ODYS: An Approach to Building a Massively-Parallel Search Engine Using a DB-IR Tightly-Integrated Parallel DBMS for Higher-Level Functionality

Kyu-Young Whang[†], Tae-Seob Yun[†], Yeon-Mi Yeo[†], Il-Yeol Song[‡], Hyuk-Yoon Kwon[†], In-Joong Kim[†]
[†]Department of Computer Science, Korea Advanced Institute of Science and Technology (KAIST)
Daejeon, Korea
{kywhang, tsyun, ymyeo, hykwon, ijkim}@mozart.kaist.ac.kr

[‡]College of Information Science and Technology, Drexel University
Philadelphia, USA
songiy@drexel.edu

ABSTRACT

Recently, parallel search engines have been implemented based on scalable distributed file systems such as Google File System. However, we claim that building a massively-parallel search engine using a parallel DBMS can be an attractive alternative since it supports a higher-level (i.e., SQL-level) interface than that of a distributed file system for easy and less error-prone application development while providing scalability. Regarding higher-level functionality, we can draw a parallel with the traditional O/S file system vs. DBMS. In this paper, we propose a new approach of building a massively-parallel search engine using a DB-IR tightly-integrated parallel DBMS. To estimate the performance, we propose a hybrid (i.e., analytic and experimental) performance model for the parallel search engine. We argue that the model can accurately estimate the performance of a massively-parallel (e.g., 300-node) search engine using the experimental results obtained from a small-scale (e.g., 5-node) one. We show that the estimation error between the model and the actual experiment is less than 2.13% by observing that the bulk of the query processing time is spent at the slave (vs. at the master and network) and by estimating the time spent at the slave based on actual measurement. Using our model, we demonstrate a commercial-level scalability and performance of our architecture. Our proposed system ODYS is capable of handling 1 billion queries per day (81 queries/sec) for 30 billion Web pages by using only 43,472 nodes with an average query response time of 194 ms. By using twice as many (86,944) nodes, ODYS can provide an average query response time of 148 ms. These results show that building a massively-parallel search engine using a parallel DBMS is a viable approach with advantages of supporting the high-level (i.e., DBMS-level), SQL-like programming interface.

SIGMOD'13 June 22–27, 2013, New York, New York, USA.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Performance evaluation (efficiency and effectiveness), Distributed systems*; C.4 [**Computer Systems Organization**]: Performance of Systems—*Measurement techniques, Modeling techniques*

Keywords

massively-parallel search engines, parallel DBMSs, DB-IR tight integration

1. INTRODUCTION

1.1 Motivation

A Web search engine is a representative large-scale system, which handles billions of queries per day for a petabyte-scale database of tens of billions of Web pages [9, 19]. Until now, commercial Web search engines have been implemented based on a scalable distributed file system such as Google File System (GFS) [12] or Hadoop Distributed File System (HDFS) [16]. These distributed file systems are suitable for large-scale data because they provide high scalability using a large number of commodity PCs. A storage system proposed for real-world scale data with better functionality is the key-value store. It stores data in the form of a key-value map, and thus, is appropriate for storing a large amount of sparse and structured data. Representative key-value stores are Bigtable [6], HBase [15], Cassandra [5], Azure [2], and Dynamo [11]. These systems are based on a large-scale distributed storage such as a distributed file system [6, 15] or a distributed hash table(DHT) [2, 5, 11].

However, both distributed file systems and key-value stores, the so-called "NoSQL" systems, have very simple and primitive functionality because they are low-level storage systems. In other words, they do not provide database functionality such as SQL, schemas, indexes, or query optimization. Therefore, to implement high-level functionality, developers need to build them using low-level primitive functions. Research for developing a framework for efficient parallel processing of large-scale data in large storage systems has been

proposed. MapReduce [8] and Hadoop [14] are the examples of parallel processing frameworks. These frameworks are known to be suitable for performing extract-transform-load (ETL) tasks or complex data analysis. However, they are not suitable for query processing on large-scale data because they are designed for batch processing and scanning of the whole data [27]. Thus, commercial search engines use these frameworks primarily for data loading or indexing instead of user query processing.

High-level functionalities such as SQL, schemas, or indexes that are provided by the DBMS allow developers to implement queries that are used in search engines easily because they provide a higher expressive power than primitive functions in key-value stores, facilitating easy (and much less error-prone) application development and maintenance. In this sense, there have been many research efforts to support SQL even in the NoSQL systems. Fig. 1 shows the representative queries used in search engines that can be easily specified using the high-level functionality. Fig. 1 (a) shows a schema of a relation *pageInfo* that represents the information of Web pages. Fig. 1 (b) shows a SQL statement that represents a keyword query. The query finds the Web pages that contain the word "Obama" from the *pageInfo* relation. Fig. 1 (c) shows a SQL statement that represents a site-limited search. *Site-limited search* limits the scope of a user query to the set of Web pages collected from a specific site [29]. The query finds the Web pages that contain the word "Obama" from the site having *siteId* 6000. Fig. 1(d) shows one of its optimized versions. It requires an index-level join operation on *docId* with text predicates involving *contents* and *siteIdText* attributes (see Fig. 4(a) in Section 2).

Attribute Name	Attribute Type	Description
docId	integer	System defined identifier
pageId	integer	Page identifier
siteId	integer	Site identifier
siteIdText	text	Site identifier
title	text	Page title
URL	varchar	Page URL
content	text	Page content

(a) *pageInfo* relation.

```
SELECT p.pageId
FROM pageInfo p
WHERE MATCH(p.content, "Obama")>0;
```

(b) SQL statement for keyword search

```
SELECT p.pageId
FROM pageInfo p
WHERE MATCH(p.content, "Obama")>0
AND p.siteId = 6000;
```

(c) SQL statement for *site-limited search.*

```
SELECT p.pageId
FROM pageInfo p
WHERE MATCH(p.content, "Obama")>0 AND MATCH(p.siteIdText, "6000")>0;
```

(d) An optimized vertion of SQL statement for *site-limited search.*

Figure 1: An example of a schema and SQL statements.

These high-level functionalities allow us to easily develop advanced search engines with multiple search fields such as on-line discussion board systems as shown in Fig. 2 (a). The presented advanced search involves multiple fields as well as community-limited search capability. It requires complex index-level join operations among multiple fields, which require an implementation with high-level complexity. However, SQL allows us to implement those operations with a simple specification. Moreover, an index can be defined on any column by a simple declaration using SQL so that these complex index-level joins on multiple columns can be processed efficiently. Fig. 2 (b) shows a simple SQL statement for an advanced search that requires a four-way index-level join on *docId* with text predicates involving *title*, *content*, *communityIdText*, and *reg_date* attributes. Likewise, other search related applications can be easily developed by using SQL.

A parallel DBMS is a database system that provides both storage and parallel query processing capabilities. It could be considered an alternative to a large-scale search engine

(a) Advanced search with multiple fields.

```
SELECT p.pageId
FROM page p
WHERE MATCH(p.title, "database") > 0
AND MATCH(p.content, "index") > 0
AND MATCH(p.communityIdtext, "3") > 0
AND p.reg_date>="2001-01-01";
```

(b) An SQL statement for an advanced search using attribute embedding.

Figure 2: An example of advanced search.

because it has higher scalability and performance than traditional single node DBMSs and also has rich functionality such as SQL, schemas, indexes, or query optimization. Stonebreaker et al. [27] argue that parallel DBMSs are scalable enough to handle large-scale data and query loads. They claim that parallel DBMSs are linearly scalable and can easily service multiple users for database systems with multi-petabytes of data. However, parallel DBMSs have been considered as not having enough performance and scalability to be used as a large-scale search engine [1, 8], one outstanding reason being the lack of efficient information retrieval (IR) functionality.

To enable a DBMS to efficiently handle keyword search, tight integration of database (DB) and information retrieval (IR) functionalities has been proposed [28, 29]. The tight DB-IR integration implements IR functionality within the core of a DBMS, and thus, IR queries become efficient due to short access paths to the IR functionality. Two techniques for providing DB-IR integrated queries also have been proposed: 1) IR index join with posting skipping [28, 29] and 2) attribute embedding [29] to be explained in Section 2.

1.2 Our Contributions

In this paper, we make the following three contributions. First, we show that we can construct a commercial-level massively parallel search engine using a parallel DBMS, which to date has not been considered practical. Our proposed architecture, featuring a shared-nothing parallel DBMS, consists of masters and slaves. Our system, ODYS, following the proposed architecture achieves commercial-level scalability and efficiency by using Odysseus, which features DB-IR tight integration [28, 29], as its slaves. We have verified that each Odysseus is capable of indexing 100 million Web pages (loading and indexing in 9.5 days in a LINUX machine[1]), and thus, ODYS is capable of supporting a large volume of data with a small number of machines. Furthermore, the DB-IR tight integration enables the system to efficiently process a large number of queries arriving at a very fast rate. We show that ODYS can achieve a commercial-level performance especially for single-keyword searching, which is the most representative query.

Second, we propose an analytic and experimental performance model (simply, a *hybrid model*) that estimates the performance of the proposed architecture of the parallel DBMS, and then, validate the accuracy of the model. We argue that this model can accurately estimate the performance of a massively-parallel engine using the experimental results obtained from a small-scale one. For the master and network, we model each system component using the queuing model, and then, estimate the performance. For the slave, we propose an experimental method for accurately predicting the performance of a scaled-out (e.g., 300-node) system

[1]The machine is with a quad-core 2.5 GHz CPU, 4 Gbytes of main memory, and a RAID 5 disk having 13 disks (disk transfer rate: avg. 83.3 Mbytes/s) with a total of 13 Tbytes, a cache of 512 Mbytes, and 512 Mbytes/s bandwidth.

using a small-scale (e.g., 5-node) one. We note that the bulk (say, 92.28% ∼ 96.43%) of the query processing time is spent at the slave compared with the master and network. Our experimental method ensures high predictability of the slave side query processing time (which contributes most of the total processing time) since the estimation is directly derived from actual measurement. To verify the correctness of the model, we have built a ten-node parallel system of the proposed architecture and performed experiments with query loads compatible to those of a commercial search engine. The experimental results show that the estimation error between the model and the actual experiment is less than 2.13%. The proposed hybrid approach allows us to substantially reduce costs and efforts in building a large-scale system because we can accurately estimate its performance using a small number of machines without actually building it.

Last, by using the performance model, we demonstrate that the proposed architecture is capable of handling commercial-level data and query loads with a rather small number of machines. Our result shows that, with only 43,472 nodes, ODYS can handle 1 billion queries/day[2] for 30 billion Web pages with an average query response time of 194 ms. We also show that, by using twice as many (i.e., 86,944) nodes, ODYS can provide an average query response time of 148 ms. This clearly demonstrates the scalability and efficiency of the proposed architecture, and supports our argument that building a massively-parallel search engine using a parallel DBMS can be a viable alternative with advantages such as the high-level (i.e., DBMS-level) and SQL-like programming interface.

The rest of this paper is organized as follows. Section 2 introduces techniques of DB-IR integration as a preliminary. Section 3 proposes the architecture of ODYS, a massively parallel search engine using a DB-IR tightly integrated parallel DBMS. Section 4 proposes the performance model of ODYS. Section 5 presents the experimental results that validate the proposed performance model and demonstrate the scalability and performance of ODYS. Section 6 concludes the paper.

2. DB-IR INTEGRATION

In the database research field, integration of DBMS with IR features (simply, *DB-IR integration*) has been studied actively as the need of handling unstructured data as well as structured data is rapidly increasing. There are two approaches to DB-IR integration: loose coupling and tight coupling. The loose coupling method—used in many commercial systems—provides IR features as user defined types and functions outside of the DBMS engine (e.g., Oracle Cartridge and IBM Extender). This method is easy to implement because there is no need to modify the DBMS engine, but the performance of the system gets degraded because of long access paths to the IR feature. On the other hand, the tight coupling method [28, 29, 30] directly implements data types and operations for IR features as built-in types and functions of a DBMS engine (e.g., Odysseus [28, 29, 30] and MySQL [20]). The implementation of the method is difficult and complex because the DBMS engine should be modified but the performance is accelerated. Thus, the tight coupling method is appropriate for a large-scale system to efficiently handle a large amount of data and high query loads. The IR index of Odysseus [28][3] and MySQL are very close, but Odysseus has more sophisticated DB-IR algorithms for IR features as discussed below.

In a tightly integrated DB-IR system, an IR index is embedded into the system as shown in Fig. 3 (a). As in a typical DBMS, a B+-tree index can be constructed for an integer or formatted column. Similarly, an IR index is (automatically) constructed for a column having the text type. Fig. 3 (b) shows the structure of the IR index. The IR index consists of a B+-tree index for the keywords, where each keyword points to a *posting list*. The leaf node of the B+-tree has a structure similar to that of an inverted index. Each posting list for a keyword consists of the number of postings and the postings for the keyword. A *posting* has the document identifier (docId), and the location information where the keyword appears (i.e., docId, offsets). On the other hand, distinct from the inverted index, the IR index has a sub-index [28][3] for each posting list to search for a certain posting efficiently.

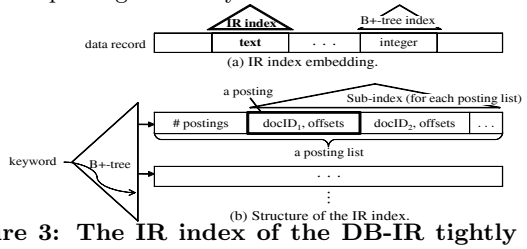


Figure 3: The IR index of the DB-IR tightly integrated DBMS.

In the DB-IR tightly integrated DBMS, two methods are used to improve the search performance: IR index join with posting skipping [28, 29] and attribute embedding [29]. *IR index join with posting skipping* is a technique for efficiently searching documents (e.g., Web pages) that have multiple co-occurring keywords. To search for documents having co-occurring keywords, the posting lists of the keywords should be joined. The posting skipping method identifies the part of the posting lists that need to be merged and skips the rest by using *sub-indexes* [28]. *Attribute embedding* is a technique for efficiently processing a DB-IR query that joins an attribute of a structured data type and an attribute of the text data type. For example, suppose that there are two attributes A and B having the text type and the integer type, respectively, and they are often accessed together. The attribute embedding method embeds the value of attribute B in each posting of attribute A. In this case, a DB-IR query that joins attributes A and B can be simply processed by one sequential scan of the posting list. In summary, it is the tightly integrated IR features, such as the embedded IR index with posting skipping, and attributed embedding, that makes ODYS a powerful search engine in the proposed parallel DBMS.

Example 1. Fig. 4 shows the processing of an IR query in a tightly integrated DB-IR system. Fig. 4 (a) shows an example of IR index join with posting skipping. When a site-limited query as in Fig. 1 (c) is given, $siteIdText$ of type text is used instead of $siteId$ of type integer as in Fig. 1 (d). Thus, the postings to be merged from each posting list are found efficiently using sub-indexes, as in the multiple keyword query processing. Fig. 4 (b) shows an example of attribute embedding. The values for $siteId$ of type integer are embedded in the postings of $Content$. We can efficiently

[2]Nielsenwire [23] reports that Google handled 214 million queries/day in the U.S. in February 2010.

[3]Patented in the US in 2002; application filed in 1999.

process the queries involving both *siteId* and *Content* as in Fig. 1 (c) by one sequential scan of the posting list. □

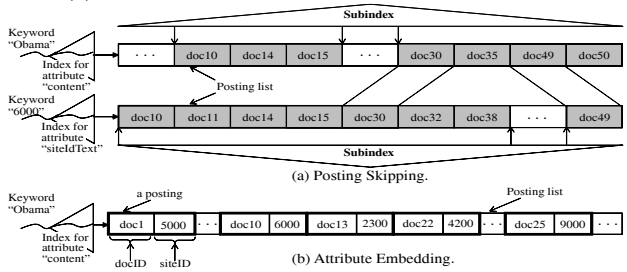


Figure 4: IR query processing in a tightly integrated DB-IR system [29].

3. ODYS MASSIVELY-PARALLEL SEARCH ENGINE

3.1 Architecture

Fig. 5 shows the architecture of ODYS. ODYS consists of masters [4] and slaves [5]. The masters share the slaves, and the slaves have a shared-nothing architecture. Each slave is Odysseus storing data in a disk array. The master and the slaves are connected by a gigabit network hub, and they communicate by using an asynchronous remote procedure call (RPC)[6].

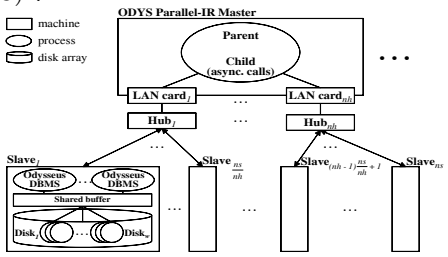


Figure 5: The architecture of ODYS.

The master stores metadata such as slaves' IP addresses, slaves' database paths (i.e., the location of the disk device storing each slave database), and schemas. The slaves store crawled Web pages and their IR indexes. There are two well-known methods for partitioning the index [9]: 1) *partitioning by documents* and 2) *partitioning by keywords*. For performance reasons, most commercial search engines including Google use the former method [9], which makes slaves work in parallel for processing the same query. Thus, we also employ the same method. That is, the entire set of Web pages is partitioned horizontally. Each slave stores a segment of the partitioned data and creates an IR index for each text-type attribute in the segment. To build a scalable shared-nothing parallel DBMS, we store tables into slaves in such a way to avoid a cross-node join. In search engines, typically there is one large-scale table, namely, the one for Web pages (say, 227 TBytes) and many small-scale tables (say, < 1 GBytes) such as tables for schema information, site information, and database statistics. We partition only the large-scale table into slaves and duplicate small-scale tables in every slave[7].

[4]The ODYS Parallel-IR Master consists of 58,000 lines of C and C++ code.

[5]The Odysseus (slave) consists of 450,000 lines of C and C++ code.

[6]We use socket-based RPC consisting of 17,000 lines of C, C++, and Python code developed by the authors.

[7]In this design, if two or more large-scale tables were used, join operations among those tables would not be allowed.

ODYS processes a user query as follows. When a user query arrives at a master, the master distributes the query to all slaves. Then, the master merges the results returned from the slaves and returns the final results to the user. The slaves process the query and return the results to the master. Each slave returns top-k results in the ranking order, and the master performs a merge operation for the results returned from the slaves to get the final top-k results. The slaves store each posting list of the IR index in the PageRank order (i.e., we are using query-independent ranking) to make top-k search efficient. Since the posting lists are stored in the PageRank order, the query processing performance is not much affected regardless how long the posting lists are or how big the database is. In this paper, we focus on the performance issues of the search engines and not on the effectiveness of ranking results. For performance reasons, we use query-independent ranking, which can be computed in advance, as many large-scale commercial search engines do, but any other ranking methods can be applied[8].

3.2 Updates and Fault Tolerance

In this paper, we focus on the performance issues of the search engine. Thus, we only briefly mention update and fault tolerance issues.

Updates and concurrency control: The search engines should handle a tremendous number of concurrent queries, which mostly consist of read-only transactions. Thus, in this paper, we focus on read-only transactions and the need for locking can be obviated[9]. Nevertheless, ODYS supports updates on a per-node basis with strong consistency [30]; thus, any transaction pertaining to individual nodes can be properly handled [29]. Update transactions can be processed on dedicated nodes and the nodes on service can be replaced with the updated nodes periodically.

Fault tolerance: Currently, fault tolerance features are being implemented in ODYS. We adopt an approach similar to the one proposed in Osprey [31]. In Osprey, Yang et al. [31] proposed a method implementing MapReduce-style fault tolerance functionality in a parallel DBMS. The proposed method maintains replicas and allocates small sized tasks dynamically to the nodes according to the loads of each node. As in Osprey, availability and reliability are achieved by maintaining multiple replicas of ODYS, and a middleware can be used for dynamically mapping masters and slaves of the multiple ODYS replicas. We call a replica of ODYS an *ODYS set*.

3.3 Other Parallel Processing Systems

In this section, we discuss the architectural relationships between ODYS and other recently developed parallel processing systems. We classify the existing DFS-based systems and parallel DBMSs into four types of layers as shown in Fig. 6 [10]. In Fig. 6, the storage layer represents a distributed storage for large-scale data. The key-value store or a table layer represents a data storage storing data in

[8]Most commercial search engines use PageRank as a base ranking measure, and additionally, combines the query-dependent ranking (e.g., TF-IDF). However, since query-dependent ranking is only applied only to the top results that have been retrieved based on query-independent ranking [25], its processing cost is somewhat limited.

[9]Locking can be switched off in the search engine mode by selecting the consistency level 0.

[10]Modified and extended from the figure in p.4 of [4].

the form of key-value pairs or records in tables. The parallel execution layer represents a system for automatically parallelizing the given job. The language layer represents a high-level query interface.

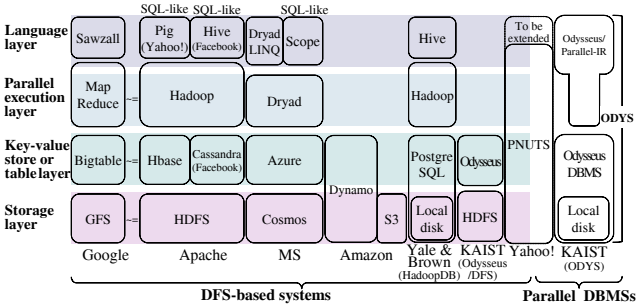


Figure 6: The map of ODYS and other parallel processing systems.

Most DFS-based systems that have been recently developed follow or modify Google's architecture. DFS-based systems developed by Apache and Microsoft (MS) have an architecture very close to Google's. On the other hand, Amazon's Dynamo and HadoopDB can be considered as variations of Google's architecture. Dynamo has a fully decentralized architecture and uses a relaxed consistency model called eventual consistency. HadoopDB has a hybrid architecture of the DFS-based system and the DBMSs; it is Hadoop on top of multiple single-node DBMSs. PNUTS is a highly scalable parallel DBMS that provides carefully chosen functionalities. It shares some design choices with the DFS-based system in that it provides simple functionalities, a relaxed consistency model, and flexible schema.

ODYS consists of two layers: Odysseus and Odysseus/ Parallel-IR. The Odysseus corresponds to the table layer. In ODYS, Odysseus DBMSs with local disks are used in parallel rather than key-value stores with a DFS system. The Odysseus/Parallel-IR is an integrated layer that combines the parallel execution layer and the language layer. Because ODYS uses a DBMS for the table layer, it can provide rich functionality for query processing by directly using most DBMS features including SQL.

There are several open source projects for search engines. Solr and Nutch are parallel search engines based on Apache Lucene [21], which is a search engine library for a single machine. They have a similar architecture consisting of masters and slaves where each slave is built on HDFS or a local file system. However, they do not support the high-level language layer such as SQL, but only support keyword queries. In addition, their performance and scalability are not enough to support commercial-level query loads. Moreira et al. [22] analyze performance and scalability of Nutch. The experimental results indicate that Nutch can process 33 queries/second (i.e., 2.8 million queries/day) for 480 GBytes dataset using 12 machines [22][11].

4. PERFORMANCE MODEL

In this section, we present a performance model of the proposed search engine architecture. Except for the specialized search engine companies, it is very difficult for a research center or an academic institute to build a real-world scale search engine because of limited resources including availability of hardware, space, and manpower. Therefore, an elaborate analytic or experimental model is needed to test and project the performance and scalability of a large-scale system without actually building it. For massively-parallel processing systems, analytic models using the queuing theory have been proposed to estimate the performance of the systems [17, 26]. However, those analytic models cannot be simply applied to our architecture because of the following reasons. First, the existing methods use simple parameters. In practice, however, to accurately estimate the performance of a large-scale system, all the specific parameters related to the performance of the system should be identified. Second, the existing methods assume that there is only one query type, while we consider multiple types of queries. Last, as the phenomenon that most significantly affects the performance, we show that the query response time is bounded by the maximum among slaves' query processing times, but no existing analytic method takes this into account. Therefore, we propose a performance model based on the queuing theory as well as intricate measurement schemes.

We claim that our performance model using a small-scale reference system (i.e., 5-node) can quite accurately predict the performance of a large-scale system (i.e., 300-node) due to the following reasons[12]. The performance model consists of two parts: 1) the master and network time and 2) the slave time. We show that the estimation error of the former is quite low, i.e., maximum 10.15% as shown in Fig. 11 in Section 5.2.2. Moreover, even if the estimation error of the master and network time were sizable, it could not affect the overall performance in a significant way since the overall performance largely depends on the performance of the slave time (e.g., 96.43% for 15.5 million queries/day) as shown in Fig. 14. We can be assured that the estimated performance of slaves is very close to the actual measurement since the estimation is directly derived from the measurement as presented in Section 4.2. Thus, our estimation of the total query response time is quite accurate (e.g., the estimation error is less than 2.13%) as shown in Fig. 11.

The assumptions related to query execution are as follows. We assume that every query is a top-k query where k is one of 10, 50 or 1000, and the set of input queries is a mix of single-keyword queries, multiple-keyword queries, and limited search queries as will be explained in Section 4.1.1. To evaluate a *lower-bound performance* of our system, we take two very conservative assumptions: 1) we run ODYS at "semi-cold start" and 2) we wait until "all" the slaves return the results. First, *semi-cold start* means that a query is executed in the circumstance where the internal nodes of the IR indexes (which normally fit in main memory) are resident in main memory while the leaf nodes (which normally are larger than available main memory), posting lists, and the data (i.e., crawled Web pages) are resident in disk. Typical commercial search engines process queries at *warm start* by storing the entire (or a large part of) indexes and data

[11]Moreira et al. [22] do not specify the ranking measure used. However, it must be that they used query-dependent ranking (e.g., TF-IDF) only since their results are significantly ($8 \sim 9$ times) worse than those of ten-node ODYS to be explained in Section 5.2.2. Thus, we do not directly compare their results with ours here.

[12]Cloud services (e.g., Amazon EC2, Microsoft Azure) cannot meet our requirements for constructing the precise performance model. We need to *exclusively* use and control physical machines (including CPUs and memory buses), network hubs, and disks to precisely reflect their effects to the performance model. However, cloud services cannot guarantee that environment.

in a massive-scale main memory [9]. This significantly helps reduce query response time. However, we do it at semi-cold start for the sake of evaluating a lower-bound performance[13]. To enforce semi-cold start, we use a buffer of only 12 MBytes sufficient for containing the internal nodes (occupying 11.5 MBytes) of the IR index for each slave[14]. Second, we wait until all the slaves return the results. This strategy guarantees the completeness of the results and projects the lower-bound performance. Performance can be improved by a deadline-driven technique, which waits for the results from slaves within limited time as in some commercial search engines, but then, we lose the completeness.

In the proposed performance model, each system component is modeled as a queue, and the query response time is estimated by summing up the expected sojourn time [7] of each queue. The following are the major considerations of the performance model. First, since each system component executes various types of tasks, the queue representing the component would receive various types of tasks that take different processing times. For example, a master CPU performs tasks of distributing the query to the slaves, merging the results returned from the slaves, etc. To simplify the problem, however, we regard the summation of all the types of tasks for a component as one request for the corresponding queue. Second, the service time is different depending on search condition types and the value of k of the top-k query. (The service time is the time for a request to be serviced in the queue, excluding the waiting time.) For instance, the time taken by a master CPU increases as the value of k gets larger because the merging cost increases. Therefore, we adopt the single-keyword top-10 query type—which has the shortest processing time—as the unit query, and transform(i.e., *normalize*) other query types in terms of the unit query. We explain this transformation in detail in Section 4.1. Finally, the times taken by slaves are bounded by the maximum value among all the sojourn times of slaves. In other words, we should calculate the expected maximum value of multiple sojourn times. However, this estimation is known to be very hard in the queuing theory [18]. Thus, we propose an estimation method for the slave maximum time through experiments, and it will be discussed in detail in Section 4.2.

4.1 Queuing Model

Among the ODYS system components, times taken by master CPUs, master memory bus, and network hubs are estimated by using a queuing model. Table 1 shows the notation used in the performance model.

4.1.1 Query Model

We consider nine types of queries. First, the queries are classified into three search condition types: single-keyword, multiple-keyword, and limited search queries. For each of the three search condition types, we further consider three types of top-k queries, where k = 10, 50, and 1,000. The performance of the master CPUs, master memory bus, and network hubs depend only on the k value while the performance of a slave depends on both the search condition type and the k value.

Table 1: The notation used in the performance model.

Symbols	Definitions
nm	the number of master nodes
ncm	the number of CPUs per master
ns	the number of slave nodes
nh	the number of network hubs
λ	the arrival rate of the ODYS
λ_C	the arrival rate of the system component C
λ'_C	the weighted arrival rate of the system component C
$w_C(k)$	the weight of the top-k query type in the system component C
$qmr(sct, k)$	the query mix ratio of top-k queries where sct is the search condition type
ST_C	the service time of the system component C
L_C	the average queue length of the system component C
X_C	the sojourn time of a customer in the queue of the system component C
r	the number of repetitions of a query set execution
np	the number of slaves in the small-sized system built
m_i	the master processing time for the ith slave
s_i	the query processing time of the ith slave
nt_i	the network transfer time of the ith slave's results

A *single-keyword query* is a query that has one keyword search condition. It is processed by finding the posting list that corresponds to the keyword from the IR index and by returning the first k results of the posting list. Single-keyword queries having the same k value can be processed within almost the same time regardless of the keyword because the top-k results have been stored according to a query-independent ranking calculated apriori.

A *multiple-keyword query* is a query that has two or more keyword search conditions. It is processed by finding the posting list for each keyword, performing a multi-way join for the posting lists, and returning the first k results of the joined results. A *limited search query* is a query having a keyword condition together with a site ID or a domain ID condition that limits the search scope. It is processed in the same way as a multiple-keyword query is. To process a limited search query using the multi-way join, site IDs or domain IDs are stored as text types and the IR indexes are created for them (see Example 1). Multiple-keyword or limited search queries take much longer processing time than single-keyword queries do because they require more disk accesses to find top-k results from the postings having the common docID's in multiple document lists.

As stated above, the query processing time varies depending on the search condition type and the k value of the top-k query. To simplify the queuing model, we normalize every query into one equivalent query type, i.e., the single-keyword top-10 query. For example, let us assume that a single-keyword top-10 query takes 2 ms while a multiple-keyword top-50 query takes 4 ms in some system component. Then, we regard a multiple-keyword top-50 query as two single-keyword top-10 queries for the component.

4.1.2 Arrival Rate (λ)

Suppose the query arrival rate is λ, and the requests are uniformly distributed to the components of the same type. Then, the arrival rate for a component is inversely proportional to the number of the components of the type, i.e., the number of queues. Table 2 shows the arrival rates for each component where nm, ncm, ns, and nh represent the

[13]We also performed experiments at warm start. It turns out that the total response time of our system at warm start is approximately 8.33% ∼ 18.01% of that at semi-cold start for the single-keyword top-10 queries in Fig. 11(a).

[14]In our system, the size of the IR index excluding the posting lists is 1.34 Gbytes per slave, out of which leaf nodes occupy 1.33 Gbytes and the internal nodes 11.5 Mbytes for indexing 114 million Web pages.

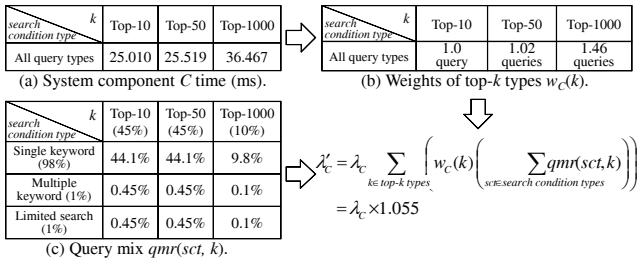
numbers of masters, of CPUs per master, of slave nodes, and of network hubs, respectively. The queues for the master CPUs and the master memory bus process one request per user query while the queues for the network hubs process ns requests per user query because every slave processes the same query and returns the results through the network hubs.

Table 2: Arrival rates.

Component	A master CPU	A master memory bus	A network hub
Arrival rate	$\frac{\lambda}{ncm \cdot nm}$	$\frac{\lambda}{nm}$	$\frac{ns}{nh}\lambda$

4.1.3 Weighted Arrival Rate (λ')

In the query model, we normalize a query of an arbitrary type to an equivalent number of single-keyword top-10 queries. A *weighted arrival rate* of a component is the arrival rate of the normalized queries. For each component, we measure the processing time of each top-k query and calculate the relative weights of the processing time of top-50 and top-1000 queries compared with that of a single-keyword top-10 query as the unit. As an example, Fig. 7 (a) shows the average query processing time in the system component C of each top-k query type, and Fig. 7 (b) shows the weight $w_C(k)$ of each top-k query type in C. For the component C, the average processing time of a top-10 query is 25.01 ms, and its weight is considered to be 1.0. The weights of other top-k queries are calculated based on this value. The weighted arrival rate at the component C, λ_C', is obtained by calculating the sum of products of the weight and the query mix ratio of each query type. For example, consider an example of query mix ratio $qmr(sct, k)$ for top-k queries in Fig. 7 (c), where *sct* is a search condition type. Then, the weight of the arrival rate at the system component C for this query mix is calculated as 1.055.



(a) System component C time (ms). (b) Weights of top-k types $w_C(k)$.

(c) Query mix $qmr(sct, k)$.

Figure 7: An example of calculating a weighted arrival rate at the system component C.

The weighted arrival rate at each system component is as shown in Formulas (1) $\sim$ (3). Here, we separate master CPUs from the master memory bus since we have multiple CPUs for each memory bus, and there is more contention in the memory bus than in a CPU. Since it is impossible to measure the weights of the master CPU and the master memory bus separately, we measure the master's total weight $w_{master}(k)$ for each top-k type and assume the same weight for both the master CPU and master memory bus.

4.1.4 Parameters of the Performance Model

To estimate a sojourn time at each component by using a queuing model, we measure the service time of each component. The service time at a component for each query type is obtained by measuring and averaging the processing times of all the queries of that type in a query set when executed alone. The master has two types of components: CPUs and the memory bus. Since the master CPU time and master memory access time cannot be measured independently, we first measure the total time taken by a master $ST_{master}(k, ns)$. Then, we obtain the service time of a master CPU $ST_{master\text{-}CPU}(k, ns)$ and that of the master memory bus $ST_{master\text{-}memory\text{-}bus}(k, ns)$ by dividing the total time by a given ratio, which will be discussed below. The total service time of a master $ST_{master}(k, ns)$ is obtained by Formula (4). In Formula (4), $T_{parent\text{-}proc}$ is the time spent by the parent process in the master for checking syntax of the query. $T_{child\text{-}proc}$ is the time spent by the child process for distributing the query to the slaves and storing the results received from the slaves into the result buffer. $T_{master\text{-}RPC}(k)$ is the time taken by the communication (RPC) module of the master; it varies according to k. $T_{child\text{-}proc}$ and $T_{master\text{-}RPC}(k)$ are multiplied by ns since they are repeated as many times as the number of slaves. $T_{merge}(k, ns)$ is the time for merging the results from the slaves to get the final top-k results. $T_{context\text{-}switch}(k, ns)$ is the time for the master processes to perform context switching. Formulas (5) and (6) show how to get the service times of a master CPU and a master memory bus. The service time of a master measured, $ST_{master}(k, ns)$, is divided according to the ratio of $\alpha : (1 - \alpha)$ to estimate the ratio of CPU time : memory access time ($0 \leq \alpha \leq 1$). The value of α is chosen in such a way that the estimated results fit the experimental results actually measured using the five-node reference system.

$$\lambda'_{master\text{-}CPU}(\lambda, nm, ncm) = \frac{\lambda}{ncm \cdot nm} \sum_{\substack{k \in top\text{-}k \\ types}} \left\{ w_{master}(k) \sum_{\substack{sct \in search \\ condition \\ types}} qmr(sct, k) \right\} \quad (1)$$

$$\lambda'_{master\text{-}memory\text{-}bus}(\lambda, nm) = \frac{\lambda}{nm} \sum_{\substack{k \in top\text{-}k \\ types}} \left\{ w_{master}(k) \sum_{\substack{sct \in search \\ condition \\ types}} qmr(sct, k) \right\} \quad (2)$$

$$\lambda'_{network}(\lambda, ns, nh) = \frac{ns}{nh}\lambda \sum_{\substack{k \in top\text{-}k \\ types}} \left\{ w_{network}(k) \sum_{\substack{sct \in search \\ condition \\ types}} qmr(sct, k) \right\} \quad (3)$$

$$ST_{master}(k, ns) = T_{parent\text{-}proc} + (T_{child\text{-}proc} + T_{master\text{-}RPC}(k)) * ns + T_{merge}(k, ns) + T_{context\text{-}switch}(k, ns) \quad (4)$$

$$ST_{master\text{-}CPU}(k, ns) = ST_{master}(k, ns) \times \alpha \quad (5)$$

$$ST_{master\text{-}memory\text{-}bus}(k, ns) = ST_{master}(k, ns) \times (1 - \alpha) \quad (6)$$

In Formula (4), $T_{parent\text{-}proc}$, $T_{child\text{-}proc}$, and $T_{master\text{-}RPC}(k)$ are measured using a small-sized (five-node) system since these values are independent of the number of slaves. On the other hand, since $T_{merge}(k, ns)$ and $T_{context\text{-}switch}(k, ns)$ depend on the number of slaves, we obtain them by using Formulas (7) and (8). To get the final top-k results, the results from the slaves are merged using a loser tree. In Formula (7), $t_{comparison}$ is the time spent by one comparison in the loser tree, and $\lceil \log_2 ns \rceil$, the height of the loser tree -1, is the number of comparisons for one result. t_{base} is the time spent for selecting a winner including the time to read streams and the time to copy the result to the result buffer but excluding the time for comparison. In Formula (8), $t_{per\text{-}context\text{-}switch}$ is the time spent by one context switch in the master. $ncs_{base}(k)$ is the initial number of context switches, and $ncs_{per\text{-}slave}(k)$ is the additional number of context switches per slave.

$$T_{merge}(k, ns) = k \times (\lceil \log_2 ns \rceil \times t_{comparison} + t_{base}) \quad (7)$$

$$T_{context\text{-}switch}(k, ns) = t_{per\text{-}context\text{-}switch} \times (ncs_{base}(k) + (ns \times ncs_{per\text{-}slave}(k))) \quad (8)$$

$ST_{network}(k)$, the service time of a network hub, is obtained by measuring the time taken by a network hub to transfer top-k results of a slave. Fig. 8 shows how to measure the service time of a network hub. For each top-k result, we measure the total time C of an RPC call. We then subtract M and S from C, where M is the CPU time taken by a master node, and S is the CPU time taken by a slave node. The CPU times are measured by using the time utility of LINUX. The CPU times M and S overlap the time O, which is the CPU time of the operating system to transfer data. However, the time is spent concurrently with the network transfer time, so we measure the time O and add it back. To measure the time O, we make a dummy RPC function by removing data transfer from the original RPC function, and measure the CPU time of the master node. The measured value corresponds to M−O, so we can obtain O from M and M−O. We assume that the CPU time of the operating system for network transfer at a slave is the same as that at the master. Therefore, we measure the service time of a network hub as (C−M−S)+2·O.

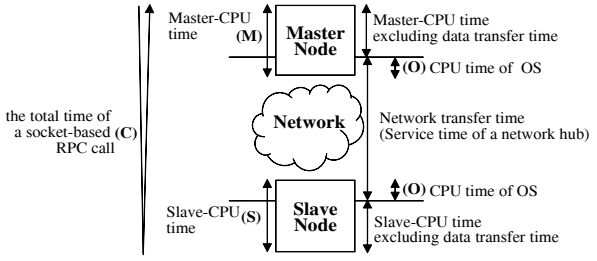


Figure 8: Measurement of a service time at a network hub.

4.1.5 Average Queue Length and the Estimated Sojourn Time

The average queue length of a component is the sum of the number of customers waiting in the queue and the number of customers being serviced. The sojourn time, i.e., the response time of a queue, is determined by the length of the queue. In this paper, every queue used in the performance model is an M/D/1 queue with a Poisson arrival process, a constant service time, and a single server. The average length of an M/D/1 queue is obtained by Formula (9). The average queue length for each component is obtained by Formulas (10)∼(12). To obtain the average queue length for the component C, we substitute λ in Formula (9) with the weighted arrival rate, and ST with the service time of top-10 query type for the component because the queries are normalized in terms of the single keyword top-10 query. We use a fixed value of ST, which is the average service time for the queue.

Meanwhile, the sojourn time X of a customer in a queue is the total time the customer spends inside the queue for waiting and for being served. The average sojourn time is obtained by Formula (13), where L is the average queue length and λ the arrival rate. Thus, the average sojourn time of single-keyword top-10 queries for each component is obtained by dividing the average length of the corresponding queue by the weighted arrival rate. The average sojourn times of queries having *other* top-k values are obtained by multiplying the weight for the top-k of the component.

$$L(\lambda, ST) = \frac{\lambda^2 E[ST^2]}{2(1 - \lambda E[ST])} + \lambda E[ST] \quad (9)$$

$$L_{master\text{-}CPU}(\lambda, nm, ncm, ns) = L(\lambda'_{master\text{-}CPU}(\lambda, nm, ncm), ST_{master\text{-}CPU}(k = 10, ns)) \quad (10)$$

$$L_{master\text{-}memory\text{-}bus}(\lambda, nm, ns) = L(\lambda'_{master\text{-}memory\text{-}bus}(\lambda, nm), ST_{master\text{-}memory\text{-}bus}(k = 10, ns)) \quad (11)$$

$$L_{network}(\lambda, ns, nh) = L(\lambda'_{network}(\lambda, ns, nh), ST_{network}(k = 10)) \quad (12)$$

$$E[X] = \frac{L}{\lambda} \quad (13)$$

$$E[X_{master\text{-}CPU}] = \frac{L_{master\text{-}CPU}(\lambda, nm, ncm, ns)}{\lambda'_{master\text{-}CPU}(\lambda, nm, ncm)} \times w_{master}(k) \quad (14)$$

$$E[X_{master\text{-}memory\text{-}bus}] = \frac{L_{master\text{-}memory\text{-}bus}(\lambda, nm, ns)}{\lambda'_{master\text{-}memory\text{-}bus}(\lambda, nm)} \times w_{master}(k) \quad (15)$$

$$E[X_{network}] = \frac{ns}{nh} \times \frac{L_{network}(\lambda, ns, nh)}{\lambda'_{network}(\lambda, ns, nh)} \times w_{network}(k) \quad (16)$$

For a network hub, ns/nh is additionally multiplied because ns/nh slaves are connected to one network hub.

4.2 Estimation of the Expected Slave Max Time

In ODYS, given a user query, all of the slaves process the query in parallel at semi-cold start. When a query is executed at semi-cold start, different slaves have much different processing times because the disk access time has a lot of variation. We note that, the total processing time is bounded by the maximum slave sojourn time (briefly, *slave max time*) since we must receive the results from all the slaves to answer the query. Here, we propose a method for estimating the slave max time based on measurement.

Fig. 9 shows the algorithm of the proposed estimation method for the slave max time. We call this algorithm the *partitioning-based estimation method* (simply, the *partitioning method*). The partitioning method estimates the slave max time of a large-sized (e.g., 300-node) target system by running a small-sized (e.g., 5-node) test system multiple times. Suppose r is the number of repetitions, np the number of slaves in the test system, and ns the number of slaves in the target system. In Step 1, the algorithm generates a sequence of $np \times r$ slave sojourn times for each query by running the test system r times. In Step 2, the algorithm partitions the sequence into segments of size ns, find the maximum value per segment, and average these values. Since the partitioning method provides the maximum value by measurement, the result must be very close to the actual measurement of the target system.

Algorithm *Partitioning Method* for estimating the expected *Slave Max Time*:
Input: (1) Q: the query set,
(2) r: the number of repetitions of the query set execution
(3) np: the number of slaves of the prototype
(4) ns: the number of slaves of the target system
Output: The estimated slave max time for each query in Q
Algorithm:
Step1. Generate a sequence of slave sojourn times for each query:
1.1 Execute Q for r times at semi-cold start by using the np-node prototype and measure the slave sojourn times.
1.2 For the ith query in Q, make a sequence of the slave sojourn times as $< t_{i,1,1}, t_{i,1,2}, \ldots, t_{i,1,m}, t_{i,2,1}, \ldots, t_{i,2,m}, \ldots, t_{i,r,1}, \ldots, t_{i,r,np} >$, where $t_{i,p,q}$ is the slave sojourn time for the ith query in the pth repetition at the qth slave.
Step2. Estimate the average slave max time for ns slaves:
For each sequence obtained in Step1,
2.1 Partition the sequence into segments of size ns.
2.2 Find the maximum value per segment and average those values.

Figure 9: The algorithm for estimating the expected slave max time by using the partitioning method.

4.3 Estimation of the Average Total Query Response Time

Fig. 10 shows an overview of the estimation method for computing the average total query response time. In Fig. 10, m_i is the master processing time for a slave (i.e., for sending the query and receiving results from a slave), s_i, the query processing time of the ith slave, and nt_i, the network transfer time of the ith slave's result. Suppose that for all i, m_i, s_i, and nt_i have the same value m, s, and nt, respectively. Then, there are two cases for estimating the response time of a query. If m is less than or equal to nt, the total query response time is obtained as $(m_1+s_1+ns\times nt) = (m+s+ns\times nt) \approx (ns\times nt+s)$. Or, if m is greater than nt, the total query response time is obtained as $(ns\times m+s_{ns}+nt_{ns}) = (ns\times m+s+nt) \approx (ns\times m+s)$. (One m or one nt is negligible.) In other words, the total query response time is obtained by adding the bigger of the network hub's total sojourn time ($ns\times nt$) and the master's total sojourn time ($ns\times m$) to the slave sojourn time (s). Here, for s, we use the slave max time as discussed in Section 4.2. Formula (17) shows how to get the estimated value of the average total query response time.

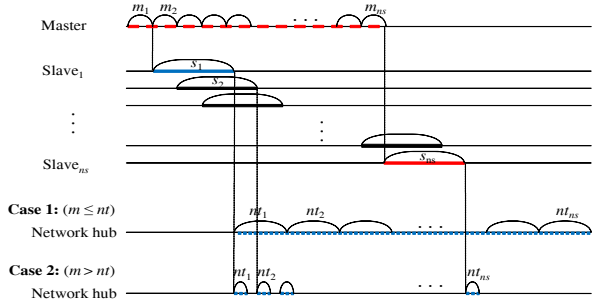


Figure 10: Overview of estimating the average total query response time.

5. PERFORMANCE EVALUATION

5.1 Experimental Data and Environment

We have built a ten-node parallel search engine according to the architecture shown in Fig. 5. Here, we have used one master, one 1-Gbps hub, and ten slaves each running 100 Odysseus processes with a shared buffer of 12 Mbytes. We have used one machine for the master, a Linux machine with a quad-core 3.06 GHz CPU[15] and 6 Gbytes of main memory. We have used two replicated five-node clusters: Cluster 1 and Cluster 2. Cluster 1 consists of four Linux machines each with two dual-core 3 GHz CPU, 4Gbytes of main memory, and a RAID 5 disk array. Each disk array has 13 disks (disk transfer rate: average 59.5 Mbytes/s) with a total of 0.9 ~ 3.9 Tbytes disk space, a cache of 0.5 ~ 1 Gbytes, and 200 Mbytes/s bandwidth. The remaining one is a Linux machine with a quad-core 2.5 GHz CPU, 4 Gbytes of main memory, and a RAID 5 disk array. The disk array has 13 disks (disk transfer rate: average 83.3 Mbytes/s) with a total of 13 Tbytes, a cache of 512 Mbytes, and 512 Mbytes/s bandwidth. Cluster 2 consists of five Linux machines each with a quad-core 2.4 GHz CPU, 8 Gbytes of main memory, and an internal RAID 5 disk array. The disk array has 10 disks (disk transfer rate: average 81.2 Mbytes/s) with a total of 5 Tbytes, a cache of 256 Mbytes, and 768 Mbytes/s bandwidth. In general, Cluster 2 is a bit faster than Cluster 1. The master and the slaves are connected by a 1-Gbps network hub.

We perform experiments using 114 million Web pages crawled. For each Web page, if its size is larger than 16 Kbytes, we extract and store important part of a fixed size (front 8 KBytes + rear 8 KBytes) to uniformly control the experiments. We build two replicated 5-node clusters (228 million Web pages total). For each cluster, we evenly partition the set of 114 million Web pages into five segments and allocate each segment to a slave. Thus, each slave stores and indexes 22.8 million Web pages[16]. We perform experiments using two query sets. One consists of only single-keyword top-10 queries, and the other consists of a mixed type of queries with different k (in top-k) values. We call the query sets SINGLE-10-ONLY and QUERY-MIX, respectively. For QUERY-MIX, we use the query mix ratio in Fig. 7 (c). Each query set includes 10,000 queries in which the keywords, site IDs, and domain IDs are all unique. Queries are generated at a Poisson arrival rate and issued by a separate machine.

We perform the following experiments. First, we measure the estimation error of the performance model by comparing its projected output with the results measured as the arrival rate and number of nodes are varied. We use the *estimation error* as defined in Formula (18). Second, we estimate the slave max time using the partitioning method using the five-node reference system (Cluster 1 only). Last, we project the performance of ODYS for real-world scale data and query loads by using the performance model. The value of α used in the performance model is 0.25. All the experimental results are measured at semi-cold start. To measure the time at semi-cold start, we first empty the DBMS buffers and disk array caches of all the slaves, and then, run 10,000 queries that are completely independent of the experimental queries (i.e., no keywords, site IDs, or domain IDs overlapping with those of the experimental queries) to load the internal nodes of the IR index into the DBMS buffer.

5.2 Experimental Results

5.2.1 Measurement of Parameters

We measure the parameters of the queuing model using the five-node system. The values are measured as described in Section 4.1.4. Table 3 shows the parameters measured.

5.2.2 Accuracy of the Performance Model

We vary the query loads from 5 to 24 million queries/day for the two query sets: SINGLE-10-ONLY and QUERY-MIX. Fig. 11 shows the average total query response time and the average processing time of the master and network for each query load. In Fig. 11[17], TOTAL-EXP-10 repre-

[15]State of the art CPUs use several features (e.g., EIST, Turbo Boost, C-State) for power saving. These features can cause a significant variance in the service time. To minimize this variance, we turn-off all these features.

[16]One ODYS slave is capable of indexing 100 million Web pages. But, we used 22.8 million/slave since we had only 114 million crawled. Nevertheless, this does not affect the query performance significantly since the postings are sorted according to the PageRank order, and only up to 1000 postings are retrieved for a single-keyword top-k query. For a multiple-keyword or limited search query, a larger number of postings are accessed, but a sufficient number of postings are available from 22.8 million Web pages.

[17]To avoid clutter, Fig. 11 shows only the results of the 10-node system. Others show similar trends.

$$
\begin{aligned}
&t_{parallel\text{-}n\text{-}node}(sct, k, \lambda, nm, ncm, ns, nh) \\
&= max\Bigg(\Big(E[X_{master\text{-}CPU}] + E[X_{master\text{-}memory\text{-}bus}]\Big), E[X_{network}]\Bigg) + t_{slave\text{-}max\text{-}time}(sct, k, \lambda, ns) \\
&= max\Bigg(\Big(\frac{L_{master\text{-}CPU}(\lambda, nm, ncm, ns)}{\lambda'_{master\text{-}CPU}(\lambda, nm, ncm)} \times w_{master}(k) + \frac{L_{master\text{-}memory\text{-}bus}(\lambda, nm, ns)}{\lambda'_{master\text{-}memory\text{-}bus}(\lambda, nm)} \times w_{master}(k)\Big), \\
&\quad \frac{ns}{nh} \times \frac{L_{network}(\lambda, ns, nh)}{\lambda'_{network}(\lambda, ns, nh)} \times w_{network}(k)\Bigg) + t_{slave\text{-}max\text{-}time}(sct, k, \lambda, ns) \qquad (17)
\end{aligned}
$$

$$
\begin{matrix} estimation \\ error \end{matrix} = \frac{\left| \begin{matrix} estimated\ average \\ time\ of\ a\ query \end{matrix} - \begin{matrix} measured\ average \\ time\ of\ a\ query \end{matrix} \right|}{\begin{matrix} measured\ average \\ time\ of\ a\ query \end{matrix}} \qquad (18)
$$

Table 3: Parameters of the queuing model measured.

Parameters	Values
$T_{parent\text{-}proc}$	1.516 ms
$T_{child\text{-}proc}$	0.0081 ms
$T_{master\text{-}RPC}(k)$	0.01 ms, $k = 10$ 0.011 ms, $k = 50$ 0.031 ms, $k = 1000$
$t_{comparison}$	0.191 μs
t_{base}	0.28 μs
$t_{per\text{-}context\text{-}switch}$	2.105 μs
$ncs_{base}(k)$	56.490, $k = 10$, 50 97.728, $k = 1000$
$ncs_{per\text{-}slave}(k)$	1.917, $k = 10$, 50 3.316, $k = 1000$
$ST_{network}(k)$	0.129 ms, $k = 10$ 0.222 ms, $k = 50$ 0.318 ms, $k = 1000$

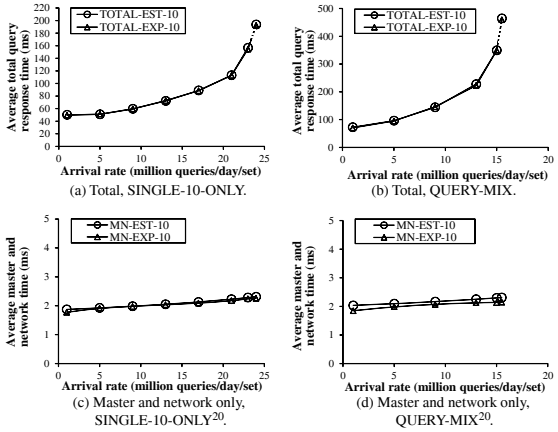


Figure 11: The estimated and experimental results of the ten-node system (ns=10) as the query arrival rate is varied. EST-10 represents estimated results for the 10-node system; EXP-10 experimental results.

sents the experimental results of the total query response time measured using the ten-node system as the query arrival rate is varied; TOTAL-EST-10 the estimated results from the hybrid performance model. That is, the slave max time is estimated from the measurement of the five-node reference system using the algorithm in Fig. 9 in Section 4.2; the master and network time is estimated from the queuing model; TOTAL-EST-10 is a sum of these two. MN-EXP-10 represents the experimental results of the master and network time; MN-EST-10 the estimated results. The results show that the maximum estimation error of the total query response time is 1.77% for SINGLE-10-ONLY, and 2.13% for QUERY-MIX. The maximum estimation error of the master and network time (i.e., the part modeled by the queuing model) is 6.29% for SINGLE-10-ONLY and 10.15% for QUERY-MIX[18]. For the same query load, the result of SINGLE-10-ONLY and QUERY-MIX shows a big difference. The reason is that the response time of the multiple-keyword and limited search queries are much longer than those of the single-keyword queries as discussed in Section 4.1.1[19]. In Fig. 11, dotted lines represent the regions where measurements become unstable since the number of input queries becomes close to the maximum possible throughput. The result in Fig. 11 (a) shows that the ten-node ODYS can stably process 266 queries/sec (23 million queries/day) with an average response time of 154 ms on a 1.52 TBytes dataset.

Fig. 12[21] shows the average total query response time and the average master and network time as the number of nodes is varied from 1 to 10 for two query sets: SINGLE-10-ONLY and QUERY-MIX. Here, EST-x represents the estimated results for x million queries/day/set; EXP-x the experimental results. In Figs. 12(a) and (b), due to variations of performance in individual machines, for 1- or 3-node configurations, the results vary depending on which machines we use in the experiment and do not average out. Thus, we represent the result as a range (i.e., from min using the fastest machine(s) to max using the slowest machine(s)). The results show that the maximum estimation error of the total query response time is 2.13%, and that of the master and network time is 10.15% when the number of nodes ≥ 5.

5.2.3 Estimation of the Slave Max Time

We analyze the effect of the number of slaves on the performance of ODYS. We estimate the slave max times for QUERY-MIX while increasing the segment size of the partitioning method for each query load. We measure 300 slave sojourn times for each query by running the query set 60 times using the five-node reference system. Fig. 13 shows the expected slave max time for each segment size. The results show that the expected slave max time increases up

[18]For sensitivity analysis, we have tested a different query mix having 20% of top-1000 queries obtaining a similar result where the maximum estimation error was 8.64%. We have not conducted sensitivity analysis on search condition types since the master and network time does not depend on search condition types, but only on the top-k value.

[19]It can be optimized by constructing separate ODYS sets dedicated to limited search queries. For the IR indexes of these ODYS sets, we can order the postings of each posting list by the domain ID, and then, by ranking the sets of the postings that have the same domain ID's independently of one another, significantly reducing the search over the posting list. However, these additional optimizations are beyond the scope of this paper and will be left as a further study.

[20]For Figures 11 (c) and (d), the master and network time is measured by subtracting the slave max time from the total query response time.

[21]To avoid clutter, Figs. 12(a) and (b) show the results only for three representative arrival rates; Figs. 12(c) and (d) only for one.

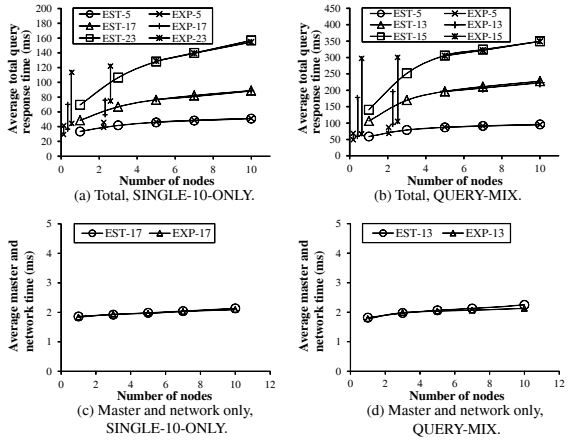


Figure 12: The estimated and experimental result as the number of nodes is varied. EST-x represents the estimated results for x million queries/day/set; EXP-x the experimental results.

to $1.5 \sim 2$ times of the minimum value as the segment size increases, i.e., as the number of slaves increases. Interestingly, the slave max time gradually converges to a value less than twice the minimum instead of increasing indefinitely. Detailed analysis of this phenomenon is beyond the scope of this paper. We leave it as a further study.

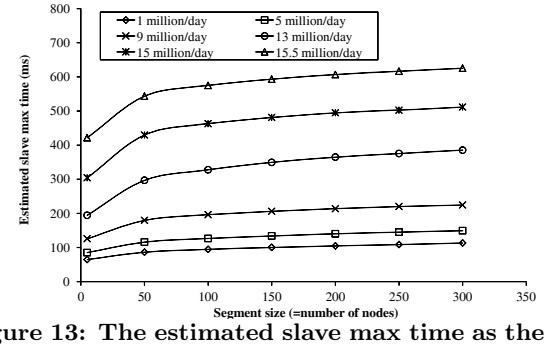


Figure 13: The estimated slave max time as the segment size is varied (QUERY-MIX, r=60, ns=5).

5.2.4 *Performance Projection of a Real-World-Scale (300-Node) ODYS*

By using the performance model, we estimate the average response time of ODYS for 30 billion[22] Web pages, which is considered real-world scale data [19]. Fig. 14 shows the estimated performance of a 300-node system for the query set QUERY-MIX. In the estimation, one ODYS set consists of 4 masters, 300 slaves, and 11 Gbit network hubs. Each master has a Quad-Core 3.06 GHz CPU, each slave has one 3 GHz CPU, 4 Gbytes of main memory, and 13×300 Gbytes SATA hard disks. Here, we select the number of masters (4) and network hubs (11) to make the queue lengths of master memory and network hubs similar to each other to avoid bottlenecks. In Fig. 14, TOTAL-EST-300 represents the estimated total response time of queries, and SLAVE-MAX-EST-300 represents the slave max time estimated using the partitioning method. SLAVE-MAX-EST-300 is identical to the results of the segment size 300 in Fig. 13.

[22]Google (as well as other commercial search engines such as MS Bing and Yahoo!) indexes approximately 25 billion Web pages. This is roughly indicated by the result of querying with frequently occurring keywords such as "a," "the," or "www."

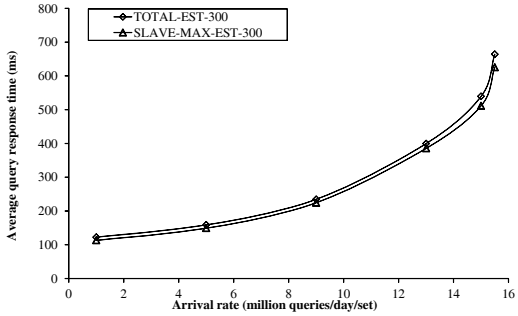


Figure 14: The projected average response time of ODYS for real-world scale service (ns=300). TOTAL-EST-300 is the average estimated total query response time; SLAVE-MAX-EST-300 the average estimated slave max time.

As we observed in Fig. 14, if an ODYS set were to take a higher query load, the total number of nodes required to handle the load could be reduced, but the performance would be degraded. Therefore, the trade-off between the number of nodes and the performance exists in providing reasonable performance with a minimal number of nodes. For example, suppose we run 7 million queries/day (81 queries /sec) per ODYS set, then ODYS can handle Google-scale service using 143 sets of 304 nodes (4 masters and 300 slaves), a total of 43,472 nodes, for 1 billion queries/day. In this case, the average query response time is only 194 ms. In contrast, if we ran 3.5 million queries/day (40.5 queries/sec), ODYS would need 286 sets with a total of 86,944 nodes with an average query response time of 148 ms[23]. The experiments show that our approach is capable of providing a commercial-level service with a rather small number of nodes.

6. CONCLUSIONS

In this paper, we have shown that a massively parallel search engine capable of processing real-world scale data and query loads can be implemented using a DB-IR tightly integrated parallel DBMS. We have also presented the detailed implementation of such a system, ODYS. The DBMS used in the slaves of ODYS is Odysseus, which is a highly scalable object-relational DBMS that is tightly integrated with IR features [29]. Odysseus is capable of indexing 100 million Web pages, and thus, ODYS is able to handle a large volume of data even with a small number of machines. The tightly integrated DB-IR functionality enables ODYS to have a commercial-level performance for keyword queries. Furthermore, ODYS provides rich functionality such as SQL, schemas, and indexes for easy (and less error-prone) development and maintenance of applications [30].

We have also proposed a performance model that can estimate the performance of the proposed architecture. The model is a hybrid one that employs both an analytic approach based on the queuing model and an experimental approach of using a small-sized (five-node, in this paper) reference system to project the performance of a large target system. We have validated this model through comparison of the result projected by the model with the results measured using a ten-node system. Our estimation of the total response time is quite accurate since the bulk of the total response time is spent at the slave, and we derive the slave

[23]The average response time of a typical commercial search engine is known to be 200-250 ms [9, 13].

max time by measurement with accuracy. The result of the comparison indicates that the estimation error of the total query response time of the ten-node system is less than 2.13%. Such a modeling method is helpful in realistically estimating the performance of a system by using limited resources—without actually building a large-scale system.

Finally, we have estimated the performance of ODYS for real-world scale data and query loads. According to the performance model, with a small number of (i.e., 43,472) nodes, ODYS is capable of handling 1 billion queries/day for 30 billion Web pages at an average query response time of 194 ms. With twice as many nodes (i.e., half the query load per node), it can provide an average query response time of 148 ms. This clearly demonstrates the scalability and efficiency of our architecture. These results are even more marked since these are conservative results from a semi-cold start environment reflecting a lower-bound performance.

7. ACKNOWLEDGEMENTS

This work was supported by the National Research Foundation of Korea (NRF) grant funded by Korean Government (MEST) (No. 2012R1A2A1A05026326). We deeply appreciate incisive comments of anonymous reviewers, which made the paper far more complete. We also would like to acknowledge contributions of many people who worked on the Odysseus DBMS project over 23 years—in particular, Young-Koo Lee, Min-Jae Lee, Wook-Shin Han, Jae-Gil Lee, and Jae-Jun Yoo.

8. REFERENCES

[1] Abouzeid, A. et al., "HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads," In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pp. 922-933, Aug. 2009.

[2] Azure, http://www.microsoft.com/windowsazure.

[3] Barroso, L., Dean, J. and Holzle, U., "Web Search for a Planet: the Google Cluster Architecture," *IEEE Micro,* Vol. 23, No. 2, pp. 22-28, Mar. 2003.

[4] Budiu, M., 2009, available at http://budiu.info/work/dryad-talk-berkeley09.pptx.

[5] Cassandra, http://cassandra.apache.org.

[6] Chang, F. et al., "Bigtable: A Distributed Storage System for Structured Data," In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 205-218, Nov. 2006.

[7] Cooper, R., *Introduction to Queuing Theory*, North Holland, 2nd ed., 1981.

[8] Dean, J., and Ghemawat, S., "MapReduce: Simplified Data Processing on Large Clusters," In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 137-150 , Dec. 2004.

[9] Dean, J., "Challenges in Building Large-Scale Information Retrieval Systems," In *Proc. ACM Int'l Conf. on Web Search and Data Mining (WSDM)* (an invited talk), p. 1, Feb. 2009 (presentation slides available at http://research.google.com/people/jeff/WSDM09-keynote.pdf).

[10] Dean, J., "Designs, Lessons and Advice from Building Large Distributed Systems," a keynote at ACM SIGOPS Int'l Workshop on Large Scale Distributed Systems and Middleware (LADIS), Oct. 2009 (presentation slides available at http://www.odbms.org/download/dean-keynote-ladis2009.pdf).

[11] DeCandia, G. et al., "Dynamo: Amazon's Highly Available Key-Value Store," In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.

[12] Ghemawat, S., Gobioff, H., and Leung, S., "The Google File System," In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.

[13] Google, http://www.google.com/about/corporate/company/tech.html.

[14] Hadoop, http://hadoop.apache.org.

[15] HBase, http://hbase.apache.org.

[16] HDFS, http://hadoop.apache.org/hdfs.

[17] Javadi, B., Khorsandi, S., and Akbari, M., "Queuing Network Modeling of a Cluster-Based Parallel System," In *Proc. Int'l Conf. on High Performance Computing and Grid in Asia Pacific Region*, pp. 304-307, July 2004.

[18] Kemper, B. and Mandjes, M., Approximations for the Mean Sojourn Time in a Parallel Queue, Technical Report PNA-E0901, Centrum Wiskunde & Informatica, Mar. 2009.

[19] Kunder, M., http://www.worldwidewebsize.com.

[20] Lentz, A., "MySQL Storage Engine Architecture," In MySQL Developer Articles, MySQL AB, May 2004 (available at http://ftp.nchu.edu.tw/MySQL/tech-resources/articles/storage-engine).

[21] Lucene, http://lucene.apache.org.

[22] Moreira, J. et al., "Scalability of the Nutch search engine," In *Proc. Int'l Conf. on Supercomputing (ICS)*, pp. 3-12, June 2007.

[23] Nielsenwire, "Nielsen Reports February 2010 U.S. Search Rankings," Nielsen Report, Mar. 15, 2010 (available at http://blog.nielsen.com/nielsenwire/ online_mobile/nielsen-reports-february-2010-u-s-search-rankings).

[24] Özsu, M. and Valduriez, P., "Distributed Reliability Protocols," In Book *Principles of Distributed Database Systems,* Prentice Hall, 2nd ed., pp. 379-400, 1999.

[25] Richardson, M., Prakash, A., and Brill, E., "Beyond PageRank: machine learning for static ranking," In *Proc. Int'l Conf. on World Wide Web (WWW)*, pp. 707-715, May 2006.

[26] Shahhoseini, H. and Naderi, M., "Design Trade off on Shared Memory Clustered Massively Parallel Processing Systems," In *Proc. Int'l Conf. on Computing and Information*, Nov. 2000.

[27] Stonebraker, M. et al., "MapReduce and Parallel DBMSs: Friends or Foes?," *Communications of the ACM (CACM)*, pp. 64-71, Jan. 2010.

[28] Whang, K. et al., 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, Feb. 15, 1999.

[29] Whang, K. et al., "Odysseus: A High-Performance ORDBMS Tightly-Coupled with IR Features," In *Proc. Int'l Conf. on Data Engineering (ICDE)*, pp. 1104-1105, Apr. 2005. This paper recieved the Best Demonstration Award.

[30] Whang, K. et al., "DB-IR Integration Using Tight-Coupling in the Odysseus DBMS," submitted for publication, 2013.

[31] Yang, C. et al., "Osprey: Implementing MapReduce-Style Fault Tolerance in a Shared-Nothing Distributed Database," In *Proc. IEEE Int'l Conf. on Data Engineering (ICDE)*, pp. 657-668, Mar. 2010.