World Wide Web
https://doi.org/10.1007/s11280-018-0647-1

Two-dimensional indexing to provide one-integrated-memory view of distributed memory for a massively-parallel search engine

Tae-Seob Yun[1] · Kyu-Young Whang[1] · Hyuk-Yoon Kwon[2] · Jun-Sung Kim[1] · Il-Yeol Song[3]

Received: 18 February 2018 / Revised: 5 June 2018 / Accepted: 31 October 2018 /
Published online: 13 November 2018

Abstract

We propose *two-dimensional indexing*—a novel in-memory indexing architecture that operates over distributed memory of a massively-parallel search engine. The goal of two-dimensional indexing is to provide a one-integrated-memory view as in a single node system using one large integrated memory. In two-dimensional indexing, we partition the entire index into $n \times m$ fragments and distribute them over the memories of multiple nodes in such a way that each fragment is entirely stored in main memory of one node. The proposed architecture is not only scalable as it uses a scaled-out shared-nothing architecture but also is capable of achieving low query response time as it processes queries in main memory. We also propose the concept of the *one-memory point*, which is the amount of the memory space required to completely store the entire index in main memory providing a one-integrated-memory view. We first prove the effectiveness of two-dimensional indexing with single-keyword queries, and then, extend the notion so as to be able to handle multiple-keyword queries. To handle multiple-keyword queries, we adopt *pre-join* that materializes a multiple-keyword query a priori as well as a new notion of *semi-memory join* that obviates extensive communication overhead to perform join across multiple nodes. In experiments using the real-life search query set over a database consisting of 100 million Web documents crawled, we show that two-dimensional indexing can effectively provide a one-integrated-memory view without too much of additional memory compared with the single node system using one large integrated memory. We also show that, with a six-node prototype, in an ideal case, it significantly improves the query processing performance over a disk-based search engine with an equivalent amount of in-memory buffer but without two-dimensional indexing — by up to 535.54 times. This improvement is expected to get larger as the system is scaled-out with a larger number of machines.

Keywords Massively-parallel search engine · DB-IR integration · Pre-join · Multiple-keyword search queries · Distributed memory

✉ Kyu-Young Whang
kywhang@mozart.kaist.ac.kr

Extended author information available on the last page of the article.

1 Introduction

1.1 Motivation

With the advent of various Web services on Internet, the amounts of data on the Web are explosively increasing. The Web search engine stores petabyte-scale data consisting of tens of billions of Web pages and handles billions of queries per day [7, 13, 15]. Since such large-scale data can be neither stored nor processed in a single node, commercial search engines use hundreds of thousands nodes to store large-scale data in a distributed manner and process queries in parallel [7].

As many people use search engines routinely, their expectations for low query response time are very high. However, satisfying users' expectation is not easy since the random access speed of disk is slow. To solve this problem, both academic research reported and commercial search engines heavily rely on cache severs. That is, keeping a set of cache servers outside the search engine for caching the query results [8, 10, 16, 20, 26] is widely-accepted as an orthodoxy, and many other caching techniques have been developed at different levels such as the document server [1, 2] and the index server [4, 29].

Meanwhile, as semiconductor technology rapidly advances, it has become viable to construct a large-scale, main-memory system based on the memories of multiple machines in a distributed environment. As a result, many large-scale systems based on distributed memory have been developed [9, 17, 19, 27]. We can decrease the level of dependency on cache servers by storing the entire index in distributed memory and process queries in memory. In the academic world, however, only some single-node in-memory IR methods have been reported [5, 28], but no work reported for massively-parallel search engines using multiple nodes. Google, which is a massively-parallel commercial search engine, is known to store the entire index in distributed memory [7], but it heavily relies on cahe servers that stores instance of query results in the memory to speed up query processing [11].

Figure 1 shows the architecture of the ODYS search engine [32]. ODYS is a massively-parallel search engine using the DB-IR tightly-integrated DBMS architecture [33]. It consists of masters and slaves. The entire Web documents and indexes are partitioned and stored in the slaves. The user query is given to a master, and the master distributes the query to the slaves. Then, the master merges intermediate results retrieved from the slaves and returns the final result to the user.

Here, the entire index is stored in disks of the slaves, and only the part of the index that is recently accessed is brought into main memory of the slaves via buffering. Thus, while the queries that have been recently processed can be rapidly processed in main memory, other queries that need to access the data in disk cause a major performance penalty. To handle a large query load, ODYS needs to keep multiple ODYS sets (i.e., replicas). However, even if the collective memory space from all the slaves in multiple ODYS sets is bigger than the size of the entire index, the entire index cannot be stored in main memory if the memories of multiple nodes are not properly coordinated. Thus, we need a method of coordinating memories of multiple nodes as if we had one integrated memory storing the entire index.

1.2 Query semantics in the search engine

Queries in a search engine are classified into single-keyword ones and multiple-keyword ones by the number of keywords used in a query. A single-keyword query returns documents that contain the given keyword. A multiple-keyword query finds documents containing keywords satisfying a Boolean expression consisting of AND, OR, NOT operators specified in

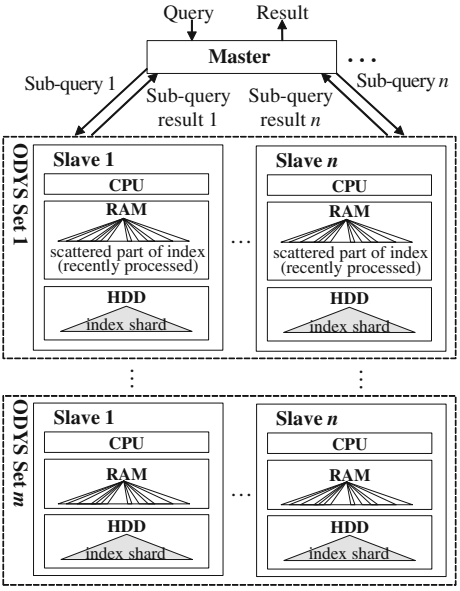


Figure 1 The architecture of the ODYS search engine [32]

the query. We use the AND semantics for multiple-keyword queries since the user generally wants to find documents containing all the query keyword.[1] In general, a Boolean expression can be transformed into the equivalent disjunctive normal form. Thus, in order to find the results of a multiple-keyword query, we can first process each conjunct as a query and union the results.

A search engine usually retrieves only top-k results since there could be a large number of documents that contain the query keywords. For the ranking measure, we use PageRank [21], which is one of the most popular ranking methods for the search engine. PageRank is very efficient since it is query-independent, i.e., the score of a document can be determined before the query is given. To improve ranking quality of PageRank, we can additionally apply query dependent ranking (e.g., TF-IDF) into the top results returned by PageRank. In this paper, however, we do not deal with ranking quality since we focus on the performance of the search engine.

We focus on the index access time only (i.e., excluding the document retrieval time).[2] The document retrieval time can be easily reduced by using other scalable in-memory systems such as in-memory key-value stores [17, 19] or in-memory DBMSs [9, 27] since each document can be accessed independent of index access. Hence, we define the *query response time* as the index access time.

1.3 Challenging issues

Challenging issues in constructing an in-memory search engine are summarized below. We focus on how to solve these issues effectively.

[1] Since ODYS/2D-Indexing is a DBMS-based search engine supporting SQL, it gently supports arbitrary queries including various operators such as AND, OR, NOT.

[2] Many research studies [5, 28] focus on the index search time without including the document retrieval time.

- **Partitioning of the entire index:** In a distributed environment, we need a method that effectively partitions the entire index into multiple index fragments to distribute over multiple nodes.
- **Inter-node communication:** During query processing in a distributed environment, we may need to transfer partial results from one node to another. This inter-node communication incurs cost. Thus, we need an efficient query processing method avoiding inter-node communication as much as possible.
- **Dynamic join:** When processing a multiple-keyword query, we need to join the posting lists of the keywords in the query to find documents containing all the query keywords. Here, the join among posting lists is a very expensive operation since we generally need to access a much larger number of postings than k for a top-k query from posting lists to find documents containing all the query keywords. Thus, we need to reduce the cost of dynamic join.

1.4 Our contributions

The goal of this paper is to show that, by carefully coordinating collective memories in multiple nodes of a search engine, we can achieve the same effect as one integrated memory in a single-node system. We summarize our contributions as follows.

- We propose a novel notion of *two-dimensional indexing* that simulates collective distributed memory as one integrated memory. Figure 2 shows the concept of providing a one-integrated-memory view of collective memories by using two-dimensional indexing over $n \times m$ two-dimensional distributed memory. Here, the horizontal axis of the array indicates n index shards that are partitioned from the entire index; the vertical axis in each column of the array indicates m index fragments that are partitioned from each index shard. The architecture supports parallel processing among n index shards based on document partitioning(to be defined in Section 2.2) and distributes the query load of each index shard to m index fragments based on keyword partitioning(to be defined in Section 2.2). We prove its usefulness with single-keyword queries (Section 3.3). The

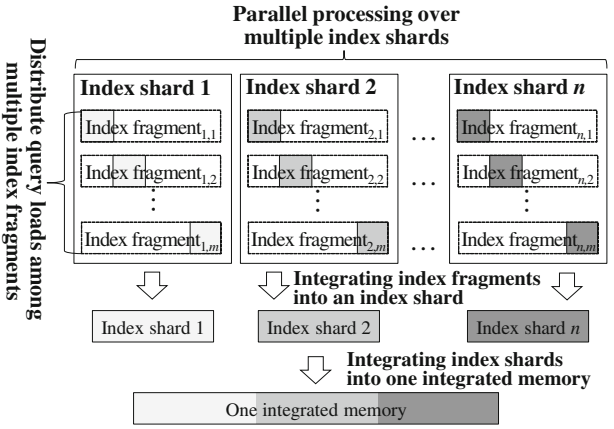


Figure 2 Integrated view of collective distributed memories as one memory

two-dimensional indexing architecture is scalable as it uses a shared-nothing architecture. Moreover, we can achieve low query response time as it allows processing queries in main memory.

– We newly propose the concept of the *one-memory point*, which is defined as the amount of memory space required for a system with distributed memory to completely store the entire index in main memory providing a one-integrated-memory view. We find the one-memory point of a system by examining a real data set (Section 4), and then, validate its correctness by comparing the examined result with the experimental result (Section 6). The significance of the one-memory point is that it allows us to estimate the total size of distributed memory required for building an in-memory massively-parallel search engine employing two-dimensional indexing, which can be an important practical guideline.
– We extend the notion of two-dimensional indexing so as to be able to handle multiple-keyword queries. We first propose the concept of two-dimensional indexing with single-keyword queries. However, in practical situations, we must handle multiple-keyword queries as well. To handle multiple-keyword queries in two-dimensional indexing, we propose the notion of *pre-join* that can handle a multiple-keyword query just like a single-keyword query (Section 5.1). Then, we propose the notion of *semi-memory join* that eliminates the costly inter-node communication among the nodes at a cost of some disk accesses (Section 5.2).
– We implement two-dimensional indexing in the ODYS search engine [32] building a system, ODYS/2D-Indexing. Through the experiments, we show that ODYS/2D-Indexing can provide a one-integrated-memory view for single-keyword queries only with 0.25% of additional memory space (i.e., one-memory point = 1.0025 $\times$ index size); for a real-world query set consisting of single-keyword and multiple-keyword queries with 10.97% of additional memory space (i.e., one-memory point = 1.1097 $\times$ index size). We further show that, in an ideal case when all the pre-joined multiple-keyword queries are in the in-memory buffer, ODYS/2D-Indexing significantly outperforms the ODYS disk-based search engine that do not use two-dimensional indexing [32] by 2.84 $\sim$ 6.05 times for single-keyword queries and 99.66 $\sim$ 535.54 times for real-world queries including single-keyword and multiple-keyword queries.

2 Preliminaries

In Section 2.1, we introduce recent trends and issues on in-memory systems. In Section 2.2, we briefly review the inverted index, and the partitioning methods of the inverted index.

2.1 Distributed main memory

Since the data transfer rate of RAM is orders of magnitude faster than that of the disk [24, 25], we can effectively improve the system performance by storing data in RAM. However, when we deal with large-scale data, it is not feasible to store the entire data into the memory of one machine since the size of RAM equipped in a single machine is limited. Therefore, the concept of distributed memory consisting of those of multiple machines has been introduced [23].

If the collective amount of distributed memory is larger than the size of data, we can store the entire data, in theory, into the distributed memory. However, since distributed memory is located over multiple nodes, the system using distributed memory, in general, does not have

the same performance as that of a single node system using one large integrated memory. This comes from the fact that query processing usually requires data stored in multiple nodes causing performance degradation due to the communication cost of inter-node joins.

Recently, many large-scale systems based on distributed memory have been developed. **In-Memory DBMSs** [9, 27] process queries by loading the necessary data into distributed memory. However, they do not deeply deal with the inter-node join problem in a distributed environment; they just adopt the traditional methods such as semi-join [3, 9] or replicating small tables to obviate inter-node communication [27]. Moreover, they neither support efficient information retrieval (IR) functionality such as an IR index, nor the architecture of partitioning/storing the IR index. **In-Memory Key-Value Stores** [17, 19] store data in the key-value format into distributed memory and accesses the data using a key uniquely assigned. It is suitable for storing data in a distributed manner. However, to model multiple normalized relations in the key-value format, we need to transform related relations into a de-normalized form (i.e., pre-joined). De-normalization renders the space utilization inefficient since a lot of data is duplicated. Moreover, we cannot use an index for a non-key attribute that is embedded in the value field. Thus, it is not suitable for processing search queries that have search conditions on arbitrary attributes. There are many other variants of in-memory key-value stores. **In-Memory Data Grid (IMDG)** [12, 18] stores data in distributed memory in a key-value format where the value is a serialized object (e.g., JVM objects). It inherits disadvantages of key-value stores. Similarly, **Spark** [34, 35] stores materialized intermediate results (i.e., RDDs) for iterative algorithms (e.g., MapReduce jobs, calculating PageRank) in a key-value format in distributed memory to accelerate the processing speed of those algorithms. They are appropriate for batch analytics using scanning of the whole data [35].

2.2 Inverted index

The inverted index is a representative index for searching the documents containing query keywords [6, 30]. Figure 3 shows the structure of an inverted index. The inverted index consists of a keyword index and posting lists. The keyword index is a B+-Tree indexing all the keywords in the entire set of documents where each keyword points to a posting list. A posting in the posting list represents the information for a document such as the document identifier (docID) and term frequency (TF). In the head of the posting list, we store the document frequency (DF), which is the frequency of documents containing the corresponding keyword among the entire set of documents.

A large-scale search engine partitions the inverted index to multiple components storing them in multiple nodes. There are two types of index partitioning: 1) document partitioning [14] and 2) keyword partitioning [14]. In *document partitioning*, we partition the inverted index using document identifiers as the partitioning criterion. Figure 4 shows an

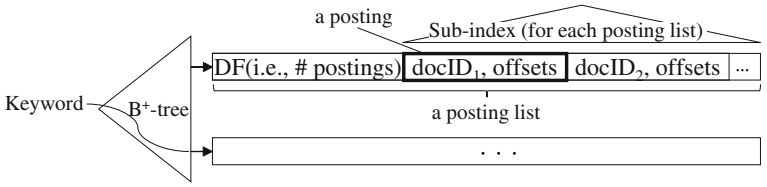


Figure 3 The structure of an inverted index [30]

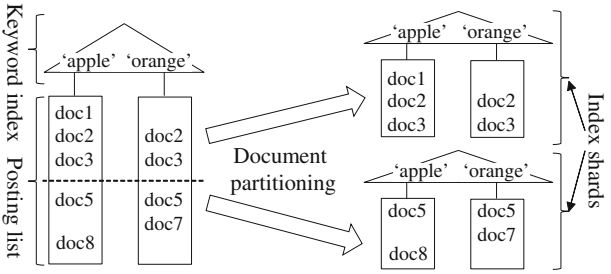


Figure 4 An example document partitioning

example document partitioning where an inverted index is partitioned into two components: 1) one for documents 1, 2, 3 and 2) another for documents 5, 7, 8. We call each partitioned index component obtained by document partitioning an *index shard*. To process a keyword query, we need to search all the shards for a complete answer. An advantage of document partitioning is that we can process a query in parallel by simultaneously accessing all the index shards. On the other hand, its disadvantage is that we must access all the index shards to process a query adversely affecting the throughput of the system and incurring an overhead of unioning the search results from all the shards.

In *keyword partitioning*, on the other hand, we partition the inverted index using keywords as the partitioning criterion. Figure 5 shows an example keyword partitioning where the inverted index is partitioned into two components: 1) one for documents containing 'apple' and 2) the other for those containing 'orange.' We call each partitioned index component obtained by keyword partitioning an *index fragment*. Here, to process each query, we need to access only those index fragments that correspond to the query keywords. Hence, the advantage of keyword partitioning is that the query load is naturally distributed over index fragments. Its disadvantage is that it does not allow us to process a single-keyword query in parallel among index fragments.

3 Two-dimensional indexing

Two-dimensional indexing is a two-dimensional ($n \times m$) way of storing the inverted index, partitioning it into n shards and partitioning each shard into m fragments. The purpose of

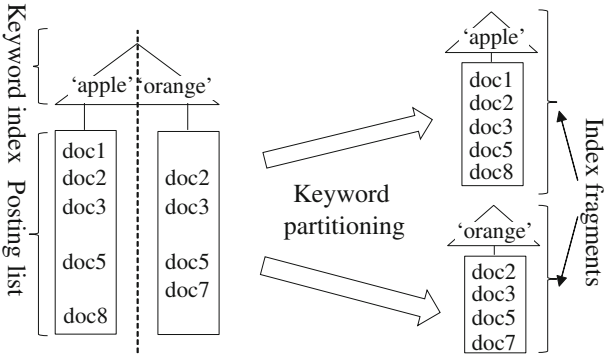


Figure 5 An example keyword partitioning

two-dimensional indexing is to provide a one-integrated-memory view of distributed memory storing the inverted index. In Section 3.1, we explain the architecture of two-dimensional indexing. In Section 3.2, we explain the concept. In Section 3.3, we present the query processing method for single-keyword queries in two-dimensional indexing.

3.1 Architecture

Figure 6 shows the architecture of two-dimensional indexing. It is a 3-level architecture consisting of one supermaster, n shard masters, and $n \times m$ slaves. We call a set of $n \times m$ slaves that store the entire index the *slave grid*. $\text{Slave}_{i,j}$ indicates the (i, j)th element in the slave grid where i indicates the number for the index shard and j the number for the index fragment. The supermaster manages all the slaves by communicating with the shard masters each of which, in turn, manages a column of the slave grid. Here, each supermaster and each shard master keeps meta information of nodes that they manage in their catalogs. In the global database catalog for a supermaster, we store the database schema and the IP addresses of the shard masters that the supermaster manages. In the local database catalog for each shard master, we store the database schema, the IP addresses of the slave nodes that the shard master manages, and the keyword ranges of those slave nodes.

3.2 Concept

In two-dimensional indexing, we employ document partitioning and keyword partitioning in a two-dimensional way, as we explained in Section 2.2. Hence, we partition the entire index in two-phases as shown in Figure 7. The first phase uniformly partitions the entire index into n index shards by document partitioning. The purpose of this phase is to enable parallel processing among index shards. The second phase partitions each index shard into m index fragments by keyword partitioning. The purpose of this phase is to distribute the query

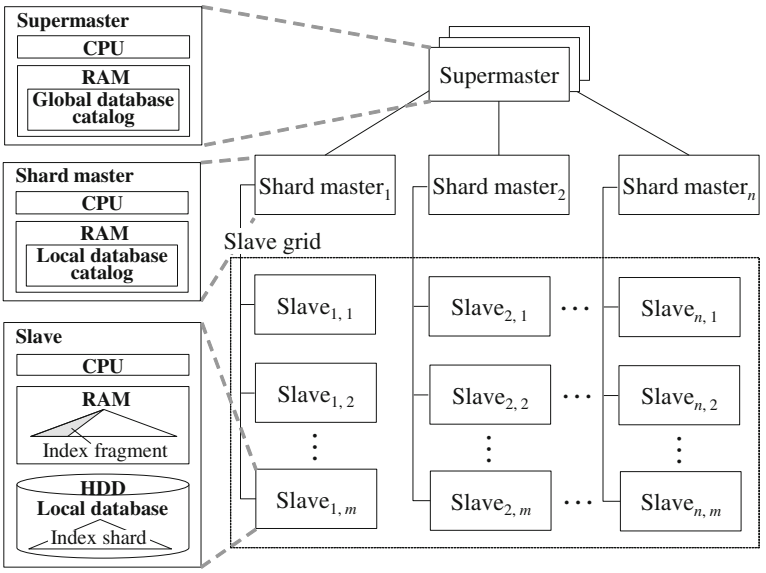


Figure 6 The architecture of two-dimensional indexing

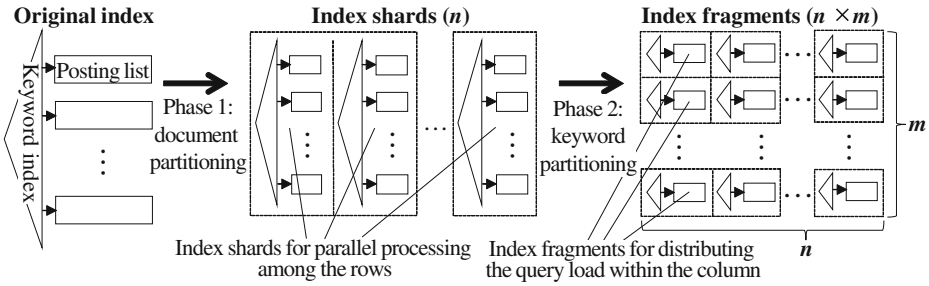


Figure 7 Two-dimensional indexing

load to the index fragments. Here, we partition each index shard into m index fragments so that each index fragment has the same number of postings.[3] Thus, we obtain the number of postings for each index fragment according to (1) where $n\text{-}total\text{-}postings$ is the total number of postings and m the number of slaves in each shard.

$$n\text{-}postings_{index\text{-}fragment} = \frac{n\text{-}total\text{-}postings}{m} \tag{1}$$

Then, we calculate the keyword range for each index fragment so that index fragments in an index shard are evenly responsible for disjoint keyword ranges, and the query load is evenly distributed among index fragments. We finally distribute the partitioned index fragments to the slaves and load them in main memory (i.e., DBMS buffer) of each slave. By allocating sufficient amount of main memory to each slave, we make sure those index fragments stay in main memory during query processing. Specifically, when we load the DBMS pages for the index fragment in the DBMS buffer, we fix them with the *LongFix* flag and control those pages not to be selected as a victim by the buffer replacement algorithm.[4]

We summarize the strength of two-dimensional indexing as follows.

- Two-dimensional indexing provides a one-integrated-memory view of distributed main memory. That is, it makes collective main memories of a large distributed system look like one large integrated main memory of a single system.
- Two-dimensional indexing is a new scalable in-memory IR indexing architecture for massively-parallel search engines, which has not been introduced in the literature before. Since most existing in-memory IR methods are based on a single-node, they are not scalable and are unsuitable for massively-parallel search engines. In contrast, the two-dimensional indexing architecture can be easily scaled-out by adding new machines as it stores the entire IR index into collective main memories of multiple machines.
- We can reduce dependency on cache servers for a search engine. The existing methods for massively-parallel search engines use a disk-based architecture and heavily rely on cache servers such as Memcached [17] at various levels (i.e., web server, index server, document server) to reduce the query response time. In contrast, two-dimensional

[3]For simplicity, we assume that all slave machines have the same amount of main memory. However, in the case where slave machines have different amounts of main memory, we can partition the index shard in proportion to the amount of main memory each slave has.

[4]The number of LongFixed pages is limited to 100% of the size of the index fragment. We allocate 2GB of more pages for the ordinary buffer without LongFix. Any page loaded after the LongFixed buffer is full is treated as an ordinary buffer page without LongFix. Thus, they may be swapped out later if necessary.

indexing can obtain a low query response time without cache servers as it processes the query efficiently using the inverted index where all the index fragments are stored in main memory.

– We can inherit the advantages of traditional index partitioning methods: parallelism and load distribution. Thus, we can efficiently process search queries on large-scale data in a massively-parallel manner.

3.3 Single-keyword query processing

Each (disjoint) index fragment in two-dimensional indexing represents a part of the entire index. Thus, we can select only those nodes that contain index fragments relevant for processing a specific query. The method of selecting those nodes depends on the index partitioning method used as we have explained in Section 2.2.

Figure 8 shows the processing method of a single-keyword query (e.g., Q_{orange}) using two-dimensional indexing. First, the supermaster copies the query Q_{orange} and distributes it to shard masters. Second, each shard master selects a slave to process the given keyword 'orange' by referring to the keyword ranges of the slaves and routes the query to the one selected. Later, the supermaster obtains the final result by unioning the results for the shard masters.

4 One-memory point

We store an index fragment generated by two-dimensional indexing into the memory space of each node. Since the sizes of index fragments vary, the total memory space requirement

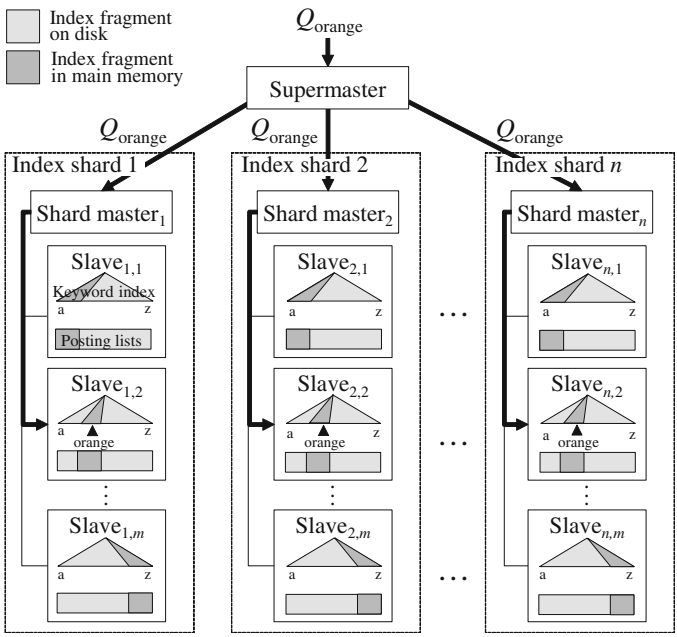


Figure 8 Processing a single-keyword query in two-dimensional indexing

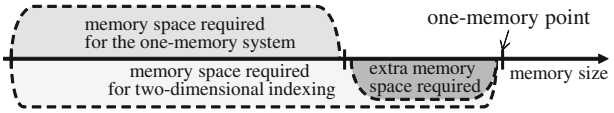


Figure 9 The concept of the one-memory point

tends to be larger than the size of the entire index. We first define the *one-memory system* as a single node system storing the entire index in main memory. Next, we define the *one-memory point* as the amount of memory space required for a system with distributed memory to completely store the entire index in main memory providing a one-integrated-memory view. We further illustrate this notion in Figure 9. If the memory space of the distributed system is lower than the one-memory point, the query processing speed will slow down since some posting lists are not properly loaded into main memory. Therefore, we should be able to project the exact one-memory point and provide sufficient amount of memory space to distributed nodes. At the one-memory point, two-dimensional indexing can provide nearly the same performance as that of the one-memory system.

We now investigate the one-memory point of two-dimensional indexing by examining the actual dataset we use. Table 1 shows the size of the index fragments for a set of 100 million documents. Specifically, the document set is partitioned into two shards each with 50 million documents evenly. Each shard is partitioned into index fragments where each index fragment has almost the same number of postings. While the sizes of index shards can vary since they are built with different document subsets, the sizes of index fragments within an index shard are almost the same since they are partitioned so that the postings are evenly distributed over fragments. Table 2 shows the one-memory point that is derived from Table 1. The size of the required memory for the one-memory system is calculated as $SUM(|fragment1|, ..., |fragment6|) = 459.75\text{GB}$ while that of two-dimensional indexing is calculated as $MAX(|fragment1|, ..., |fragment6|) \times nFragments = 460.90\text{GB}$ assuming that every slave has the same amount of main memory. One-memory point is calculated as $\frac{MAX(|fragment1|,...,|fragment6|)\times nFragments}{SUM(|fragment1|,...,|fragment6|)} = 100.25\%$. In other words, if we have 0.25% of additional memory, we can process every single-keyword query in main memory. As we showed, the one-memory point for single-keyword queries is close to 100% for a dataset of a large size and can even be computed statistically from the deviations in the uniformity of the size of index fragments partitioned. However, for multiple-keyword queries, the one-memory point can be much larger than 100%. We will discuss this issue in Section 5.4.

Table 1 The sizes of index fragments

Index fragment size(GB)					
Shard 1	Fragment 1	76.50	Shard 2	Fragment 4	76.42
	Fragment 2	76.70		Fragment 5	76.59
	Fragment 3	76.81		Fragment 6	76.70
Total		230.02	Total		229.72

Table 2 One-memory point

Required memory size(GB)	
One-memory system	459.75
Two-dimensional indexing	460.90
One-memory point	100.25%

5 Multiple-keyword query processing

In Sections 3 and 4, we have discussed the architecture and model of two-dimensional indexing in terms of only single-keyword queries. In practice, however, multiple-keyword queries represent a large portion of the query load in a search engine, and they must be incorporated in the model. In this section, we extend two-dimensional indexing so as to incorporate multiple-keyword queries. We first present two processing methods for multiple-keyword queries: *Pre-Join* and *Semi-Memory Join*. Then, we discuss the issues arising from the proposed methods: selection of multiple-keywords to be pre-joined, extra memory space required for multiple-keyword queries, required buffer size(i.e., one-memory point), and the expected hit ratio. We define the *hit ratio* as the probability that a query can be processed by accessing a pre-joined multiple-keyword set.

5.1 Pre-join

The basic idea of *pre-join* is that we join the posting lists for multiple keywords and store the results in the index in advance. This way, a multiple-keyword query can be processed just like a single-keyword query. We call a pre-calculated posting list a *pre-joined posting list*. We denote the memory space for storing the pre-joined posting lists by $Space_{pre\text{-}joined}$.

Figure 10 shows an example of pre-join for a multiple-keyword query consisting of two keywords. This process is just the same as a general process of joining posting lists of a multiple-keyword query. Then, we assign a unique keyword to the pre-joined posting list for storing and querying. We generate the unique keyword by concatenating the multiple keywords in a specific order. We note that using a simple alphabetical order will make the distribution of the posting lists highly skewed since a larger portion of the pre-joined posting lists will be assigned to the former slaves than the latter ones. To avoid this problem,

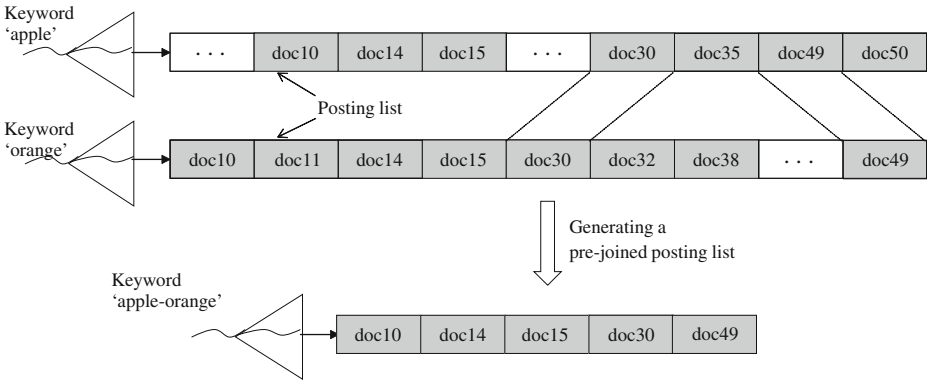


Figure 10 Generating a pre-joined posting list by joining two posting lists

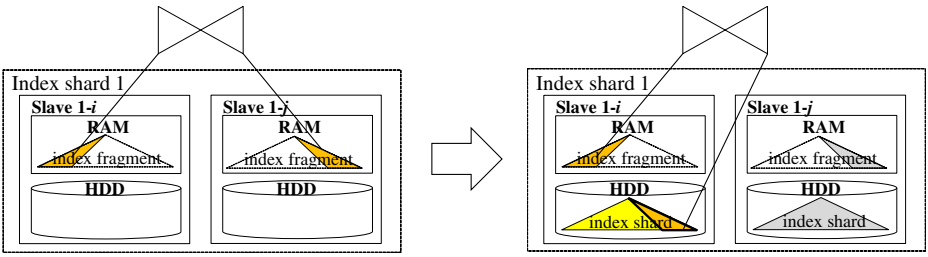


Figure 11 Semi-memory join

we generate the unique keyword by concatenating the multiple keywords by first applying a hash function, and then, ordering them in the order of resulting hash values. For example, a multiple-keyword consisting of two keywords 'apple' and 'banana' would be transformed into one composite-keyword 'banana-apple' if the hash value of 'apple' were bigger than that of 'banana'. This way, we can distribute the pre-joined posting lists evenly to slaves. Then, we insert the composite-keyword and the pre-joined posting list into the inverted index just like a single keyword. Thus, when processing the query, we do not need to distinguish the composite-keyword from the single-keyword as long as we transform the multiple-keyword query to a composite-keyword query by using the same algorithm for concatenation.

5.2 Semi-memory join

Processing a multiple-keyword query could easily incur the need to join multiple posting lists that are stored in fragments in different nodes, causing inter-node communication as we explained in Section 2.1. To avoid inter-node communication, we propose a join method that joins a posting list stored in main memory with another posting list stored in disk within the same node. We call this method the *semi-memory join*. To implement semi-memory join, we duplicate an entire index shard to disks of all the slaves in the same column of the slave grid shown in Figure 8.[5] Then, we join a posting list in the index fragment stored in main memory with another posting list in the index shard stored in disk in the same machine. Thus, query keywords that are included in the keyword range of the index fragment are processed in main memory while those outside the keyword range are processed by accessing disk (Figure 11). This way, we can avoid more costly inter-node communication at a lesser cost of disk access. As we explained in Section 3.2, we store the index fragment in the DBMS buffer with the LongFix flag. Thus, we can ensure the index fragment is not swapped out to disk even if we access posting lists stored in disk.

When we process a multiple-keyword query, we need to select a specific slave to process the query among the slaves storing the same index shard in disk. Since the slaves store the posting lists in different keyword ranges in main memory, the number of postings that are required to be read from the memory depends on which slave is selected. Naturally, it is

[5]The disk space required for duplication is proportional to the number of fragments (m) used in the column. These replicas play a role as replication servers for fault-tolerance as well as the semi-memory join. That is, even if a failure occurs on one machine in the column, the shard master can process any queries on disk by adjusting the keyword range. We note that it is common for large-scale search engines that adding search engines in parallel as replication servers to handle large volumes of search queries. We are exploiting those redundant replication servers to two-dimensional indexing.

the most efficient to select the slave that can read the largest number of postings from main memory. Thus, we select the slave having the largest collective number of postings of the query keywords. The collective number of postings can be obtained by summing up the document frequencies(DF) of the query keywords included in the index fragment, which can be easily obtained by referring to a hash table[6] of ⟨keyword, DF⟩ and the keyword range that are managed by the shard master.

5.3 Pre-joined multiple-keyword set

Although we can achieve a higher hit ratio if we use a larger pre-joined multiple-keyword set, the number of possible combinations for multiple-keywords is nearly infinite(2^n, n is the total number of distinct keywords in the document set) while the memory space for storing pre-joined posting lists is limited. Therefore, we cannot store the pre-joined posting lists for all the combinations of multiple-keywords in main memory. Thus, we need to select a set of multiple-keywords that have a high probability of being used frequently.

Since it is difficult to theoretically analyze the expected hit ratio according to the size of the pre-joined multiple-keyword set, we empirically examine a well-known query set, namely, the AOL search query set[7][22] and generate pre-joined multiple-keyword sets for simulating expected hit ratios. Specifically, we select a set of top-$X\%$ of multiple-keyword queries having high frequencies of occurrence from the AOL search query set. We call this set *MKSet-X*. Then, we generate the pre-joined posting lists of the multiple-keyword queries in *MKSet-X* and add them to the inverted index as composite-keywords. We then measure the memory space required for storing the pre-joined posting lists of *MKSet-X*, which we denote by $Space_{MKSet}(X)$. When processing other multiple-keyword queries not included in *MKSet-X*, we use semi-memory join. We now discuss issues related to *MKSet-X* and $Space_{MKSet}(X)$ in detail.

– ***MKSet-X*:** We find that most multiple-keyword queries can be processed in memory by using a relatively small size of *MKSet-X* since a small set of frequent queries covers most of the total query frequency. Figure 12 shows the distribution of the frequencies of the multiple keyword queries in the AOL search query set. The horizontal axis represents the ranking based on the occurrence frequency, and the vertical axis the occurrence frequency of each multiple keyword. As shown in the graph, the multiple-keyword queries used in the search engine follows the power-law distribution. The total frequency of the multiple keyword queries in *MKSet*-20, *MKSet*-30, and *MKSet*-40 over that of the entire set of the multiple-keyword queries in the AOL search query set is 66.45%, 74.68%, and 80.58% respectively. It means that if we select top 40% of multiple-keyword queries as a pre-joined multiple-keyword query set, we can achieve the hit ratio of 80.58% by using the pre-joined posting lists. In Section 6, we measure the query processing performance through experiments as X in *MKSet-X* is varied to show the effect of using using pre-join.

[6]The shard master initializes it by executing a query "SELECT keyword, nPostings FROM ⟨*inverted_index_name*⟩" that reads DFs of all posting lists (shown in Figure 3 in Section 2.2) in the slave database.

[7]The AOL(America On-line) search query set is a collection of search queries consisting of 35,020,000 queries collected from 650,000 users over 3 months. The portion of single-keyword queries is 36.43%. For other specific application domains, a similar query set can be collected over a period of time.

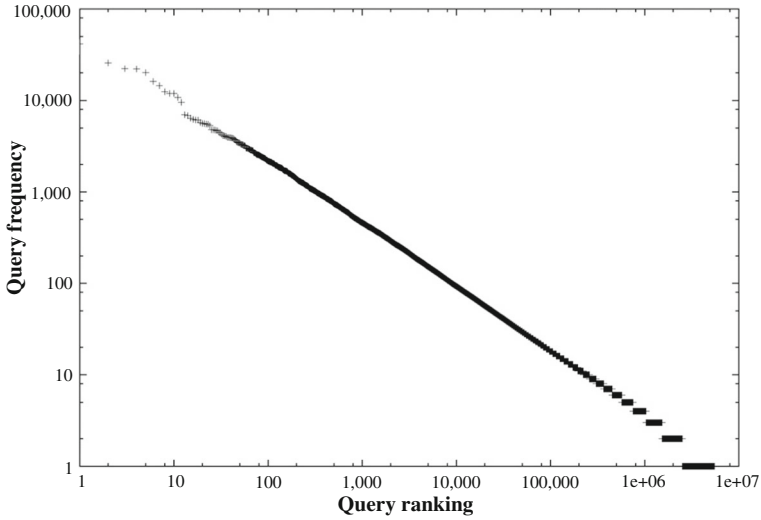


Figure 12 The distribution of AOL search query set[22]

– $Space_{MKSet}(X)$**:** At the first thought, one might think that $Space_{MKSet}(X)$ would be much smaller than the memory space for the entire index including all the single keywords since it is normally expected that a pre-joined posting list would be short being the intersection of the posting lists of multiple keywords. However, we find, in practice, that $Space_{MKSet}(X)$ is not small since each keyword in a popular multiple-keyword query 1) tends to have very a high document frequency (DF), and 2) tends to be highly correlated with one another. Thus, intersecting the posting lists does not significantly reduce the size of the result. Our strategy for solving this problem is to store only top-500 postings in the LongFixed buffer for pre-joined posting lists. Choosing a larger k value results in better search results but at the same time increases the space overhead. The minimum k value required to satisfy the users of the search engine, from the perspective of the quality of search results, tends to be determined by habitual behavior of the search engine user. Therefore, theoretical analysis of the appropriate k value is left as a future work. Instead, we directly examined the case of Google and selected the appropriate k value as 500.[8] This strategy is very space efficient with a marginal performance loss since most users are interested only in a few top results, and low-ranked results are rarely retrieved. In case we need more than top-500 results, the system just issues a normal multiple-keyword query rather than a composite-keyword query. Then, we can compute the result by dynamic join.

We now discuss the tradeoff between the size of *MKSet-X* and the performance. We can easily expect that a larger *MKSet-X* yields a better query processing performance, eventually outperforming the one-memory system, but instead, the space overhead(i.e., $Space_{MKSet}(X)$) will increase. Thus, we need to find the most effective *MKSet-X* by considering both the performance and the space overhead. In Section 6.2.2, we find the relevant

[8]Specifically, we tested the maximum number of search results that Google provided and observed that they provided up to top-500 search results.

Table 3 The size of index fragments with pre-joined posting lists(Top-500)

Index fragment size(GB)		*MKSet*-20	*MKSet*-30	*MKSet*-40	*MKSet*-60	*MKSet*-80	*MKSet*-100
Shard 1	Fragment 1	78.98	79.83	80.72	82.24	83.61	84.73
	Fragment 2	78.95	79.89	80.74	82.25	83.63	84.88
	Fragment 3	78.98	79.94	80.74	82.25	83.64	85.03
Total		236.91	239.66	242.21	246.74	250.87	254.64
Shard 2	Fragment 1	78.88	79.75	80.65	82.15	83.53	84.65
	Fragment 2	78.87	79.80	80.64	82.16	83.53	84.78
	Fragment 3	78.87	79.81	80.63	82.13	83.52	84.91
Total		236.62	239.36	241.91	250.58	252.52	254.34

X in *MKSet-X* that makes the performance of two-dimensional indexing equivalent to that of the one-memory system through experiments.

5.4 Discussion on one-memory point

In Section 4, we have defined the one-memory point as the amount of memory space required for a system with distributed memory to completely store the entire index in main memory providing a one-integrated-memory view. With multiple-keyword queries, the one-memory point become larger since index size increases to store pre-joined posting lists for composite-keywords, i.e., the pre-joined multiple-keyword set. We note that, from the perspective of the buffer size, the one-memory point implies the size of the buffer required to store the desired amount of pre-joined posting lists, which determines the hit ratio. In Section 5.5, we will discuss practical issues on the method of managing composite-keywords in a buffer of a limited size and the expected hit ratio.

Table 3 shows the size of the index fragments for 100 million documents with the pre-joined posting lists. Table 4 shows the one-memory point as *X* is varied. As we can see in Table 4, despite that we store pre-joined posting lists, the one-memory point is relatively low. For example, if we use *MKSet*-40, the one-memory point is 105.37%. In other words, if we allocate 5.37% of additional memory compared to the one-memory system, we can treat 80.58% of the multiple-keyword queries as single-keyword queries, and they can be processed in main memory. In Section 6.2, we will verify through experiments the correctness of the one-memory point examined.

We note that, as the size of the document set increases (say, beyond 100 million), we can store more top-*k* postings in main memory where *k* is bigger than 500 while maintaining a small one-memory point. Table 5 shows the trend of the index size as *k* in top-*k*

Table 4 One-memory point examined with pre-joined posting lists(Top-500)

Required memory size(GB)	*MKSet*-20	*MKSet*-30	*MKSet*-40	*MKSet*-60	*MKSet*-80	*MKSet*-100
One-memory system	459.75	459.75	459.75	459.75	459.75	459.75
ODYS/2D-Indexing	473.89	479.61	484.46	493.52	501.81	510.18
One-memory point	103.07%	104.32%	105.37%	107.35%	109.15%	110.97%

Table 5 The trend of the index size as k in top-k for the hot part varies

Index size(GB) for document sets		10 million	100 million
Top-10	Entire index(w/o pre-join)	44.52	459.75
	$Space_{MKSet}(X = 100)$	1.17	1.69
	MKSet ratio	2.62%	0.37%
Top-100	Entire index(w/o pre-join)	44.52	459.75
	$Space_{MKSet}(X = 100)$	7.09	13.16
	MKSet ratio	15.93%	2.86%
Top-1000	Entire index(w/o pre-join)	44.52	459.75
	$Space_{MKSet}(X = 100)$	31.80	83.27
	MKSet ratio	71.42%	18.11%

for the hot part varies. We observe that the ratio of $Space_{MKSet}(X)$ over the entire index size (*MKSet ratio*) stays in a similar range as both k in top-k and the size of the document set are increased by the same factor. For example, for the set of 10 million documents, the ratio of $Space_{MKSet}(X{=}100)$ for top-100 (top-10) pre-joined posting lists is 15.93% (2.62%). For the set of 100 million documents, the ratio is 18.11% (2.86%) for top-1000 (top-100). By extending this observation, we can project that, for a set of 1 billion (10 billion) documents, we can store top-10000 (top-100000) pre-joined posting lists without significantly increasing the *MKSet ratio*. Thus, we have sufficiently large k in top-k for a real-life large database so that we may additionally employ various query dependent ranking measures without limitation.

5.5 Load balancing and dynamic update of pre-joined multiple-keyword set

We have discussed on the concept of two-dimensional indexing. However, in a real-life search engine, the query keywords used in the search engine can dynamically change as the users' current interests change, and this might incur various performance optimization issues, which requires a lot of space to illustrate them. Therefore, here, we only briefly explain the issues and possible solutions leaving the details as a future work.

– **Dynamic Index Partitioning:**
  According to (1) in Section 3.2, we partition each index shard into multiple index fragments by the document frequency (DF) so that each index fragment has the same number of postings. In this scheme, the memory space is balanced among slaves, but the query loads are not since the query load is proportional to the frequency of queries processed in a slave, which we define as *Query Keyword Frequency (QKF)*. QKF dynamically changes as time passes since it reflects users' current interests. To accommodate this change, we can balance the query load among the slaves by dynamically adjusting the keyword ranges for the index fragments. If we partition the index shard by QKF, the query load among slaves is balanced. Instead, the memory space becomes unbalanced. Then, we should allocate to all the slaves the maximum of the memory requirement among index fragments. Thus, balancing the memory space and the query loads is a trade-off.

  We present the result of brief experiments in Section 6.2.1. If load balancing is employed, the performance is improved by 1.00% $\sim$ 90.55% as the arrival rate varies,

but 22.29% of more memory space is required. Thus, we conclude that load balancing can be employed without allocating an excessive amount of memory space.

– **Dynamic Update of In-Memory MKSet:**

In Sections 5.3 and 5.4, we have discussed on the selection of *MKSet-X* and have noted their one-memory points are relatively low. For instance, if we pre-join all the multiple-keywords occurring in the query set (i.e., *MKSet*-100), one-memory point is 110.97%. This indicates that, without much space overhead, we can pre-join a large number of the multiple-keyword queries and store them in main memory. However, in a real-life search engine, all the pre-joined multiple-keyword queries cannot be maintained in main memory since multiple-keyword queries that have not been issued and stored in main memory previously can be newly generated anytime. We define the set of multiple-keyword queries whose results have been brought into the in-memory buffer the *in-memory MKSet*. Thus, we need to dynamically update the in-memory MKSet with new multiple-keyword queries. We employ the Least Recently Used (LRU) policy for selecting victims to delete and updating the in-memory MKSet. This is easily implemented by enforcing a separate LRU policy only for the buffer pages that are LongFixed. Updating the in-memory MKSet involves dynamically creating the pre-joined posting lists of new multiple-keyword queries and inserting them as composite-keywords in the inverted index stored in disk, and unLongFixing from the buffer those composite keywords deleted.

Table 6 shows the the cost of dynamic updating the in-memory MKSet by the LRU policy. We process 24 multiple-keyword queries(consisting of 2 $\sim$ 4 keywords) retrieving top-500 results over a database consisting of 50 million Web documents and measure the average elapsed time of 1) deleting a pre-joined posting list, 2) inserting a pre-joined posting list, 3) processing dynamic join, and 4) total query processing time. As we can see in the experimental result, the time required for dynamic update of the inverted index and the in-memory MKSet is limited to tens of milliseconds, which is relatively small compared to that of dynamic join.

The expected buffer hit ratio and the performance of query processing depends on how often we update the in-memory MKSet with new multiple-keywords. By examining the AOL search query set, we find that dynamically updating the in-memory MKSet yields a hit ratio of 74.14% when we store queries accumulated for a three-month period in main memory. Moreover, we find that we can achieve a higher hit ratio if we have a larger(i.e., dense) query set during the same period. For instance, Google achieves typically 70 $\sim$ 90% of hit ratio using Google Global Cache (GGC) for various services including the search engine [11]. As illustrated in Table 7, we can use *MKSet*-30 for simulating the hit ratio of 74.68% (close to the hit ratio of the AOL query set), and *MKSet*-60 for simulating the hit ratio of 88.18% (close to the hit ratio of Google). In

Table 6 The cost of dynamic update of in-memory MKSet

# keywords in the query	# retrieved results	time elapsed (ms)			
		delete	insert	dynamic join	total
2	500.00	67.40	24.56	126.46	218.41
3	500.00	43.81	18.17	588.08	650.07
4	301.21	46.26	57.07	8702.08	8805.41

Table 7 The expected buffer hit ratio as *MKSet-X* varies

MKSet-X	*MKSet*-20	*MKSet*-30	*MKSet*-40	*MKSet*-60	*MKSet*-80	*MKSet*-100
Query freq. in *MKSet-X*	12,332,323	13,859,600	14,956,104	16,365,616	17,462,120	18,558,625
Query freq. in *MKSet-100*	18,558,625	18,558,625	18,558,625	18,558,625	18,558,625	18,558,625
Buffer size (one-memory point)	103.07%	104.32%	105.37%	107.35%	109.15%	110.97%
Hit ratio(%)	66.45%	74.68%	80.58%	88.18%	94.09%	100.00%

Section 6.2.3, we will simulate the performance of two-dimensional indexing by using both *MKSet*-30 and *MKSet*-60.

6 Performance evaluation

6.1 Experimental data and environment

Through experiments, we show two-dimensional indexing effectively provides the view of one integrated memory. We implement two-dimensional indexing in the ODYS search engine [32], which we call ODYS/2D-Indexing.[9] The experiments are organized in three parts:[10] 1) experiments for a set of single-keyword queries, 2) experiments for a set of real-world queries consisting of single-keyword queries and multiple-keyword queries, and 3) performance comparison with a disk-based search engine without two-dimensional indexing. For parts 1 and 2, we first show the effect of two-dimensional indexing and associated algorithms. Then, we find the one-memory point through experiments and compare the result with the examined result that we obtained in Section 4. In part 3, we compare ODYS/2D-Indexing with the ODYS search engine [32] with an equivalent amount of main memory buffer but without extension of two-dimensional indexing.

We have built a prototype of ODYS/2D-Indexing, One-Memory-System, and ODYS according to the architecture shown in Figure 13. Since we must use only one slave for One-Memory-System, for the fairness, we use instead the same number of CPU cores for each system (six cores with hyper-threading disabled). For ODYS/2D-Indexing and ODYS, we use six slave machines. Each slave is a Linux machine with a quad-core 2.4 GHz CPU (with only one core enabled) and 96 GB of main memory. Therefore, the slaves in the slave grid can store 576 GB of data in their collective main memory. Each slave has an internal RAID 5 disk array. The disk array has 10 disks (disk transfer rate: average 81.2 MB/s) with a total of 5 TB, a cache of 256 MB, and the 768 MB/s bandwidth. For One-Memory-System, we use one Linux machine with two quad-core 2.4 GHz CPUs (with only six cores enabled)

[9]The goal of the experiment is to show the net effect of applying two-dimensional indexing, but is not a comprehensive performance comparison of parallel search engines. Therefore, we used the same search engine(i.e., ODYS) to test the net effect of using two-dimensional indexing.

[10]Dynamic update of the multiple-keyword query set in the buffer is not incorporated in the experiments to obviate the need to recover the database to the initial state for a number of repeated experiments, which takes significant time. It takes only tens of milliseconds of additional time to dynamically update a multiple-keyword query in the in-memory *MKSet* as we have shown in Table 6.

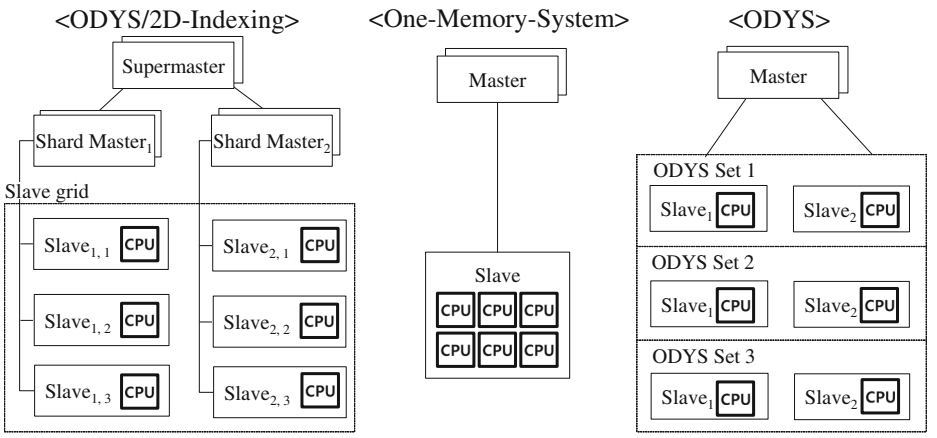


Figure 13 The architecture of ODYS/2D-Indexing, One-Memory-System, and ODYS used in the experiments

and 192 GB of main memory. For all slaves, we use Odysseus DBMS/search engine[11] with IR features tightly integrated into the DBMS [31]. In all these systems, each slave runs 100 Odysseus processes.

In ODYS/2D-Indexing, we use two supermasters and two shard masters to avoid their becoming the bottleneck and to focus on the performance of the slaves. We use two masters for One-Memory-System and ODYS for the same reasons. The supermasters (or masters) are Linux machines with a octa-core 3.00GHz CPU and 16 GB of main memory, and the shard masters are Linux machines with a quad-core 3.06 GHz CPU and 6 GB of main memory. All nodes are connected by a 1-Gbps network hub.

We use a dataset reduced from the one used in [32], which originally consists of 114 million Web documents crawled from all over the world. Table 8 shows the statistics of the reduced dataset. We partition the entire index into two index shards (i.e., $n = 2$) with each index shard duplicated in the disks of the slaves in the same column of the slave grid. We also partition each index shard into three index fragments (i.e., $m = 3$) with each index fragment stored in the main memory of a slave in the column of the slave grid.

For the query sets, from the AOL search query set [22], we generate two subsets consisting of 10,000 queries: AOL-QUERY-SUBSET(SK) and AOL-QUERY-SUBSET. We first generate the former, which consists of single-keyword queries only, by randomly selecting the ones among the single-keyword queries appearing in the AOL search query set. Then, we generate the latter, which is a mix of single-keyword and multiple-keyword queries, by randomly selecting the ones from the entire AOL search query set. We use AOL-QUERY-SUBSET to simulate real-world queries.

To simulate a steady-state environment, we first initialize by loading the entire index in main memory(i.e., DBMS buffer) of each machine. Specifically, for ODYS/2D-Indexing, we allocate a sufficient amount of main memory required to store the entire index as shown

[11] In ODYS [32], the authors show that a massively-parallel search engine can be built using a DBMS tightly integrated with IR features (Odysseus).

Table 8 The document set used in experiments

Document set	
Number of documents	100,000,000
Size of documents	667.96GB
Number of unique keywords	104,029,346
Number of total postings	20,568,950,762
Size of total postings	459.75GB
Average DF	197.72

in Tables 2 and 4 according to *MKSet-X* used in each experiment (e.g., 89.29GB for for each slave when we use *MKSet*-100: 85.03GB(510.18GB/6) for the index fragment + 2.26GB for the keyword index + 2GB for the ordinary buffer), and then, load posting lists into the DBMS buffer of each machine according to the keyword range assigned. For ODYS, we allocate memory and load the index in the same way as for ODYS/2D-Indexing, but we note that a large range of posting lists must be swapped out because an ODYS slave does not have sufficient memory to accommodate the (unpartitioned) entire index. In case of One-Memory-System, we were not able to store the entire index(459.75GB) in main memory due to our machine limitation that can accommodate only 192GB. To get around this problem, we loaded the entire index first, and then, run the query set a priori so as to ensure the set[12] of all the posting lists required by test queries are completely loaded in main memory.

In the experiments, we issue the queries in the test query set to each system and measure its average query response time. The query response time defined as the elapsed time from the time of the query arrived at the waiting queue of the supermaster to the time of the supermaster obtaining the top-k document IDs as the final query result. To simulate an environment of a real-life search engine, we generate queries with a Poisson arrival and issue them from a separate machine. We adjust the query load by varying the arrival rate. We retrieve the top-500 results for each query.[13]

6.2 Results of experiments

6.2.1 Comparison of ODYS/2D-indexing with one-memory-system(single-keyword queries)

In this section, we discuss the effect of two-dimensional indexing for single-keyword queries. We first show two-dimensional indexing can provide as close performance for single-keyword queries as that of One-Memory-System. Then, we find the one-memory point of ODYS/2D-Indexing through experiments and compare it with the analytic result derived in Section 5.4.

– **Effect of Two-Dimensional Indexing**

  Figure 14 shows average query response times of ODYS/2D-Indexing and One-Memory-System for the query set AOL-QUERY-SUBSET(SK) as the query arrival rate is varied from 1 to 90 million queries/day. In Figure 14, we note that the performance

[12]The size is much smaller (maximum 51.74GB) than those of the entire index(459.75GB) and the buffer(192GB).

[13]We use top-500 results to allow for additional query-dependent rankings (e.g., TF-IDF).

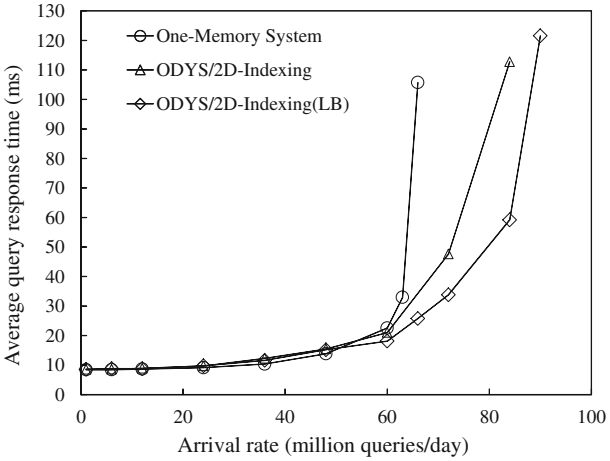


Figure 14 Performance of ODYS/2D-Indexing(w/w or w/o load balancing) vs. One-Memory-System for AOL-QUERY-SUBSET(SK)(single-keyword queries)

of ODYS/2D-Indexing is comparable to that of One-Memory-System. At higher arrival rates, we even see that ODYS/2D-Indexing shows a better performance (i.e., at over 60 million queries/day). This result is due to the difference in the architecture of resource sharing among CPU cores even if we use the same number of CPU cores for each system. Specifically, in ODYS/2D-Indexing, every CPU core has an independent memory channel and an L3 CPU cache. In One-Memory-System, on the other hand, multiple CPU cores have to competitively share those system resources, rendering some of them to finally become a bottleneck.

If we employ load balancing, which we call ODYS/2D-Indexing(LB), the average query response time is improved by 1.00% $\sim$ 90.55% over that of ODYS/2D-Indexing as the arrival rate is varied from 1 to 84 million queries/day. However, the total memory space required is also increased by 22.29%[14] since the maximum memory requirement of a slave increases (see Figure 15). We use ODYS/2D-Indexing for all other experiments to focus on the core concept of two-dimensional indexing.

– **One-Memory Point**

To measure the one-memory point, we find the specific memory amount where the query processing speed is saturated(flattened) with no further improvement as the memory size is varied. We conduct experiments at a low arrival rate (1 million queries/day) to obtain elaborate results. High arrival rates can distort the experimental results as some system component (e.g., the memory bus) becomes a bottleneck.

Figure 16 shows the one-memory point of ODYS/2D-Indexing for AOL-QUERY-SUBSET(SK). It is identified between 100% $\sim$ 105% of the memory requirement of One-Memory-System as the average query response time flattens in this range. This result coincides with the analytic result of 100.25% in Section 4.

[14]Since we let every machine have the same amount of main memory, the total required memory space of ODYS/2D-Indexing(LB) is calculated by 93.7GB $\times$ 6 = 562.2GB while that of One-Memory-System is 459.7GB.

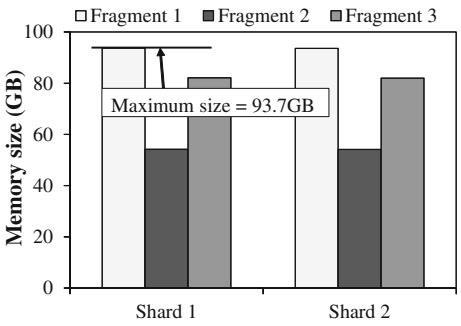


Figure 15 Required memory space when load balancing is employed

6.2.2 Comparison of ODYS/2D-indexing with one-memory-system(multiple-keyword queries)

For multiple-keyword queries, we use real-world queries (AOL-QUERY-SUBSET) containing both single-keyword and multiple-keyword queries. Here, we employ pre-join and semi-memory join. We first compare the performance of ODYS/2D-Indexing with that of One-Memory-System. Next, we measure the one-memory point of the former.

- **Two-Dimensional Indexing with Semi-Memory Join**
  Figure 17 shows the performance comparison of ODYS/2D-Indexing using only semi-memory join with One-Memory-System for AOL-QUERY-SUBSET. The average query response time of the former is 2.71 $\sim$ 12.04 times higher than that of the latter due to the disk access cost. We do not have figures for higher arrival rates since they exceed the system's maximum throughput.
- **Two-Dimensional Indexing with Pre-Join and Semi-Memory Join**
  Figure 18 shows the performance comparison of ODYS/2D-Indexing with pre-join and semi-memory join vs. One-Memory-System for AOL-QUERY-SUBSET. In

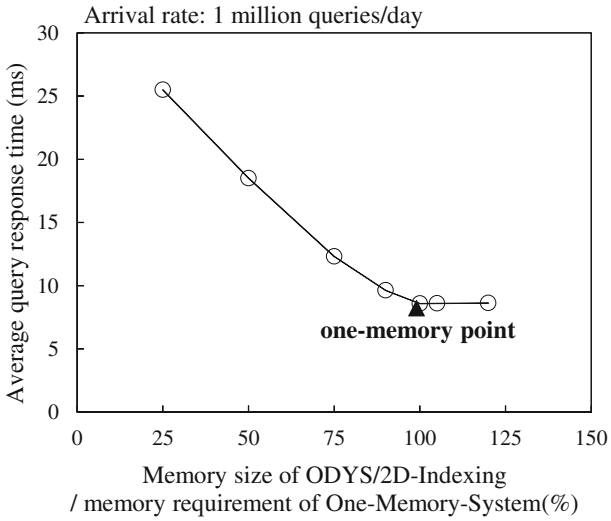


Figure 16 One-memory point of ODYS/2D-Indexing for AOL-QUERY-SUBSET(SK)

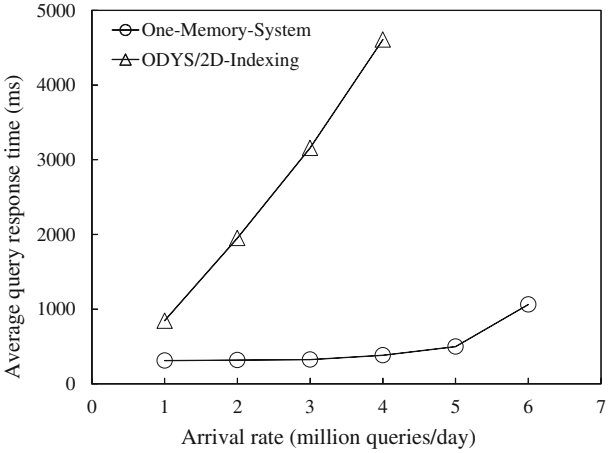


Figure 17 Performance of ODYS/2D-Indexing using only semi-memory join vs. One-Memory-System for AOL-QUERY-SUBSET

Figure 18, we observe that ODYS/2D-Indexing outperforms One-Memory-System as *X* in *MKSet-X* increases. One-Memory-System can process queries at a maximum of 6 million queries/day since the CPU cost is rapidly increasing as it processes dynamic join in main memory. On the other hand, the performance of ODYS/2D-Indexing is much more solid even at higher arrival rates since it avoids dynamic join by pre-joining a large portion of multiple-keyword queries. Moreover, when we use *MKSet*-20, ODYS/2D-Indexing shows a performance comparable to that of One-Memory-System. This result means that, if we pre-join only 20% of multiple-keyword queries, we can obtain a performance comparable to that of One-Memory-System having a large integrated memory.

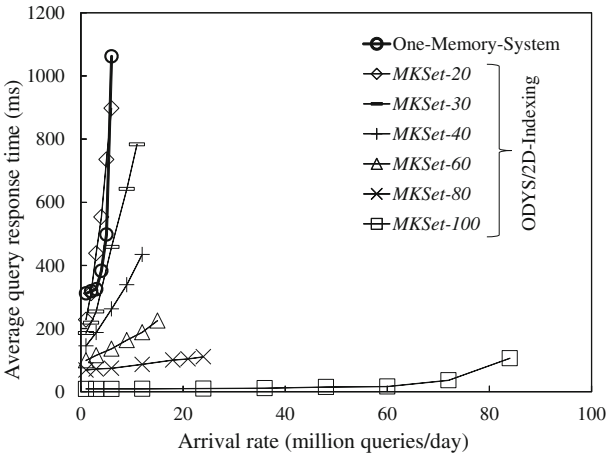


Figure 18 Performance of ODYS/2D-Indexing using both pre-join and semi-memory join vs. One-Memory-System for AOL-QUERY-SUBSET

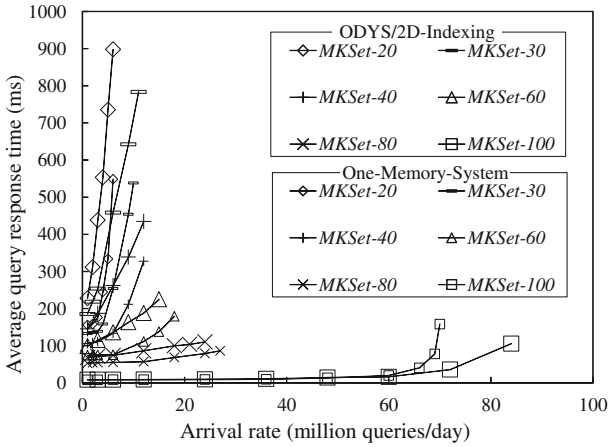


Figure 19 Performance of ODYS/2D-Indexing using both pre-join and semi-memory join vs. One-Memory-System using pre-join for AOL-QUERY-SUBSET (real-world queries)

Figure 19 shows the cases where both systems use pre-join. Here, we observe that One-Memory-System generally outperforms ODYS/2D-Indexing. This happens because ODYS/2D-Indexing needs to read posting lists from disk using semi-memory join while One-Memory-System reads all of them from main memory when the given multiple-keyword query is not in *MKSet-X*. In the case of *MKSet*-100, however, ODYS/2D-Indexing shows a performance better than that of One-Memory-System since it also can process every query in main memory as if it were a single-keyword query(see Figure 14). Considering the fact that One-Memory-System is infeasible with real-life big data where the data size certainly exceeds the main memory size of one machine, ODYS/2D-Indexing is very worthwhile for its fast performance and, more important, scalability. For comparison purpose, we use One-Memory-System without pre-join for the rest of this section.

– **One-Memory Point**

  Figure 20 shows the one-memory point of ODYS/2D-Indexing for AOL-QUERY-SUBSET at an arrival rate of 1 million queries/day. The one-memory point measured and the one examined are also consistent. In the case of *MKSet*-40, for example, it is between 105% and 110% coinciding with the examined result of 105.37% in Table 4, Section 5.4.

– **Finding a Relevant *MKSet-X***

  Figure 21 shows the average query response time at the one-memory point and the memory size required as X in *MKSet-X* varies for AOL-QUERY-SUBSET at an arrival rate of 1 million queries/day. It shows that, as X increases, the average query response time decreases while the memory size required increases. Thus, there is a trade-off between the two measures. In particular, at *MKSet*-20, the average query response time of ODYS/2D-Indexing becomes smaller than that of One-Memory-System. Thus, we can conclude that *MKSet*-20 is a minimum requirement to achieve a comparable performance as that of One-Memory-System. Obviously, if more memory space can be spared, we can achieve a better performance by increasing X as we have discussed in Section 5.5.

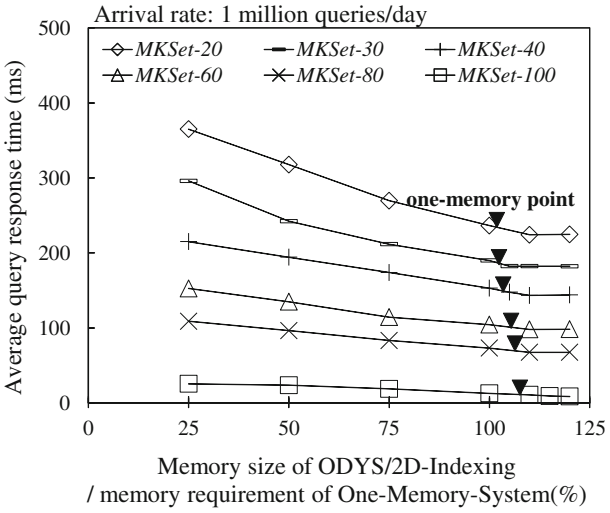


Figure 20 One-memory point for AOL-QUERY-SUBSET

6.2.3 Comparison of ODYS/2D-indexing with ODYS

In this section, we compare the performance of ODYS/2D-Indexing with that of ODYS [32]. We configure an ODYS system consisting of two shards and three ODYS Sets as illustrated in Figure 13 to make a fair comparison with ODYS/2D-Indexing.

Figure 22 shows the effect of two-dimensional indexing compared with ODYS for AOL-QUERY-SUBSET. Figure 22a shows average query response times of ODYS and ODYS/2D-Indexing when neither of them uses pre-join (i.e., using *MKSet-0*). The effect of two-dimensional indexing turns out to be 1.09 $\sim$ 2.33 times as the arrival rate varies 1 $\sim$ 2 million queries/day. Figure 22b shows the effect of two-dimensional indexing when we employ pre-join (i.e., using *MKSet*-100) in both systems, which turns out to be 2.84 $\sim$ 6.05

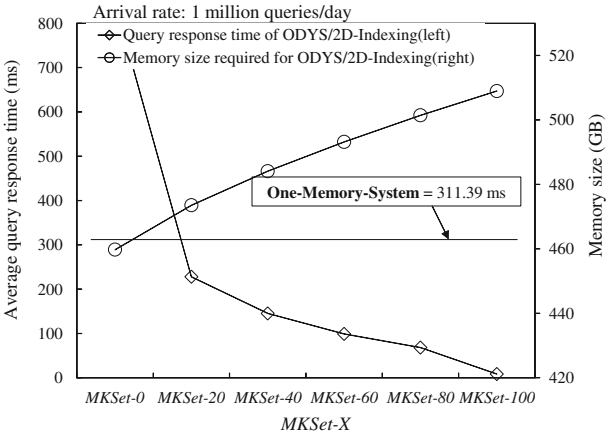
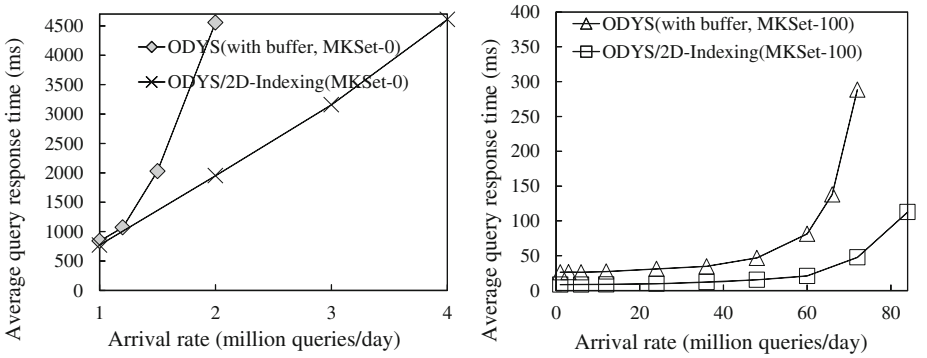


Figure 21 The trade-off between the average query response time and the memory size required



(a) With *MKSet*-0 (w/o pre-join employed) (b) With *MKSet*-100 (w/w pre-join employed)

Figure 22 Effect of Two-Dimensional Indexing for AOL-QUERY-SUBSET

times of performance improvement as the arrival rate varies $1 \sim 72$ million queries/day. We have these results because ODYS/2D-Indexing processes most queries in main memory while ODYS processes a large portion of queries in disk despite that ODYS uses the DBMS buffer whose collective size is equivalent to that of ODYS/2D-Indexing. We further expect this ratio will increase as we add more replicas to handle a much larger query load (say, 1 billion queries/day) since, in this case, the buffer size of an individual slave of ODYS would become much smaller degrading the performance of ODYS. In contrast, the performance of ODYS/2D-Indexing will not degrade since the keyword range of an individual slave is also proportionally reduced.

Now, we finally compare the overall performance of ODYS/2D-Indexing with that of ODYS. Figure 23 shows the performance comparison for AOL-QUERY-SUBSET(SK); Figure 24 the comparison for AOL-QUERY-SUBSET. In Figure 24, for ODYS/2D-Indexing, we use *MKSet*-20, which makes the performance equal to that of One-Memory-System, *MKSet*-60, which simulates the buffer hit ratio of a practical search engine (i.e.,

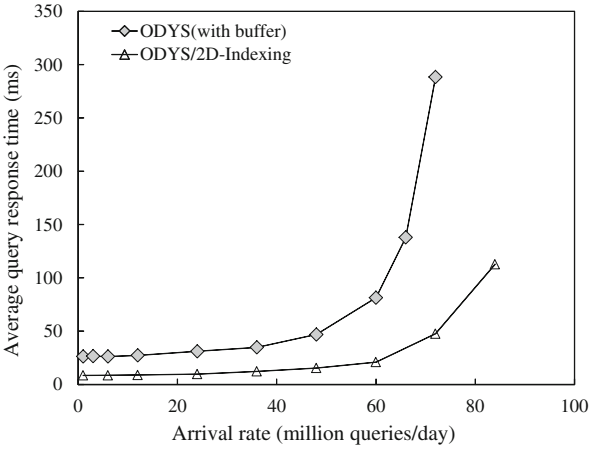


Figure 23 Performance Comparison of ODYS/2D-Indexing with ODYS for AOL-QUERY-SUBSET(SK)

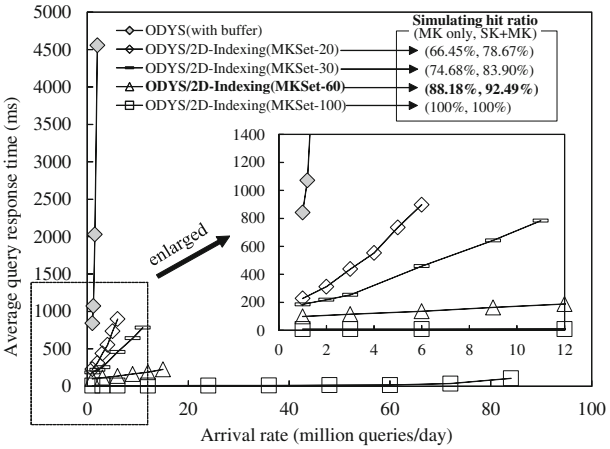


Figure 24 Performance Comparison of ODYS/2D-Indexing using *MKSet-X* (X = 20, 30, 60, 100) with ODYS for AOL-QUERY-SUBSET

the buffer hit ratio = 88.18%[15]), and *MKSet*-100, which represents the ideal case where the search engine keeps every multiple-keyword query in the in-memory buffer as a pre-joined composite-keyword query (i.e., the buffer hit ratio = 100%).[16] Figures 23 and 24 show the experimental results. Figure 23 shows that ODYS/2D-Indexing outperforms ODYS by 2.84 $\sim$ 6.05 times for AOL-QUERY-SUBSET(SK). Figure 24 shows that ODYS/2D-Indexing outperforms ODYS that does not use pre-join by 3.69 $\sim$ 14.62 times for *MKSet*-20, by 8.51 $\sim$ 39.57 times for *MKSet*-60, and by 99.66 $\sim$ 535.54 times for *MKSet*-100 for AOL-QUERY-SUBSET.

7 Conclusions

In this paper, we have proposed a scalable in-memory IR indexing architecture of *two-dimensional indexing* that aims at providing the one-integrated-memory view of collective memories distributed over multiple slave nodes. We have implemented the two-dimensional indexing architecture in ODYS—namely, ODYS/2D-Indexing—which is a massively-parallel in-memory search engine using distributed memory. For two-dimensional indexing, we partition the entire index into a two-dimensional array of multiple index fragments and completely store them into main memories of the multiple machines. As a result, two-dimensional indexing achieves low query response time as it processes all the queries in main memory. Moreover, as two-dimensional indexing inherits the merits of traditional index partitioning methods—parallelism and load distribution—we can efficiently process search queries on large-scale data in a massively-parallel manner, and can effectively distribute the query loads.

We have also proposed the new concept of the one-memory point, which is the amount of memory space required for a system with distributed memory to completely store the entire

[15]The values are calculated when we use multiple-keyword queries only. If we consider single-keyword queries together, which always achieves 100% of hit ratio, the overall hit ratio becomes 92.49%.

[16]We note that Google achieves typically 70 $\sim$ 90% of hit ratio using Google Global Cache(GGC)[11].

index in main memory providing a one-integrated-memory view. We have investigated the one-memory point of ODYS/2D-Indexing by examining an actual dataset and comparing with the result of experiments, which largely coincide with each other.

Then, we have extended the notion of two-dimensional indexing to accommodate multiple-keyword queries. We have proposed two efficient multiple-keyword query processing methods in distributed memory. Pre-join reduces the cost of dynamic join by treating multiple-keyword queries as single-keyword ones. Semi-memory join guarantees elimination of costly inter-node communication at a lesser cost of local disk access.

Through experiments, we have shown that ODYS/2D-Indexing can provide an effect comparable to or better than that of One-Memory-System, a system with one large integrated main memory. We have further shown that ODYS/2D-Indexing significantly outperforms ODYS[32] that has an equivalent amount of main memory buffer but without extension of two-dimensional indexing. For real-world queries(i.e., AOL-QUERY-SUBSET), ODYS/2D-Indexing outperforms ODYS by 3.69 $\sim$ 535.54 times as X in *MKSet-X* varies from 20 to 100. We expect this improvement will further increase as we handle a much larger data set (equivalently, as the number of replicas in each shard increases).

Acknowledgements This work was supported by the National Research Foundation of Korea(NRF) grant funded by Korean Government(MSIT) (No. 2016R1A2B4015929).

References

1. Baeza-Yates, R., Gionis, A., Junqueira, F., Murdock, V., Plachouras, V., Silvestri, F.: The impact of caching on search engines. In: Proceedings of the 30th Int'l Conference on Information Retrieval (SIGIR), pp. 183–190 (2007)
2. Baeza-Yates, R., Gionis, A., Junqueira, F., Murdock, V., Plachouras, V., Silvestri, F.: Design trade-offs for search engine caching. ACM Transactions on the Web (TWEB) **2**(4), 1–28 (2008)
3. Bernstein, P., Chiu, D.: Using semi-joins to solve relational queries. J. ACM **28**(1), 25–40 (1981)
4. Ceccarelli, D., Lucchese, C., Orlando, S., Perego, R., Silvestri, F.: Caching query-biased snippets for efficient retrieval. In: Proceedings of the 14th Int'l Conference on Extending Database Technology (EDBT), pp. 93–104 (2011)
5. Culpepper, J., Petri, M., Scholer, F.: Efficient in-memory top-k document retrieval. In: Proceedings of the 35th Int'l Conference on Information Retrieval (SIGIR), pp. 225–234 (2012)
6. Cutting, D., Pedersen, J.: Optimization for dynamic inverted index maintenance. In: Proceedings of the 13th ACM Int'l Conference on Information Retrieval (SIGIR), pp. 405–411 (1990)
7. Dean, J.: Building Software Systems at Google and Lessons Learned, Stanford Computer Science Department Distinguished Computer Scientist Lecture, Nov. 2010. (presentation slides available at http://research.google.com/people/jeff/Stanford-DL-Nov-2010.pdf)
8. Fagni, T., Perego, R., Silvestri, F., Orlando, S.: Boosting the performance of web search engines caching and prefetching query results by exploiting historical usage data. ACM Transactions on Information Systems (TOIS) **24**(1), 51–78 (2006)
9. Färber, F., et al.: The SAP HANA database - an architecture overview. IEEE Data Eng. Bull. **35**(1), 28–33 (2012)
10. Gan, Q., Suel, T.: Improved techniques for result caching in web search engines. In: Proceedings of the 18th Int'l Conference on World Wide Web (WWW), pp. 431–440 (2009)
11. Google Global Cache: https://peering.google.com/about/ggc.html, https://peering.google.com/about/faq.html (referenced in Jan. 2016)
12. IBM WebSphere eXtreme Scale: http://www.ibm.com/software/products/en/websphere-extreme-scale (referenced in Jan. 2018)
13. Internet Live Stats: http://www.internetlivestats.com/google-search-statistics (referenced in Jan. 2018)

14. Jung, B., Omiecinski, E.: Inverted file partitioning schemes in multiple disk systems. IEEE Trans. Parallel Distributed Syst. **6**(2), 142–153 (1995)
15. Kunder, M.: http://www.worldwidewebsize.com (referenced in Jan. 2018)
16. Markatos, E.: On caching search engine query results. Comput. Commun. **24**(2), 137–143 (2001)
17. Memcached - A Distributed Memory Object Caching System, http://memcached.org
18. Oracle Coherence, http://www.oracle.com/technetwork/middleware/coherence
19. Ousterhout, J. et al.: The case for RAMClouds: scalable high-performance storage entirely in DRAM. In: ACM SIGOPS Operating Systems Review, vol. 43, pp. 92–105 (2010)
20. Ozcan, R., Altingovde, I., Ulusoy, Ö.: Static query result caching revisited. In: Proceedings of the e17th Int'l Conference on World Wide Web (WWW), pp. 1169–1170 (2008)
21. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank citation ranking: bringing order to the Web, technical report(SIDL-WP-1999-0120) Stanford University (1999)
22. Pass, G., Chowdhury, A., Torgeson, C.: A picture of search. In: Proceedings of the 1st ACM Int'l Conference on Scalable Information Systems, Article No. 1 (2006)
23. Protic, J., Tomasevic, M., Milutinović, V.: In Book Distributed Shared Memory-Concepts and Systems. Wiley, New York (1998)
24. Samsung Semiconductor, http://www.samsung.com/semiconductor/global/file/insight/2015/08/DDR4_Brochure_July2015-0.pdf
25. Seagate, http://www.seagate.com/internal-hard-drives/desktop-hard-drives/desktop-hdd/#specs
26. Skobeltsyn, G., Junqueira, G., Plachouras, V., Baeza-Yates, R.: Resin: a combination of results caching and index pruning for high-performance Web search engines. In: Proceedings of the 31th Int'l Conference on Information Retrieval (SIGIR), pp. 131–138 (2008)
27. Stonebraker, M., Weisberg, A.: The voltDB main memory DBMS. IEEE Data Eng. Bull. **36**(2), 21–27 (2013)
28. Strohman, T., Croft, W.: Efficient document retrieval in main memory. In: Proceedings of the 30th Int'l Conference on Information Retrieval (SIGIR), pp. 175–182 (2007)
29. Turpin, A., Tsegay, Y., Hawking, D., Williams, H.: Fast generation of result snippets in web search. In: Proceedings of the 30th Int'l Conference on Information Retrieval (SIGIR), pp. 127–134 (2007)
30. 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, Feb. 19, 2002, Application No. 09/250,487 (1999)
31. Whang, K., Lee, M., Lee, J., Han, W.: Odysseus: a high-performance ORDBMS tightly-coupled with IR features. In: Proceedings of the 21st Int'l Conference on Data Engineering (ICDE), pp. 1104–1105 (2005)
32. Whang, K., Yun, T., Yeo, Y., Song, I., Kwon, H., 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: Proceedings of the 2013 ACM Int'l Conference on Management of Data (SIGMOD), pp. 313–324 (2013)
33. Whang, K., Lee, J., Lee, M., Han, W., Kim, M., Kim, J.: DB-IR Integration using tight-coupling in the Odysseus DBMS. The World Wide Web J **18**(3), 491–520 (2015)
34. Xin, R., Xin, R., Rosen, J., Zaharia, M., Franklin, M.J., Shenker, S., Stoica, I., et al.: Shark: SQL and rich analytics at scale. In: Proceedings of the 2013 ACM Int'l Conference on Management of Data (SIGMOD), pp. 13–24 (2013)
35. Zaharia, M.: An architecture for fast and general data processing on large clusters, PhD Dissertation, University of California, Berkeley (2013)

Affiliations

Tae-Seob Yun[1] · Kyu-Young Whang[1] · Hyuk-Yoon Kwon[2] · Jun-Sung Kim[1] · Il-Yeol Song[3]

Tae-Seob Yun
tsyun@mozart.kaist.ac.kr

Hyuk-Yoon Kwon
hyukyoon.kwon@seoultech.ac.kr

Jun-Sung Kim
jskim@mozart.kaist.ac.kr

Il-Yeol Song
songiy@drexel.edu

1 Department of Computer Science, KAIST, Daejeon, Korea

2 Department of Global Fusion Industrial Engineering, Seoul National University of Science and Technology, Seoul, Korea

3 College of Information Science and Technology, Drexel University, Philadelphia, PA 19104, USA