a more general sense of the overall performance of the system. In addition to total query time, we break time down into textual query time (generating the standard histograms for text fields) and numeric query time (generating the numeric range histograms). Here, we see that title queries have the greatest variation in overall time, and that for all fields except content, the numeric range query processing can take a significant fraction of

the overall query time (30 - 50%). Note, content queries always have a 0 numeric query time. This is because all of our queries are seeded with text terms and numeric range processing is only computed when the filtered cells are numeric. However, since the query term is textual content, there can be no numeric cells in the result, and thus no numeric range processing is computed.

	Textual Query Time (ms)		**Numeric Query Time (ms)**		**Total Query Time (ms)**	
Category	**Avg.**	**Std. dev**	**Avg.**	**Std. dev**	**Avg.**	**Std. dev**
title	139	112	93	18	232	114
content	147	27	0	0	147	27
rowContext	122	21	121	23	243	36
columnName	126	31	111	32	237	49
random	157	86	87	47	244	92

Tab. 6: Average and standard deviation of time consumed by textual and numerical query on one-term query

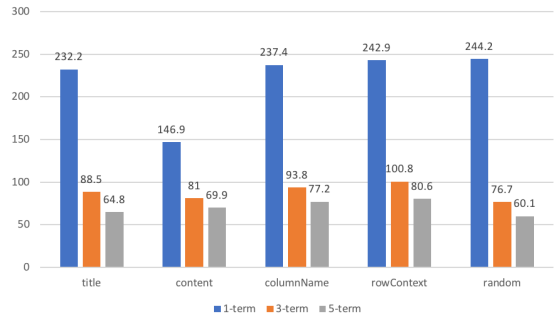

Fig. 7: Queries from 4.0 GB collection: average time (ms) grouped by query length and first term field

Using the largest index, Figure 7 displays the average query times across the five different field types on three different query lengths. For most types of queries, the one-term query takes 230-250 ms. However, queries starting with a content term take less: only 146.9 on average. Recall from Table 6 that content queries do not need numeric range processing, and are thus faster. Furthermore, it can also be observed that queries starting with rowContext will often take more time than other types of queries. These queries have the longest average times for both three-term and five-term queries (100.8 ms and 80.6 ms, respectively). We hypothesize this is because many rowContext terms have a large inverted index; that is they are associated with many cells.

One oddity of Figure 7 is that for one-terms queries, rowContext is actually faster on average than for random queries. In order to understand if outliers play a role in this observation, we create a box plot of the same data. From Figure 8, we can see that certain types of queries have more variability in execution times than others. In particular, one-term queries starting with a random category have more outliers above the third quartile than rowContext one-term queries. We also see that the median is slightly below the median of rowContext one-term queries. Thus, it is these outliers that lead to a higher average query time.

Note, given that both title queries and columnName queries have many outliers, we assume that these are the types of queries that contribute the most to the variability in the random query result. It can also be observed that, of over 20,000 random queries issued over the index, only three took longer than a second, and only one of those was (slightly) over two seconds.

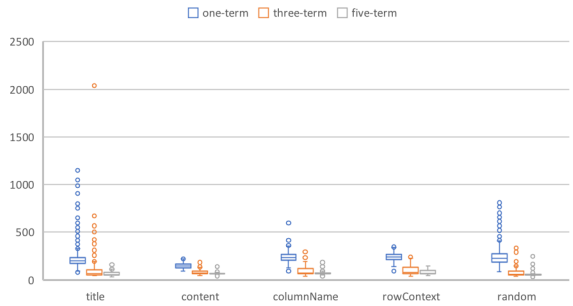

Fig. 8: Box plot of query times by query length and starting field. The boxes show the first quartile, median and third quartile. Circles indicate outliers.

6 Conclusion

Cell-centric indexing is an innovative architecture that enables anyone to conduct schema-less queries on datasets in an exploratory fashion. In this paper, we discuss the scalability and query efficiency of our system. We were able to load the entire 1.6 million table WikiTables corpus in 106 minutes and one-term queries against this index averaged less than 250 ms each. Queries with more terms took less time. In general, the server is able to parse and upload source files in linear time; the index will be approximately 2.6 times as large as the size of the source folder, which is an acceptable trade off between inexpensive disk space and running time.

Despite the small size of our test server and being unable to handle a large number of requests and responses as currently configured, we believe it can provide an easy and efficient way of locating a dataset of interest. Future work includes system performance enhancements, more usability features and a thorough user evaluation of the interface.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. III-1816325. Alex Johnson, Xuewei Wang, dePaul Miller, Drake Johnson, and Keith Register contributed to the implementation of the system.

References

1. Bozzon, A., Brambilla, M., Ceri, S., Fraternali, P.: Liquid query: Multi-domain exploratory search on the web. In: Proceedings of the 19th International Conference on World Wide Web. p. 161–170. WWW '10, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1772690.1772708

2. Chen, Z., Jia, H., Heflin, J., Davison, B.D.: Generating schema labels through dataset content analysis. In: Companion Proceedings of the The Web Conference 2018. p. 1515–1522. WWW '18, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE (2018). https://doi.org/10.1145/3184558.3191601
3. Derthick, M., Kolojejchick, J., Roth, S.F.: An interactive visual query environment for exploring data. In: Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology. p. 189–198. UIST '97, Association for Computing Machinery, New York, NY, USA (1997). https://doi.org/10.1145/263407.263545
4. Dunaiski, M., Greene, G.J., Fischer, B.: Exploratory search of academic publication and citation data using interactive tag cloud visualizations. Scientometrics **110**(3), 1539–1571 (2017)
5. Fan, J., Keim, D.A., Gao, Y., Luo, H., Li, Z.: Justclick: Personalized image recommendation via exploratory search from large-scale flickr images. IEEE Transactions on Circuits and Systems for Video Technology **19**(2), 273–288 (2008)
6. Ferré, S., Hermann, A.: Semantic search: Reconciling expressive querying and exploratory search. In: International semantic Web conference. pp. 177–192. Springer (2011)
7. Ianina, A., Golitsyn, L., Vorontsov, K.: Multi-objective topic modeling for exploratory search in tech news. In: Conference on Artificial Intelligence and Natural Language. pp. 181–193. Springer (2017)
8. Koh, E., Kerne, A., Hill, R.: Creativity support: Information discovery and exploratory search. In: Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. p. 895–896. SIGIR '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1277741.1277963
9. Peltonen, J., Strahl, J., Floréen, P.: Negative relevance feedback for exploratory search with visual interactive intent modeling. In: Proceedings of the 22nd International Conference on Intelligent User Interfaces. p. 149–159. IUI '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3025171.3025222
10. Ruotsalo, T., Peltonen, J., Eugster, M.J.A., Głowacka, D., Floréen, P., Myllymäki, P., Jacucci, G., Kaski, S.: Interactive intent modeling for exploratory search. ACM Trans. Inf. Syst. **36**(4) (Oct 2018). https://doi.org/10.1145/3231593
11. Singh, M., Cafarella, M.J., Jagadish, H.V.: Dbexplorer: Exploratory search in databases. In: Pitoura, E., Maabout, S., Koutrika, G., Marian, A., Tanca, L., Manolescu, I., Stefanidis, K. (eds.) Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016. pp. 89–100. OpenProceedings.org (2016). https://doi.org/10.5441/002/edbt.2016.11
12. White, R.W., Roth, R.A.: Exploratory Search: Beyond the Query-Response Paradigm. Synthesis Lectures on Information Concepts, Retrieval, and Services, Morgan & Claypool Publishers (2009). https://doi.org/10.2200/S00174ED1V01Y200901ICR003
13. Yogev, S., Roitman, H., Carmel, D., Zwerdling, N.: Towards expressive exploratory search over entity-relationship data. In: Proceedings of the 21st International Conference on World Wide Web. p. 83–92. WWW '12 Companion, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2187980.2187990
14. Zhang, X., Song, D., Priya, S., Heflin, J.: Infrastructure for efficient exploration of large scale linked data via contextual tag clouds. In: International Semantic Web Conference. pp. 687–702. Springer (2013)