# Toward DB-IR Integration: Per-Document Basis Transactional Index Maintenance


Jinsuk Kim jinsuk@kisti.re.kr
Du-Seok Jin dsjin@kisti.re.kr
Yunsoo Choi armian@kisti.re.kr
Chang-Hoo Jeong chjeong@kisti.re.kr
Kwangyoung Kim kykim@kisti.re.kr
Sung-Pil Choi spchoi@kisti.re.kr
Minho Lee cokeman@kisti.re.kr
Min-Hee Cho mini@kisti.re.kr
Ho-Seop Choe hschoe@kisti.re.kr
Hwa-Mook Yoon hmyoon@kisti.re.kr
Jeong-Hyun Seo jerry@kisti.re.kr

Information System Development Team
Information Technology Development Division
Korea Institute of Science & Technology Information
52-11, Eoeun-dong, Yuseong-gu, Daejeon, 305-806, Republic of Korea



## Abstract

*While information retrieval* (IR) *and databases* (DB) *have been developed independently, there have been emerging requirements that both data management and efficient text retrieval should be supported simultaneously in an information system such as health care systems, bulletin boards, XML data management, and digital libraries. Recently DB-IR integration issue has been budded in the research field. The great divide between DB and IR has caused different manners in index maintenance for newly arriving documents. While DB has extended its SQL layer to cope with text fields due to lack of intact mechanism to build IR-like index, IR usually treats a block of new documents as a logical unit of index maintenance since it has no concept of integrity constraint. However, towards DB-IR integration, a transaction on adding or updating a document should include maintenance of the postings lists accompanied by the document – hence* per-document basis transactional index maintenance. *In this paper, performance of a few strategies for per-document basis transaction for inserting documents – direct index update, stand-alone auxiliary index and pulsing auxiliary index – will be evaluated. The result tested on the* `KRISTAL-IRMS` *shows that the pulsing auxiliary strategy, where long postings lists in the auxiliary index are in-place updated to the main index whereas short lists are directly updated in the auxiliary index, can be a challenging candidate for text field indexing in DB-IR integration.*


## 1. Introduction

"The world of data has been developed from two main points of view: the structured relational data model and the unstructured text model. The two distinct cultures of databases and information retrieval now have a natural meeting place in the Web with its semi-structured XML model. As web-style searching becomes an ubiquitous tool, the need for integrating these two viewpoints becomes even more important." – Consens and Baeza-Yates [13].

DB-IR integration is a recent budding area of interest in the research fields of database area [15, 1, 10] and information retrieval area [3, 13, 6].

There have been several approaches to merge DB's structured data management and IR's text search facilities. The first approach, which is the most commonly adopted architecture for many real information systems, is simply to glue existing DB and IR engines at the application or middleware level, hence called *middleware approach* [10]. In the middleware approach, a DBMS engine manages the documents and an information retrieval system (IRS) takes part in building indexes and supporting user query evaluations. Web search engines belong to this category, of which documents are collected by web crawlers with DB-like but very simplified data management features, and of which indexes are built – although usually built off-line – by separate index servers. Though this middleware architecture is prevailed in real field of information services, it is very difficult to synchronize DBMS' document contents and IRS' index information, resulting in *document-index gap*. Furthermore this approach fails to be powerful enough to fully utilize the DB and IR facilities, since DB is a black box in IR's view, and vice versa

The second approach is to extend DBMS by loosely coupling IR functionalities – hence *loose coupling* [30]. Commercial DBMS vendors provide extensible architectures implemented using a high-level (typically, SQL-level) interface. Oracle's data blade, DB2's cartridge and Informix's extender belong to this class. Although loose coupling can be implemented easily, it is argued that loose-coupling violates many assumptions hardwired into current database systems [1], and it is not preferable for implementing new data types and operations in large databases when high performance is required [30]. Another example of loose coupling is the QUIQ engine, a hybrid IR-DB system of which DBMS holds all the base data and an external index server maintains the index [21]. The TopX engine, which combines search in semi-structured data (XML) with top-k retrieval techniques, is built on top of a commercial RDBMS (Oracle) [28]. MonetDB/X100, an experimental relational database engine, showed comparable efficiency and effectiveness by building an inverted list for TREC-TeraByte text data as a separate relational table [20].

The third approach is to extend DBMS by tightly coupling IR features [30], or vice versa – *tight-coupling*. In tight-coupling of DBMS with IR features, new data types and their corresponding IR operations are integrated into the core of the DBMS engine in the extensible type layer. One of the well-known examples of this category is `Odysseus` [30], an object-relational database management system (ORDBMS) with tightly-coupled IR features. In the reverse direction of tight-coupling where an IRS integrates some DB features, transactional data management is incorporated into the core of the IRS engine [17]. We call this kind of IR-extended engine as an `IRMS`, *Information Retrieval & Management System*.

The last approach is to design an integrated DB-IR on the scratch. This approach is under discussion [10, 1]. It is suggested that the new system architecture of DB-IR unification will have advantages of *structural data independence, generalized scoring*, and *a flexible and powerful query language* [1]. Chaudhuri and his colleagues suggested "the storage-level core system with `RISC`-style functionality in DB-IR integration" as the most plausible architecture [10].

Figure 1 summarizes these approaches. Web crawlers and web search engines are included for the purpose of comparison. A web crawler is program that collects web documents to be indexed, where web page contents are managed by URLs as keys. In the context of Figure 1, data management is essential for performance but retrieval is not important feature in the web crawlers. On the other hand, a web search engine is a kind of IRS customized to deal tens

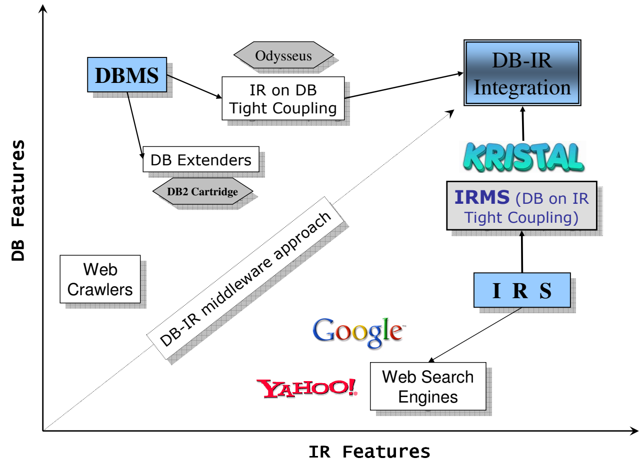



**Figure 1. Evolution of Information Systems. DB and IR are climbing up to the ridge of DB-IR Integration.**

of millions to tens of billions of web pages. Web search engines commonly rebuild the index from the documents collected by the web crawlers periodically. Therefore, data management is not an important factor but the retrieval feature is essential for web search engines. In web search engines, web page contents are managed by web crawlers which can be regarded as extremely simplified database system. Thus web search engines, briefly to say, belong to the class of DB-IR middleware architecture. DB (DBMS) has adopted many additional features which were out of interest in traditional database world [18], including IR features. IRS is being faced with user requirements such as transaction processing, join, materialized view, trigger, etc. Although their start points were different, we see, they will converge into a single ridge of DB-IR integration.

Getting to the ridge, it will be a rough and long way with many issues to be solved (see [10, 18, 3, 1] for more detail of these issues). One of the fundamental issues in DB-IR unification is the dynamic index maintenance, of which performance for text fields is a main obstacle both in IR and DB. We will mention studies in index maintenance at the last of this section.

From a viewpoint of IR, index maintenance strategies for text retrieval systems in dynamic search environments, where index update operations are interleaved with search queries, have been studied intensively over the past few years [25, 24, 9, 23]. To implement an index update strategy, it is straightforward to construct and index for *a block of new documents*, then merge it with an existing index [24, 23]. Buffering new documents, as they will share many terms, the per-document cost of index update can be significantly reduced. These approaches maintain the postings lists for incoming documents *in-memory* data structures and, upon meeting certain threshold conditions such as given memory size, time interval and number of new documents, in-memory index is merged to on-disk main index.

As in the DB-IR middleware approach, IR usually shifts the responsibility of data management to another tools or software such as DBMS. Furthermore there is no equivalent in a text retrieval system to the concept of integrity constraint, since index can be built at any time from the source documents. Thus updates of index for incoming documents, in IR, do not necessarily have to be instantaneous and most of the IR approaches in index maintenance regards *a bunch* of new documents as a logical unit of work.

The situation is changing rapidly. During more than ten years of field experience in IR, we are facing user requirements of fast immediate index update as well as fast crash recovery on IR contents. The bunch of documents approach does not meet these requirements since upon crash in-memory postings can be lost and its re-construction can be a too time-consuming and complex job to transfer to the field managers who are usually unaware of the depth of their text systems. In addition to this kind of practical viewpoint, in our academic consideration of true DB-IR integration, there should be data integrity constraint between a document and its accompanying index. If there is a possibility that a part of index can be lost upon certain circumstances, it is far from data integrity since data integrity can be simply defined as preserving data for their intended use. Since retrieval will be the basic "intended use" of data, *document-index integrity* should be guaranteed in DB-IR integration. To reserve document-index integrity, DB-IR integration will support per-document transaction with the ACID properties, of which work unit includes one document and its index information as well. And all transactions are written into a log that can be played back to recreate the system to its state right before a failure or abort, as DB typically does. We call this policy as *per-document basis transactional index maintenance*.

In this paper, we propose two on-disk auxiliary-based strategies for per-document transaction of index maintenance and compares their performance with direct update strategy. (The transactional index maintenance is expected to be inferior to bulk documents strategies since it has inevitable overhead of heavy disk accesses for hundreds of postings updates and on-disk transaction logs.) An auxiliary index is an *on-disk supplementary data structure* which contains the postings lists of newly arriving documents and changing documents. The first auxiliary strategy, *stand-alone auxiliary index*, accumulates postings lists of upcoming documents without any limitation. The second, *pulsing auxiliary index*, accumulates postings lists in the auxiliary data structure while certain threshold is met. If a postings list exceeds the threshold, it is merged to the main index. These approaches were tested on `KRISTAL-IRMS`[1] engine using bibliographic data for journal articles, each data containing more than one hundred unique terms. Auxiliary approaches outperform direct update strategy and pulsing method outperforms stand-alone auxiliary index. Our results show, as expected from frequent disk-access overhead, per-document insertion speed of 2–5 seconds in the on-disk auxiliary approaches, which is significantly (more than a magnitude) inferior to about 0.1 second of the bunch of documents approaches [24, 9, 23], leaving a vast room for future research.

In Section 2, we give a brief overview of related work on on-line index maintenance. Section 3 describes transactional index maintenance of which transaction unit is an incoming document rather than a block of documents in traditional index maintenance approaches in the IR field. Experimental methods and results are described in Section 4 and Section 5, respectively. Finally we conclude and suggest future work in Section 6.

## 2. Related Work

Inverted files have been proved to be the most efficient data structure in high-performance retrieval systems for large text data [33]. There have been vast body of work on devising inverted list index [16, 31, 32]. There has also been work on devising inverted lists that can efficiently handle document insertions, deletions or updates [29, 21, 25, 24, 23, 9]. For efficiency, these work approached the index maintenance issue in the manner of "as many as possible documents" as a unit task. Pulsing technique, an in-memory buffering scheme keeping short lists in the vocabulary structure while pulsing long lists to on-disk supplementary heap file, was introduced in [14]. The system described in [8] and the Spider IR system [22] propagates index changes in-place. The Gold Mailer system [4, 5] propagates changes periodically. The publicly available search engine framework `Lucene` [7] applies changes based on a memory threshold. The studies in [29, 27] explored space allocation strategies in in-place strategy where short lists are grouped in fixed-size buckets to reduce the problem of managing space for numerous small inverted lists. And the overallocation scheme was further extended in [26]. The granularity of propagation of index updates to disk was considered in [12]. A technique, called *landmark-diff update* method was introduced to improve the index update performance for existing documents of which contents are changed [25].

Most of the index maintenance strategies work by accumulating postings for *a block of incoming documents* in

[1] `KRISTAL` stands for Knowledge Retrieval In Science and Technology Affiliated Literatures. It aims at an IRMS finally intending to a full DB-IR integration. Source is open at http://www.kristalinfo.com

main memory, building an in-memory inverted file. To amortize disk-access costs, writing or propagating these in-memory postings to existing on-disk main index is deferred to the moment meeting a certain, predefined threshold. The QUIQ engine, a DBMS extender for text fields, applies index changes to the on-disk index based on a fixed time interval [21]. Work in IR usually propagate changes to disk driven by events such as exceeding a memory threshold or the number of updated documents [23, 9]. This way, disk accesses can be amortized over a large number of index update operations, resulting in increased indexing performance. However, under crashes, data integrity between documents and indexes can be damaged in these bunch of documents approaches.

Chunk method, where the document collection is divided into "chunks" of ordering documents' scores, is newly introduced to deal with ranking queries for structured data values in relational databases under update-intensive environment [19]. However, since chunk method was designed for numeric data types and for only update-intensive systems, it is difficult to be applied to text data types and insert-intensive environment. In CompleteSearch system [6], a novel kind of index data structure, `HYB`, was introduced to support *context-sensitive prefix search and completion* mechanism. The block index HYB consists of bags of (*word_id*, *document_id*) pairs and can be augmented with word positions. The performance of HYB data structure in on-line index maintenance is still unclear.

# 3. Transactional Index Maintenance

Since most information retrieval systems lack the concept of integrity constraint, they have opportunities to regard a block of new documents – for efficiency – as a unit of index maintenance. However, towards DB-IR integration and to meet the field requirements from database managers in text service area, index maintenance should be supported in the on-disk storage level, not in-memory. This means the logical unit of index maintenance process should consist of each document (not multiple documents) and its accompanying index information. And, to support `ACID` property of database systems, an integrated DB-IR must support logged processing of index maintenance. We have tested three strategies for per-document basis index maintenance regarding each document and its index as a transaction unit: naive *direct index update*, *stand-alone auxiliary index*, and a hybrid strategy of these two, *pulsing auxiliary index*.

For all experiments in this work, every changes in index data is logged to disk in per-document basis transaction unit. If the transaction is successful, the log is discarded. Otherwise, if the transaction fails for some reason, the log is used to roll-back to the state of just before the transaction. In the context of this paper, all inverted lists contain full positional information, *i.e.* the exact locations of all occurrences of a given term. Since our auxiliary approaches contains two separate on-disk indexes, we call primary index for existing documents as *main index* and supplementary index for newly incoming documents as *auxiliary index*.

## 3.1. Direct Index Update

The direct method for updating index is to simply amend the index, list by list, to include information about a new document. The naive manner of *direct index update* is an intuitive and very simple index update strategy. Whenever a document is arrived to the system, the posting list for each term is appended to the main index. In general this requires relocating existing on-disk postings lists for each term in the new document. Since a new document will usually have hundreds of unique terms, this kind of direct, *in-place*, update for main index requires hundreds of time-consuming relocations of postings list in the main index. Overallocation of postings list – which leaves some amount of free space after every postings list – in the main index is proposed to relieve these costly operations [9]. However, overallocation size is not predictable in real databases and overallocation can be degenerated into garbages if there is few update and addition of documents, or falls into same situation of no-overallocation after a huge amount of documents are inserted hence filling over-allocated free spaces in the postings list. Therefore we do not regard the overallocation strategy in this experiment.

In direct update strategy, query processing performance will be comparable to totally re-built databases since postings lists are guaranteed to be contiguous. However, the frequent relocations of postings lists in the main index will degrade the update performance seriously. As a typical document contains hundreds of distinct terms, such an update involves hundreds of disk accesses, a cost that is only likely tolerable if the database is very small or the rate of update is very low indeed.

## 3.2. Stand-alone Auxiliary Index

Direct update strategy is expected to suffer from frequent relocations of long postings lists. If the size of each relocation is reduced to be small enough, the overall update performance can be improved. To reduce each relocation size, we introduced an on-disk auxiliary index for incoming documents. Whenever a new document is arrived, the indexing results are appended to the auxiliary index rather than to the main index. (As depicted in Figure 2, deletion is implemented as `delete list` and update of existing document as (delete+insert) with `update list`.) At initial state, the auxiliary index is empty or small, hence new postings lists from the incoming documents will be short and can be easily inserted to the auxiliary index. However as new documents are accumulated, postings lists in the auxiliary index grow in size resulting in deteriorated performance due to frequent relocations of grown postings lists.

In our implementation of the stand-alone auxiliary index strategy, data structure of the auxiliary index is the same as the main index which uses B$^+$-tree for key storage in lexicographical order. This is an inevitable choice to support the range operations such as `(PUBDATE >= 20010101 AND PUBDATE <= 20061231)` and right truncations or prefix wild-carding such as `TITLE:(index* AND strateg*)`. Since the auxiliary index has the same data structure with the main index, updating new postings lists of the auxiliary index can be worse to the level of the main index as the size of new documents grows. At this point, whole postings lists of the stand-alone auxiliary index should be merged to main index and vacated to guarantee the update performance – we call this bulk procedure as *index optimization*. `KRISTAL-IRMS` uses in-place update strategy [29, 24, 23] for this purpose. During in-place update of the auxiliary index to the main index, query processing is halted.

In our index maintenance strategies, 'delete' of a document is achieved simply tagging it as deleted in the delete list and update of a document is processed as `delete+insert` – the cost of delete can be ignored and the cost of update of an existing document is similar to insertion. Therefore we only focus on insertion of new documents in this paper.

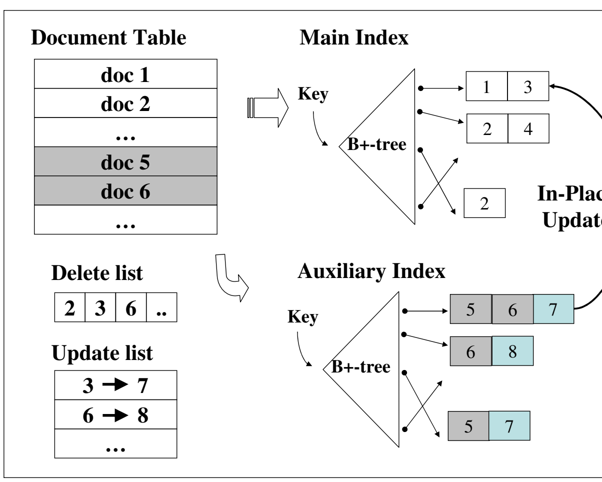



**Figure 2. The Pulsing Auxiliary Index with Pulsing Threshold of DF=3 (Detail structure including term frequencies and physically separated positions lists are omitted for clarity). Without In-Place Update, it depicts the Stand-alone Auxiliary Index.**

### 3.3. Pulsing Auxiliary Index

Introduction of an auxiliary index will be benefited from the small size of the auxiliary index itself. However, as new documents are accumulated, the stand-alone auxiliary index also grows to the point where the index optimization is needed to support fast index maintenance for future incoming documents. However, if the size of auxiliary index can be maintained to be compact enough throughout the processing of incoming documents, index update performance will not be degenerated significantly, as in the case of initial stage of the stand-alone auxiliary strategy.

In both the direct update strategy and the stand-alone auxiliary strategy, the most expensive job is the relocation of long postings lists. Furthermore, since terms with high document frequencies such as "`a`", "`of`", and "`the`" are commonly co-occur in a document, the direct index update and the stand-alone auxiliary strategies have the high probability of relocations of more long postings lists in a single transaction. The pulsing auxiliary index strategy is a hybrid strategy of direct index update and the stand-alone auxiliary index strategy. In this hybrid strategy, to reduce the number of long postings list to process simultaneously in the auxiliary index, every auxiliary postings list longer than a given threshold is in-place updated or *pulsed* to the main index and discarded from the auxiliary index. Since the terms with high frequencies are usually – not exactly – co-occurs, pulsing long postings lists to the main index is dispersed throughout transactions, not co-occurring in a transaction. Since this strategy is similar to the *pulsing* technique described by Cutting and Pedersen [14], we named this strategy as *pulsing auxiliary index* after their work.

Say, the pulsing threshold for long postings lists is the document frequency of 500. And already sufficient documents have been indexed and thus some terms' postings lists in the auxiliary index have grown nearly about to reach the pulsing threshold. For example, the document frequency of "`a`" has grown to 497, "`of`" to 498, and "`the`" to 499. For an incoming document which contains "`a`", "`of`", and "`the`", the pulsing method directly updates the postings lists of auxiliary index for "`a`" and "`of`" since their current document frequencies are 498 and 499 respectively. On the other hand, for the term "`the`", its document frequency has just reached the threshold of 500, then the postings list of the main index is in-placed updated and the postings list in the auxiliary index is cleared to be empty. This means that processing of terms with high document frequencies are limited to be smaller than the threshold in the auxiliary index while there is overhead to in-place update to the main index. In addition, since high frequency terms are not exactly co-occur in a document, in-place updates to the main index is dispersed throughout insertion tasks of new documents.

As any auxiliary postings list longer than a given threshold is in-place updated to the main index, the size of pulsing auxiliary index will be nearly constant regardless of update numbers. This way, there is no postings list longer than the given threshold, and updating of postings lists for all unique terms in a document can be very small compared with the direct update and the stand-alone auxiliary index strategies. Therefore, inserting a document will have hundreds of unique terms to be processed but the transaction consists of hundreds of updates of *small* postings lists in the auxiliary index compared with direct updates or stand-alone auxiliary.

@DOCUMENT (1296)
**#TITLE=**Regression with Doubly Censored Current Status Data
**#AUTHOR=**Rabinowitz, Daniel ; Jewell, Nicholas P.
**#JOURNAL=**Journal of the Royal Statistical Society. Series B (Methodological)
**#VOLUME=**58
**#NUMBER=**3
**#PAGE_START=**541
**#PAGE_END=**550
**#PUBDATE=**20010324
**#ABSTRACT=**Data from settings in which an initiating event and a subsequent event occur in sequence are called doubly censored current status data if the time of neither event is observed directly, but instead it is determined at a random monitoring time whether either the initiating or subsequent event has yet occurred. This paper is concerned with using doubly censored current status data to estimate the regression coefficient in an accelerated failure time model for the length of time between the initiating event and the subsequent event. Motivated by a problem in the epidemiology of acquired immune deficiency syndrome, attention here is focused on a special case, the case in which the initiating event, given that it has occurred before the monitoring time, may be assumed to follow a uniform distribution. The main result is that the likelihood in the special case has the same structure as the likelihood in a simpler setting, the setting in which the time of the initiating event is known. The result allows methods developed for the simpler setting to be applied in the special case. The results of the application of the approach to real data are reported.
**#KEYWORDS=**Accelerated Failure Time ; Acquired Immune Deficiency Syndrome ; Current Status Data ; Double Censoring ; Survival Analysis

**Figure 3. A Sample Data in Semi-formatted Format**

## 4. Experiments

Experiments were performed on a dual Pentium Xeon 3GHz machine with 8GB of memory and RAID-5 SCSI storage while the machine in under light load where there is no other processes using disk, memory, or CPUs significantly.

**Table 1. Brief database schema and statistics of index terms per document**

| Section | Data Type | Index Type | Unique Terms | Total Terms |
|---|---|---|---|---|
| Title | string | by token | 11.2 | 11.9 |
| Author | string | by token | 3.0 | 3.0 |
| Journal | string | by token | 5.0 | 3.4 |
| Volume | integer | no index | - | - |
| Number | integer | no index | - | - |
| Page_Start | integer | no index | - | - |
| Page_End | integer | no index | - | - |
| PubDate | char[8] | index as is | 1.0 | 1.0 |
| Abstract | string | by token | 85.3 | 143.0 |
| Keywords | string | by token | 2.5 | 2.4 |
| | | SUM | 107.9 | 166.8 |

We prepared three sets of text collections from 1 million and 10 thousands of bibliographic data for journal and proceeding articles (Figure 3). A set of the first 10 thousand items is referred to as `10K` set (12MB in ASCII texts), the first 100 thousand items to `100K` set (121MB in ASCII texts), and the first million items to `1M` set (1,310MB in ASCII texts). Each set was bulk-loaded and updated with the rest 10 thousand items (12.2MB of ASCII texts) for three update strategies (Experiments of direct update for `100K` set and `1M` set were omitted due to prohibitive time constraint). The different collection sizes were chosen to explore the maintenance cost of the three strategies with different amount of data. The sizes of the final indexes which contains full location information for all terms in `10K`, `100K`, and `1M` data sets were 29MB, 246MB and 2,373MB, respectively. For the pulsing auxiliary strategy, document frequency of 500 was chosen for all experiments, since it balances the trade-off between query processing and index maintenance (data not shown).

Figure 3 shows the 1,296th example item which consists of @DOCUMENT(*record delimiter*), TITLE, AUTHOR, JOURNAL, VOLUME, NUMBER, PAGE_START, PAGE_END, PUBDATE(*publish date*), ABSTRACT and KEYWORDS. Table 1 shows average term counts in every section in the text collection (a section is the equivalent to a field or column in database world). Remind that to keep the lexicographical order of the terms in an inverted file, usually a vocabulary data structure is separated from the postings list, and in the `KRISTAL-IRMS` postings lists are further separated physically into two parts, postings of document identifiers with term frequencies and exact locations in the documents. Now refer to Table 1 where a document contains more than one hundred unique terms, which means that addition of a document should access the on-disk index more than three hundred times since KRISTAL should access three disk data structures – the vocabulary list, posting list and location list – for each unique term from the input document.

As shown in Table 1, KRISTAL-IRMS supports data types such as string (var_char equivalent but with length limit of 2GB (Usually DB has 4K limit); KSTRING in KRISTAL nomenclature), character array (KCHAR[]), integer (KINT), and float (KFLOAT), of which index type can be any one among INDEX_AS_IS (whole section value as an indexing unit), INDEX_NUMERIC (indexing section value as a floating point), INDEX_BY_CHAR (char level indexing), INDEX_BY_TOKEN (token level indexing), several more indexing types for Korean and Chinese characters, and a few other types related to biological sequences such as DNA and proteins.

In the auxiliary strategies, the non-contiguous nature of postings lists, being separated in the main index and auxiliary index, may result in deteriorated postings access, compared with contiguous posting lists. To test the trade-off between query processing and index maintenance of the auxiliary strategies, three sets of terms with different document frequency ranges – 27,920 terms with ($100 <= DF < 1000$), 7,071 terms with ($1000 <= DF < 10,000$), and 1,427 terms with ($DF >= 10,000$) – were prepared from 1M database. Queries against TITLE, ABSTRACT and KEYWORDS sections were formed with these terms – (TITLE : *term*) OR (ABSTRACT : *term*) OR (KEYWORDS : *term*)[2] in KRISTAL query formula. To assess trade-off between query evaluation and non-contiguous postings lists of auxiliary indexes, we compared times to process theses queries in Boolean model against a 1M+10,000 documents database of which index was constructed by *re-build* method [23], 1M database with 10,000 additional documents indexed with the stand-alone auxiliary index and 1M with 10,000 documents indexed with the pulsing auxiliary strategy.

Finally, against 10K, 100K and 1M sets we evaluated 2,994 complex queries, formulated by experts and currently used as subject queries in a bibliography retrieval service[3]. The target sections for these complex queries were same as the above experiment – "(TITLE : *complex_query*) OR (ABSTRACT : *complex_query*) OR (KEYWORDS : *complex_query*)" in KRISTAL query formula. Several examples of the 2,994 complex queries are shown below.

- yellow* /N8 (polyurethane* OR urethane*)
- silicon AND (optic* /N8 signal*) AND module*
- food* /N3 (wastewater* OR (waste /W1 water*)) AND treat*
- ceramic* AND (bulletproof* OR (bullet /W1 proof*) OR (bullet /W1 resist*) OR (bullet* /N2 (protect* OR resist*)))
- wood* /N5 (substitut* OR replacement*)
- (catalyst* OR catalyzer*) /N5 (regenerat* OR ((precious OR valu* OR noble*) /N2 metal* /N5 recover*))

Since NEAR and WITHIN operations – in the form of /N$n$ and /W$n$, respectively – were heavily used in addition to AND and OR Boolean operators, both postings lists and locations lists are used to evaluate each complex query (remind that KRISTAL separates locations lists from postings lists). Furthermore, from the frequent prefix wild-carding, the efficiency of accessing the vocabulary structure also can be reflected in evaluating complex queries. Complex queries were applied to 10K, 100K and 1M databases with additional 10,000 documents inserted with re-build, stand-alone auxiliary and pulsing auxiliary strategies.

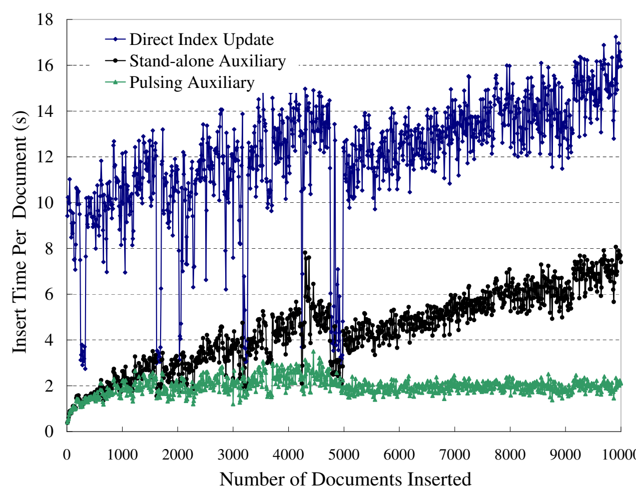



**Figure 4. Insert times averaged over every 10 of 10,000 input documents for 10K set**

## 5. Results

The results for three index update strategies are shown in Figure 4, 5 and 6. Each point in the figures was obtained by averaging insertion times of every 10 insertions, to reduce the effects of the biases in document lengths. All experiments are tested while the machine is under light load, that is, no other significant tasks are accessing the disk, memory, or CPUs.

Figure 4 shows the results of pouring 10 thousand documents to the 10K database where 10 thousand of documents were indexed in bulkload mode prior to the experiment. As mentioned in Section 3.1, Figure 4 shows directly updating new documents' index information to the

[2] The colon serves to distinguish the target section name from the query term or terms.

[3] YesKISTI at http://www.yeskisti.net

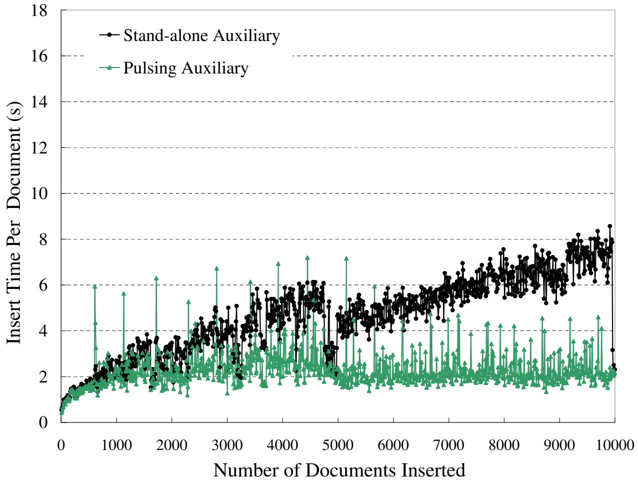



**Figure 5. Insert times averaged over every 10 of 10,000 input documents for `100K` set (Direct update test was omitted due to time constraints)**

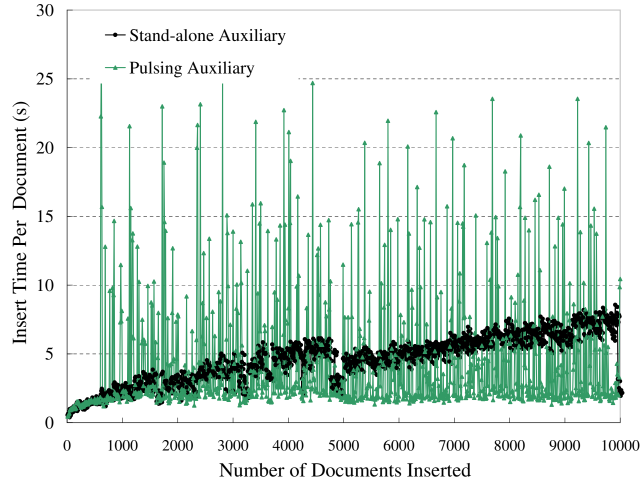



**Figure 6. Insert times averaged over every 10 of 10,000 input documents for `1M` set (Direct update test was omitted due to time constraints)**

main index is very poor compared with auxiliary index approaches (Averaged over 10,000 insertion transactions to `10K` database, direct index update, stand-alone auxiliary index, and pulsing auxiliary index strategy show processing speeds of 12.0, 4.37 and 1.97 seconds per document, respectively). This means direct update of main index is not feasible for even in very small text databases. On the other hand, stand-alone auxiliary index can improve the update efficiency but slows down more and more documents are inserted. The pulsing auxiliary index, a hybrid strategy of the stand-alone auxiliary index for short postings lists and direct update for long postings lists, gives even better performance than the stand-alone auxiliary index. Furthermore this strategy shows nearly stable performance though many inserts are done. Therefore index optimization (bulk-mode in-place update of the auxiliary postings lists to main index) is inevitable for the stand-alone auxiliary index but not necessarily for the pulsing auxiliary index.

As incoming documents are accumulated, the postings lists in the auxiliary index grow longer. For stand-alone auxiliary, there is no limit in the length of a postings list. On the other hand, any posting list in the pulsing auxiliary does not exceed the length of pulsing threshold. Excluding the pulsing postings lists, every postings list to process in the pulsing auxiliary is smaller than (or equal to) that in the stand-alone auxiliary. For example, assume that the term "`the`" occurs in all upcoming documents and the pulsing threshold is the document frequency of 500. For the 10,137th new document, postings list of the term "`the`" in the stand-alone auxiliary is 10,137 times of posting data structure, while only 137 times in the pulsing auxiliary. In the worst case, where the same document is repeatedly inserted $n$ times, the size of total postings lists to treat in the stand-alone auxiliary is at least $\frac{n}{\text{pulsing threshold}}$ times larger than that of the pulsing auxiliary resulting in the performance differences in Figure 4.

In Figure 5, the experiments were done with same condition but with `100K` database omitting direct update strategy due to prohibitive time constraint of more than 3 minutes per document. Averaged over 10,000 insertion transactions to `100K` database, the stand-alone and pulsing auxiliary indexes show the processing speeds of 4.62 and 2.26 seconds per document, respectively. In this experiment for `100K` data set, stand-alone and pulsing auxiliary indexes show very similar behavior to those seen for `10K` data set (Figure 4). In Figure 5, stand-alone auxiliary strategy's performance degraded as addition of new documents is accumulated, while pulsing auxiliary method has stable tendency but with frequent biased points that have not been observed in Figure 4. Except these biased points, pulsing method is superior to stand-alone method and its performance is stable regardless of accumulation of new documents.

Figure 6 shows result on `1M` data set. Direct index update is omitted due to time constraint, too. Averaged over 10,000 insertion transactions to `1M` database, the stand-alone auxiliary index and pulsing auxiliary index show the insertion speed of 4.71 and 4.38 seconds per document, respectively. As we have seen the performance biases of pulsing method for `100K` data set (Figure 5), even broader performance biases of the pulsing method are observed in the processing time plot for `1M` database (Figure 6). In pulsing scheme, once a term's document frequency reaching the pulsing

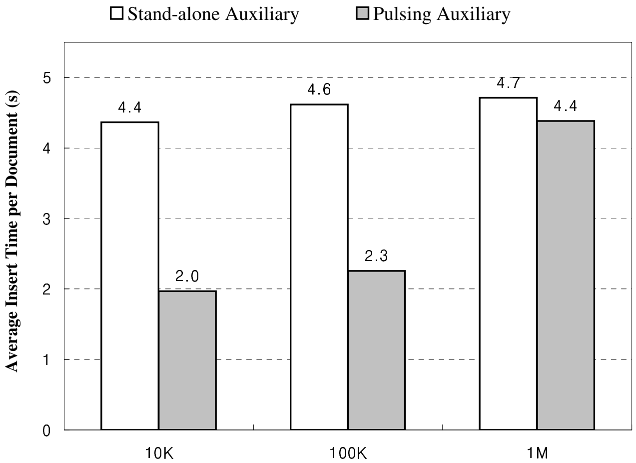
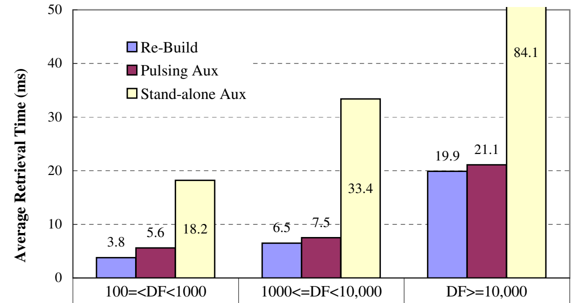



**Figure 7. Comparison of average insert times per document for updating 10,000 new documents to `10K`, `100K` and `1M` sets with the stand-alone auxiliary index and pulsing auxiliary index**

threshold, its postings list is appended to the corresponding posting list in the main index. However the `1M`'s main index size is 100 times larger than `10K` and 10 times larger than `100K`. In general, it means a postings list is 100 times or 10 times longer in average. A postings list in the main index can reach tens to hundreds of mega-bytes in length. Appending auxiliary postings list will apparently result in relocations in the main index and logging for such large transaction takes much time. Hence bigger biases are seen as the main index gets larger.

The experimental results shown in Figure 4, 5, and 6 are summarized again in Figure 7. For all three data sets, the pulsing auxiliary outperforms the stand-alone auxiliary strategy. The average performances in stand-alone auxiliary index are nearly constant regardless of main index size – it is natural since the stand-alone is truly independent of main index; on the other hand the pulsing strategy, of which performance is related to the main index size due to pulsing, is deteriorates as the main index size grows. Thus it is expected, unfortunately, that in a very large document table with more than 1 million bibliographic data, performance of the pulsing auxiliary might be worse than the stand-alone auxiliary index (This situation can be avoided if the document collection is divided into smaller tables with less than 1 million records).

Figure 8 shows `1M` postings access patterns of low frequency terms ($100 <= DF < 1000$), middle frequency terms ($1000 <= DF < 10,000$), and high frequency terms ($DF >= 10,000$) for re-build strategy as base line, pulsing auxiliary and stand-alone auxiliary strategies. For




| | 100=<DF<1000 | 1000<=DF<10,000 | DF>=10,000 |
|---|---|---|---|
| Re-Build | 3.8 | 6.5 | 19.9 |
| Pulsing Aux | 5.6 | 7.5 | 21.1 |
| Stand-alone Aux | 18.2 | 33.4 | 84.1 |

**Figure 8. Average query processing times of terms with varying DF ranges after adding 10,000 new documents to the `1M` set**

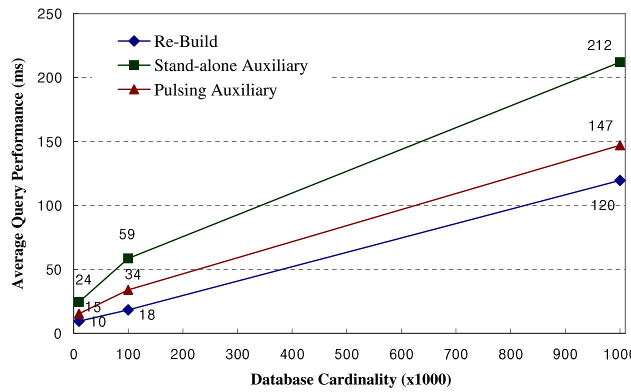



**Figure 9. Average query processing time for complex queries after insertion of 10,000 new documents to `10K`, `100K` and `1M` sets**

all cases, the stand-alone auxiliary strategy is significantly inferior to the re-built database while the pulsing strategy is close to the re-rebuilt (Stand-alone auxiliary strategy is 2.6–3.7 times slower than re-build while the performance of pulsing auxiliary strategy is deteriorated only 23% compared with the re-build).

As in our auxiliary strategies with non-contiguous nature of postings lists, index maintenance strategies inevitably contain a trade-off between index maintenance performance and query processing performance. Figure 8 shows the trade-off may be significant in the stand-alone auxiliary but not in the pulsing auxiliary index. This is verified again in the experiment for evaluating complex queries, depicted in Figure 9. For `1M` database, the query processing performance of the stand-alone and pulsing auxiliary index strategies are 56% and 81% of re-build strategy. Forcing the posting lists to be contiguous minimizes the the number of disk seeks necessary to fetch a posting list and thus maximizes the query processing performance. Re-build strategy guarantee this constraint. However, both in the stand-alone and pulsing auxiliary index strategies, the postings lists are separated into two parts, postings list in the main index and in the auxiliary index, thus more disk accesses are required than the re-build strategy resulting in deteriorated query performance (Figure 9). But, keeping the postings to be small, the pulsing auxiliary strategy has overhead of only 19% in query processing performance while the stand-alone has 44%.

## 6. Conclusion and Future Work

We have presented an evaluation of a few per-document basis transaction strategies for dynamic index maintenance. Our experimental evaluation shows that the pulsing auxiliary index strategy can be a competing candidate for text indexing schemes in true DB-IR integration, while it has also shows shortcomings of biased index processing times and slightly deteriorated query processing performance. Frankly to say, the efficiency of pulsing auxiliary is still unsatisfactory, compared those of a block of documents approaches. Buffering a block of new documents' index in memory and re-merging it to on-disk main index shows the update speed of 0.1 seconds per document for large text collections [24, 23], while our approach gives the best speed of 4.4 seconds per document in average for 1 million document collection.

The main problem of the pulsing auxiliary method of in-place update and auxiliary index is the costly relocations of long postings lists. As a previous work in information retrieval [9] designed a strategy to avoid costly relocations of long postings list, the performance of transactional index maintenance can also be greatly improved, if this costly operation can be avoided or relinquished. It also might be helpful to introduce an overallocation strategy for the auxiliary index. Inverted index compression algorithms [31, 2] will improve the performance further by reducing the lengths of postings lists.

Although `KRISTAL-IRMS` has been evolved from the different direction, but intending to the same point of DB-IR integration, many parts of our design principles in `KRISTAL` system architecture agree with "the storage-level core system with *RISC*-style functionality in DB-IR integration" argued in [11, 10]. `KRISTAL-IRMS`, adopting Berkeley DB as its storage engine[4], supports full-fledged concurrency control and recovery (even for both long text fields and their index information) and storage-level single-table queries. Since `KRISTAL` has aimed at an IRMS (an evolved form of IRS – See Figure 1), it supports text abstract data type at the storage level in addition to various structured data types. `KRISTAL`'s query layer is built on top of the storage layer extending storage-level single-table queries to multi-tables or multi-schemas. In the query layer, `KRISTAL` supports ranked query evaluation, structured query evaluation, group by, sorting and efficient XML retrieval, while leaving some DB features, such as SQL-like query language, table join, materialized view, ranking on structured values, trigger and query optimizer, as problems to be solved.

We live in an extremely changing time where all design assumptions for any system are being re-evaluated. Information retrieval and database researchers have eschewed each other for a long time, residing in the fixed and separated worlds of IR and DB. However our clients have started to ask table joins and triggers in IR and unstructured text processing and ranked query evaluation in DB.

[4] At the beginning, `KRISTAL` used its own storage layer customized to IR features only. However in-filed experiences have driven us to consider new storage engine with data (index) management feature. Rather extending our own storage layer, we chose open-source but reliable BerkeleyDB as our storage engine to support concurrency control and recovery.


## Acknowledgments

We thank Mr./Mrs. ... for proofreading this manuscript. The first author especially thanks Jieun and Changmin for supporting this work.


## References


[1] S. Amer-Yahia, P. Case, T. Roelleke, J. Shanmugasundaram, and G. Weikum. Report on the DB/IR panel at SIGMOD 2005. *SIGMOD Record*, 34(4):71–74, December 2005.

[2] V. N. Anh and A. Moffat. Improved word-aligned binary compression for text indexing. *IEEE Transactions on Knowledge and Data Engineering*, 18(6):857–861, 2006.

[3] R. Baeza-Yates, Y. S. Maarek, T. Roelleke, and A. P. de Vries. Third edition of the "XML and information retrieval" workshop. First workshop on integration of IR and DB (WIRD) jointly held at SIGIR'2004, Sheffield, UK, July 29th, 2004. *SIGIR Forum*, 38(2):24–30, 2004.

[4] D. Barbará, C. Clifton, F. Douglis, H. Garcia-Molina, S. Johnson, B. Kao, S. Mehrotra, J. Tellefsen, and R. Walsh. The Gold Mailer. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*, pages 92–99, 1993.

[5] D. Barbará, S. Mehrotra, and P. Vallabhaneni. The Gold Text Indexing Engine. In *ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering*, pages 172–179, 1996.

[6] H. Bast and I. Weber. The CompleteSearch Engine: Interactive, efficient, and towards IR&DB integration. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 88–95, 2007.

[7] A. Bialecki, D. Cutting, S. Ganyo, C. Goller, O. Gospodnetic, M. Harwood, E. Hatcher, D. Naber, Y. Seeley, and S. Siren. The Apache Lucene Project. http://lucene.apache.org/.

[8] E. W. Brown, J. P. Callan, and W. B. Croft. Fast incremental indexing for full-text information retrieval. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 192–202, 1994.

[9] S. Büttcher, C. L. A. Clarke, and B. Lushman. Hybrid index maintenance for growing text collections. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 356–363, 2006.

[10] S. Chaudhuri, R. Ramakrishnan, and G. Weikum. Integrating DB and IR technologies: What is the sound of one hand clapping? In *Proceedings of the Second Biennial Conference on Innovative Data Systems Research (CIDR 05)*, pages 1–12, Asilomar, CA, USA, 2005.

[11] S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning RISC-style database system. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 1–10, 2000.

[12] T. Chiueh and L. Huang. Efficient real-time index updates in text retrieval systems. Technical Report ECSL-TR-66, Computer Science Department, SUNY at Stony Brook, 1999.

[13] M. P. Consens and R. A. Baeza-Yates. Database and information retrieval techniques for XML. In *Advances in Computer Science - ASIAN 2005, Data Management on the Web, 10th Asian Computing Science Conference, Kunming, China, December 7-9, 2005, Proceedings*, volume 3818, pages 22–27, 2005.

[14] D. R. Cutting and J. O. Pedersen. Optimization for dynamic inverted index maintenance. In *SIGIR '90: Proceedings of the 13th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 405–411, 1990.

[15] A. P. de Vries and A. N. Wilschut. On the integration of IR and Databases. In *Database issues in multimedia; short paper proceedings, 8th International IFIP 2.6 Working Conference on Database Semantics (DS-8), Rotorua, New Zealand*, volume 138, pages 16–31, 1999.

[16] W. B. Frakes and R. A. Baeza-Yates. *Information Retrieval: Data Structures & Algorithms*. Prentice-Hall, 1992.

[17] GIIS. KRISTAL-IRMS: Information Retrieval & Managment System. http://www.kristalinfo.com/.

[18] J. Gray. The next database revolution. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 1–4, 2004.

[19] L. Guo, J. Shanmugasundaram, K. Beyer, and E. Shekita. Efficient inverted lists and query algorithms for Structured Value Ranking in update-intensive relational databases. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 298–309, 2005.

[20] S. Héman, M. Zukowski, A. P. de Vries, and P. A. Boncz. Efficient and flexible information retrieval using MonetDB/X100. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 96–101, 2007.

[21] N. Kabra, R. Ramakrishnan, and V. Ercegovac. The QUIQ engine: A hybrid IR-DB system. In *Proceedings of the 19th International Conference on Data Engineering (ICDE), March 5-8, 2003, Bangalore, India*, pages 741–, 2003.

[22] D. Knaus and P. Schäuble. The system architecture and the transaction concept of the SPIDER information retrieval system. *IEEE Data Eng. Bull.*, 19(1):43–52, 1996.

[23] N. Lester, J. Zobel, and H. Williams. Efficient online index maintenance for contiguous inverted lists. *Inf. Process. Manage.*, 42(4):916–933, 2006.

[24] N. Lester, J. Zobel, and H. E. Williams. In-Place versus Re-Build versus Re-Merge: Index Maintenance Strategies for Text Retrieval Systems. In *Computer Science 2004, Twenty-Seveth Australasian Computer Science Conference (ACSC2004), Dunedin, New Zealand, January 2004*, pages 15–22, 2004.

[25] L. Lim, M. Wang, S. Padmanabhan, J. S. Vitter, and R. Agarwal. Dynamic maintenance of web indexes using landmarks. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 102–111, 2003.

[26] W.-Y. Shieh and C.-P. Chung. A statistics-based approach to incrementally update inverted files. *Inf. Process. Manage.*, 41(2):275–288, 2005.

[27] K. Shoens, A. Tomasic, and H. Garcia-Molina. Synthetic workload performance analysis of incremental updates. In *SIGIR '94: Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 329–338, 1994.

[28] M. Theobald, R. Schenkel, and G. Weikum. An efficient and versatile query engine for TopX search. In *VLDB '05: Proceedings of the 31st International Conference on Very Large Data Bases*, pages 625–636, 2005.

[29] A. Tomasic, H. Garcia-Molina, and K. A. Shoens. Incremental updates of inverted lists for text document retrieval. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, pages 289–300, 1994.

[30] K.-Y. Whang, M.-J. Lee, J.-G. Lee, M.-S. Kim, and W.-S. Han. Odysseus: A high-performance ORDBMS tightly-coupled with IR features. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 1104–1005, 2005.

[31] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[32] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.

[33] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Trans. Database Systems*, 23(4):453–490, 1998.