World Wide Web (2015) 18:491–520
DOI 10.1007/s11280-013-0264-y

DB-IR integration using tight-coupling in the Odysseus DBMS

Kyu-Young Whang · Jae-Gil Lee · Min-Jae Lee · Wook-Shin Han · Min-Soo Kim · Jun-Sung Kim

Received: 18 January 2013 / Revised: 15 July 2013 /
Accepted: 8 October 2013 / Published online: 15 December 2013
© Springer Science+Business Media New York 2013

Abstract As many recent applications require integration of structured data and text data, unifying database (DB) and information retrieval (IR) technologies has become one of major challenges in our field. There have been active discussions on the system architecture for DB-IR integration, but a clear agreement has not been reached yet. Along this direction, we have advocated the use of the tight-coupling architecture and developed a novel structure of the IR index as well as tightly-coupled query processing algorithms. In *tight-coupling*, the text data type is supported from the storage system just like a built-in data type so that the query processor can efficiently

K.-Y. Whang (✉) · M.-J. Lee · J.-S. Kim
Department of Computer Science, Korea Advanced Institute of Science and Technology (KAIST), 291 Daehak-ro, Yuseong-gu, Daejeon 305-701, Korea
e-mail: kywhang@cs.kaist.ac.kr

M.-J. Lee
e-mail: mjlee@mozart.kaist.ac.kr

J.-S. Kim
e-mail: jskim@mozart.kaist.ac.kr

J.-G. Lee
Department of Knowledge Service Engineering, Korea Advanced Institute of Science and Technology (KAIST), 291 Daehak-ro, Yuseong-gu, Daejeon 305-701, Korea
e-mail: jaegil@kaist.ac.kr

W.-S. Han
Department of Creative IT Engineering/Department of Computer Science and Engineering, Pohang University of Science and Technology (POSTECH), 77 Cheongam-ro, Nam-gu, Pohang-si, Gyeongbuk 790-784, Korea
e-mail: wshan@postech.ac.kr

M.-S. Kim
Department of Information and Communication Engineering, Daegu Gyeongbuk Institute of Science & Technology (DGIST), 333 Technojungang-daero, Hyeonpung-myeon, Dalseong-gun, Daegu 711-873, Korea
e-mail: mskim@dgist.ac.kr

handle queries involving both structured data and text data. In this paper, for archival purposes, we consolidate our achievements reported at *non-regular* publications over the last ten years or so, extending them by adding greater details on the IR index and the query processing algorithms. All the features in this paper are fully implemented in the Odysseus DBMS that has been under development at KAIST for over 23 years. We show that Odysseus significantly outperforms two open-source DBMSs and one open-source search engine (with some exceptional cases) in processing DB-IR integration queries. These results indeed demonstrate superiority of the tight-coupling architecture for DB-IR integration.

Keywords Tight-coupling · Information retrieval · DB-IR integration · Odysseus

1 DB-IR integration

Databases (DB) and information retrieval (IR) have been parallel universes. However, many recent applications such as customer support, health care, digital libraries, and advanced web search require integration of structured data and text data [6, 13, 34, 40]. Seamless integration of structured data and text data is of prime importance as discussed in the Lowell report [1] and the Claremont report [3]. Thus, unifying DB and IR technologies has become one of major challenges in our field.

The requirements for DB-IR integration have been actively discussed in the literature [6, 13, 34]. Among those requirements, we focus on the capability of efficiently executing the queries involving both structured data and text data. This capability forms the basis of DB-IR integration, and such queries can be easily found in many applications.

Example 1 Consider a query that finds papers about "cloud computing" published after "2005" in digital libraries. The former condition with "cloud computing" is evaluated on text data such as titles or abstracts. The latter condition with "2005" is evaluated on structured data such as the "publication_year" attribute of type integer.

For the requirement mentioned above, mainly four alternatives have been addressed in the literature [6, 13]: (i) IR on top of a relational DBMS (RDBMS), (ii) a middleware layer on top of an RDBMS and an IR system, (iii) IR supported via user-defined functions in RDBMSs, and (iv) IR on top of a *relational storage* engine. Among them, the first and second alternatives have not been widely adopted. One example of the first alternative is the SRAM system [16] implemented on top of MonetDB/X100. Commercial DBMS vendors are using the third alternative to add IR features into their DBMSs, e.g., Oracle Cartridge [30] and IBM DB2 Extender [26]. We call the third alternative as *loose-coupling*.

Chaudhuri et al. [13] argue that the fourth alternative would be the right architectural approach to explore. Nevertheless, they admit that, since a relational storage engine does not support the text data type, trying to use a B^+-tree implementation and a traditional relational storage layer without modification can result in very poor performance. Thus, design and implementation of a core storage-level DB-IR platform remains a major research challenge.

For a new architecture of DB-IR integration, we have proposed the *tight-coupling* architecture [36–38, 40, 41] and have contended that it is the most efficient approach. In tight-coupling, unlike the fourth alternative, the storage system of the DBMS

engine directly supports the text data type. In contrast, in loose-coupling, the text data type is supported from add-on packages running on top of the query processor. Due to this characteristic of tight-coupling, the text data type is treated just like a built-in data type. Hence, tight-coupling allows us to support the text data type as a "first-class citizen" [1] within the DBMS architecture.

The tight-coupling architecture has been used to incorporate IR and spatial database features into the Odysseus DBMS [38, 41],[1] which has been under development at KAIST for 23 years. The tight-coupling with the IR features makes Odysseus a DBMS and, at the same time, a search engine. Excellence of the tightly-coupled IR features[2] has been demonstrated through the parallel web search engine implemented using Odysseus, which is capable of managing 100 million web pages in a non-parallel configuration and should be able to support tens of billions of web pages in a parallel configuration [40, 42].[3]

Tight-coupling has many advantages for DB-IR integration over loose-coupling. The query processor can run sophisticated algorithms for the queries involving both structured data and text data. By taking advantage of tight-coupling, we earlier proposed two algorithms: (i) *IR index join with posting skipping* and (ii) *attribute embedding* [37–40], which are feasible only in tight-coupling. They are fully implemented into Odysseus to boost DB-IR integration queries.

In this paper, for archival purposes, we consolidate our achievements reported at *non-regular* publications such as a patent (2002) [36], a panel presentation (2003) [37], a demo (2005) [38], and a keynote (2007) [39]. This paper significantly extends them in two ways. First, we add greater details on the structure of the patented IR index and the two query processing algorithms. Second, we compare the performance of processing DB-IR integration queries with two widely-used open-source DBMSs and one open-source search engine. Odysseus is shown to outperform other systems by virtue of our tight-coupling architecture. The main contribution of this paper is addressing DB-IR integration in the *performance* point of view.

One might think that our achievements presented in this paper are not very novel *at this point of time*. Nevertheless, we claim that they were innovative at the time they were reported in these non-regular publications.

- To the best of our knowledge, the U.S. patent on DB-IR tight coupling (applied in 1999; granted in 2002) [36] is the first patent that addresses DB-IR integration. Furthermore, this patent first proposed an index structure created on each posting list of an inverted index, which we call the *subindex*. The concept of the subindex was novel since it was the first time the index was defined on an "instance" rather than on the "schema." As we discuss in Section 3.1, a posting list is considered a part of a tuple instance for a relation involving an attribute of the text type. This notion was used in later work such as the ZigZag join [23] in the context of XML query processing in 2003.
- The IR index join has been the standard method for processing multiple-keyword queries. An optimization technique, which we call *posting skipping*,

[1]The Odysseus DBMS consists of approximately 450,000 lines of C and C++ high precision codes.

[2]This work received the best demonstration award at IEEE ICDE 2005 [38].

[3]The sister paper [42] recently published at ACM SIGMOD discusses application of the Odysseus DBMS to a massively-parallel search engine architecture.

allows us to skip unqualified postings in the process of the IR index join. Although posting skipping was first reported in the demo paper [38] in 2005, it had been fully implemented in 2001. Posting skipping in the IR index join employed the key idea of the subindex, which had been proposed in the U.S. patent [36] granted in 2002 (applied even earlier in 1999).

– Attribute embedding is to insert any attribute values into postings for use in query processing. One may find that attribute embedding sounds very similar to the *payload* implemented in Lucene. However, the payload supports only a placeholder for an attribute value, and the method of processing the queries using the value of the payload is yet to be implemented in Lucene. Attribute embedding was fully implemented in 2001 and was first reported at the panel [37] in March 2003. In contrast, the payload of Lucene is still under active development [29].

The rest of this paper is organized as follows. Section 2 introduces Odysseus and its tight-coupling architecture. Section 3 presents the tightly-coupled IR index and algorithms in Odysseus. Section 4 discusses the architecture of other DBMSs with regard to DB-IR integration. Section 5 shows the results of performance evaluation. Finally, Section 6 concludes the paper.

2 Background

2.1 Overview of Odysseus

Figure 1 shows the system architecture of Odysseus. Odysseus consists of a storage system (Odysseus/COSMOS) and a query processor (Odysseus/OOSQL). Odysseus/COSMOS is a sub-system that stores and manages objects in the database. Most important, Odysseus/COSMOS contains the *extensible type layer* for tight-coupling. Thus, IR and spatial database operations are processed at the level of the storage system. Disk Manager controls raw disks independent of O/S files. Small Object Manager manages objects smaller than one page, and Large Object Manager manages those larger than one page and up to 2^{63} bytes. Index Manager manages the B+-tree index and Multilevel Grid File (MLGF) spatial index. Cursor Manager processes sequential and index scan operations. Recovery Manager manages recovery functions, and Transaction Manager concurrency control functions and transactions. Odysseus/OOSQL is a sub-system for processing SQL queries. Query Analyzer analyzes given SQL queries. Query Plan Generator and Optimizer generate and optimize query plans. Query Plan Executor executes optimized query plans to return query results using Odysseus/COSMOS.

The Odysseus DBMS supports most features of the SQL3 standard. Its 64-bit architecture allows large-scale databases up to 32 ZBytes (10^{21}) per table and up to 8 EBytes (10^{18}) per record. It supports fast bulk loading and bulk deletion. Concurrency control and recovery can be done in fine or coarse granularity.

The Odysseus DBMS is tightly-coupled with IR and spatial database features. It provides extensions of an SQL query language so as to offer users tightly-coupled IR and spatial database features. The inverted index [36] is integrated for indexing IR contents; the Multi-Level Grid File (MLGF) [35] for indexing spatial contents. Due to the tight-coupling architecture, concurrency control and recovery on IR and spatial contents can be done in fine as well as coarse granularities. Odysseus provides fast immediate update capability on IR contents. That is, there is no need to halt

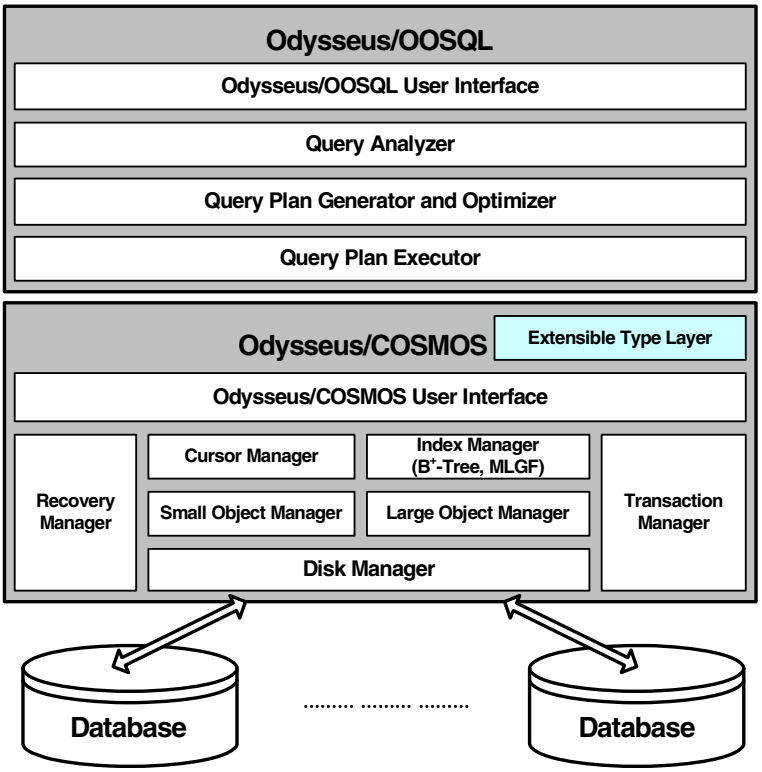

Figure 1 The architecture of the Odysseus DBMS

the system to perform updates on IR contents. The tightly-coupled IR features are implemented within the extensible type layer of Figure 1. We elaborate on those features in Section 3.

2.2 Tight-coupling architecture

Here, we define the tight-coupling architecture by using the concept of the *extensible type layer* [41], which is the layer that provides facilities required for using new data types (i.e., non built-in types). That is, the extensible type layer provides new data types, operations, and indexes.

We now define *tight-coupling* and *loose-coupling* based on the location of the extensible type layer. *Tight-coupling* is a mechanism of supporting new data types by locating the extensible type layer *inside the storage system*; *loose-coupling* by locating the extensible type layer *on top of the query processor*. Figure 2 contrasts tight-coupling with loose-coupling. In loose-coupling, a Cartridge or Extender in a commercial DBMS corresponds to the extensible type layer. MySQL has also adopted tight-coupling, while PostgreSQL a slight variation of loose-coupling. The architectures of MySQL and PostgreSQL will be explained later in Section 4.

As shown in Figure 2, in loose-coupling, the extensible type layer is isolated from the DBMS server in order to protect the DBMS server from errors occurring in

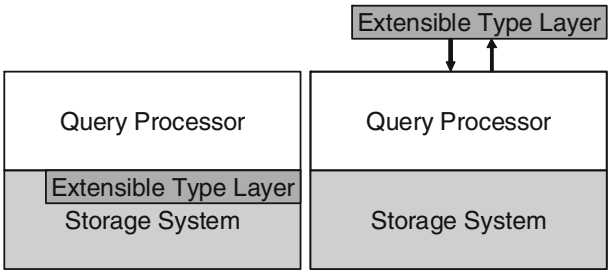

Figure 2 Comparison between tight-coupling and loose-coupling [41]

the extensible type layer. Since the extensible type layer in loose-coupling may be implemented by ordinary users, safety of the extensible type layer is generally not guaranteed. That is, the extensible type layer may be terminated abnormally due to incomplete or inadvertent programming. To prevent this abnormal termination from influencing the DBMS server, the extensible type layer runs in a process separate from the DBMS server. In contrast, since the extensible type layer in tight-coupling is implemented by DBMS developers, we consider that safety is guaranteed.

Different programming interfaces are used to implement new data types in tight-coupling and loose-coupling. In tight-coupling, the storage system API is employed; in loose-coupling, typically, the SQL interface is employed.

In general, tight-coupling has many advantages over loose-coupling [36–38, 41]. First, performance of query processing is superior. Second, flexible concurrency control is possible on new data types. Third, implementable data types and operations are more flexible since the extensible type layer uses the storage system API. In Section 3, we focus on the advantages of tight-coupling for DB-IR integration. Highly efficient algorithms for DB-IR integration can be implemented to speed up DB-IR integration queries.

3 Tightly-coupled IR features

The Odysseus DBMS supports the *text* type for storing text data (e.g., web documents) and the IR index for performing keyword search on the text data. Users can use the text type and the IR index just like other built-in types and indexes when defining the database schema. Suppose we define a schema involving the text type, an IR index, the integer type, and a B^+-tree index. Figure 3 shows the physical structure of a data record conforming to the schema. As shown in the figure, the text type is treated just in the same way as the integer type is. Similarly, an IR index is treated in the same way as a B^+-tree index is.

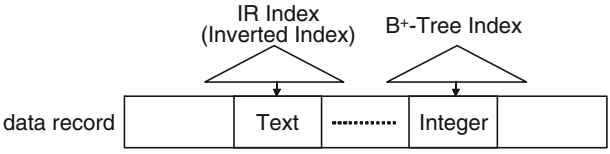

Figure 3 The structure of a data record involving the text type and an IR index

In this section, we discuss the tightly-coupled IR features of the Odysseus DBMS. Section 3.1 explains the structure of the IR index. Section 3.2 explains two algorithms for processing DB-IR integration queries. Section 3.3 introduces the site-limited search as an application of the tightly-coupled IR features.

3.1 The IR index

Overview of the IR index Figure 4 shows the structure of the IR index [36] implemented in Odysseus. It is analogous to a traditional inverted index [5] widely used for information retrieval. The inverted index of Odysseus consists of keywords and posting lists. A *posting list* exists for each keyword and consists of postings. A *posting* contains the document identifier (docId), the object identifier (OID), the number of occurrences, and the offsets in the document where the keyword appears. Here, docIds are logical identifiers, and OIDs are physical ones. Postings are maintained in the order sorted by the docId. This can be done easily by assigning the docId in the order of storing the documents in the database. Besides, a B^+-tree index is constructed on keywords in order to quickly find the posting list for a specific keyword.

The inverted index of Odysseus has two distinct features compared with traditional ones. These features have first been proposed in a patent [36] in 2002. First, it uses large objects [11] to store posting lists. We manage the storage space of the

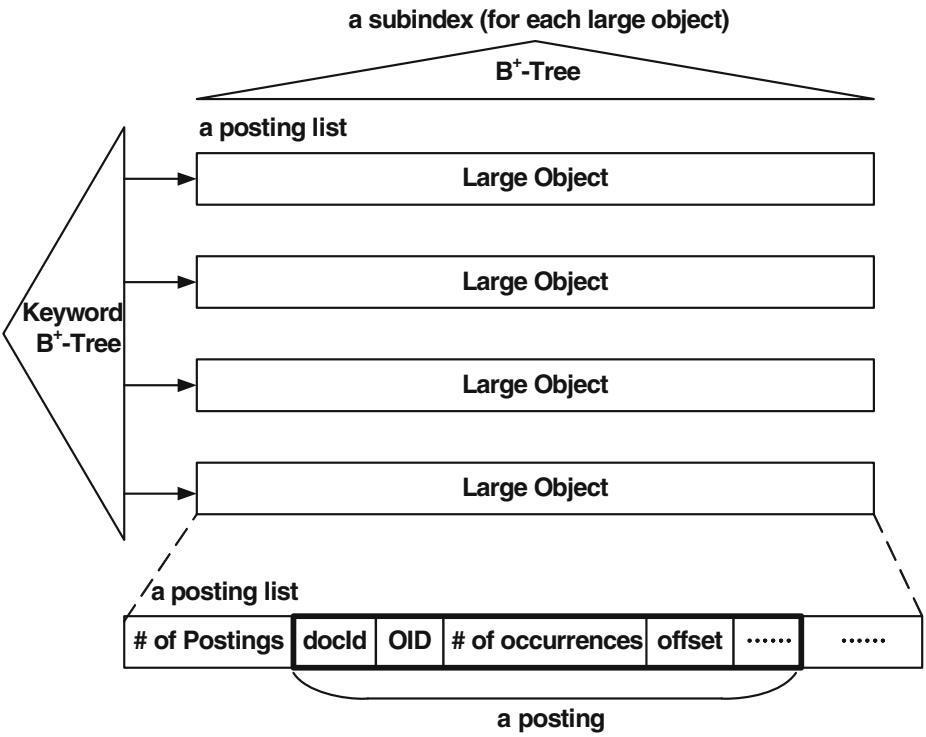

Figure 4 The IR index structure using large objects and subindexes

posting list by using the large object tree, which has been proposed by Biliris [11] for Exodus. The advantage of this method [36] is that a small amount of disk I/O's is required to insert a new posting into or remove it from a posting list without much sacrificing sequential search performance. That is, insertion or deletion of a posting does *not* require reading and writing the *entire* posting list unlike a BLOB. Second, it uses subindexes [36] to index postings in each posting list. The subindex is used for locating the posting with a given docId within a posting list. Using subindexes, we can quickly find the location of a new posting to be inserted or of an existing posting to be deleted or modified.

A *subindex* is a B^+-tree created on each large object that stores a posting list. Figure 5 shows the detailed implementation of the subindex. The key of a subindex is docId; a leaf node points to an entry of the offset array stored in the large object. An entry of the offset array stores the byte offset of a specific posting within the posting list. This offset array enables us to quickly insert or delete a posting. We note that, if a posting is inserted into or deleted from a posting list, the locations of the postings after that one are changed. Suppose that the offset array does not exist and that the leaf nodes of the subindex store the offsets of postings. Then, the leaf nodes of the subindex must be updated whenever the locations of postings are changed. In contrast, in Figure 5, a change of the locations affects only the offset array, but not the subindex. The leaf nodes of the subindex are not guaranteed to be physically contiguous while the entries of the offset array are. Hence, the cost of updating the latter is much cheaper than that of updating the former. This advantage becomes more prominent especially when insertion, deletion, and update are done in the bulk mode.

Currently, in information retrieval, it is very common that a skipping list of pointers to postings is used as an auxiliary structure [28]. The role of a skipping list is exactly the same as that of the subindex, which allows us to lookup a posting efficiently and skip unqualified postings.

Implementation of the IR index The IR index of Odysseus is implemented using tables in the extensible type layer. It consists of three tables as in Figure 6: the *inverted table*, *docId table*, and *content table*. We call these three tables as the *text metadata tables*. In contrast, we call a table containing text attributes as the *text data table*. Text metadata tables are created for each text attribute of a text data table. For example, for the `webpages` table in Figure 6, text metadata tables are

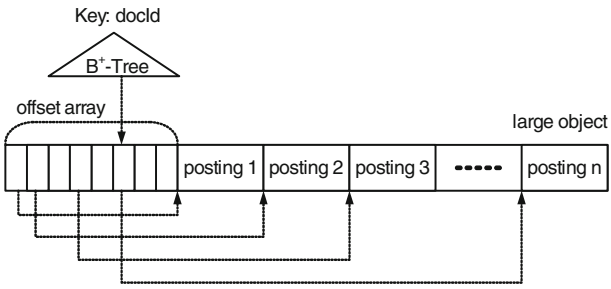

Figure 5 The detailed implementation of the subindex

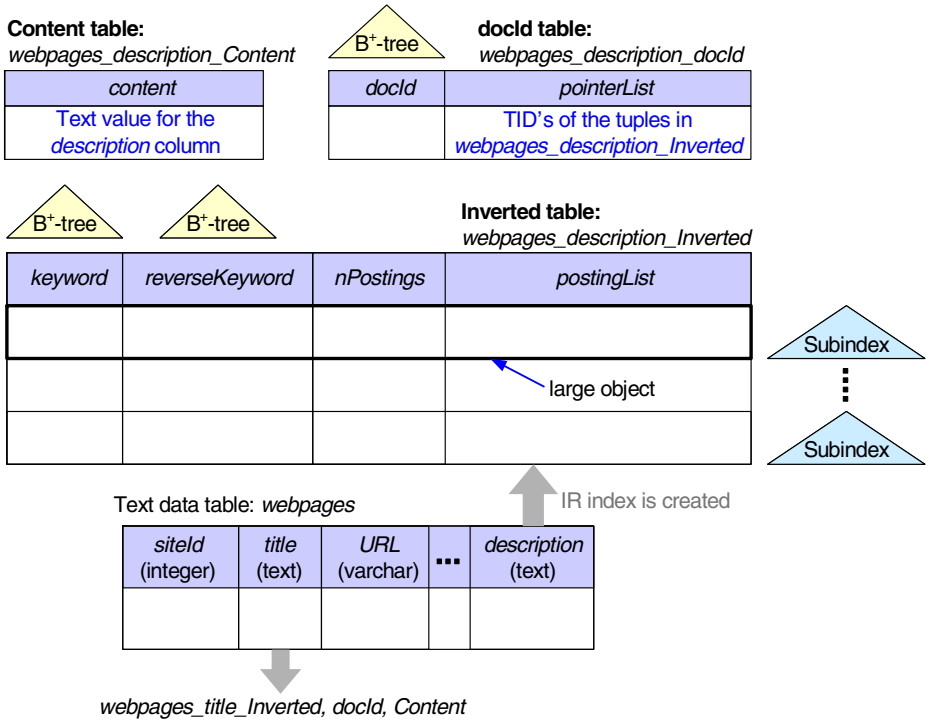

Figure 6 Implementation of the IR index using tables

created separately for the `title` attribute and the `description` attribute, where both attributes are of type text.[4]

The *inverted table* stores the inverted index in Figure 4 in a form of a table. The `keyword` attribute stores a keyword indexed, and the `reverseKeyword` attribute its reversed string; e.g., `reverseKeyword` is "tenretni" if `keyword` is "internet." `reverseKeyword` is useful for processing left-wildcard operators. For example, "*net" is processed by finding the tuples whose `reverseKeyword` contains "ten" as a prefix. This can be efficiently processed using the B$^+$-tree index created on `reverseKeyword`. We note that the values of `reverseKeyword` are not actually stored in the table since they can be derived from those of `keyword`. The `nPostings` attribute stores the number of postings in a posting list, and the `postingList` attribute the posting list itself. B$^+$-tree indexes are created on `keyword` and `reverseKeyword`, respectively. In addition, a subindex is created if the size of a large object exceeds a threshold, which is the size of data where an index scan begins to outperform a sequential scan.

[4]The names of the text metadata tables are constructed in such a way that the names of the text data table and the text attribute are prefixed, and then, `Inverted`, `docId`, or `Content` are appended.

The *docId table* points to the posting lists for the keywords extracted from each document. We use tuple identifiers (TIDs) of the tuples in the inverted table as pointers. A TID, a part of an OID, is a physical pointer of twelve bytes, and thus, allows us to directly access the corresponding tuple. The list of TIDs is stored in the `pointerList` attribute of type varchar. The docId table is mainly used for processing immediate update, more specifically deletion. Using the docId table, we can quickly locate the posting lists affected by a deletion of a document.

The *content table* stores the values of a text attribute. Since the size of these values tends to be very large, storing them in the text data table can drastically degrade clustering of tuples. Thus, we store these values in a separate table, i.e., the content table; then, the TIDs of tuples in the content table are stored in the text attribute of the text data table.

Customization of the IR index The Odysseus DBMS supports customization of the IR index, allowing us to add or remove components of the IR index. Due to this customization, we can adjust the size of the IR index to fit the requirement. For example, if proximity operators (finding the documents containing two keywords within n words) are not required, we do not need to store pairs ⟨sentence offset, word offset⟩ in a posting. This can be done easily by specifying an option when defining a schema. For another example, if left-wildcard operators are not required, we do not need to create the `reverseKeyword` attribute in the inverted table and the B^+-tree on that attribute. Besides, we can add or remove other components such as the subindex and the docId table according to our need. Experimental results indicate that the size of the compressed IR index, excluding the size of source data, can be varied in the range of 90 % $\sim$ 400 % of the size of source data through customization.

Compression of the IR index The Odysseus DBMS supports compression of the IR index. Since the size of the posting lists takes a large proportion (over 60 %) of the total size of the IR index, we compress primarily the posting lists of the IR index. Compression is performed by storing a d-gap instead of a docId in a posting. The *d-gap* [43] is defined as the difference between successive docId values. Hence, a docId can be obtained by summing up d-gaps. The important feature of the d-gap is that it can be stored using a smaller number of bits than the docId due to its smaller value. The average compression ratio of the IR index is approximately 60 %. Besides, query processing time is improved by approximately 20 % at cold start since the amount of disk I/O's is decreased due to a smaller size of the posting list; on the other hand, query processing time gets worse by approximately 5 % at warm start because of the decompression overhead.

3.2 IR algorithms for DB-IR integration queries

In this section, we present two tightly-coupled DB-IR algorithms: (1) IR index join with posting skipping and (2) attribute embedding. We then show how DB-IR integration queries in Example 1 benefit from these algorithms. Since they take advantage of our patented IR index, the superiority of our index structure can be proven by their performance. We note that users can easily use these features simply by issuing SQL queries with a proper schema definition. Insertion and deletion

(i.e., immediate update) are fully supported in Odysseus, but are discussed here only briefly because they are beyond the scope of the paper.

3.2.1 IR index join with posting skipping

Overview The IR index join technique has been a standard way of processing multiple-keyword queries in the IR area. Interestingly, this technique is also useful for queries involving both keyword and attribute conditions [37, 38]. For this purpose, an attribute value is also treated as a text and is stored in a newly added attribute of type text. For example, an integer value 2010 is stored as a text "2010." Then, the attribute condition is evaluated as a keyword condition on the text attribute. In this way, a mixture of keyword and attribute conditions can be represented using a set of purely keyword conditions.[5]

The performance of the IR index join can be significantly improved if the subindex explained in Section 3.1 is exploited. Using subindexes, the exact parts of posting lists that need to be merged—where the docId matches among multiple posting lists—can be identified. Performance is enhanced since needless parts of the posting lists are skipped. We call this optimization technique *posting skipping*.

Algorithm Figure 7 shows the algorithm of the IR index join. We present a binary join algorithm for ease of explanation. Binary joins can be extended to multi-way joins by sophisticated join ordering. This algorithm finds the pairs of postings that have the same docId between two posting lists. To accomplish this, the algorithm compares the docIds of two postings where the cursor is currently located in each posting list, and then, moves the cursor of the posting having the smaller docId to a posting having the docId greater than or equal to the docId of the current posting of the other posting list. We note that there are two methods of finding the posting to which the cursor is to be moved. If the difference in docId between the two posting lists exceeds a specified threshold, we find the posting by using the subindex, i.e., by performing *posting skipping* (Lines 6 and 11). Otherwise, we use sequential scan over the posting list (Lines 8 and 13).

The *threshold* for posting skipping is a tuning parameter. It indicates the maximum difference in docId's between two postings within a posting list where the cost of sequentially reading the postings between them is smaller than that of performing posting skipping.

Example 2 Figure 8 shows the process of joining two posting lists to find documents that contain the keyword "Hadoop" and that are published in the year 2010. Suppose that a text attribute is added for the integer attribute "year." The gray-colored parts of the posting lists are to be sequentially searched since these parts include the postings having the matching docId values. These gray-colored parts can be easily found using subindexes. We can skip to the posting whose docId is *doc672* in the posting list for "Hadoop" and to the posting whose docId is *doc154* in the posting list for "2010."

[5]We note that =, <, <=, >, and >= operators on numeric values can be translated to IR operators in Odysseus.

```
Algorithm IR_Index_Join
Input: Two posting lists plist1 and plist2
Output: Query results (pairs of postings with the same docId)
/* currPosting1 denotes the posting where the cursor is currently located in plist1 */
/* currPosting2 denotes the posting where the cursor is currently located in plist2 */
01  Set currPosting1 and currPosting2 to be the first posting in plist1 and plist2, respectively;
02  while (there remain postings in both plist1 and plist2) do
        /* posting.docId denotes the docId of the posting */
03      while (currPosting1.docId != currPosting2.docId) do
04          if (currPosting1.docId < currPosting2.docId) then
05              if (currPosting2.docId – currPosting1.docId > threshold) then
06                  Skip to the first posting where docId ≥ currPosting2.docId
                    in plist1 by using the subindex;   /* perform posting skipping */
07              else
08                  Scan sequentially to the first posting
                    where docId ≥ currPosting2.docId in plist1;
09          else if (currPosting1.docId > currPosting2.docId) then
10              if (currPosting1.docId – currPosting2.docId > threshold) then
11                  Skip to the first posting where docId ≥ currPosting1.docId
                    in plist2 by using the subindex;   /* perform posting skipping */
12              else
13                  Scan sequentially to the first posting
                    where docId ≥ currPosting1.docId in plist2;
14      Return the pair of currPosting1 and currPosting2 as the query result;
15      Advance currPosting1 and currPosting2 to the next posting, respectively;
```

Figure 7 The algorithm of the IR index join with posting skipping

Constructing a subindex on each posting list (a tuple in Odysseus) is a unique feature of Odysseus, and this structure has been U.S. patented in 2002 [36]. To take advantage of our unique structure, we had developed the IR index join [37] and then posting skipping. IR index join with posting skipping was fully implemented into Odysseus in 2001. Later in 2003, a few methods adopting the notion of the subindex were incidentally developed in the area of XML query processing. The methods proposed by Guo et al. [22] and Halverson et al. [23] store XML elements in the inverted index and create a B^+-tree index (corresponding to a subindex) on each posting list. Using the B^+-tree indexes, these methods avoid accessing unnecessary postings during query execution. Halverson et al. called this method the ZigZag join [23].

3.2.2 Attribute embedding

Overview We describe the *attribute embedding* technique—an alternative query processing technique for speeding up queries involving both keyword and attribute conditions. In this technique, the values of the attributes of documents are embedded

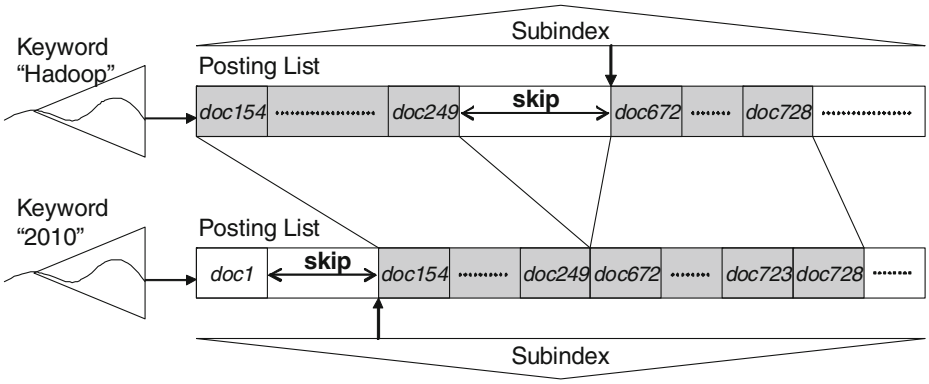

Figure 8 Processing a DB-IR integration query using the IR index join

into postings; i.e., they are stored in the posting structure shown in Figure 4. This technique is a kind of materialized join in the context of DB-IR integration. Using this technique, we are able to evaluate keyword and attribute conditions together by reading a single posting list, in which the values of the attributes of documents are embedded. Without attribute embedding, we have to access data records to obtain the values of the attributes. Accessing the data records is very expensive since it incurs random disk accesses over a large amount of storage space. Thus, attribute embedding significantly enhances the performance. On the other hand, this technique incurs additional storage requirement for embedded attributes.

Example 3 Figure 9 shows the process of finding documents that contain the keyword "Hadoop" and that are published in the year 2010. Since the values of the attribute "year" are embedded into the posting list for the keyword "Hadoop," the query processor can check whether each document was published in the year 2010 while reading only one posting list.

Specifying embedded attributes Users can specify embedded attributes when defining a table schema by using the option *embedded_attributes* for a text attribute. That is, attribute embedding is easily activated just by using a schema definition.

Example 4 The DDL statement below makes the attribute "year" of type integer embedded in the posting list for the attribute "description" of type text.

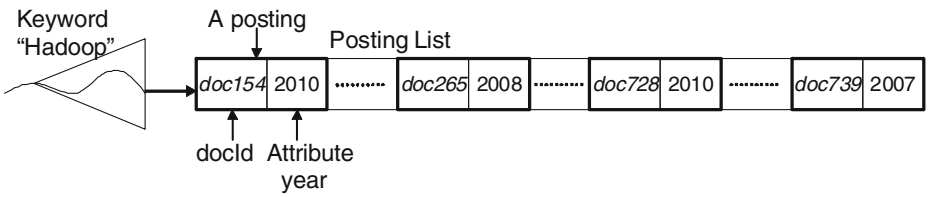

Figure 9 Processing a DB-IR integration query using attribute embedding

```
CREATE TABLE webpages ( year integer, ···
    description text(embedded_attributes(year))
    ··· )
```

Updating embedded attributes The values of embedded attributes are kept consistent with those of the original attributes of the document. That is, update of the latter triggers update of the former. This operation requires finding postings generated from the updated document. It can be easily processed by using the docId table and the subindex shown in Figure 6. Let us call the docId of the updated document $docId_{\text{updated}}$. We first find the posting lists that contain postings generated from the updated document. This can be done by searching for the tuple with $docId_{\text{updated}}$ for the `docId` attribute in the docId table. We then select the postings whose docId is $docId_{\text{updated}}$ from these posting lists by using subindexes.

3.2.3 Advantages of the tightly-coupled DB-IR algorithms

A primary advantage of the two tightly-coupled DB-IR algorithms with the DBMS is efficient processing of queries involving *both* keyword and attribute conditions, which inherently involves join. We note that, in the tight-coupling architecture, the query processor can employ the IR index join or attribute embedding to evaluate both keyword and attribute conditions together. In contrast, in the loose-coupling architecture, the add-on packages evaluate keyword conditions, and then, the query processors merge the results of the attribute conditions with those of the keyword conditions. Here, performance is degraded since a number of intermediate results that satisfy only the keyword or the attribute condition are retrieved. We contend that this advantage of the tight-coupling architecture is highly practical since DB-IR integration requires this kind of queries as discussed in Section 1.

The two algorithms are hard to be *efficiently* implemented in the loose-coupling architecture since loose-coupling typically employs the SQL interface. To efficiently implement them, the extensible type layer should have the ability to interpret a tuple as a list of postings. For example, to support posting skipping, we should be able to *selectively* access postings within a posting list (i.e., tuple). However, the SQL interface lacks this ability since it provides only tuple-level accesses.

3.3 An application: site-limited search

To show usability of the IR features tightly-coupled with the DBMS, we present implementation of site-limited search using the Odysseus DBMS. *Site-limited search* allows us to limit the scope of a query to the web pages collected from a specific site.[6] It is a very useful notion in that it can provide customized search service to individual users limiting the search within their own web pages obviating the need to install a separate web search engine. Site-limited search is a primary example of a DB-IR

[6]Google provides a similar service by allowing the users to specify the site URL together with the query keywords.

integration query since it involves *both* keyword and attribute conditions. Here, the query to the web pages constitutes the keyword condition, and the specific site constitutes the attribute condition. Figure 10 shows the interface for site-limited search in the web search engine implemented using Odysseus. A search window is displayed on the upper side of a web page. If users issue a query into that window, the scope of the query is limited to the site that the current web page being displayed belongs to.

We first define the database schema required for implementing site-limited search. The schema consists of two tables as in Figure 11: the `websites` and `webpages` tables. The `websites` table stores the information of sites, and the `webpages` table that of web pages. The attribute `webpages.siteId` represents the site from which the web page has been collected. This site identifier is stored also in `webpages.siteIdText` of type text. Besides, the attribute `webpages.description` stores the content of a web page after removing HTML tags. The keyword condition is resolved against this attribute.

We offer two methods for implementing site-limited search. These methods utilize the IR index join and attribute embedding explained in Section 3.2, respectively. The

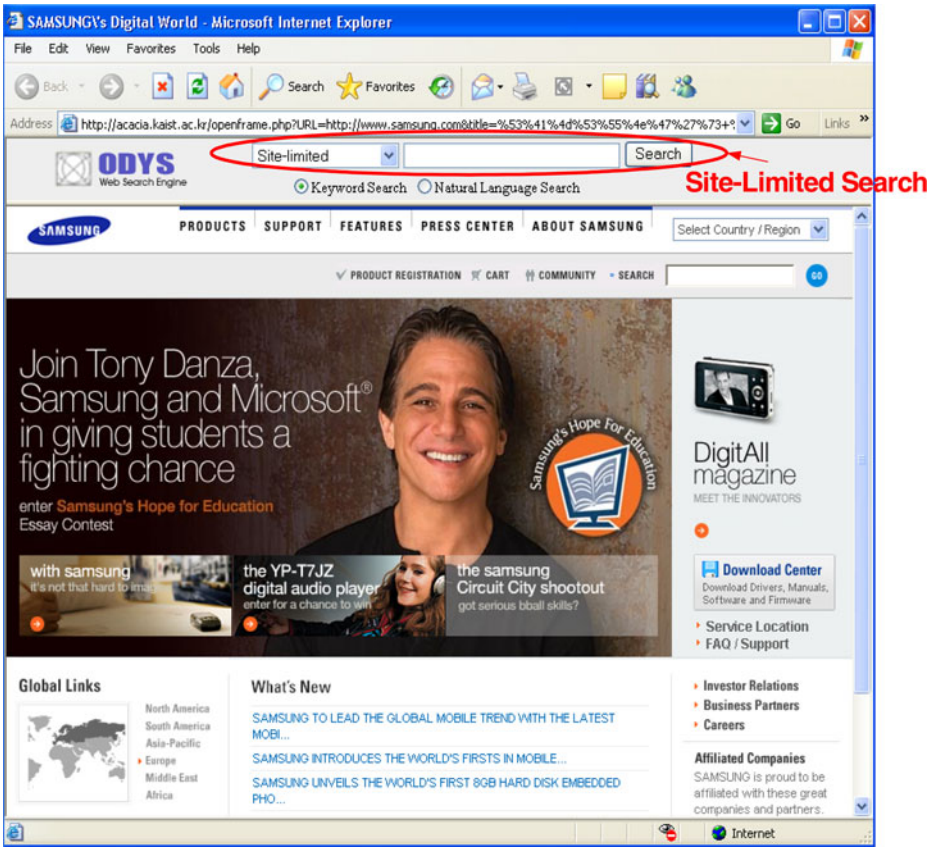

Figure 10 The interface for site-limited search

Figure 11 The database schema used to implement site-limited search

Attribute name	Attribute type	Description
`siteId`	integer	Site identifier
`URL`	varchar	Site URL
`title`	text	Site title
`description`	text	Site description

(a) The *websites* table.

Attribute name	Attribute type	Description
`siteId`	integer	Site identifier
`siteIdText`	text	Site identifier
`URL`	varchar	Page URL
`title`	text	Page title
`description`	text	Part of page content

(b) The *webpages* table.

system designer or the query optimizer can choose one of the two methods depending on the cost.

- IR index join: This method handles site-limited search as a multiple-keyword query. We note that we can evaluate the site condition by executing keyword search on `siteIdText`. `SiteIdText` is the text type version of `SiteId` and has to be declared additionally in the schema. (Let us recall that, from Section 3.2.1, the IR index join handles only text type attributes.) Hence, we perform the IR index join between the two posting lists of `siteIdText` and `description`. Thanks to posting skipping, a large number of postings within the posting list of `description` can be skipped while performing the IR index join; i.e., the postings generated from the web pages of other sites are skipped. Thus, performance is enhanced significantly.
- Attribute embedding: This method handles site-limited search as a query involving both keyword and attribute conditions. We note that we can embed the values of `siteId` into the posting lists of `description`, and the embedding can be specified in the schema. We can check whether each web page is

```
CREATE TABLE webpages ( siteId integer,
        siteIdText text, description text, ··· );
SELECT p.oid
FROM    webpages p
WHERE MATCH(p.description, "internet") > 0 AND MATCH(p.siteIdText,
"50000") > 0;
```

(a) Site-limited search using the IR index join.

```
CREATE TABLE webpages ( siteId integer,
        description text(embedded_attributes(siteId)),
        ··· );
SELECT p.oid
FROM    webpages p
WHERE MATCH(p.description, "internet") > 0 AND p.siteId = 50000;
```

(b) Site-limited search using attribute embedding.

Figure 12 Query statements for processing site-limited search

from a specific site while reading only one posting list rather than two. Thus, performance is enhanced significantly.

Example 5 Figure 12 shows queries for finding web pages that contain the keyword "internet" from the site whose siteId is equal to 50,000. Figure 12a shows a query using the IR index join; Figure 12b a query using attribute embedding with relevant schema definition. We note that, while processing the query in Figure 12b, the query processor accesses the embedded attribute, but not the data records.

4 Related work

4.1 DBMSs with IR features

4.1.1 MySQL with MyISAM

In MySQL, IR features are implemented in MyISAM [27], which is a storage system of MySQL. Thus, MySQL can be considered as tight-coupling like Odysseus. Odysseus has supported tight-coupling since Apr. 1997 (version 2.0)[7] while MySQL since Sept. 2000 (version 3.23.23). Although IR features are tightly integrated into MySQL, sophisticated DB-IR algorithms such as the IR index join (between two or more text attributes) and attribute embedding have not been implemented yet. However, the IR index join *without* posting skipping within *one* attribute (i.e., processing a query with multiple keywords on the same attribute) has been implemented.

The IR index of MySQL is the closest to our patented IR index. In MySQL, each posting list is implemented using a B-tree, and postings are stored in the leaf nodes of the B-tree. Thus, MySQL has a structure similar to our subindex even if the implementation details have differences. The IR index of Odysseus is more update-friendly than that of MySQL. In Odysseus, as described in Figure 5, updating a posting list is cheap since only the offset array needs to be updated. On the other hand, in MySQL, updating a posting list may involve a large amount of disk I/O's to move postings stored in leaf nodes of the B-tree.

Another notable difference is that Odysseus is an object-relational DBMS, and thus, supports the notion of the OID, which is an essential feature for web search engines. Odysseus can retrieve a list of OID's as a search result whereas MySQL should retrieve a list of *attribute values* (e.g., the identifier or the title). Retrieving a list of OID's does *not* require accessing data records. MySQL, however, cannot avoid accessing data records since the attribute value is stored inside a data record. We note that accessing data records is expensive since it incurs random I/O's. The advantage of using the list of OID's is that we only need to access the data records of those web pages that need to be displayed.

4.1.2 PostgreSQL with Tsearch2

Tsearch2 [33] is a package[8] for adding IR features to the PostgreSQL DBMS. In Tsearch2, new text data types, functions, and operators are implemented in the

[7] A patent was applied in 1999 [36].

[8] Tsearch2 is bundled with PostgreSQL since PostgreSQL Version 8.3.

C language. Then, the C code, which is compiled into shared libraries, is integrated into the PostgreSQL server through dynamic loading [19]. For IR indexing, Tsearch2 uses the GIN (Generalized Inverted iNdex) index [20] implemented in PostgreSQL. Tsearch2/PostgreSQL can be viewed as a slight variation of loose-coupling since its IR algorithms are implemented using the user-level APIs although it supports IR indexing at the storage system level. In general, not every algorithm can be implemented using the user-level APIs due to their restricted capability. Specifically, the user-level API of PostgreSQL provide only tuple-level accesses, so the IR index join and attribute embedding are hard to be implemented since these algorithms need to manipulate inside of tuples (i.e., posting lists).

4.1.3 Commercial DBMSs

In Oracle Cartridge and IBM DB2 Extender, new data types are added by using user-defined types, and their operations by using user-defined functions [7, 21]. However, Cartridge and Extender have differences in *extensible indexing* schemes that are employed for adding indexing schemes on new data types. Extensible indexing in Oracle is called *cooperative indexing* because the Cartridge module and the DBMS server cooperate to provide an indexing scheme [7, 18]. Extender employs the notion of *key transform* for extensible indexing [14] to exploit the B-tree index for new types. The details are omitted here since they are not a focus of this paper.

4.2 IR search engines with DBMS features

Among several open-source projects for search engines, Apache Lucene [4] has been the most representative one. Apache Solr is a search engine platform built upon Lucene. Lucene does not have full DBMS features, but has some DBMS-like features with regard to DB-IR integration. It supports fielded searching that enables us to specify a keyword condition on each field (attribute), which corresponds to structured query languages for DBMSs. In addition, it supports multiple-index searching, which is similar to the IR index join with posting skipping. A data structure, called a *skip list*, is used to boost query performance. We conjecture that, using the skip list, Lucene has implemented a technique similar to posting skipping or the ZigZag join.

Lucene now offers a mechanism, called a *payload*, that attaches arbitrary bytes to a token in the search index. It "looks" very similar to attribute embedding in that it can be used for storing attribute values in a posting. However, in the current version of Lucene (version 3.6.1), the payload supports *only a placeholder*, but the sophisticated method of checking the value of the payload for query processing is yet to be implemented. Its current usage is to store the weight of a term and to use the weight to boost the ranking of a document that contains the term [29].

Although Lucene has some DBMS-like features, it is not a full-blown DBMS. Thus, we believe the usability of Odysseus is better than that of Lucene. In Odysseus, users can develop applications simply by defining a schema and writing SQL queries. In contrast, to implement the same features in Lucene, programmers should write Java applications using the Lucene APIs. Especially, a significant amount of programming efforts will be required for implementing a technique similar to attribute embedding. Thus, in general, developing applications is much more systematic and less error-prone in Odysseus than in Lucene.

4.3 Research on DB-IR integration

In this section, we briefly review well-known studies on DB-IR integration. These studies are related to our research in a broad perspective, but their focuses are quite different from ours.

- Zobel and Moffat [44] presented a really nice survey on text search engines. They mainly discussed query evaluation for single-keyword queries and phrase queries as well as index construction and maintenance, but did *not* discuss query evaluation for DB-IR integration queries.
- Cheng and Chang [15] proposed a new web-search model, *entity search*, and its query processing method. Entity search can be considered as DB-IR integration since it enables us to specify search fields for unstructured data. That is, users can execute *structured* queries over *unstructured* data. Here, the type of an entity serves as a field. The focus of this paper is an index structure for entities rather than documents such as Web pages.
- Theobald et al. [32] described an XML-IR system called TopX. The system handles an XML query language extended with IR features. This work can be considered as one of DB-IR integration systems since it handles both path conditions (DB features) and text conditions (IR features). However, this work is quite different from ours since their DB-IR integration deals with XML queries, more specifically, path operations.
- Bast and Weber [9] described CompleteSearch, an interactive search engine that offers users several features including the *autocompletion* feature. Since this engine is mainly based on prefix search (IR features) and join (DB features), it can be considered as one of DB-IR integration systems. However, their DB-IR integration is quite different from ours in that it focuses on autocompletion. The same authors proposed a system called ESTER [10] for combined full-text and ontology search, and ESTER shares some concepts with CompleteSearch.
- Heman et al. [24] built a search engine on top of their relational DBMS MonetDB/X100. They argued that their approach is reasonable since modern DBMSs have benefited from recent hardware developments. This work belongs to the first category introduced in Section 1, which is IR on a relational DBMS.
- Several methods were developed to support keyword search in relational databases. DBXplorer proposed by Agrawal et al. [2] and DISCOVER proposed by Hristidis and Papakonstantinou [25] are the representative systems. The logical units of information may be fragmented and scattered across multiple tables. Thus, the key issue of these systems is to join the tuples from these multiple tables on the fly so that the tuples contain all the keywords specified. In this way, the systems enable users to perform keyword search on a relational database without knowing the schema of the database. These studies are orthogonal to ours since the forms of the queries considered are totally different. Only a set of keywords is given as a query in these systems whereas an SQL statement in Odysseus.

Finally, we introduce some well-known research projects related to DB-IR integration. They mostly concentrate on *information extraction* or *entity search*, and this direction has been regarded as one area of DB-IR integration [34] while it is different from the focus of our approach. The primary goal of this direction is to impose a structure on *unstructured* data such as the Web contents. For example, by

extracting speakers and departments (i.e., entities) from the Web pages for seminar announcements in a university, structured (i.e., database-style) queries can be issued over those extracted entities; an example query is finding all speakers who gave a talk at a specific department. On the other hand, we have concentrated on the *performance* of DB-IR integration queries in the setting where the data are *already organized* into the documents with multiple attributes of either a built-in type or the text type. Since the projects mentioned below do not provide a software package but a demo system with their proprietary data set, a direct comparison between Odysseus and the projects is not possible.

- Libra: For supporting entity search on the Web, Microsoft Research Asia developed information extraction methods. These methods include pattern-matching algorithms tailored to typical Web-page layouts. The methods were used to build a portal site (http://libra.msra.cn) for scholarly search on the extracted records about authors, papers, conferences, and communities.
- Cimple/DBLife: The Cimple project [17] is jointly carried out by the University of Wisconsin and Yahoo! Research. Similar to Libra, it aims at generating and maintaining community-specific portals with structured information gathered from the Web. Cimple's flagship application is the DBLife portal (http://dblife.cs.wisc.edu). For gathering and reconciling different types of entities on the Web, Cimple has a suite of DB-style extractors based on pattern matching and dictionary lookups.
- KnowItAll/TextRunner: The KnowItAll project [8] is carried out by the University of Washington. Its query model involves multiple entities and a relationship between the entities. For example, "What countries are located in Africa?" translates to the query *Argument*1 = "type: Country," *Relation* = "is located in," and *Argument*2 = "Africa." This project provides a demo system TextRunner (http://openie.cs.washington.edu).
- YAGO: The YAGO project [31] is carried out by Max Plank Institute. It shares the goal with KnowItAll and TextRunner, but emphasizes high accuracy and consistency. To this end, YAGO primarily gathers its knowledge from Wikipedia and WordNet. In addition, it employs text-mining-based techniques as well.

5 Performance evaluation

In this section, we evaluate the performance of Odysseus by using real data. Section 5.1 compares the two query processing algorithms presented in Section 3. Section 5.2 presents the performance comparison with two open-source DBMSs—MySQL and PostgreSQL. Section 5.3 presents the performance comparison with Lucene, which is the most representative open-source search engine.

Approximately 16 million web pages have been crawled and then converted to documents conforming to the database schema in Figure 11. The size of the source data is approximately 100 GBytes.

The main goal of our experiments is to compare the performance of processing *DB-IR integration queries* shown in Figure 12. Odysseus supports various types of queries including multiple-keyword queries and phrase queries, but they are not discussed in this paper because they are out of scope of this paper. We use a *query-independent ranking* scheme, PageRank [12]. The documents are sorted in the order

of PageRank and are indexed according to the order. Thus, no sorting is required for ranking in the middle of query processing. Unlike the typical setting of a Web search engine where only top-k results are retrieved, we decide to retrieve *all* relevant results in the experiments. This is because we want to see performance trends as the number of query results varies.

We measure the cold start time and the warm start time, which are two standard measures for database system performance. The *cold start* time is defined as the wall clock time for executing a query when no data are in the DBMS buffer. Thus, the data are accessed from disk. The *warm start* time is defined as the wall clock time for executing a query when all the relevant data are in the DBMS buffer. Thus, the data are accessed from main memory. The cold start time essentially measures the disk time; the warm start time the CPU time for processing a query.

For the three DBMS (Odysseus, MySQL, and PostgreSQL), the experiment programs are written using their call-level interface in C/C++. For Lucene, the programs are written using its Java APIs since Lucene is purely Java-based.

All experiments are conducted on a Linux PC with 3.6 GHz dual-core CPU, 1 GB of main memory, and four 1.5 TB disks. The transfer rate of the disks is 61∼125 MB/sec (95 MB/sec on the average), and the bandwidth is 187.5 MB/sec.

5.1 Comparison between the two tightly-coupled algorithms

We compare the IR index join with attribute embedding. For this comparison, 800 site-limited queries are generated as follows: a keyword and a site identifier are randomly chosen from a given subset of the data set, and they are inserted into the template query in Figure 12. The same set of queries will be used for subsequent experiments.

Since the two algorithms perform differently according to the selectivity of a query, we elaborate on the performance trend as selectivity varies. For a given site-limited query, the *posting selectivity* is defined as the number of query results to the total number of documents containing the keyword. More specifically, given a keyword K and a site S, the posting selectivity is defined by (1), where $postings\_K$ (or $postings\_S$) is the set of the postings in the posting list for K (or S).

$$posting\ selectivity\ (K, S) = \frac{|postings\_K \cap postings\_S|}{|postings\_K|} \tag{1}$$

Because the execution time is heavily dependent on the size of the posting lists involved, to avoid variations with the size, we *normalize* the execution time as in (2) instead of adding the size as a new parameter. Here, we set the standard size for normalization to be 10,000. The *base time* means a fixed time spent regardless of the queries. At cold start, the base time is *not* negligible because it includes the time for accessing the keyword B^+-tree and locating the posting lists from disk. It is determined to be 70 ms, which is the shortest execution time. On the other hand, at warm start, the base time is negligible and is set to be 1 ms.

$$\begin{gathered} T_n = (T_o - base\ time) \times \frac{10000}{|postings\_K|} + base\ time \\ T_n : normalized\ time \quad T_o : original\ time \end{gathered} \tag{2}$$

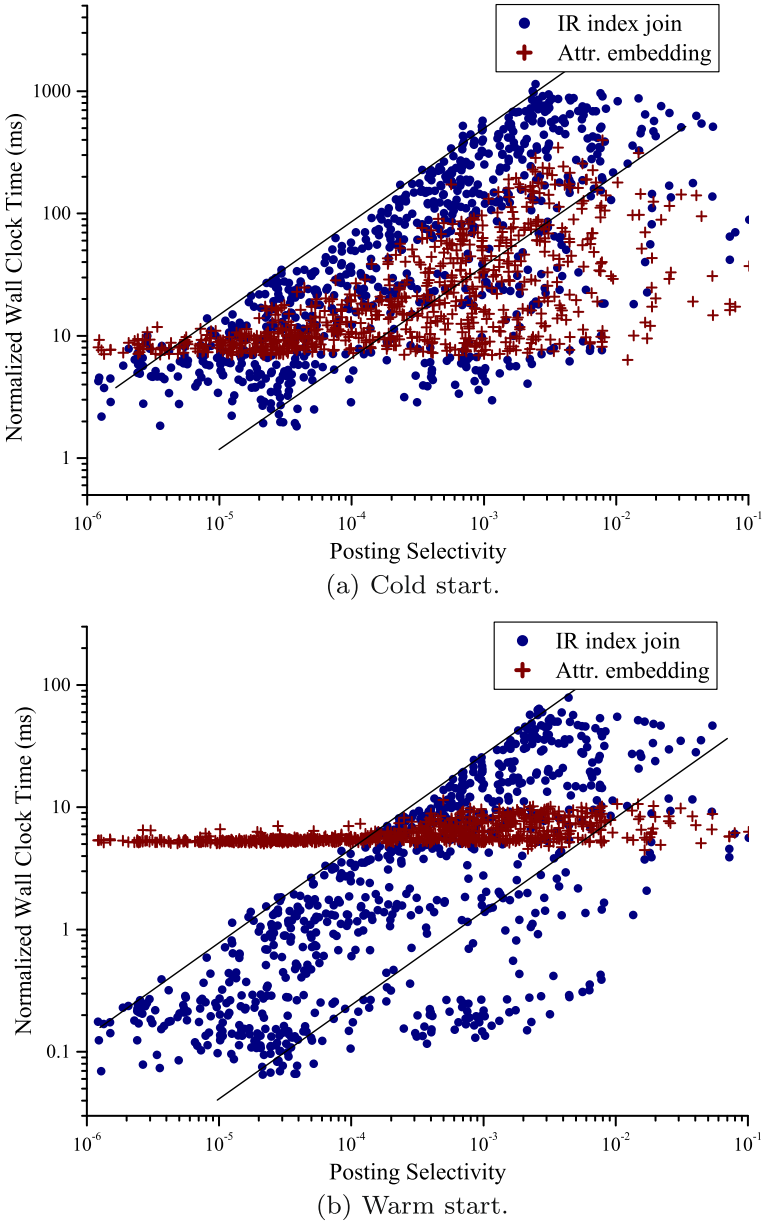

Figure 13 Comparison between the two tightly-coupled algorithms

Figure 13 shows scatter plots between the posting selectivity and the *normalized* wall clock time. Each point represents a query. Let's look into the warm-start case of Figure 13b first for ease of explanation. The performance of attribute embedding is not affected by the posting selectivity since all postings are touched regardless of the

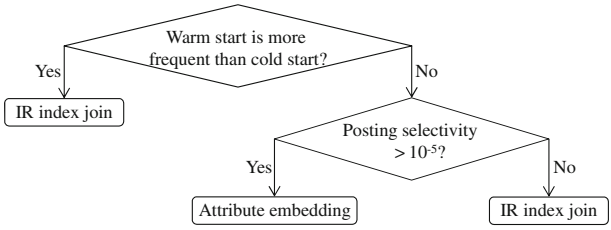

Figure 14 Rule-of-thumb optimization for DB-IR integration queries

posting selectivity.[9] The performance of the IR index join improves as the posting selectivity decreases since more postings can be skipped by posting skipping. The threshold parameter (explained in Section 3.2) for posting skipping is empirically set to be 1000 throughout the paper.

The points for the IR index join are observed to fall into the diagonal strip which is enclosed by two lines in the figure. In other words, the performance of the IR index join is not totally determined by the posting selectivity, though it is a dominant factor. In fact, the performance is also affected by the distribution of the query results within the posting lists. That is, if query results are clustered, posting skipping becomes very effective, achieving good performance; however, if query results are scattered with small gaps, posting skipping is not triggered frequently. Therefore, performance at a specific posting selectivity is represented by a range due to different distributions.

The performance of attribute embedding is shown to be not much different between at cold start and at warm start. This is because attribute embedding performs only sequential scan. In Odysseus, the performance of sequential scan is highly optimized, obviating significant overhead at cold start (see Section 5.2.1).

In general, we conclude that IR index join with posting skipping has better performance for low-selectivity queries while attribute embedding has uniform performance regardless of posting selectivity. The choice between the two methods can be done depending on the characteristics and usage pattern of the application. As a rule of thumb, we suggest an optimization rule in Figure 14. Here, we always suggest the IR index join at warm start to simplify the rule even though attribute embedding outperforms the IR index join at high selectivity. We note that, if we consider the *unnormalized (i.e., real)* execution time, a loss caused by choosing the IR index join is very small because the execution time itself is usually small (<10 ms) at high selectivity.

5.2 Comparison of Odysseus and open-source DBMSs

Odysseus, MySQL, and PostgreSQL are compared for the performance of processing DB-IR integration queries. We built three web search systems using these three DBMSs. We use MySQL version 5.1 and PostgreSQL version 8.4, which are the currently recommended versions. Odysseus and MySQL adopt tight-coupling, while PostgreSQL a variation of loose-coupling.

[9]The result at cold start in Figure 13a appears to be a bit cluttered because of normalization errors, which are likely to happen in short-running queries.

As an example of DB-IR integration queries, we use site-limited search, which involves both keyword and attribute conditions. The keywords are classified according to the number of documents in which a keyword appears: the *small keyword set* is composed of the ones appearing in [0, 10000) documents, the *medium keyword set* in [10000, 100000) documents, the *large keyword set* in [100000, 1000000) documents, and *huge keyword set* in [1000000, ∞) documents. The sites are classified according to the number of documents crawled from a site: the *small site set* is composed of the ones having less than 10,000 documents, and the *large site set* the ones having more than 10,000 documents.

To make the comparison as fair as possible, we use the same parameter values for all three DBMSs: the DBMS buffer size is 76 MB, the sort buffer size is 100 MB, the page size is 4 KB except for MySQL.[10]

5.2.1 Single-keyword queries

As a baseline, we first show the performance of processing single-keyword queries in Figure 15. One hundred keywords are randomly selected from each keyword set. As noted earlier, Odysseus can retrieve a list of OID's as a search result, whereas MySQL and PostgreSQL should retrieve the attribute value itself. For a fair comparison, we substract the time for retrieving the attribute value in MySQL and PostgreSQL.

In Figure 15, Odysseus improves the performance by 7.1∼21.4 times at cold start and by 5.0∼27.2 times at warm start compared with PostgreSQL. MySQL improves the performance by 2.0∼6.2 times at cold start and by 1.1∼3.1 times at warm start compared with PostgreSQL. These results demonstrate the superiority of the tight-coupling architecture of Odysseus (and MySQL) compared with the loose-coupling architecture of PostgreSQL. In the loose-coupling architecture, the performance degrades due to the overhead of calling functions implemented in the external module (i.e., the tsearch2 module). Furthermore, user-defined types and functions employed in the loose-coupling architecture incur the overhead of accessing the database catalog.

In order to provide good sequential search performance, in Odysseus, we maintain sequential contiguity of posting lists by using the notion of the physical extent (typically, set to 64 Kbytes) and by using the notion of the train, a physically contiguous sequence of disk pages retrieved as a unit.

5.2.2 Site-limited search

Figure 16 shows the wall clock time for processing site-limited search. We vary both keywords and sites used in the experiment. For each pair of a keyword set and a site set, we generate one hundred queries randomly and present the average wall clock time. The X axis represents the initials of the keyword set and the site set; e.g., LS denotes the pair of the large keyword set and the small site set. In this way, we control the size of the posting lists involved in site-limited search.

We compare six query processing methods by selecting two methods from each of three DBMSs. Odysseus-Join, MySQL-IR, and PostgreSQL-IR indicate the methods

[10]The page size of MySQL cannot be changed and is set to be the default value of 1 KB.

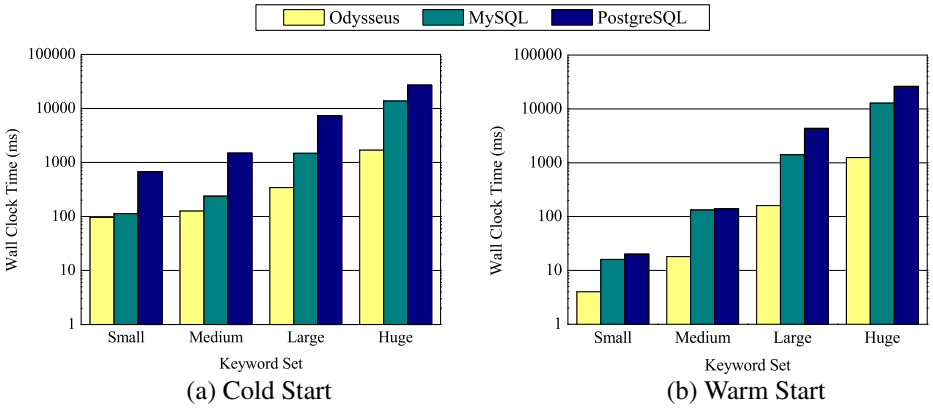

Figure 15 The wall clock time for processing single-keyword queries

that execute the `SELECT` query in Figure 12a. Odysseus-Embedding, MySQL-BTree, and PostgreSQL-BTree indicate the methods that execute the `SELECT` query in Figure 12b. Odysseus-Join and Odysseus-Embedding exploit IR index join with posting skipping and attribute embedding, respectively.

Figure 16 shows that Odysseus improves the performance by 27.8∼623 times at cold start and by 138∼3650 times at warm start compared with PostgreSQL. MySQL improves the performance by 1.46∼1.71 times at cold start and by 15.1∼19.6 times at warm start compared with PostgreSQL. Odysseus improves the performance by 19.1∼385 times at cold start and by 9.18∼190 times at warm start compared with MySQL. Odysseus performs the best owing to (i) IR index join with posting skipping and (ii) attribute embedding. These results indeed demonstrate the effectiveness of our tightly-coupled algorithms for DB-IR integration.

In MySQL and PostgreSQL, due to lack of such advanced features, performance degrades mainly because they have to spend time retrieving intermediate results that

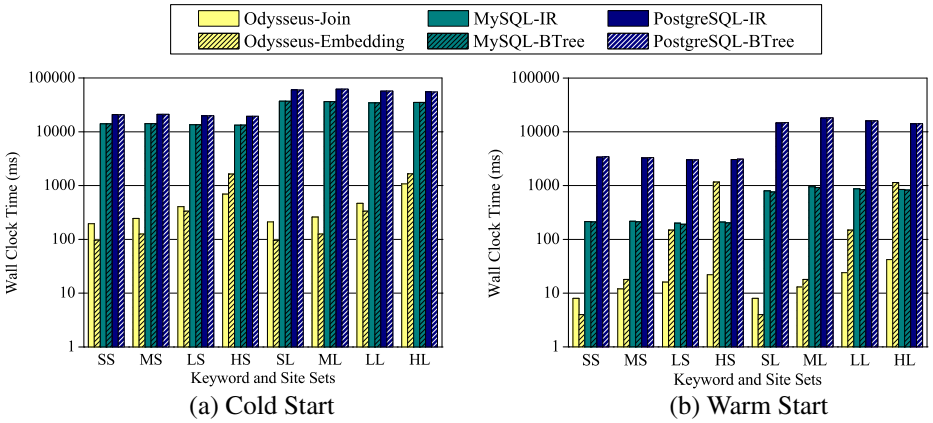

Figure 16 The wall clock time for processing site-limited search queries

satisfy only one condition. MySQL-IR and PostgreSQL-IR first evaluate one keyword condition in Figure 12a using the inverted index, and then, evaluate the other keyword condition by fetching each intermediate result obtained from the first step. MySQL-BTree and PostgreSQL-BTree first evaluate the attribute condition (i.e., the site condition) in Figure 12b using the B(B^+)-tree, and then, evaluate the keyword condition by fetching each intermediate result obtained from the first step.

For MySQL-IR and PostgreSQL-IR, their optimizers use only one inverted index which appears earlier in the schema definition. Since `p.siteIdText` appears earlier than `p.description`, the query optimizers always pick up the inverted index on `p.siteIdText`. It turns out that using the inverted index on `p.siteIdText` is more advantageous than using that on `p.description` since the condition on `p.siteIdText` is more selective than that on `p.description`. That is, for a given site-limited query, the number of documents that satisfy the site condition is less than or equal to the number of documents that satisfy the keyword condition. Despite that we give MySQL and PostgreSQL this advantage, Odysseus significantly outperforms MySQL and PostgreSQL.

MySQL and PostgreSQL perform poorly since they cannot exploit two inverted indexes but only one for the query in Figure 12a or none for the query in Figure 12b. This limitation is inevitable in PostgreSQL since it is based on loose-coupling. On the other hand, since MySQL adopts tight-coupling, we expect that MySQL would perform comparable to Odysseus if our sophisticated algorithms were also implemented into MySQL.

Some might be curious why MySQL-IR and MySQL-BTree (or PostgreSQL-IR and PostgreSQL-BTree) show similar performance. The only difference between two methods is the index—the inverted index or the B(B^+)-tree index—used for obtaining the intermediate results satisfying the site condition. Both methods do not use the inverted index on the `p.description` attribute. We observe that the type of the index does not make a big difference in performance of evaluating the site condition itself.

In summary, Odysseus significantly outperforms MySQL and PostgreSQL for both single-keyword queries and site-limited queries. In general, Odysseus can execute queries involving *both* keyword and attribute conditions very fast—which is exactly the advantage of the tight-coupling architecture.

5.3 Comparison of Odysseus and Lucene

The purpose of this comparison is to show that Odysseus provides higher performance for processing DB-IR integration queries than Lucene. We use Lucene version 3.6, which is the currently recommended version. For Lucene, only the IR index join, which corresponds to Odysseus-Join, is implemented and compared with the two methods of Odysseus. The IR index join is not specific to Odysseus but is a common query processing technique also supported by Lucene. Moreover, it is commonly known that specialized search engines may have better search performance than database systems. Thus, in fact, this comparison is even more favorable to Lucene than to Odysseus. On the other hand, a method corresponding to Odysseus-Embedding is hard to implement in Lucene as discussed in Section 4.2.

We implement the experiment programs for Lucene using its Java APIs. For data loading, a `Document` object is created so that it has the schema in Figure 11

and is filled with attribute values; then, every `Document` object is added to an `IndexWriter` object to build an IR index. For query processing, a `QueryParser` object translates a query string to a `Query` object, which is an internal structure of Lucene; then, the `Query` object is passed to an `IndexSearcher` object for searching the documents that satisfy the query conditions. Lucene is known to use a data structure called the *skip list* to speed up multiple-index searching although the details of the skip list are not reported in the literature.

We now explain the comparison results. First, Figure 17 shows the performance of processing single-keyword queries in Odysseus and Lucene. The queries used are the same as those in Figure 15. Odysseus is shown to outperform Lucene for processing these basic IR queries by 3.1∼6.2 times at cold start and by 1.5∼3.9 times at warm start.

Then, Figure 18 shows the performance of processing site-limited search queries in Odysseus and Lucene. The queries used are the same as those in Figure 16. Odysseus improves the performance by up to 7.4 times at cold start. However, Lucene shows a better performance than Odysseus at warm start. This is because our IR index is optimized for disk accesses while the skip list seems to be optimized for main-memory accesses. We claim that cold start is more prevalent than warm start in practical environments since, typically, all data cannot be loaded into main memory owing to their huge size.

Surprisingly, it is observed that the performance of Lucene is almost constant regardless of the length of posting lists involved. We conjecture that the performance of Lucene is mainly dependent on the number of query results possibly because of the skip list. We note that the number of query results does not vary that much across the keyword-site sets.

In addition to the performance, yet another important advantage of Odysseus is that it provides higher-level interface than Lucene. The high-level functionalities such as SQL, schemas, or indexes that are provided by Odysseus allow developers to easily implement query processing modules in search engines because they provide stronger expressive power than primitive functions in Lucene, facilitating easy (and much less error-prone) application development and maintenance [42].

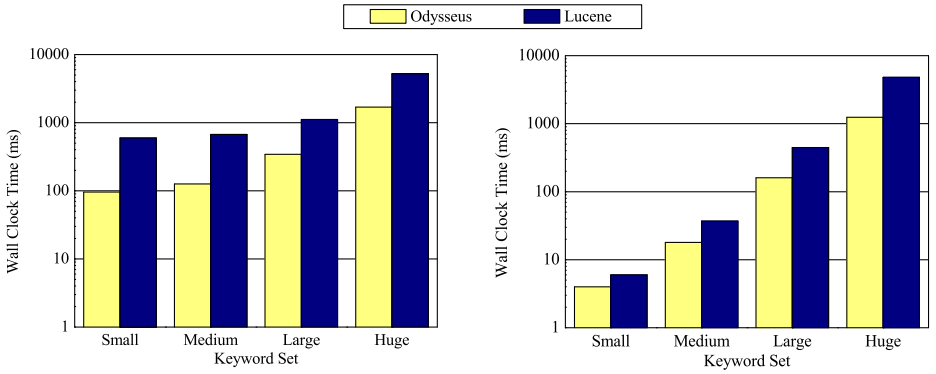

Figure 17 The wall clock time for processing single-keyword queries

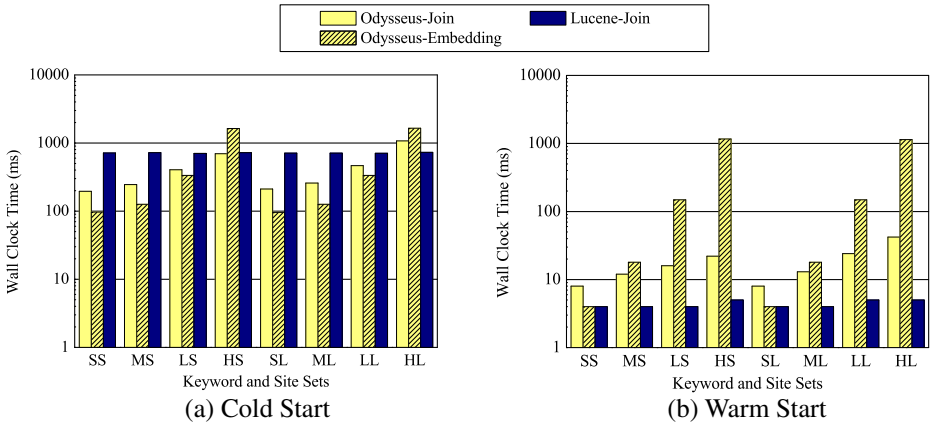

Figure 18 The wall clock time for processing site-limited search queries

6 Conclusions

In this paper, we have contended that the tight-coupling architecture has many advantages for DB-IR integration over the loose-coupling architecture. We have verified our arguments by using the Odysseus DBMS that has tightly-coupled IR features. Especially, two tightly-coupled DB-IR algorithms—IR index join with posting skipping and attribute embedding—have been proven to be very effective for DB-IR integration queries. The IR index join is effective for highly selective queries whereas attribute embedding has uniform performance regardless of the selectivity. We have compared three DBMSs and one search engine for the performance of processing DB-IR integration queries—more specifically, site-limited search queries. As a result, Odysseus outperforms MySQL and PostgreSQL up to orders of magnitude by virtue of the IR index join and attribute embedding, which are only possible in the tight-coupling architecture; Odysseus significantly outperforms Lucene but with some exceptional cases (i.e., site-limited search queries at warm start). Overall, we believe that tight-coupling with IR features will be the right direction in database systems for supporting DB-IR integration. In this perspective, Odysseus is the first DBMS that truly implements DB-IR integration.

Acknowledgements This work was supported by the National Research Foundation of Korea (NRF) grant funded by Korean Government (MEST) (No. 2012026326).

References

1. Abiteboul, S., et al.: The Lowell database research self-assessment. Commun. ACM **48**(5), 111–118 (2005)
2. Agrawal, S., Chaudhuri, S., Das, G.: DBXplorer: a system for keyword-based search over relational databases. In: ICDE, pp. 5–16 (2002)
3. Agrawal, R., et al.: The Claremont report on database research. ACM SIGMOD Rec. **37**(3), 9–19 (2008)
4. Apache Lucene: http://lucene.apache.org/ (2013). Accessed 22 Nov 2013

5. Baeza-Yates, R.A., Ribeiro-Neto, B.A.: Modern Information Retrieval. ACM Press/Addison-Wesley (1999)
6. Baeza-Yates, R.A., Consens, M.P.: The continued saga of DB-IR integration. In: VLDB (2004) (a tutorial)
7. Banerjee, S., Krishnamurthy, V., Murthy, R.: All your data: the oracle extensibility architecture. Oracle White Paper. Oracle Corp. (1999)
8. Banko, M., Cafarella, M.J., Soderland, S., Broadhead, M., Etzioni, O.: Open information extraction from the web. In: IJCAI, pp. 2670–2676 (2007)
9. Bast, H., Weber, I.: The completeSearch engine: interactive, efficient, and towards IR & DB integration. In: CIDR, pp. 88–95 (2007)
10. Bast, H., Chitea, A., Suchanek, F.M., Weber, I.: ESTER: efficient search on text, entities, and relations. In: SIGIR, pp. 671–678 (2007)
11. Biliris, A.: The performance three database storage structures for managing large objects. In: SIGMOD, pp. 276–285 (1992)
12. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. In: WWW, pp. 107–117 (1998)
13. Chaudhuri, S., Ramakrishnan, R., Weikum, G.: Integrating DB and IR technologies: what is the sound of one hand clapping. In: CIDR, pp. 1–12 (2005)
14. Chen, W., Chow, J., Fuh, Y., Grandbois, J., Jou, M., Mattos, N.M., Tran, B.T., Wang, Y.: High level indexing of user-defined types. In: VLDB, pp. 554–564 (1999)
15. Cheng, T., Chang, K.C.-C.: Beyond pages: supporting efficient, scalable entity search with dual-inversion index. In: EDBT, pp. 15–26 (2010)
16. Cornacchia, R., Heman, S., Zukowski, M., de Vries, A.P., Boncz, P.A.: Flexible and efficient IR using array databases. VLDB J. **17**(1), 151–168 (2008)
17. DeRose, P., Shen, W., Chen, F., Doan, A., Ramakrishnan, R.: Building structured web community portals: a top-down, compositional, and incremental approach. In: VLDB, pp. 399–410 (2007)
18. DeFazio, S., Daoud, A.M., Smith, L.A., Srinivasan, J., Croft, W.B., Callan, J.P.: Integrating IR and RDBMS using cooperative indexing. In: SIGIR, pp. 84–92 (1995)
19. Ewald, G., Hans-Jurgen, S.: PostgreSQL developer's handbook. Sams Publishing (2001)
20. Full-Text Search in PostgreSQL: http://www.postgresql.org/docs/8.3/static/textsearch.html (2013). Accessed 22 Nov 2013
21. Fuh, Y., Deßloch, S., Chen, W., Mattos, N., Tran, B., Lindsay, B., DeMichel, L., Rielau, S., Mannhaupt, D.: Implementation of SQL3 structured types with inheritance and value substitutability. In: VLDB, pp. 565–574 (1999)
22. Guo, L., Shao, F., Botev, C., Shanmugasundaram, J.: XRANK: ranked keyword search over XML documents. In: SIGMOD, pp. 16–27 (2003)
23. Halverson, A., Burger, J., Galanis, L., Kini, A., Krishnamurthy, R., Rao, A.N., Tian, F., Viglas S., Wang, Y., Naughton, J.F., DeWitt, D.J.: Mixed mode XML query processing. In: VLDB, pp. 225–236 (2003)
24. Heman, S., Zukowski, M., de Vries, A.P., Boncz, P.A.: Efficient and flexible information retrieval using MonetDB/X100. In: CIDR, pp. 96–101 (2007)
25. Hristidis, V., Papakonstantinou, Y.: DISCOVER: keyword search in relational databases. In: VLDB, pp. 670–681 (2002)
26. IBM: DB2 UDB Text Extender Administration and Programming Version 8 (2003)
27. Lentz, A.: MySQL Storage Engine Architecture. MySQL Developer Articles. MySQL AB (2004) (available from http://dev.mysql.com/tech-resources/articles). Accessed 22 Nov 2013
28. Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Cambridge University Press (2008)
29. McCandless, M., Hatcher, E., Gospodnetic, O.: Lucene in Action, 2nd edn. Manning Publications (2010)
30. Oracle: Oracle Data Cartridge Developer's Guide 11g Release 1 (2008)
31. Suchanek, F.M., Kasneci, G., Weikum, G.: YAGO: a core of semantic knowledge. In: WWW, pp. 697–706 (2007)
32. Theobald, M., et al.: TopX: Efficient and versatile top-k query processing for semistructured data. VLDB J. **17**(1), 81–115 (2008)
33. Tsearch2—Full Text Extension for PostgreSQL: http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2 (2013). Accessed 22 Nov 2013
34. Weikum, G.: DB&IR: both sides now. In: SIGMOD, pp. 25–30 (2007)
35. Whang, K., Krishnamurthy, R.: The multilevel grid file—a dynamic hierarchical multidimensional file structure. In: DASFAA, pp. 449–459 (1991)

36. Whang, K., Park, B., Han, W., Lee, Y.: An inverted index storage structure using subindexes and large objects for tight coupling of information retrieval with database management systems. U.S. Patent No. 6,349,308 (2002) (Appl. No. 09/250,487 (1999))
37. Whang, K.: Tight-coupling: A way of building high-performance application specific engines. DASFAA (2003) (presented at the panel session, available on-line from http://www.dasfaa.org/dasfaa2003/file/Prof_Kyu-Young_Whang_5.pdf). Accessed 22 Nov 2013
38. Whang, K., Lee, M., Lee, J., Kim, M., Han, W.: Odysseus: a high-performance ORDBMS tightly-coupled with IR features. In: ICDE, pp. 1104–1105 (2005) (this paper received the Best Demonstration Award)
39. Whang, K.: A new DBMS architecture for DB-IR integration. In: APWeb/WAIM, pp. 4–5 (2007) (a keynote presentation)
40. Whang, K.: DB-IR integration and its application to a massively-parallel search engine. In: CIKM, pp. 1–2 (2009) (a keynote presentation)
41. Whang, K., Lee, J., Kim, M., Lee, M., Lee, K., Han, W., Kim, J.: Tightly-coupled spatial database features in the Odysseus/OpenGIS DBMS for high-performance. GeoInformatica **14**(4), 425–446 (2010)
42. Whang, K., Yun, T., Yeo, Y., Song, I., Kwon, H., and Kim, I.: ODYS: an approach to building a massively-parallel search engine using a DB-IR tightly-integrated parallel DBMS for higher-level functionality. In: SIGMOD, pp. 313–324 (2013)
43. Witten, I.H., Moffat, A., Bell, T.C.: Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd edn. Morgan Kaufmann Publishers (1999)
44. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comput. Surv. **38**(2), 1–56 (2006)