The QUIQ Engine: A Hybrid IR–DB System

Navin Kabra Raghu Ramakrishnan Vuk Ercegovac

Veritas Inc. Department of Computer Sciences,
University of Wisconsin–Madison

E-mail: navin.kabra@veritas.com, {raghu, vuk}@cs.wisc.edu

Abstract

For applications that involve rapidly changing textual data and also require traditional DBMS capabilities, current systems are unsatisfactory. In this paper, we describe a hybrid IR-DB system that serves as the basis for the QUIQ-Connect product, a collaborative customer support application. We present a novel query paradigm and system architecture, along with performance results.

1. Introduction

Internet-based customer support has grown ubiquitous in recent years because it costs less than traditional channels such as phone support. The QUIQConnect application enables users to *post* a question if they cannot find a satisfactory answer in the knowledge-base. Others can post answers; questions and related answers are automatically combined into searchable *knowledge units*.

QUIQConnect content is a combination of structured and unstructured data. We require a query paradigm that adequately bridges the exact answers of relational database (DB) systems and the ranked answers of information retrieval (IR) systems. Further, updates must be applied immediately in order to meet the application requirements. All data is stored in a relational DBMS. However, the query and update performance was not adequate (in either speed or quality of results), and we developed the *QUIQ Query Engine (QQE)* to address this problem. The main contributions that arise from our approach are: (1) a novel data model and query paradigm that combines ideas from DB and IR approaches, (2) fast updates and queries through the use of a self-organizing differential index structure to avoid in-place updates, and (3) an integration architecture that leverages the DBMS for concurrency and recovery. The rest of this paper is organized as follows. Section 2 describes the unified query paradigm. Section 3 describes the QQE architecture and implementation details. Performance is summarized in Section 4, followed by a discussion of related work in Section 5.

2. A Hybrid DB-IR Query Paradigm

In the QUIQConnect model, each data object (or tuple) has a *TID* and is described by a set of ⟨ tag-name, tag-type, tag-value ⟩ triples (which we often refer to as *tags*). In contrast to the relational model, we do not require a fixed set of tags within a collection. We focus on selection queries over a single collection, with results ranked in terms of how well they match the query. In contrast to XML systems, we concentrate on a simpler structural model that allows us to focus on text *vs.* non-text attributes and relevance-ranked retrieval. In contrast to IR systems, the tags provide some structure and semantics.

We now describe the types of constraints that are supported, followed by a description of how we can apply the constraints to both text and non-text fields. A query in QUIQConnect can be decomposed into three sets of constraints—*match*, *filter*, and *quality* constraints. The query result is essentially the result of all the *match* and *filter* constraints AND-ed together. The *quality* constraints are used to adjust the relevance of those results.

Intuitively, *Match* constraints are *approximate* constraints that specify what the user is looking for. A tuple can appear in the result of a query only if it satisfies at least one match constraint. The *relevance* of a tuple is mainly determined by how many match constraints it matches, and how well. While relevance is not tied to a particular algorithm, for concreteness we use the well-known TF-IDF formula.

Filter constraints are *exact* constraints, and act like a `WHERE` clause in a SQL query; only tuples that satisfy all filter constraints are in the query result. On the other hand, tuples that satisfy the filter constraints but do not match any match constraints are not in in the query result.

The preceding discussion is applicable to tokens in general and does not explain how QQE evaluates constraints over non-text attributes such as integers and dates. A typical approach for handling both types of attributes in the same system is to manage text data with an IR system and the

other data using an independent database engine. However, this approach has performance limitations since a constraint from one engine cannot be used to prune results in the other. Furthermore, it does not support the range of queries that we aim to support. For example, we cannot express find "inexpensive car under 5 years".

In our hybrid approach, text and non-text data can be combined in a single index. The basic idea is to map non-text data to pseudo-keywords that cannot be confused with actual keywords of text. Now, each distinct value that a non-text attribute in the database might take is mapped to one or more of the pseudo keywords. The mapping scheme can differ based on the data-type of the attribute. In some cases, the mapping scheme might contain collisions, necessitating the use of post-processing to remove false positives.

This approach allows us to compute relevance even for constraints on non-text fields, using a common TF-IDF framework. Relevance calculation for quality constraints is a special case of this framework.

3. System Architecture

QQE consists of a DBMS that holds all the base data and an external *index server* that maintains the unified index. Inserts/Updates are made directly to the DBMS. The index server monitors these updates to keep its indexes current. It can also be updated in bulk-load mode. Data retrieval is done by querying the index server.

The primary data structure in the index server is an *inverted index* that maps each token appearing in an attribute to a *TIDLIST*. Each entry of the TIDLIST is a TID and a count that represents the number of times the token appears in the given attribute of the TID's tuple. Entries are sorted in descending TID order, i.e., youngest tuples first, and an entry does not appear for counts of zero token occurrences.

Our basic idea is to *defer applying update operations* to the persistent store of an index server. Updates are handled in three steps: (1) Changes to the database are written to a special JOBS table as a part of the same transaction. (2) The JOBS table is continually polled by the index server and changes are incorporated into an in-memory differential index structure, referred to as the *dynamic index*. (3) The persistent on-disk index, referred to as the *static index*, is periodically refreshed to absorb the dynamic index. These details must be transparent to data retrieval operations, and therefore retrieval operations have an additional step of checking results against the *dynamic index* to adjust for changes that have not yet made it to the *static index*.

This approach allows us to disregard random updates and optimize persistent index structures as if they were static, since they are refreshed offline and accessed sequentially. Since query processing time is dominated by processing constraints to identify the top results, these index optimizations dramatically improve performance. Periodic refreshes can also be combined with analysis/mining of query and update traces to make the system self-tuning at the storage level, extending the current state of the art (which has concentrated on the choice of indexes). Furthermore, related data such as statistics are regularly refreshed as a side-effect. A final benefit of a self-organizing index is in evolution flexibility. For example, format changes may be incorporated by simply restarting the server process with a new version of the executable, which can read the old format and write the new format during a refresh.

Concurrency requires only short term latches protecting the in-memory data-structures since the disk structures are never written. For recovery, the JOBS table is effectively used as a *redo log*. All jobs that have a timestamp newer than the timestamp of the static index are fetched and reapplied at system start-up time.

4. Performance

In this section, we report on the results of a performance study of QQE. Details can be found in [8]; we just summarize the results below.

We studied how performance scales with query size. It scales close to linearly with increasing document size and number of documents. It scales super-linearly as the number of query tokens is increased, but this can be offset significantly by certain optimizations we implemented.

For studying updates, we measure the throughput for insert-only, update-only, mixed insert-update, and bulk-load workloads as the number of jobs increase and as the average job size increases. For studying the effect of merging static and dynamic indices on queries, we vary the time interval for merging all partitions versus workloads consisting of queries, insert-only, update-only, and a mix of inserts and updates. Details can be found in [8].

Finally, we designed a comparative study against a DBMS Text Extension (*DBMS-TE* for short) of a commercial database. Several issues had to be addressed. First, QUIQConnect supports data-types whose content is stored on the file-system and referenced by names stored in relational tables. These data-types were converted to native data-types supported by DBMS-TE. Second, queries used for QQE have to be translated to SQL with text extensions. Finally, when making measurements during the experiments, we need to insure that the state of DBMS-TE corresponded to that of QQE (e.g., whether the inserted tuple is visible to queries).

The query experiments show QQE out-performing DBMS-TE by 60% to over an order of magnitude. However,the DBMS-TE performed better for bulk-loads and converting an unoptimized structure to an optimized structure. However, QQE out-performed DBMS-TE for work-

loads composed of straight inserts, straight updates, and a mix of inserts and updates.

5. Related Work

This section outlines work related to QQE in terms of the query paradigm and managing a dynamic corpus. It is intended to highlight different approaches along these two dimensions and is not intended to be comprehensive. We consider three types of systems: (1) DBMSs extended to handle text, (2) IR systems implemented to handle a dynamic corpus, and (3) other systems that defer updates for a combination of performance and application reasons.

Commercial RDBMSs have been extended to allow keyword searches over textual attributes but they do not have a very sophisticated notion of relevance, and simply apply an external text-search engine on a per-field basis. With respect to handling a dynamic corpus, RDBMSs typically provide facilities that allow an administrator to control when an update is made visible to queries.

Database systems not based on the relational model have also been extended to incorporate IR style queries. The HySpirit system described in [6] combines Datalog with IR using a probabilistic scheme. Additionally, MOA [5] provides an integrated DBMS and IR query paradigm. It is unclear how their performance would compare to QQE.

IR systems allow a document to have multiple attributes, but they only use the attributes to filter results after the regular "relevance" query has been evaluated. QQE, through the use of a quality constraint, can additionally re-order results based on non-text attributes.

In terms of managing a dynamic corpus, *dynamic inverted index* have been extensively studied. Since most of the systems *defer applying changes*, the key differences relative to QQE are in *how* and *when* the changes are propagated to disk. QQE propagates changes by *rewriting* the entire static index whereas many other systems do *in-place* propagation. Additionally, QQE applies changes based on a fixed time interval whereas many other systems are driven by *events* such as exceeding a memory threshold. Finally, the document identifier space in QQE is shared with the RDBMS, requiring a true update operation but not requiring a remapping before retrieving the tuple.

The publicly available search engine framework Lucene [4] applies changes based on a memory threshold and uses rewrite.

The system described in [2] differs by propagating changes in-place. Similarly, the Spider system [9] propagates updates in-place but does not allow queries over pending updates. The Gold Mailer system described in [1] propagates changes periodically but it is unclear if rewrite is used. The study in [11] considers various degrees of writing in-place versus rewriting.

Finally, much work utilizing deferred updates has been done in systems that are not specific to text. For example, the work in [10] discussed differential database file structures. Data warehouses also need to accommodate changes to a structure that is highly optimized for queries. The work in [7] proposes that the changes are similarly deferred and merged into the main structure using a multi-level merge algorithm.

6. Conclusion

In conclusion, QQE is designed for applications that require the flexibility and intuitiveness of text search combined with the structured meta-data. The system is architected for dynamic environments with a significantly greater number of queries than change requests, and is optimized for fast query responses.

Acknowledgments

We thank several people at QUIQ who contributed to QQE: Andrew Baptist, Matt Hanselman, Jim Kupsch, Rajesh Raman, and Uri Shaft.

References

[1] D. Barbará, C. Clifton, F. Douglis, H. Garcia-Molina, S. Johnson, B. Kao, S. Mehrotra, J. Tellefsen, and R. Walsh. The Gold Mailer. In *Proc. ICDE*, 1993.

[2] E. Brown, J. Callan, and W. Croft. Fast Incremental Indexing for Full-Text IR. *Proc. VLDB*, 1994.

[3] T.-C. Chiueh and L. Huang. Efficient Real-Time Index Updates in Text Retrieval Systems. TR-66, SUNY at Stony Brook, Mar. 1999.

[4] D. Cutting. The Jakarta Lucene Project. `http://jakarta.apache.org/lucene`.

[5] A. de Vries and A. Wilschut. On the Integration of IR and Databases. *IFIP 2.6 DS-8 Conf.*, 1999.

[6] N. Fuhr and T. Rölleke. HySpirit–A Probabilistic Inference Engine for Hypermedia Retrieval in Large Databases. *Advances in Database Technology–EDBT*, 1998.

[7] H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View Maintenance Issues for the Chronicle Data Model. *Proc. ACM PODS*, 1995.

[8] N. Kabra, R. Ramakrishnan, and V. Ercegovac. The QUIQ Engine: A Hybrid IR–DB System. TR-1449, Department of Computer Sciences, University of Wisconsin–Madison, Nov. 2002.

[9] D. Knaus and P. Schäuble. The System Architecture and the Transaction Concept of the SPIDER Information Retrieval System. *Data Engineering Bulletin*, 19(1):43–52, 1996.

[10] D. Severance and G. Lohman. Differential Files: Their Application to the Maintenance of Large Databases. *ACM TODS*, 1(3):256–267, Sept. 1976.

[11] A. Tomasic, H. Garcia-Molina, and K. A. Shoens. Incremental Updates of Inverted Lists for Text Document Retrieval. *Proc. ACM SIGMOD Intl. Conf.*, 1994.