The TEXTURE Benchmark: Measuring Performance of Text Queries on a Relational DBMS

Vuk Ercegovac David J. DeWitt Raghu Ramakrishnan

University of Wisconsin, Madison, WI 53706, USA
{vuk, dewitt, raghu}@cs.wisc.edu

Abstract

We introduce a benchmark called TEXTURE (TEXT Under RElations) to measure the relative strengths and weaknesses of combining text processing with a relational workload in an RDBMS. While the well-known TREC benchmarks focus on quality, we focus on efficiency. TEXTURE is a micro-benchmark for query workloads, and considers two central text support issues that previous benchmarks did not: (1) queries with relevance ranking, rather than those that just compute all answers, and (2) a richer mix of text and relational processing, reflecting the trend toward seamless integration. In developing this benchmark, we had to address the problem of generating large text collections that reflected the (performance) characteristics of a given "seed" collection; this is essential for a controlled study of specific data characteristics and their effects on performance. In addition to presenting the benchmark, with performance numbers for three commercial DBMSs, we present and validate a synthetic generator for populating text fields.

1 Introduction

As applications emerge that require queries over both text and relations, supporting text as a new data type (*TextType*), has become a focal point for relational database systems. For example, consider an on-line store where each item in the catalog has an associated description and discussion forum. A user may search by relational attributes such as price, product category, or brand, or by TextTypes such as "description" and "forum content". Combining the two classes of attribute types, a user may request information about inexpensive systems for graphic design that are viewed positively by users. The items in the result may be sorted according to price, or ranked by a TextType query on descriptions.

The topic of integrating TextType into an RDBMS has been widely studied ([20, 11, 17, 8]), and most commercial RDBMS's have integrated TextTypes. However, an application developer currently has no way to assess how a system that stores text in a relational DBMS will perform. In this paper, we propose a new benchmark, called TEXTURE, that compares performance of query workloads running on relational database systems.

TEXTURE differs from other text benchmarks in three fundamental ways:

1. The definition of *performance* focuses on system response time, as is typical of relational benchmarks. Benchmarks that focus exclusively on text, such as TREC [2], often focus on the *quality* of results as a measure of system performance. While quality is important, we assume that users are sufficiently satisfied with the quality of results (or will use other complementary benchmarks to evaluate quality), and wish to assess performance.
2. Previous text database benchmarks (notably [9]), have also used response time as their primary metric. In contrast to these benchmarks, TEXTURE includes a broader class of queries. In particular, [9] does not consider workloads that combine text expressions and relational predicates. It also does not consider sorting by a score, such as *relevance*, computed from the text expression.
3. TEXTURE measures performance of the end-to-end text processing task, which includes parsing, evaluating queries (possibly using specialized IR based subsystems), and integrating results with relational data. In contrast, benchmarks such as Set Query [13] and ASP3 [22] mimic text processing tasks using relations.

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

TEXTURE is a micro-benchmark modeled after the Wisconsin Benchmark [10]. The relations and query workloads are designed to provide a detailed analysis of how systems differ with respect to performance. Systems can be compared with respect to response time obtained when combining queries over text and relational attributes. For this purpose, the selectivity of both types of queries as well as how they are combined, is varied. Finally, system performance can be compared as the data set is scaled up in size.

The TEXTURE framework consists of a database, workload, and evaluation specification. The database consists of several relations with TextType and relational attributes. Queries are constructed from several *query templates*. Given the database and queries, multiple systems can be compared by the *evaluator* according to *response time*.

The database is specified according to the *DataGen* module. This module can use either real data, if available, or it can produce synthetic data. In this paper, we focus on the latter. In this case, DataGen is composed of *TextGen* and *RelGen*, which are used to populate the text and relational attribute values respectively.

Due to the focus on performance, a key step in developing TEXTURE was the design and validation of a novel synthetic data generator, TextGen. TextGen is unique in that it is capable of accurately scaling up an input "seed" text collection, while preserving important data characteristics. The benefits of such a synthetic generator are:

1. Data Availability: The user may not always have enough data available for scaling experiments.
2. Experimental Control: Using larger, publicly available data sets may not be appropriate. For example, a collection with large documents or multiple languages may skew results.
3. Anonymity: TextGen can be parameterized using statistical model parameters estimated from a "seed" corpus. As a result, third parties can evaluate the TEXTURE benchmark on behalf of a user, without access to the user's text data.

The *QueryGen* module creates a number of queries from several query templates. The structure of the query is determined by the template, but the values used in the text expressions and relational predicates are drawn from the database. While the goal is to evaluate queries that combine TextType expressions and relational predicates, we first establish a baseline by investigating each data type independently. Respectively, these baseline query templates are referred to as *Text* queries and *Relational* queries. Queries that combine the two classes are referred to as *Mixed* queries, and are composed by combining Text and Relational predicates. Currently, the relational operators considered for Mixed queries include selections, projections, and joins. QueryGen selects values to plug into queries according to the selectivity that is chosen by the user.

Finally, TEXTURE specifies how systems are loaded and how the query workload is evaluated. Three modes of evaluation are distinguished: (1) fetching all results, (2) fetching the first result, and (3) fetching the *top-k* results. We consider the last mode to be the most significant in practice because it arises in many web applications and allows systems a great degree of flexibility for optimization. The first two modes are included for completeness, and to shed light on how different systems work.

In summary, the contributions of this paper are as follows:

1. We propose the first in-depth benchmark for mixed relational and text query workloads, utilizing the text generator.
2. We present an evaluation of three commercial database systems using the benchmark
3. We develop and validate a novel synthetic text generator (*by comparing actual query response times* for each synthetically scaled dataset vis-a-vis a corresponding real dataset).

1.1 Paper Organization

We describe TEXTURE data, queries, and methodology in Sections 2, 3, and 4. We use TEXTURE to compare three commercial systems in Section 5. Next, we describe in greater detail how we generate document collections and how we validate the generation procedure in Section 6. Finally, we review related work and present conclusions.

2 TEXTURE Data

A TEXTURE database consists of several relations, each with the same schema composed of text and relational attributes, but varying in the number of tuples. The relations are named $1x, 2.5x, 5x, 7.5x$, and $10x$. If $1x$ contains n tuples, then $10x$ will contain $10n$ tuples, and so forth. The variably sized relations are used to test system performance as the data set is scaled up.

The schema of each TEXTURE relation is based on the schema used in the Wisconsin Benchmark [10]. It consists of a number of relational attributes and two text attributes as shown in Table 1. The relational attributes are used for controlling the selectivity of a query. The text attributes are used for ranked retrieval and projection.

The attribute *TXT-SHORT* is for short strings such as a summary suitable for projecting and displaying in a query result set. The attribute *TXT-LONG* is used for searching and ranking tuples by using an Information Retrieval-style query. It is not meant to be projected with search results but can be retrieved on a per tuple basis. This is similar to current web applications: a search typically returns a set of results that are succinctly described. If a user is interested

Attribute	Values	Comment
NUM-ID	$0...N-1$	primary key, clustered
NUM-U	$0...N-1$	random, unclustered index
NUM-05	0...199	
NUM-5	0...19	
NUM-50	0, 1	
TXT-SHORT	VARCHAR (255)	use *TextGen*
TXT-LONG	CLOB	

Table 1: The schema used for a relation of N tuples.

in the details of a particular result, they can retrieve the whole tuple, including the TXT-LONG attribute. The storage of TXT-SHORT is in-lined with the rest of the tuple whereas TXT-LONG is a CLOB.

2.1 Data Generators

The TEXTURE relations are populated using the DataGen module. It is composed of RelGen, for populating the relational attributes, and TextGen for populating the TextType attributes. RelGen is implemented using DBGen [14], which efficiently populates a relation by sampling distributions that are specified for each relational attribute.

TextGen is used to populate the TXT-SHORT and TXT-LONG attributes. TXT-SHORT is a randomly generated string of length 256. In contrast, TXT-LONG is searched so is populated with documents. However, a sufficiently large collection of documents may not be available for populating the larger relations. For this purpose, TextGen uses a model of document collections that is parameterized using a real document collection. It then uses the model in order to generate n documents, one per TXT-LONG value of the n tuples of a given TEXTURE relation. We refer to this synthetic collection of documents as $Synthetic_3$ whose construction is described in Section 6. Furthermore, we validate in Section 6.4 that similar performance is measured when evaluating the same query workloads using $Synthetic_3$ and using a real collection.

Using TextGen, we generate a TEXTURE database referred to as **AP-DB**. TextGen's model is parameterized using the Associated Press (AP) newswire data, volume one, from the TREC data set [2]. The AP collection contains 84,678 documents, each of which contains several text attributes. The largest attribute, *body*, is used for parameterizing TextGen's model. The collection of body documents consumes 250 MB of storage and has on average 315 words per document. Then, RelGen and TextGen populate the $1x$ relation with 84,678 tuples. The same distributions for relational attributes as well as the same TextGen model are then used to populate the scaled-up relations. For example, the $2.5x$ relation contains 211,695 tuples, and so forth, with the $10x$ relation containing 846,780 tuples, or roughly 2.5 gigabytes of storage.

This process is repeated for a second TEXTURE database, referred to as **VLC2-DB**. The only difference with AP-DB is that the Very Large Collection (VLC2) data set [1] is used to parameterize TextGen's model. The VLC2 collection contains 18,571,671 documents obtained by a *crawl* of the .GOV Internet domain. It requires about 100 GB of storage. However, we use it in order to compare system performance while varying the values used for TXT-LONG attributes. As a result, VLC2-DB consists of relations with the same number of tuples as in AP-DB. Consequently, we use 84,678 randomly selected documents from VLC2 in order to parameterize TextGen's model.

3 TEXTURE Queries

The query workload for TEXTURE is also based on the Wisconsin Benchmark. The objective is to investigate the effect of selectivity, join conditions and text query complexity on the query evaluation plan and resulting performance. There are three broad query templates that we consider: (1) text-only queries, (2) single-relation mixed queries, and (3) multiple-relation mixed queries.

3.1 Text-Only Workloads

The text-only query workload is included in all query templates so is the baseline for all other query templates. One of our goals is to determine how the selectivity and type of text query affect performance. For this purpose, the text-only query workloads are specified by three parameters:

1. Number of words
2. Word selectivity
3. Type of query expression (connectives used: AND, OR, phrase)

In practice, text expressions can be more varied than the above list suggests. For example, systems support many types of query expansion and proximity search that are richer than phrases. In addition, it may be useful to allow a text expression to be specified by a tuple variable. However, not all systems support such syntax. Our focus on fundamental text expressions is driven by the principle of choosing the simplest workloads that exhibit the greatest difference between systems. Clearly, as system performance converges for simpler workloads, TEXTURE's choice of queries will have to expand. However, the results in Section 5 indicate that the text expressions used do differentiate the systems studied.

The text expressions consist of single or multi-word queries. The multi-word queries are phrases or a form of *fuzzy* Boolean query that is used to rank documents. OR queries give a non-zero score to a document if any query word is included whereas AND queries require that all query words are included. The two types of queries are similar in that for those documents with

non-zero score, all contained query words contribute to the score.

The word selectivity parameter determines how many documents match a given query word. A workload is constructed by grouping words into selectivity *ranges* and randomly sampling within the given range, following [9], where three ranges, *high*, *medium*, and *low* are defined. The ranges are obtained as follows: the words are ordered by descending frequency; the high range contains the first k words whose cumulative frequency is 90% of all word occurrences. The range of words whose cumulative frequency accounts for the next 5% constitutes the medium range, and all remaining words fall within the low range. Word distributions are typically skewed, with few words in the high range and most words in the low range.

Phrase workloads require that multiple words appear contiguously in a document. Each phrase has two parts: a single *anchor* word and one or more *follow* words. An anchor word w is selected from a given selectivity range. Then, documents that contain w are retrieved and a random sample of these are selected in order to find follow words for w.

Since the workload depends on the word distribution, text workloads are created independently, once per relation. All queries in a text-only workload follow the query template shown in Figure 1. It is assumed that *SCORE* assigns the score of a document with respect to the query's *TXT-QUERY* (text expression). Table 2 lists some concrete text-only queries that are used in the evaluation.

```
SELECT SCORE, txt-short, num-id
FROM 1x
WHERE CONTAINS(txt-long, TXT-QUERY, SCORE)
ORDER BY SCORE DESC
```

Figure 1: Text-only query template for ranking $1x$ tuples according to TXT-QUERY.

3.2 Mixed Workload

The Mixed workload consists of queries that combine Text-only selection conditions with relational predicates over relational attributes. The objective of the mixed relational and text query workloads is to compare query plans across systems. The relational attribute value is chosen such that a system uses an unclustered index or a scan. The text attribute varies in selectivity as described in the previous section. Note that even though the queries are over a single relation, optimizer decisions are similar to that of a two-relation join where the first relation involves the relational attributes and the second relation is the Text attribute. The choice of queries allows us to see whether or not Text selectivity plays a role in query optimization for the systems evaluated. Furthermore, by varying the complexity of the Text query, we can see if errors in estimates can lead to poorer query plans.

For simplicity, we assume that text queries and relational predicates are combined using the AND logical connective. Figure 2 shows the template used in TEXTURE for Mixed queries over a single relation.

```
SELECT SCORE, txt-short, num-id
FROM A
WHERE CONTAINS(txt-long, TXT-QUERY, SCORE)
     and RELATIONALPREDICATE
ORDER BY SCORE DESC
```

Figure 2: Mixed text, relational query template used for medMix, lowMix, and multMix queries.

Since we pick the selectivity of relational attributes so that either an index or a scan is used, how is such a selectivity picked when comparing multiple systems? Different systems have different selectivity thresholds for deciding when to use an index. As a result, using a selectivity for which an index is selected with one system may result in a scan on a different system. While differences in query plans are of interest, a difference that is due solely to this selectivity threshold is of interest if we are studying relational query optimizers. However, our focus is on how the text expression is used, so we isolate its effect by requiring that if a system uses an index, then it will make the same decision as other systems. Since this threshold changes for differently sized relations, predicates are adjusted for each TEXTURE relation.

TEXTURE also includes Mixed queries over multiple relations. We consider two basic classes of mixed workloads that join two relations A and B. Queries in the first class (see Figure 3) apply a text expression to A and a relational predicate to B. A natural example of such a query is searching for catalog items by price and description. The objective of this class of queries is to test whether the selectivity of the text query plays a role in query optimization.

```
SELECT A.SCORE, A.txt-short, A.num-id
FROM A, B
WHERE CONTAINS(A.txt-long, TXT-QUERY, A.SCORE)
     and B.RELATIONALPREDICATE
     and JOINPREDICATE ORDER BY SCORE DESC
```

Figure 3: Mixed text, relational query template used for lowJoin and medJoin queries.

Queries in the second class (see Figure 4) apply a text expression to both relations. An example of such a query is searching for catalog items using their description and the content of associated discussion forums. In both kinds of join workloads in TEXTURE, relation B has a foreign key constraint to relation A. The objective of the second class of queries is to highlight the worst-case scenario for those systems that do not reduce their text related work with the highly selective join predicate.

4 Benchmark Methodology

TEXTURE is a micro-benchmark that varies many fine-grained experimental settings in order to compare

```
SELECT A.SCORE, A.txt-short, A.num-id
FROM A, B
WHERE CONTAINS(A.txt-long,TXT-QUERY1,A.SCORE)
      and CONTAINS(B.txt-long,TXT-QUERY2,B.SCORE)
      and JOINPREDICATE ORDER BY SCORE DESC
```

Figure 4: Mixed text, relational query template used for low2Join queries.

Name	Type	Description
lowTxt	Text	low frequency words
medTxt	Text	medium frequency words
andTxt	Text	two words combined using AND (medium frequency)
orTxt	Text	two words combined using OR (low frequency)
phrTxt	Text	phrase of two words
lowMix	Mix	low frequency word, medium rel. selectivity using NUM-5 equality
medMix	Mix	medium frequency word, low rel. selectivity using NUM-U range
multMix	Mix	andTxt or orTxt, any rel. selectivity
lowJoin	Mix	low frequency word, high join selectivity
medJoin	Mix	medium frequency word, low join selectivity
low2Join	Mix	low frequency word, high rel. selectivity on both join relations

Table 2: Examples of text-only and mixed queries.

systems on specific aspects relevant to queries involving TextType attributes. An *experiment* is defined by a choice of settings for the following *experiment parameters*: (1) relation, (2) query template, and (3) evaluation mode. The database and query template are as described in Sections 2, 6, and 3. For each query template, 100 queries are generated for the query workload. Multiple queries are used in order to smooth over the variance in number of results that is inherent with typically skewed word distributions. The *evaluation mode* refers to whether each query retrieves *all* results, the first result, or specifies a *top-k* hint in order to fetch the top-k results.

The response measured for each database system is the total elapsed response time for executing all 100 queries in the experiment. The database system is loaded with the dataset, and indexes are generated on a separate disk from the data. For indexes on relational attributes, statistics are analyzed for use by the optimizer. Finally, when running a query, we verify that all systems produce the same results.

All selected experiments are run 6 times against each database system being evaluated, thereby ensuring "warm numbers". The result of the first run for each experiment is discarded, and the reported result for a database system for a given experiment is the average response time over the next 5 runs. All reported results are within 3% of actual values with 95% confidence.

Relation	1x	2.5x	5x	7.5x	10x
medTxt	2863	4663	6884	9461	13856
lowTxt	25	55	109	148	193
andTxt	513	1255	2466	3682	4887
medMix	1	3	5	8	10
lowMix	1	2	2	4	5

Table 3: Average number of results per relation and query workload for AP-DB

The evaluation of all experiments is handled by a client process running on the server machine. The interface used is JDBC and the version of Java is 1.4.0. The operating system is Microsoft Windows 2003 Server. The hardware consists of a dual processor 1.8Hz AMD machine with 2 GB of memory, and 8, 123 GB IDE Hitachi Deskstar (7200 rpm) hard drives. Each database system has access to all of available memory.

5 Benchmark Evaluation

Given the specification for TEXTURE's data, query workloads, and evaluation methodology, a wide range of experiments are conducted on three commercial relational database systems, A, B, and C [1] that provide integrated support for TextType attributes. The objective of the experiments are as follows.

1. How do the systems perform as the data set is *scaled up*?
2. We look into issues that relational optimizers face when supporting a TextType expression.
3. Motivated by our web application example, we look at sorting on relational attributes and investigate *top-k* optimizations.

Results are first reported on the **AP-DB** TEXTURE database, followed by results from the **VLC2-DB** database.

5.1 Scaling Experiments

In order to investigate how the systems perform as the data set is scaled up, the queries listed in Table 2 are evaluated over the relations whose size is scaled by a factor of 10. The average number of results per query, organized by query workload and data set is given in Table 3.

Results from text-only workloads are first discussed, followed by mixed query results.

5.1.1 Text Only Workload

The results obtained from the text-only workloads show that performance of all systems scales linearly as the relations are scaled up. For example consider the results for lowTxt queries shown in Figure 5.

The results for andTxt queries shown in Figure 6 also scale linearly. However, in terms of magnitude, response time is an order of magnitude higher. This

[1] For legal reasons, we do not name the systems.

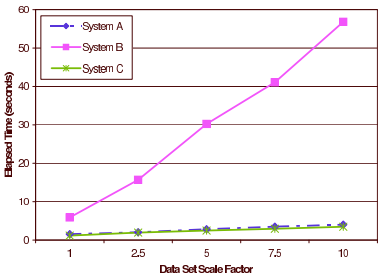

Figure 5: Average total elapsed time over 100 lowTxt queries.

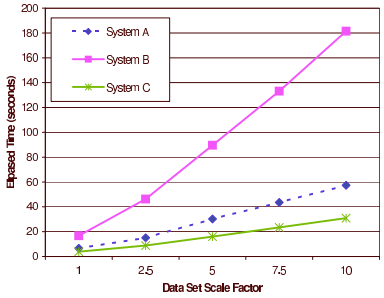

Figure 6: AndTxt costs more due to added complexity and more results.

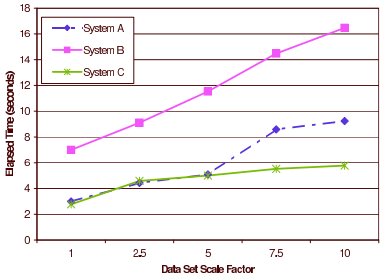

Figure 7: The systems are similarly differentiated for phrase queries.

is expected since the result set counts are an order of magnitude higher. The orTxt (OR) queries (not shown) similarly demonstrate linear scale up, both in terms of the relation size and in the number of words in the text expression.

Results for phrase queries are shown in Figure 7. Each phrase consists of two words. The anchor word is of low selectivity and its follow word is chosen from a random subset of documents that contain the anchor. For this experiment, the number of tuples (on average, 7) returned per phrase is held constant over all relations. The systems scale well for such highly selective queries and are ordered similarly as in the other experiments.

We conjecture that the difference in performance between systems A and C versus system B is due to an architectural difference. System B manages text indexing using relations whereas the other two systems utilize specialized, text indexing that can be more finely tuned for text-only workloads.

5.1.2 Mixed Query Workloads

The experiments over mixed query workloads combine the text workload from Section 5.1 with low (range predicate over NUM-U) and high (equality predicate over NUM-5) selectivity relational predicates. The workloads for single relation mixed workloads, *medMix, lowMix,* and *multMix*, are presented in Figure 2. One query workload that includes a join is *lowJoin*. The objective of the experiment is to determine whether a system's optimizer uses information about the text index in constructing an evaluation plan. First we consider the case where a single relation is referenced by the query, and this is followed by a section on queries that include joins.

The results for lowMix (not shown) are nearly identical to that of lowTxt results in Figure 5 with respect to both scaling and magnitude. Since the text expression is the same for both workloads, all systems first process the results from the text expression, retrieve the records, and filter those that do not match the relational predicate.

The medMix workload reverses the situation with a low selectivity relational predicate and a high selectivity text expression. The results are shown in Figure 8.

The reason for the difference is in the plans used by the three systems. Systems A and C use a nested loops join to combine relational and text results. However, system A always chooses to scan the results from the text expression as the *outer* access method. Thus, for low selectivity (few records selected) text queries, the overall response time is low. Conversely, for high selectivity text expressions, most probes from the text expression will not yield any results. System C on the other hand exchanges the inner and outer relations thus probing by the lower selectivity relational predicate, obtaining overall faster response times. System B takes an entirely different approach by converting each join input to a bitmap, intersecting, and fetching records by RID.

The *multMix* workload uses a two-word text expression to determine whether optimizers are sensitive to the type of text expression. The text expression used is from andTxt (AND queries) which have a higher selectivity than the relational predicate. The results (not shown) are nearly identical to the andTxt result shown in Figure 6 for systems A and C. However, the difference between System B and the the other systems decreases.

5.1.3 Multiple Relation Workloads

This section considers join queries. The lowJoin workload joins two relations, R and S with relation R being the $5x$ relation and S being the $2.5x$ relation. The join condition is on NUM-U whose values are unique so at most one record from R matches a record from S. Additionally, S has a low selectivity text expression whereas R has a high selectivity text expression. The average response time for systems A and C is 47 and 28 seconds, respectively. System B was unable to complete the experiment in a reasonable amount of time (over two orders of magnitude slower).

The query evaluation plans uncover another difference between systems A and C compared to system B. Systems A and C first select tuples using the low selectivity text expression on S. Then, relation R is joined, followed by a selection on R using high selectivity text expression. System B selects tuples using both text expressions on R and S, then join the result-

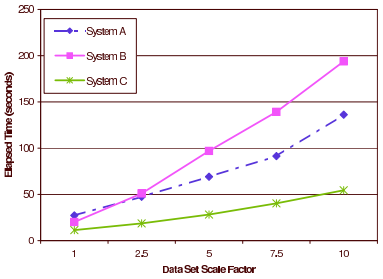

Figure 8: Scaling trends for medMix query workload.

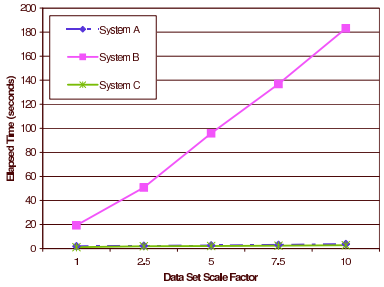

Figure 9: Scaling trends for medMix workload with top-k optimization.

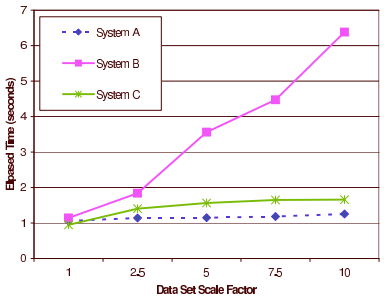

Figure 10: Scaling trends for the lowTxt workload using **VLC2-DB**.

ing tuples. Many unnecessary tuples from R are seen, thus leading to poorer performance.

5.2 Sort Order and Top-k Optimization

Thus far, our evaluation has focused on the case where the sort order is based on the score of the record with respect to its text expression. This corresponds to a web application in which a user would like to see the most relevant results where relevance is defined exclusively in terms of their text expression. Alternatively, the results could be sorted in terms of a relational attribute such as time. In either case, it is assumed that only the first few results are presented. For either sort clause, how do the systems' optimizations for top-k queries perform? The results in Table 4 show the average elapsed time for both text type queries and mixed queries where the sort order and top-k optimization hint are independently toggled. The results shown are obtained using the $10x$ relation.

	Score	S-Topk	Rel.	R-Topk
System A				
medMix	136	3.8	136	1.5
lowMix	4.0	1.5	4.0	3.8
System B				
medMix	194	183	186	1,520
lowMix	57	57	57	90
System C				
medMix	54	2.8	54	2.8
lowMix	3.4	1.5	3.5	1.5

Table 4: Average elapsed time in seconds for the medMix and lowMix workloads using the 10x relation. Query results are sorted by *Score* in the first two columns and by a relational attribute in the last two columns. All results are retrieved for columns 1 and 3 whereas the *top-k* optimization is used for columns 2 and 4.

The results in Table 4 show that for Systems A and C, performance is not affected by the choice of sort order. Similarly, the added benefit of using a top-k optimization is similar for either sort order. System B is also indifferent to the sort order used. However, the effect of a top-k clause differs depending on which sort is used. When using a score based sort, top-k does not affect the performance as significantly as it did for the other systems. However, using a relational sort in combination with a top-k hint results in a significant degradation in performance.

Now we examine how the systems fare when the data set size is scaled up. Results for the medMix workload is shown in Figure 9. The primary difference is the level of performance degradation for System B.

5.3 Results Using VLC2 Data Set

The results obtained using **VLC2-DB** highlight some of the differences between basing the TXT-LONG attribute on the VLC2 collection, rather than the AP collection. As shown in Figure 10, the response times are considerably lower than the comparable lowTxt queries over **AP-DB** shown in Figure 5. The reason is that due to the much larger vocabulary in VLC2, there are, on the average, fewer results for the low frequency workload. In addition, system C is shown to perform more poorly than system A which was not the case when using the AP data set. The results from the other experiments (not shown) are similar to basic trends observed with the AP data set: (1) systems A and C generally out-perform system B and (2) system B has significant degradation in performance for top-k query workloads. However, system A does not dominate over system C or vice-versa across all workloads.

5.4 A Note on Loading and Indexing

While loading and indexing are not the focus of TEXTURE, performance across systems was found to vary widely. The large variability was problematic for scaling up to larger dataset sizes. For example, loading $846,780$ tuples, requiring 2.5 GB of storage into system A required roughly 6 minutes. The other systems each required on the order of hours to load the same dataset. The indexing times for the VLC2 data set are summarized in Table 5. The bulk-mode indexing option for each system was used. We refer the reader to compare these times with times reported in [15] which are over the full VLC2 collection. We believe there is significant room for improvement with regard to loading and indexing TextType data.

VLC2	1x	2.5x	5x	7.5x	10x
A	38	72	145	205	242
B	17	41	80	112	146
C	5	19	40	63	80

Table 5: Index times reported in minutes.

6 Synthetic Text Generation

The objective of the TextType generator, *TextGen*, is to populate the TXT-LONG attribute with text, thus must generate as many text values, i.e., *documents* as there are tuples in a TEXTURE relation. In this section, we focus on growing a *collection of documents* by increasing the number of documents, while preserving features of the collection that affect performance. A document is modeled as a bag of *words*. TextGen is given a collection C_{input} consisting of n documents as input, and outputs a *scaled up* collection C_{output} with m documents, where $m > n$.

The design of TextGen can be understood in terms of: (1) the *features* from C_{input} that are maintained during scale up, (2) the *analytical models* used to maintain the features while scaling, and (3) the model *parameters*.

While there are many features one could pick to characterize collection growth (see [4] for a more complete discussion), we focus on the features that we expect to have the most impact on performance:

- **Word Distribution (W)**: The word distribution $W(w, c)$ associates with every unique word w in the collection, the number of times c it appears in the collection.
- **Vocabulary Growth (G)**:Vocabulary growth refers to the observation in [16] that the number of unique words grows as new documents are added to a collection.
- **Unique Words per Document (U) and Document Length (D)**.

Each feature can be characterized using one of several analytical models, as discussed in [4]. The **models** used in TEXTURE for the features listed above are as follows:

- W**, Empirical Model**: The model used for maintaining W is based on the empirical distribution derived from the input collection. The model is used as a means for approximating the probability that a word will be selected for inclusion in the collection. If word w_i appears x_i times, then it will be chosen approximately $\frac{x_i}{X}$ times where X is the total number of words in C_{input}.
- G**, Heaps Law**: Vocabulary growth G is modeled using an empirical law, Heaps Law [16], which states that the number of unique words $|V|$ in a collection is a function of the number of total words in the collection $G(x) = \alpha x^{\beta}$.
- U, D**: Average**: The unique and total numbers of words in a document are modeled using their respective *averages*, computed directly over the set of documents in the input collection.

Heaps Law requires extra **model parameters**, α and β, that are derived from the input collection. The derivation is done by fitting 20 evenly spaced points, each point being the number of total words versus the number of unique words seen in a collection. The fit is done using a least squares fit function (in Matlab).

TextGen constructs a synthetic collection in the following three phases.

1. The input collection is *preprocessed* in order to derive model parameters.
2. The models are used in the *core* algorithm that populates documents with words. This algorithm is based upon drawing random, independent samples from W.
3. Independent sampling from the previous step does not support AND query workloads. The synthetic collection is *post-processed* in order to maintain word co-occurrence.

The three phases are discussed in more detail in the following Sections.

6.1 Text Generator Pre-Processing

The pre-processing phase serves two purposes. First, the necessary model parameters are derived as discussed above. Second, the input words are "randomized" meaning that all words in the input collection are assigned new words of the same length consisting of random letters.

Following randomization of the vocabulary, the collection words are then mapped to the new words to form a new collection. Thus, the randomization replaces each word. That is, a word prior to randomization will be replaced with a randomized word of the same length at every location in the collection where it appears.

The reasons for randomization are two-fold. The first is to ensure that all systems produce the same number of results for a given text expression. Systems may differ in how words are mapped to index keys; randomizing effectively levels the playing field with respect to the amount of data that is returned to the user. Empirically, we have seen that randomization produces nearly identical sets of resulting tuples across systems, whereas without randomization, the variance in the result set size across systems can be quite high. Therefore with randomization, we can compare system performance with out needing to factor differing result set sizes into observed differences between systems.

The second reason for randomization is to anonymize a possibly sensitive sample collection. A client may request that TEXTURE be run external to her organization, in which case, anonymization may first be run locally and the result handed over to the TEXTURE user.

A drawback of randomization is that system effort

spent in improving ranking quality is partially ignored. However, comparing systems with respect to quality, irrespective of randomization, would require agreed upon quality judgments for an ad hoc collection of documents and query workload, and is beyond the scope of this paper. It is fair to ask whether randomization affects performance; the results in Section 6.4.1 show that this is not the case.

6.2 Text Generator Algorithm

The core TextGen algorithm ties together the features and models introduced thus far in a common framework, allowing us to instantiate a specific TextGen for given features and models. Assuming that the model parameters have been estimated from the input collection C_{input}, the main loop of the generator proceeds as shown in Figure 11.

TextGen
INPUT: [W]:word distribution, [G]:growth function, [U]:unique words per document, [D]:total words per document, [m]:target number of documents, [k]:increment for growing vocabulary
OUTPUT: collection
for 1 to m
1. Every k documents, *GrowVocabulary* using G
2. Create document **d**
3. *for 1 to U*
   Choose word w from distr. W;
   Add w to document **d**
4. *AddRemainingWords* up to D words to **d**
5. Add **d** to collection

Return collection

Figure 11: Pseudo code for the TextGen procedure.

Each document is constructed by sampling independently u times from the word distribution W. Subsequently, *AddRemainingWords* fills in the rest of the document with repeated words in order to maintain the total number of words per document, D. *AddRemainingWords* proceeds by constructing a distribution from the u sampled words, and re-samples to obtain the remaining words.

6.2.1 Vocabulary Growth

The function *GrowVocabulary* is used to expand the current vocabulary from which document words are drawn. Consider the $(i + k)$th invocation. First, $x = G(n_{i+k}) - G(n_i)$ new words are generated where n_i is the total number of words sampled for i documents. Next, the sampled word counts observed from the ith to the $i + k$ document are merged into W and W is normalized. Subsequent word samples favor the new x words over W until each has been included at least once in the next k documents. The new words are merged into W during the next *GrowVocabulary* invocation.

Two issues remain unresolved: how many documents k before the next invocation, and how is a new word generated? If k is too small, the algorithm renormalizes more frequently, so is slower. If it is too large, the resulting growth curve will be less precise. The value for k that was empirically found to balance these trade offs is $k = 5,000$.

According to [4], word lengths are expected to increase logarithmically as new words are introduced into the vocabulary. A simplification to word generation is made by growing existing words by a constant factor. Since most systems use a level of indirection between an actual word and its representation in a TextIndex, the word length is assumed to not have a significant effect on performance.

6.3 Text Generator Post-Processing

Once all documents are constructed, a post-process phase fixes some anomalies that may result from the algorithm. One important anomaly is word co-occurrences. Recall from Figure 11 that samples from W are independently drawn. As a result, word clusters that co-occur in documents in the input collection may not co-occur in the synthetic collection. While this sampling approach is simple and efficient, the drawback is that performance prediction for AND queries may suffer. In order to maintain word clustering, which may appear in real collections, we use a post-processing step called *FixClusters*.

FixClusters takes as input a collection of word pairs along with the *degree* to which they are clustered. The task of FixClusters is to make the appropriate modifications to the output collection in order to maintain the given clustering degree for all input pairs.

The clustering degree is based on the similarity between a word pair $< w_1, w_2 >$ found in the sample and synthetic collection. Similarity is defined in terms of how far the word pairs are from independence. Since W may differ between C_{input} and C_{output}, we cannot use the joint distribution $P(w_1, w_2)$. Thus the measure of similarity used is the ratio of the observed joint probability to the joint probability under the independence assumption: $s = \frac{P(w_1,w_2)}{P_{\perp}(w_1,w_2)}$. The numerator is the number of documents containing $< w_1, w_2 >$ over n. The denominator is defined in terms of the probability that a document contains a given word: $P(w_1)P(w_2)$.

The algorithm works for a given pair by swapping w_1 or w_2 from documents containing exclusively one into another document containing exclusively the other until the target s is achieved assuming $s > 1$. Care has to be taken in order to preserve document lengths and W. Thus, when swapping in a cluster word, the word with the closest collection-wide frequency that is not another cluster word must be cho-

Collection	W	$G(x)$	Document Length	Unique	Clustering
Real	AP	AP	315	211	AP
***Synthetic*$_1$**	Zipf, $\theta = 0.8358$	Heaps: $\alpha = 51.1273, \beta = 0.495$	311	N/A	none
***Synthetic*$_2$**	Empirical			207	
***Synthetic*$_3$**					FixClusters
Real	VLC2	VLC2	444	196	VLC2
***Synthetic*$_2$**	Empirical	Heaps: $\alpha = 3.4187, \beta = 0.7566$	438	197	none

Table 6: The features, their models, and model parameters derived from the AP and VLC2 data sets versus the synthetic collections, derived from their samples, that are used to evaluate text generation performance.

sen.

Once n such documents are produced, and post-processed by FixClusters, C_{output} is complete and TextGen terminates.

6.4 Text Generator Evaluation

Ideally, performance over document collections produced by TextGen will be identical to performance over real document collections from which TextGen is parameterized. In order to test equality, we use the following experiment: given a *Real* collection of m documents, obtain a sample of n documents, $n < m$ which are used to parameterize TextGen to create a *Synthetic* collection with m documents. In order to show that TextGen collections are reasonable for several workloads, we wish to show that workloads evaluated over commercial systems A, B, and C yield performance that is similar to evaluating the same type of workload over the *Real* collection.

We use TEXTURE to run the experiment. As opposed to using TEXTURE for identifying differences between systems as was done in Section 5, now it is used to test whether a system performs similarly on multiple document collections.

For the *Real* collection, we use the AP and VLC2 as seed collections. Each has its words replaced with random words as discussed in Section 6.1. Their model parameters are summarized in Table 6. A TEXTURE database is created for each along with the derived, *Synthetic* collection. The sample used is 10% so the $1x$ relation has $8,467$ tuples and the $10x$ relation has $84,678$ tuples.

The query workload consists of text-only queries as described in Section 3. We consider only text queries since we are interested in testing the validity of TextGen.

By using TEXTURE, we can vary the collection size as well as the collection. Thus, several competing synthetic document collections are tested by varying the model that TextGen uses. In this sense, varying TextGen models and validating is a type of search that uses TEXTURE in order to rank TextGen models according to performance that is measured on their generated document collections.

Specifically, we compare the performance of three synthetic document collections, $Synthetic_1$, $Synthetic_2$, and $Synthetic_3$ against the *Real* collection. The synthetic collections differ in the model that TextGen uses. $Synthetic_3$'s model is described in the previous section. We vary the model for the other two collections in order to shed light on which parts of TextGen's model affect performance.

Feature	***Synthetic*$_3$**
W	Empirical
$G(x)$	Heaps, $\alpha = 30.1361, \beta = 0.5217$
Document Length	315
Unique	211
Clustering	FixClusters

Table 7: $Synthetic_3$ parameters used for **AP-DB**.

$Synthetic_1$ collections are generated by replacing the empirical distribution used for W with a model based on Zipf's Law [23]. Modeling word distributions using Zipf's Law is a common approach used in [14], [7], [9], and [12] to name a few. Its advantages include simplicity of implementation and its drawback is that it has been shown not to produce the best fit for word distributions[5]. In addition, $Synthetic_1$ does not explicitly maintain the number of unique words per document U.

$Synthetic_2$ collections are generated by excluding FixClusters. We wish to know if FixClusters is useful for AND workloads and does not hurt performance for other workloads.

Before comparing performance using the synthetic collections versus the real collection, we present results showing that randomization does not have a significant affect on performance. Then we use the randomized real collection as the real collection and show that $Synthetic_3$ out-performs the other text generators. As a result, the AP based TEXTURE database, **AP-DB** used in Section 5 uses the parameters shown in Table 7. Its parameters are derived from the whole AP collection so differ from the parameters shown in Table 6 which are derived from a sample.

6.4.1 Effect of Randomization

In order to test the effect of randomization, we use two collections: AP and randomized AP, where each unique word in AP was replaced with a randomized word. Then we compared the performance of lowTxt and medTxt workloads on AP as well as their translations to randomized AP words. First, however, we removed those queries that produced different results,

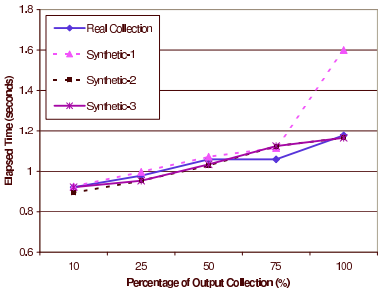

Figure 12: $Synthetic_1$ overestimates performance for lowTxt queries on System C.

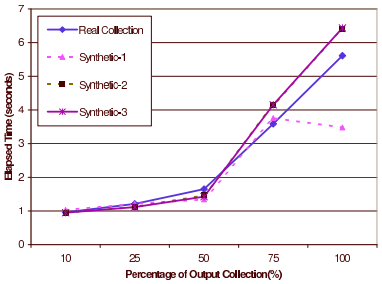

Figure 13: $Synthetic_1$ *underestimates* performance for lowTxt queries on system B so is inconsistent across systems.

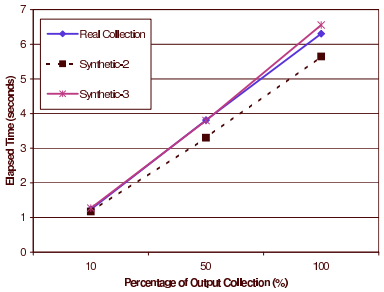

Figure 14: $Synthetic_3$ outperforms $Synthetic_2$ for andTxt queries on system A due to FixClusters.

either due to punctuation or stemming, in order to focus on the effect or randomization when the workloads and their results are held constant. The difference in response time across systems A, B, and C was found to be typically low ($< 1\%$) and did not exceed 4.4%. Therefore, we conclude that using randomized words is reasonable for the purpose of measuring performance.

6.4.2 Comparative Results

The results show that even for the simplest of workloads, the choice of text generator may lead one to incorrectly conclude that one system out-performs another. Looking at individual systems versus multiple text generators, Figures 12 and 13 show that the same generator leads to a different ranking of systems with regard to performance. More importantly, $Synthetic_3$ predicts performance well and consistently across systems, while e.g., $Synthetic_1$ predicts that System C can be expected to degrade more rapidly than the baseline indicates. The results for System A, not shown, have the same trend as System C except that $Synthetic_1$ *underestimates* the real collection performance. Also note that applying FixClusters in the case of $Synthetic_3$ does not effect single term workloads. Similar results were obtained for orTxt queries.

A significant difference between $Synthetic_1$ and the other generators is its use of Zipf's Law. The model a text generator uses for word distribution determines to a large extent the number of results that match a word. This corresponds to the length of a word's *posting list*, which a text index uses to store a word's matching documents. The other difference between $Synthetic_1$ and the others is that it does not model unique words per document as a feature. [2] Without maintaining unique words, more unique words are included per document, leading to more document matches per word.

The AND query workload poses challenges due to the independence assumption implicit in the core algorithm of text generator. Since $Synthetic_3$ and $Synthetic_2$ are found to be more accurate for single word and OR workloads, the results in Figure 14 compare only these two generators to the real collection. The results show better predictions with FixClusters ($Synthetic_3$) when applied to System A than without ($Synthetic_2$). The lower predicted response time with $Synthetic_2$ is expected since the workload with clusters returns more results when the data preserves the clusters as seen in the real collection as well as FixClusters in $Synthetic_3$.

The experiments based on the VLC2 collection are similarly encouraging. As a result of the poorer results obtained when using $Synthetic_1$, we compare the real VLC2 collection against $Synthetic_2$ using query workloads medTxt and lowTxt. The results for medTxt are shown in Figure 15. As with the experiments run against AP, System B is slower than the other two but the generator is more accurate for Systems A and B when compared to System C.

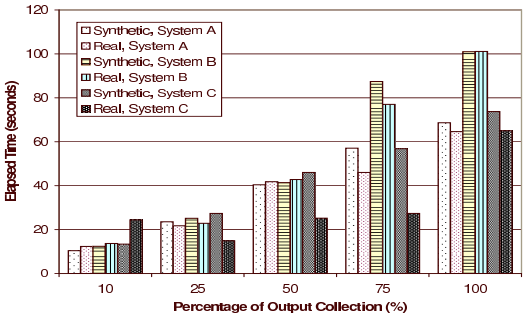

Figure 15: Results over all systems from evaluating medTxt using $Synthetic_2$ seeded with VLC2 data.

The results demonstrate that $Synthetic_3$ is more robust across workloads and systems in comparison to the other text generators. Thus, $Synthetic_3$ is used in the evaluation in Section 5.1 for scale-up experiments up to 10 times the input collection.

7 Related Work

There are several relational database benchmarks that measure performance using TextType. For example, TPC-W [3], specifies single word containment queries without a sort order or relational predicates. The ASP3 [13] and SetQuery [13] benchmarks spec-

[2] A similar generator using an empirical model for distribution was implemented without modeling unique words and proved to be uniformly worse than $Synthetic_2$ but better than $Synthetic_1$.

ify relational queries that mimic information retrieval processing. ASP3, for example includes an information retrieval type query whose objective is to test if a system utilizes index intersection plans. The Set Query benchmark specifies document retrieval queries as a count over similar types of queries found in ASP3. In contrast, TEXTURE focuses on end-to-end performance and does not make an assumption regarding how text processing is implemented.

Similarly, there are many benchmarks that focus exclusively on information retrieval systems. These include Full-Text Document Retrieval Benchmark (FTDR) [9], the Very Large collection (VLC) track from the TREC conferences [2], numerous research projects [6, 7, 21] to name a few. Amongst other differences, all differ from TEXTURE by not specifying relational predicates.

The TREC benchmarks focus on the quality of results given a user task. While each participant submits a summary of their experiment that includes indexing and query times, the granularity of this information is too coarse to make a detailed, systematic comparative analysis with the performance numbers obtained using TEXTURE.

The VLC Track from the TREC conference investigates how scale impacts performance and quality [15] when using a real data set. In addition, VLC compares systems by measuring indexing time and machine costs. These are both interesting and useful directions in which TEXTURE should be extended.

FTDR uses real data so it includes a wider variety of query workloads, including proximity queries. While scaling is specified, the assumption made is that a sufficiently large data set is available. TEXTURE, in contrast, trades diversity of the query workload in order to generate larger data sets in a systematic and potentially anonymizing manner. In addition, FTDR does not consider top-k queries or ordering results according to score, which is a significant limitation given the central role of such queries in practice.

Synthetic text generation has been used for the past fifty years [18]. It is being used in speech synthesis, benchmarking, and text retrieval research. TPC-W [3] for example includes the *WGEN* program that populates the benchmark's text attributes using a static collection of words and a grammar.

In [19], text generation is used to study the effect of a growing collection on inverted index maintenance strategies. A similar approach is taken in that an empirical word distribution and growth curve are used in text generation. However, the scale-up is limited to $3x$ as a result of the growth curve and fit that is used. In addition, clustering amongst words is not handled.

8 Conclusions

We have presented a text database benchmark and a detailed synthetic text generator that can scale up a given collection of documents. Additional benefits of synthetic generation include understanding one's dataset, the ability to finely control experiments, and to potentially anonymize a data set. Our results shed light on the support for text in current relational database systems, which is timely given the intense focus on enhancing this support. Future work includes addressing the other important issues including updates and how quickly they are propagated to indexes by different systems and at what cost, and the impact of multi-user workloads.

References

[1] http://es.csiro.au/TRECWeb/vlc2info.html.

[2] http://trec.nist.gov/.

[3] http://www.tpc.org/.

[4] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[5] Z. Bi, C. Faloutsos, and F. Korn. The dgx distribution for mining massive, skewed data. In *KDDM*, pages 17–26, 2001.

[6] E. Brown, J. Callan, and W. Croft. Fast incremental indexing for full-text information retrieval. In *VLDB*, pages 192 – 202, Santiago, Chille, September 1994.

[7] E. W. Brown. Execution performance issues in full-text information retrieval. Technical Report UM-CS-1995-081, 1995.

[8] W. B. Croft, L. A. Smith, and H. R. Turtle. A loosely-coupled integration of a text retrieval system and an object-oriented database system. In *ACM SIGIR*, pages 223–232. ACM Press, 1992.

[9] S. DeFazio. *Full-text document retrieval benchmark*, chapter 8. Morgan Kaufmann, 2 edition, 1993.

[10] D. DeWitt. *The Wisconsin Benchmark: Past, Present, and Future*, chapter 4. Morgan Kaufmann, 1991.

[11] C. Faloutsos. Access methods for text. *ACM Computing Surveys (CSUR)*, 17(1):47074, 1985.

[12] C. Faloutsos and H. V. Jagadish. On b-tree indices for skewed distributions. In L.-Y. Yuan, editor, *VLDB*, pages 363–374. Morgan Kaufmann, 1992.

[13] J. Gray. *The Benchmark Handbook For Database and Transaction Processing Systems*. Morgan Kaufmann, 1991.

[14] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In R. T. Snodgrass and M. Winslett, editors, *ACM SIGMOD*, pages 243–252. ACM Press, 1994.

[15] D. Hawking. Overview of trec-7 very large collection track. In E. M. Voorhees and D. K. Harman, editors, *The Seventh Text REtrieval Conference*, 1998.

[16] H. S. Heaps. *Information Retrieval, Computational and Theoretical Aspects*. Academic Press, Inc., New York, 1978.

[17] C. A. Lynch and M. Stonebraker. Extended user-defined indexing with application to textual databases. In F. Bancilhon and D. J. DeWitt, editors, *VLDB*, pages 306–317. Morgan Kaufmann, 1988.

[18] C. Shannon. Prediction and entropy of printed English. Technical report, Bell Systems, 1951.

[19] K. A. Shoens, A. Tomasic, and H. Garcia-Molina. Synthetic workload performance analysis of incremental updates. In *Research and Development in Information Retrieval*, pages 329–338, 1994.

[20] M. Stonebraker, H. Stettner, N. Lynn, J. Kalash, and A. Guttman. Document processing in a relational database system. *ACM Trans. Inf. Syst.*, 1(2):143–158, 1983.

[21] A. Tomasic and H. Garcia-Molina. Performance of inverted indices in distributed text document retrieval systems. In *PDIS*, pages 8–17, 1993.

[22] C. Turbyfill, C. Orji, and D. Bitton. As3ap - a comparative relational database benchmark. In *Proc. IEEE Compcon*, February 1989.

[23] G. Zipf. *Human Behaviour and the Principle of Least Effort: An Introduction to Human Ecology*. Hafner Publications, 1949.