World Wide Web
DOI 10.1007/s11280-014-0312-2

PARADISE: Big data analytics using the DBMS tightly integrated with the distributed file system

Jun-Sung Kim · Kyu-Young Whang · Hyuk-Yoon Kwon · Il-Yeol Song

Received: 10 June 2014 / Revised: 7 October 2014 / Accepted: 18 November 2014

Abstract There has been a lot of research on MapReduce for big data analytics. This new class of systems sacrifices DBMS functionality such as query languages, schemas, or indexes in order to maximize scalability and parallelism. However, as high functionality of the DBMS is considered important for big data analytics as well, there have been a lot of efforts to support DBMS functionality in MapReduce. HadoopDB is the only work that directly utilizes the DBMS for big data analytics in the MapReduce framework, taking advantage of both the DBMS and MapReduce. However, HadoopDB does not support sharability for the entire data since it stores the data into multiple nodes in a shared-nothing manner—i.e., it partitions a job into multiple tasks where each task is assigned to a fragment of data. Due to this limitation, HadoopDB cannot effectively process queries that require internode communication. That is, HadoopDB needs to re-load the entire data to process some queries (e.g., 2-way joins) or cannot support some complex queries (e.g., 3-way joins). In this paper, we propose a new notion of the DFS-integrated DBMS where a DBMS is tightly integrated with the distributed file system (DFS). By using the DFS-integrated DBMS, we can obtain sharability of the entire data. That is, a DBMS process in the system can access any data since multiple DBMSs are run on an integrated storage system in the DFS. To process big data analytics in parallel, our approach use the MapReduce framework on top of a DFS-integrated DBMS. We call this framework PARADISE. In PARADISE, we employ a job splitting method that logically splits a job based on the predicate in the integrated storage system. This contrasts with physical splitting in HadoopDB. We

J.-S. Kim · K.-Y. Whang (✉) · H.-Y. Kwon
Department of Computer Science, KAIST, Daejeon, Korea
e-mail: kywhang@cs.kaist.ac.kr

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

H.-Y. Kwon
e-mail: hykwon@mozart.kaist.ac.kr

I.-Y. Song
College of Computing & Informatics, Drexel University, Philadelphia, USA
e-mail: songiy@drexel.edu

Published online: 11 December 2014

also propose the notion of *locality mapping* for further optimization of logical splitting. We show that PARADISE effectively overcomes the drawbacks of HadoopDB by identifying the following strengths. (1) It has a significantly faster (by up to 6.41 times) amortized query processing performance since it obviates the need to re-load data required in HadoopDB. (2) It supports query types more complex than the ones supported by HadoopDB.

Keywords Big data analytics · MapReduce · DBMS · Distributed file system · Integration · HadoopDB

1 Introduction

1.1 Motivation

The era of big data, where a vast amount of information is continuously generated, has arrived and this trend will surely continue for a long time in the future [11, 24]. It was reported that the digital information content of the world amounted to 1.8 zettabytes in 2011 and was to increase by tens to hundreds of times in ten years [15]. The amount of information itself is so large (hence, the name *big data*) that it is not easy to find specific information that a user wants. Thus, the technology for extracting useful information from big data (i.e., big data analytics) have become very important [19].

MapReduce is the state-of-the-art technology for big data analytics [9]. It provides a distributed/parallel programming framework that runs a user program in parallel. Users simply write Map and Reduce functions, which will run in parallel over a cluster of commodity hardware nodes. MapReduce facilitates petabyte scale big data analytics over thousands to tens of thousands of nodes. In general, it stores and manages distributed data by using a NoSQL-based system, especially, the distributed file system (simply, DFS) such as GFS [16] and HDFS, or a key-value store such as BigTable [6]. The DFS provides not only scalability, fault-tolerance, and load-balancing but also sharability for the slaves, i.e., allows each slave to access the entire database through the network [16].

Compared to DBMSs, MapReduce applications are hard to implement and maintain since the system provides a relatively low-level storage system API and simple functionality [29]. In order to resolve lack of DBMS functionality in MapReduce, there has been a lot of efforts to combine MapReduce and the DBMS taking advantage of both systems. These efforts can be classified into two categories: (1) supporting partial functionality of the DBMS and (2) supporting full functionality of the DBMS. The former implements only some specific functionality of the DBMS such as a query language, schema, or indexes in MapReduce. Pig [26] or Hive [30] are the representative systems that support a query language for MapReduce. However, they do not support other DBMS functionalities such as secondary indexes or transactions. We have exhaustively investigated the MapReduce systems that support functionality of the DBMS and concluded that, to the best of our knowledge, all of them except HadoopDB [1] are classified as the former while HadoopDB is the only work that is classified as the latter. We will describe them in Section 2 in detail.

HadoopDB directly uses the DBMS for parallel query processing using MapReduce [1]. HadoopDB partitions the entire set of data stored in the DFS into multiple fragments and loads each fragment into the local DBMS of a slave node in a shared-nothing manner. Then, it performs parallel query processing on those local DBMSs using the MapReduce framework. Although HadoopDB is a new brand of research that uses the DBMS for big data analytics, it does not support *sharability* for the entire set of data stored in the DFS. That is,

a DBMS process can access only the data stored in the slave node where it is running. Thus, HadoopDB splits the job according to the physical partition of data mapping one Map task to one fragment database. We call this job splitting method *physical splitting*. Physical splitting can efficiently process queries that do not require accessing multiple slave nodes (e.g., scan for a relation) by partitioning the job into multiple tasks and processing those tasks in parallel. Nevertheless, it is inefficient for some queries that require re-loading the entire data (e.g., 2-way joins) or cannot support some complex queries (e.g., 3-way joins) that require internode communications. Since HadoopDB partitions data based on the join attributes to process a 2-way join so as to prevent internode communication, it needs to re-load the entire data when it cannot use the current snapshot stored in local databases, i.e., when the join attributes for an upcoming query are different from the current partitioning attributes.

We introduce a new storage system that tightly integrates the DBMS with the DFS and supporting sharability of data. We call it a *DFS-integrated DBMS*.[1] It supports not only full functionality of the DBMS and scalability of the DFS but also sharability for the entire data stored in the DFS. The salient point is that sharability allows us to regard the entire data in the DFS as *one integrated database*. Since each DBMS process in a DFS-integrated DBMS can access any data stored in the DFS, we do not have the limitations of HadoopDB described above, i.e., having to re-load the entire data or being unable to support some complex queries. In the literature, there have been no research effort using the DFS-integrated DBMS for big data analytics. Furthermore, since physical splitting in HadoopDB assumes multiple fragmented databases rather than one integrated database, we cannot directly apply processing methods based on physical splitting to the DFS-integrated DBMS.

1.2 Our contributions

We propose a new approach to big data analytics using an integrated database in the DFS, which we call PARADISE (**P**arallel **A**nalytic framework for the **R**elational DBMS **A**nd the **D**istributed file system **I**ntegrated to one **S**torage syst**E**m). PARADISE consists of the MapReduce framework providing efficient parallel processing of big data and the DFS-integrated DBMS providing the DBMS functionality and one integrated database. In addition, to resolve MapReduce job splitting issues for the DFS-integrated DBMS, we employ a job splitting method suitable for an integrated database, which we call *logical splitting*. This method splits a MapReduce job into multiple tasks based on a SQL predicate, and thus, is independent of physical partitioning of the data over multiple nodes. Figure 1 shows the architecture of PARADISE compared with that of HadoopDB. One distinguished difference of PARADISE from HadoopDB is that the former processes a MapReduce job using an integrated database in the DFS while the latter does it using multiple fragmented databases.

In this paper, we make the following contributions. First, we propose a new approach PARADISE that consists of the MapReduce framework using the DFS-integrated DBMS providing an integrated database. PARADISE effectively overcomes the drawbacks of HadoopDB by supporting sharability of the entire data in the DFS. Specifically, PARADISE has the following advantages compared with HadoopDB. (1) The amortized system performance of PARADISE is far better than that of HadoopDB. PARADISE does not require re-loading the entire data while HadoopDB does for processing certain kinds of queries.

[1] The notion of the DFS-integrated DBMS has been implemented in Odysseus/DFS [23]. Detailed issues for tightly integrating the DFS and the DBMS in a DFS-integrated DBMS are presented in Kim et al. [23].

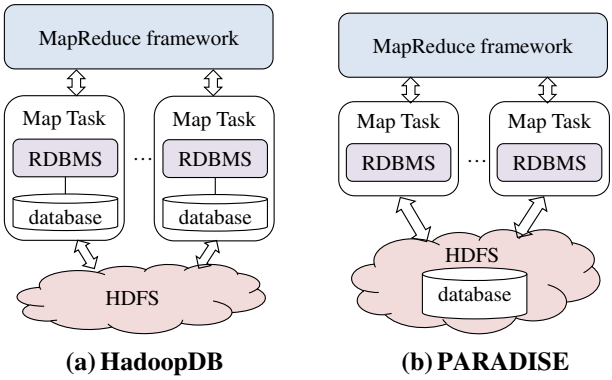


Figure 1 An architectural comparison between PARADISE and HadoopDB

Overall, PARADISE enhances the performance of HadoopDB by up to 6.41 times for a database of size 20Gbytes. (2) It supports more complex query types. Specifically, it can process queries requiring internode communications such as 3-way joins and Cartesian products, which cannot be supported by HadoopDB. Second, we propose a logical job splitting method for the DFS-integrated DBMS. It enables efficient parallel query processing by utilizing the MapReduce framework in an integrated database. We also propose the notion of *locality mapping* for further optimization of logical splitting. Third, we analyze the performance overheads of PARADISE compared with those of HadoopDB. Since PARADISE accesses an integrated database through the DFS, there is an additional overhead of accessing data through the DFS.

The rest of the paper is organized as follows. In Section 2, we review the representative systems using MapReduce or DBMSs for big data analytics. In Section 3, we present our new approach for big data analytics, PARADISE. In Section 4, we present the results of various experiments and analyze them to verify the efficiency of PARADISE. Finally, in Section 5, we conclude the paper.

2 Related work

2.1 MapReduce

Google has introduced the DFS and MapReduce framework as parallel and scalable solutions for large-scale data. We briefly introduce them for the architecture and the terminology to be used in this paper. MapReduce has evolved to Hadoop [18], an open-source project. Two major research thrusts utilizing Hadoop are implementations of Hadoop Distributed File System (HDFS) [20], a clone of Google's GFS, and utilization of the MapReduce framework [17]. Specifically, HDFS consists of a NameNode (master), multiple DataNodes (slaves), and Clients. A DFS NameNode has a role of maintaining metadata of DFS files. A DFS DataNode has a role of maintaining DFS blocks, partitions of DFS files, in replica. A DFS Client has a role of retrieving/storing DFS files to/from the user. Hadoop MapReduce framework consists of a JobTracker (master) and multiple TaskTracker (slaves). A JobTracker has a role of managing multiple tasks, and a TaskTracker a role of processing a

task. Typical Hadoop cluster is deployed with one master node consisting of a DFS NameNode and MapReduce JobTracker and multiple slave nodes consisting of a DFS DataNode, DFS Client, and MapReduce TaskTracker. A user can process queries for large-scale data in the DFS in parallel by writing Map and Reduce functions through high-level languages such as Java.

2.2 Supporting DBMS functionality in MapReduce

We discuss on the existing methods supporting partial functionality of the DBMS in MapReduce. These methods implement only some specific functionality of the DBMS. (1) There have been several efforts for supporting high-level query languages (e.g., SQL). High-level query languages provide a higher expressive power than MapReduce, and consequently, allow us easy application development and maintainance [33]. Thus, techniques have been proposed for managing parallel tasks in MapReduce using high-level languages. Specifically, they transform a user query into an equivalent MapReduce job and return the results obtained by MapReduce to the user. Representative systems include Pig [26], Hive [30], SCOPE [5], Tenzing [7], and SQL/MapReduce [14]. (2) There have been efforts for supporting indexes in the DFS. Hadoop++ [12] and HAIL [13] are the representative systems. (3) There have been efforts for supporting transactions in distributed environments for MapReduce. PNUTS by Yahoo! [8] provides a relaxed consistency, called eventual consistency, to reduce the overhead of maintaining strong consistency in distributed environments. Brantner et al. [3] have proposed a storage system that supports transactions on top of Amazon S3 system.[2] (4) There have been other related work. Blanas et al. [2] have proposed a method for supporting join operations in MapReduce. Herdotou et al. [21] have proposed a method for tuning job parameters of MapReduce by applying cost-based query optimization techniques commonly used in the DBMS. Jahani et al. [22] have proposed a method for finding the optimal query plan by applying query plan optimization techniques used in the DBMS.

Most research efforts to support the DBMS functionality have been focused on supporting partial functionality as explained above. However, Pavlo et al. [27] have shown that parallel DBMSs are more efficient than MapReduce for analytical tasks and have clarified that their performance improvement is due to 'full' functionality of the DBMS.

HadoopDB [1] is the only effort to date for supporting full functionality of the DBMS in MapReduce. Like MapReduce, HadoopDB uses the DFS to store data and uses the MapReduce framework to parallelize operations. At the same time, HadoopDB also uses DBMSs to store data and process queries. Specifically, HadoopDB partitions the data stored in the DFS and loads them into the local DBMS of each slave node in a shared-nothing manner. Then, it performs parallel processing in local DBMS units using the MapReduce framework. Since HadoopDB uses the DFS and MapReduce framework, it is scalable and fault-tolerant. It also inherits high-level functionality of the DBMS such as the schema, indexes, SQL, and query optimization that help program MapReduce operations. Thus, HadoopDB is more efficient than MapReduce. Abouzeid et al. [1] show by experiments that HadoopDB consistently outperforms MapReduce by up to $1.2 \sim 10$ times. HadoopDB proposes a parallel query processing method using SQL-to-MapReduce-to-SQL (SMS) planner where local DBMSs work in parallel by MapReduce. The SMS planner takes a SQL query given by the user and converts it to a MapReduce job, which consists of Map tasks containing SQL

[2] a distributed file system for Amazon cloud services

subqueries against local databases. Thus, each Map task has its own SQL subquery to be processed.

Odysseus/DFS is the first approach that supports scalability, fault-tolerance, and sharability by using the DFS as the storage of the DBMS [23]. The goal of Odysseus/DFS is to support NoSQL-scale scalability and fault-tolerance in the DBMS by integrating the DBMS with the DFS. It supports the data update employing the new notion of the *meta DFS file* [23]. A meta DFS file consists of multiple DFS blocks where overwrite and append can be done in the unit of a DFS block.[3] It also supports concurrency control through the distributed lock manager and recovery based on the log. In this paper, we use Odysseus/DFS as the DFS-integrated DBMS.

The F1 DBMS [28] by Google has tried to support DBMS functionality such as SQL and transactions on top of the key-value store. However, we cannot use F1 as the DFS-integrated DBMS due to the following two reasons. (1) F1 does not support sharability of the entire data when we use the system for analytics (i.e., processing a query in a 'distributed query' mode). That is, it partitions the entire data to multiple slaves in the shared-nothing manner, and each slave process manages only its own partitioned data. (2) How much DBMS functionality is provided by the F1 DBMS is not clear. It may very well inherit the drawbacks of the key-value store such as lack of flexible indexes and difficulty in modeling many-to-many relationships due to its hierarchical nature, and redundancy in data representation.

2.3 Parallel DBMSs for big data analytics

Major commercial DBMS vendors such as Oracle, IBM, and Microsoft and major open-source software DBMS developer groups such as MySQL and PostgreSQL expanded their own database engines to the shared-nothing massively-parallel processing (MPP) architecture by developing specialized engines or by composing a specialized configuration for big data analytics. The parallel DBMSs such as Aster Teradata, GreenPlum, Sybase, and Vertica also have the shared-nothing MPP architecture [4]. The shared-nothing architecture minimizes dependency between slave nodes so that each node has a high level of performance and stability like a single-node DBMS [10]. The architecture is easy to scale up by just adding new machines. However, using parallel DBMSs for big data analytics has limitation since parallel DBMSs cannot take advantage of the DFS as an integrated storage providing scalability, fault-tolerance, and load-balancing.

3 PARADISE: a new approach for big data analytics using an integrated database in the distributed file system

3.1 Architecture

We propose a new approach for big data analytics using an integrated database in the DFS. shows the architecture of PARADISE showing our approach.

In Figure 2, we use Hadoop MapReduce [17]. Here, the MapReduce TaskTracker communicates with the DBMS through the database connector, which is an interface that passes

[3] A detailed description of a meta DFS file can be found in [23].

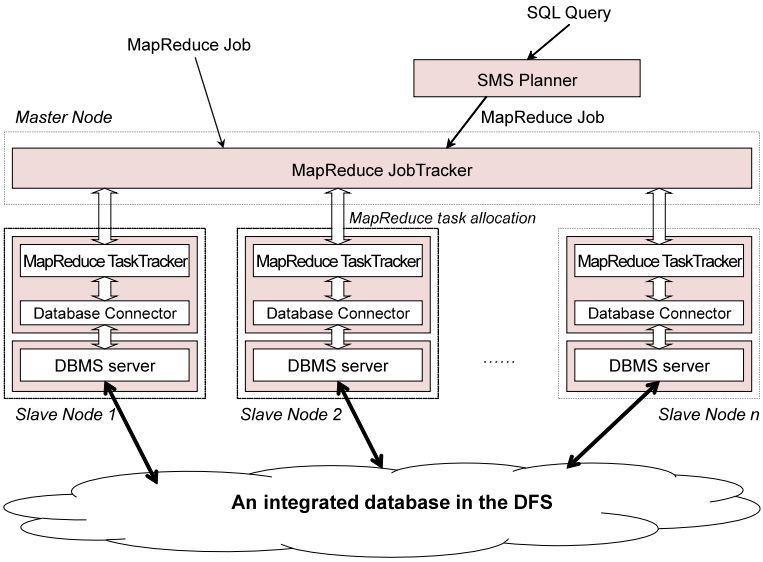


Figure 2 The architecture of PARADISE

queries and query results to and from the DBMS. We also use a DFS-integrated DBMS to store an integrated database in the DFS and to process queries. The SMS Planner converts the SQL query given by the user to multiple SQL subqueries[4] that can be processed by the DBMSs in the slave nodes using *logical splitting*, which we present in detail in Section 3.2. To operate a DFS-integrated DBMS, we should deploy DFS NameNode and DFS DataNodes in the architecture. Here, we deploy DFS DataNode at the slave nodes and DFS NameNode at the master node.

3.2 Logical splitting

3.2.1 The concept

In HadoopDB, when we convert the user query into multiple subqueries, we take advantage of the physical partition of data. That is, each subquery is processed against the fragment database stored in each local machine. However, in PARADISE, we cannot take advantage of physical partition since we have an integrated database. Thus, we use logical splitting to split the user query into multiple subqueries. The basic idea of logical splitting is to split the query based on a particular attribute of a table involved in the query. We call the table to split the *target table* and the attribute the *split attribute*. We use the clustering attribute or an attribute used in the SQL predicate as the split attribute. The logical splitting first partitions the range of the values of the split attribute and assigns each range to a different subquery, and then, for each subquery, augment the `WHERE` condition of the original SQL query with each range using the AND operation.

[4] Each subquery is processed by a Map task.

When a query is given, we choose an attribute from the target table as the split attribute using the following criteria: (1) If the query contains a selection condition and there is an index for the attribute used in the selection condition (i.e., selection attribute), we choose the selection attribute as the split attribute. Since we have an index, we can efficiently access only those selected rows. If there is more than one qualified selection attribute, we choose the most selective one as the split attribute. (2) Otherwise (i.e., if the query does not contain the selection condition or there is no index for selection attributes), we choose the clustering attribute of the target table as the split attribute. In this case, the system must scan the entire data of the target table, but each slave node can sequentially read a reduced range of the data.

3.2.2 Locality mapping

PARADISE maintains the database as an integrated one. However, since the database is stored in the DFS, it is actually stored in many DFS DataNodes networked together. Thus, if we are not careful during logical splitting, a subquery assigned to a specific node is likely to access data stored in another node through the network—incurring inefficiency. Thus, to efficiently process logical splitting, it is beneficial to store the data accessed by a subquery in the very node that processes the subquery. For this purpose, we present the notion of *locality mapping*.

Locality mapping allocates the subquery to the DFS DataNode that contains DFS blocks needed for processing the subquery. We can use locality mapping only when the second criteria of logical splitting is met, i.e., when the clustering attribute is chosen as the split attribute. In this case, we guarantee that each slave node processing a subquery reads all the DFS blocks from the local storage without incurring the network overhead of accessing data in another node. Specifically, (1) we uniformly split the value range of the clustering attribute of each table and assign them to slaves; (2) we store DFS blocks that are included in the subrange assigned to each slave in the same slave; (3) we maintain the mapping information in the master node as metadata; (4) we convert the user query according to logical splitting using this mapping information so that each slave can process the subquery by accessing the data stored in its own node. To process the step (2) above, the integrated database in the DFS should support specific features such as: (1) managing a database as a set of multiple partitioned DFS blocks and (2) the capability of storing particular DFS block to the slave node where we desire.

We can guarantee locality of data by locality mapping only when the data is loaded initially. That is, when the data is updated, we store them at an arbitrary slave node without regard to its locality. However, data update is not a serious problem when we process queries with locality mapping since (1) it does not influence correctness of the query processing and (2) the performance is not significantly affected because of a relatively small fraction of updated data.

3.2.3 The algorithm

Figure 3 shows the algorithm *Logical Splitting*. In Step 1, we choose the target table and the split attribute. A target table is determined as follows. If a query involves only one table, then it becomes the target table. If a query contains more than one table, we choose the outermost table in the query plan as the target table. We then choose the split attribute of the target table according to the criteria explained in Section 3.2.1. If the clustering attribute has been selected as the split attribute, the flag, *clustering_flag*, is set.

Algorithm ***Logical Splitting*****:**
Input: (1) an SQL query Q
(2) # of Map tasks (subqueries) M
Output: MapReduce Job J consisting of a set of Map tasks $\{T_i\}$ (i.e., $J = \{ T_1, T_2, \ldots, T_M \}$)
Algorithm:
Step1. Determine the split attribute *SPA* from the target table *TT*
1.1 If Q involves only one table T, $TT := T$
Else /* Q involves more than one table */
TT := the outermost table of Q
1.2 If a selection condition on *TT* exists, and the selection attribute has an index,
SPA := the most selective selection attribute of *TT*
clustering_flag := `FALSE`
Else
SPA := the clustered attribute of *TT*
clustering_flag := `TRUE`

Step2. Split the values of *SPA* into ranges
2.1 SPA_{min} := the minimum of *SPA* values; SPA_{max} := the maximum of *SPA* values
2.2 Uniformly split $[SPA_{min}, SPA_{max}]$ to M subranges $\{S_1, S_2, \ldots, S_M\}$

Step3.Generate SQL subqueries
3.1 For i in $[1, \ldots, M]$ Do
Q_i := AND the condition '*SPA* in range S_i' to the predicate of Q

Step4.Generate a MapReduce job
/* A task T is composed of (Q, H), where Q is the subquery and H is the host address. */
4.1 If *clustered_flag* == `FALSE`, For i in $[1, \ldots, M]$ Do $T_i := (Q_i,$ `RANDOM`$)$
/* '`RANDOM`' means that an arbitrary host can process the task T. */
Else /* If *clustered_flag* == `TRUE` */
For i in $[1, \ldots, M]$ Do
H_i := Search_metadata_for_data_location (S_i)
/* Search the metadata to find the host address of a slave node
in which the data in S_i reside */
$T_i := (Q_i, H_i)$
4.2 $J := \{T_1, T_2, \ldots, T_M\}$

Figure 3 The algorithm for the logical splitting

In Step 2, we uniformly partition the value range of the split attribute into the number of Map tasks.[5] Each subrange becomes the predicate that limits the data to be accessed by the particular subquery. This partitioning method could cause skew in load among the slaves if the data distribution is not uniform. However, MapReduce performs load balancing automatically in the unit of tasks when it assigns the tasks to the slaves. Thus, we resolve the load balancing problem by using a much larger number of Map tasks compared to the number of slave nodes[6] as suggested by Dean et al. [9]. In Step 3, the system creates a

[5]This number is configurable by the user.

[6]Here, for efficient load balancing, the number of Map tasks is recommended to be set twice or three times the number of slave nodes [9].

SQL subquery for each Map task by ANDing the subrange condition to the predicate of the original SQL query. In Step 4, we assign a subquery to each slave node, and then, generate a MapReduce job. If the *clustering_flag* is set, we assign a subquery to the node that contains the DFS blocks to be accessed achieving locality mapping. Here, we use the mapping information (metadata) stored in the master node. Example 1 illustrates the logical splitting when the split attribute is a selection attribute; Example 2 when it is the clustering attribute.

Example 1 Suppose we have the following database schema[7] for Web page visit log, PageRank of each page, and revenues from advertisement of each IP by date:

```
CREATE TABLE UserVisits ( destURL VARCHAR(100),
        sourceIP VARCHAR(16), visitDate INT, visitTime INT );
CREATE TABLE Rankings
                    ( URL VARCHAR(100), pageRank FLOAT );
CREATE TABLE AdRevenues
            ( IP VARCHAR(16), date INT, adRevenue FLOAT );
```

Suppose the UserVisits table has an index on visitTime. Consider the query that obtains the average pageRank for each sourceIP from the UserVisits table from *time_from* to *time_to*. Then, the SQL query is as follows:

```
SELECT sourceIP, AVG(pageRank)
FROM UserVisits, Rankings WHERE destURL = URL
AND visitTime BETWEEN time_from AND time_to
GROUP BY sourceIP;
```

Since this query contains an indexed selection on the visitTime attribute, visitTime is chosen to be the split attribute. Since this query needs data from *time_from* to *time_to*, we split only this range. Figure 4 shows the step by step description of the logical splitting in processing the query. Here, S_i represents the i^{th} subrange, and Q_i the subquery that reflects S_i. Typically, each slave in the MapReduce framework accesses data from one node since this is the default option of MapReduce accessing data in the unit of one DFS block, which resides in only one node. In PARADISE, however, to process a DBMS subquery, each slave should be able to access data from any other slave node. The MapReduce framework allows this option. □

Example 2 For the schema used in Example 1, suppose the UserVisits table has a clustering index on visitDate. Consider a query that obtains the count of web page visits for each sourceIP from the UserVisits table. Then, the SQL query is as follows:

```
SELECT sourceIP, COUNT(*) FROM UserVisits
GROUP BY sourceIP;
```

Since the query does not contain a selection condition, we select visitDate, the clustering attribute, as the split attribute. Figure 5 shows the step by step description of the logical

[7]This schema is the same as in Abouzeid et al. [1] except for normalizing the relations in the schema in order to show a three-way join scenario.

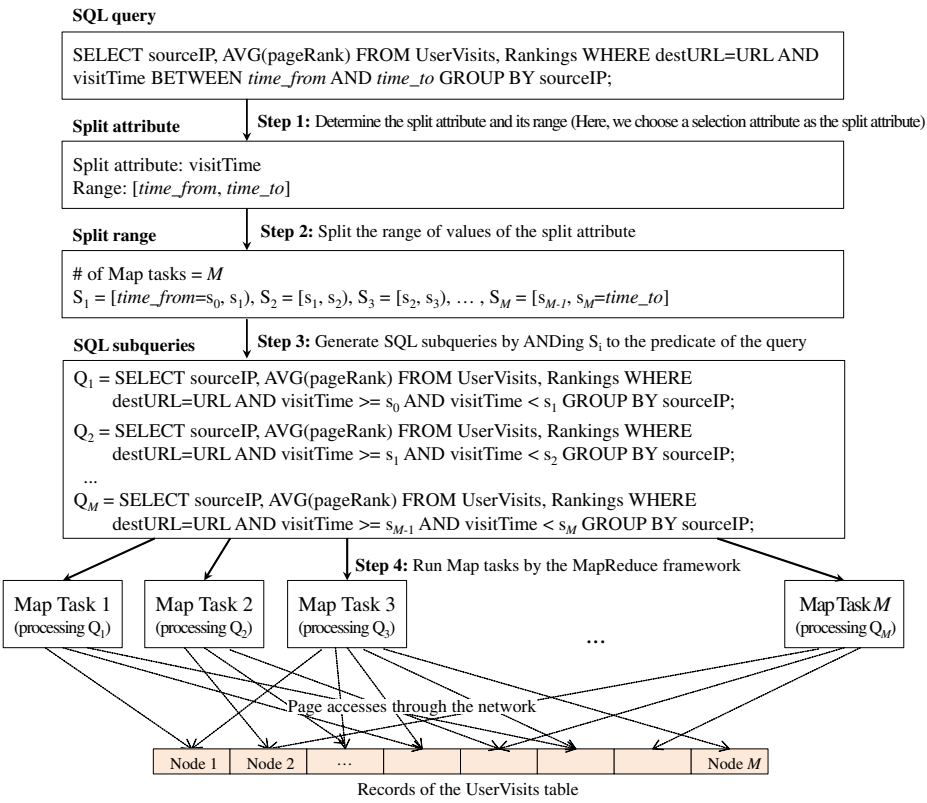


Figure 4 A step-by-step description of Example 1

splitting processing the query. Thanks to locality mapping, each slave accesses DBMS pages only locally during processing the subquery assigned to itself. □

3.3 Strengths of PARADISE compared to HadoopDB

In this section, we explain how PARADISE effectively overcomes the drawbacks of HadoopDB.

First, PARADISE does not require re-loading since it provides sharability for the entire database. In contrast, HadoopDB requires re-loading of the entire database when it cannot use the current snapshot stored in local databases since HadoopDB partitions data based on a specific attribute to process a join so as to prevent internode communication. Thus, in order to process a two-way join, the entire data should be partitioned by the join attribute before it is processed. Hence, in order to process a two-way join on a non-partitioned attribute, the entire database must be re-partitioned and re-loaded (simply, re-loaded) from the DFS to local databases. The experiments performed by Abouzeid et al. [1] show that it takes a few hundred seconds in processing a specific query, but it takes tens of thousands of seconds in partitioning and re-loading the database. This means re-loading could cause significant performance degradation. Example 3 shows this situation.

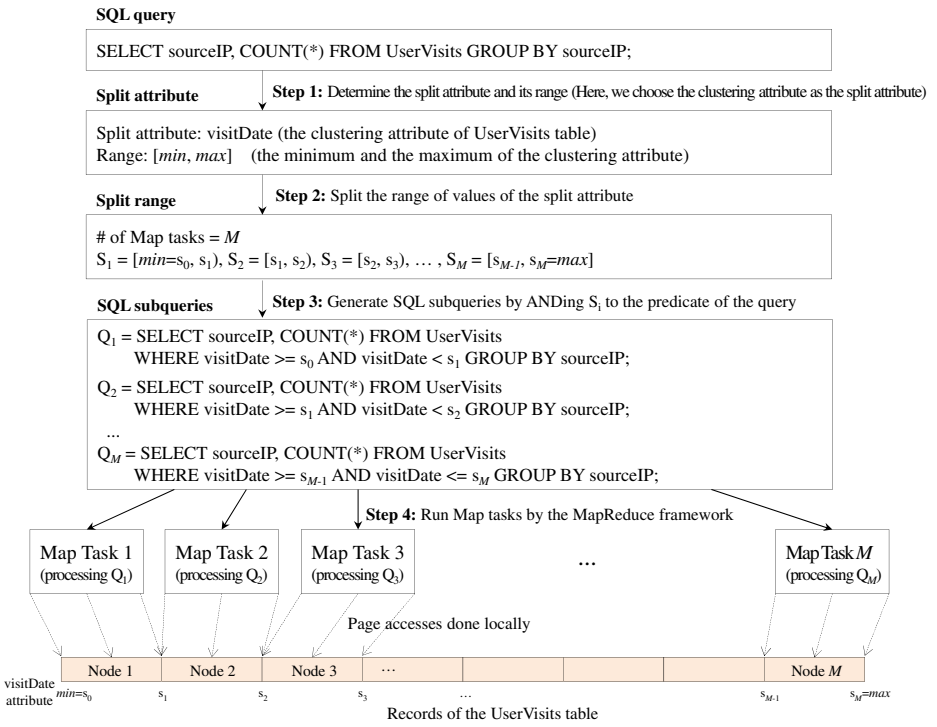


Figure 5 A step-by-step description of Example 2

Example 3 For the schema used in Example 1, consider Query 1 that obtains the average pageRank of the Web pages visited by each sourceIP on a certain date and Query 2 that obtains the sum of adRevenue for each sourceIP. Both are two-way join queries.

```
Query 1: SELECT sourceIP, AVG(pageRank)
         FROM UserVisits, Rankings
         WHERE destURL = URL GROUP BY sourceIP;
Query 2: SELECT sourceIP, SUM(adRevenue)
         FROM UserVisits, AdRevenues
         WHERE sourceIP = IP AND visitDate = date
         GROUP BY sourceIP;
```

To process Query 1, we need to join the UserVisits table with the Rankings table. To process Query 2, we need to join the UserVisits table with the AdRevenues table. HadoopDB partitions data based on the attribute used in the join predicate. For example, to process Query 1, UserVisits table should be partitioned based on the destURL attribute; to process Query 2, the table should be partitioned based on the sourceIP attribute. Suppose the UserVisits table has already been partitioned based on destURL to process Query 1. Then, to process Query 2 we need to re-partition the table based on the sourceIP attribute and re-load it. □

In contrast, in PARADISE, re-partitioning and re-loading the database is not needed. Thus, query processing performance is significantly improved. For example, Query 1 and

Query 2 in Example 3 can be continuously processed using the integrated database stored in the DFS without re-loading. In Section 4.2.3, the performance results indicate that PARADISE outperforms HadoopDB by up to 6.41 times (or larger as the database size increases) due to the re-loading overhead of HadoopDB.

Second, PARADISE can support more complex query types than HadoopDB. HadoopDB does not support queries that require internode communication. Cartesian products and three-way joins are typical queries that are not supported. In order to support these queries, some parallel DBMSs employs a broadcasting function for each node to distribute its data to the other nodes. However, HadoopDB does not support the broadcasting function. Example 4 shows a situation where internode communication is required together with an example query that cannot be processed by HadoopDB.

Example 4 Consider Query 3 that obtains the average pageRank and the sum of adRevenue for each sourceIP from the UserVisits table for the schema in Example 1. Query 3 is a three-way join.

```
Query 3: SELECT sourceIP, AVG(pageRank), SUM(adRevenue)
         FROM UserVisits, Rankings, AdRevenues
         WHERE destURL = URL AND sourceIP = IP
         AND visitDate = date GROUP BY sourceIP;
```

Since HadoopDB does not allow internode communication, Query 3 can be processed only by re-partitioning and re-loading the database. To process Query 3, we need to join the UserVisits table with the Rankings table via destURL attribute and join it with the AdRevenues table via sourceIP and visitDate attributes. In HadoopDB, to do the former join, we must partition the UserVisits table based on the destURL attribute; to do the latter join we must partition the UserVisits table based on the sourceIP or visitDate attribute. However, since it is impossible to partition a table based on two different attributes simultaneously, HadoopDB cannot process the query effectively. □

In contrast, PARADISE can process all the query types including Cartesian product or three-way joins since the data in the DFS are treated as one integrated database. For example, Query 3 in Example 4 can be processed in PARADISE.

3.4 Performance overheads of PARADISE compared to HadoopDB

In this section, we discuss the performance overheads of PARADISE compared to HadoopDB when processing the queries that do not require re-loading of the database. While HadoopDB can process those queries simply by accessing data locally, PARADISE does so by accessing the DFS. In the DFS, the DBMS that processes a query is likely to reside in a slave node different from the slave node that physically contains the data to be accessed. Overheads are incurred due to this mismatch. We classify them into three types: (1) *disk arm contention overhead*, (2) *network transfer overhead*, and (3) *network bottleneck overhead*. We explain these overheads in detail; we analyze them through extensive experiments in Section 4.3.

The disk arm contention overhead is caused by disk arm contention when DFS I/O requests are concentrated on a specific slave node. The DBMS in an arbitrary slave node can request a DFS I/O to any node. Hence, multiple DBMSs could simultaneously request an I/O request to a specific slave node, causing disk arm contention and queuing delay. This

overhead is contrasted with that in HadoopDB where I/O operations for query processing are confined only to the local database exclusively accessed by the local DBMS.

The network transfer overhead is incurred whenever an I/O request is made to the DFS, requiring data (including metadata) access through the network. A DFS Client accesses metadata in the DFS NameNode, requests the corresponding DFS block to a DFS DataNode, and then, the DFS DataNode transmits the DFS block to the DFS Client. These actions incur network transfer delay. Naturally, the network transfer overhead is proportional to the number of I/O requests to the DFS since each I/O request incurs a constant delay. For example, when we access large data sequentially, this overhead is minimized since sequential access incurs only a few I/O requests. On the other hand, when we randomly access data, this overhead is heavily incurred since random access incurs a number of I/O requests.

The network bottleneck overhead is caused by insufficient network speed. In general, the network speed (typically, about 80MB/s) is not as fast as the average disk transfer rate (typically, 120MB/s), and thus, the system cannot fully take advantage of the disk transfer rate, causing performance degradation.[8] We note that this overhead is incurred mainly when we sequentially access data. (i.e., when we are exploiting the maximum transfer rate of the disk.) In contrast, when we access data randomly, the disk speed slows down by excessive movement of the disk arm. Therefore, the disk speed does not become faster than the network speed so that this overhead becomes minimal. We also note that this overhead is not a fundamental one. Just using a network switch faster than the disk transfer rate solves the problem. Since the network speed evolves at a rate almost the same as or faster than that of the disk transfer rate[9] [25], this overhead would not be a serious factor in the future.

4 Performance evaluation

4.1 Experiment setting

In this section, we present the experiments that compare the query processing performance of three systems: HadoopDB, PARADISE, and the 'Hadoop system' that consists of HDFS, Hbase, and Hadoop MapReduce for big data analytics. In the experiments, we use the following queries for big data analytics: scan, aggregation, selection, and join. In order to set up the systems, we use a cluster of nine nodes: one master and eight slaves. Each node consists of 3.2GHz Intel Quad-Core CPU, 8GB RAM and one 1TB hard disk. Nodes are connected by 1Gbps network switches. The average transfer rate of hard disk is 120MB/s. The average network transfer rate is 80MB/s. In the case of HadoopDB, there is one local database for each slave node; thus, there are eight local databases in the cluster.

We use the same DBMS for both PARADISE and HadoopDB[10] for fair comparison. We use the Odysseus [31, 32, 34] DBMS for this purpose. For PARADISE, we use Odysseus/DFS [23], the DFS-integrated version of the Odysseus DBMS. We use Hadoop version 1.0.3, which consists of two core subsystems: HDFS and Hadoop MapReduce [18].

[8] The average transfer rate of contemporary local storage reaches more than 120MB/s and is constantly increasing. Even though the theoretical network speed of 1Gbps switch is 128MB/s, the actual maximum transfer rate is about 80MB/s due to header(non-payload) transfer [35].

[9] It is known that disk transfer rate doubles approximately in 24 months; network speed approximately in 18 months [25].

[10] Since HadoopDB is not released in public, we implemented it according to the architecture described in [1].

```
dfs.datanode.max.xcievers = 4096
mapred.map.tasks.speculative.execution = false
mapred.reduce.tasks.speculative.execution = false
mapred.tasktracker.map.tasks.maximum = 1
mapred.tasktracker.reduce.tasks.maximum = 1
```

Figure 6 The configuration of HDFS and MapReduce

```
CREATE TABLE Rankings ( pageURL VARCHAR(100)
                        PRIMARY KEY, pageRank INT, avgDuration INT );
CREATE TABLE UserVisits ( sourceIP VARCHAR(16),
                          destURL VARCHAR(100), visitDate DATE,
                          adRevenue FLOAT, userAgent VARCHAR(64),
                          countryCode VARCHAR(3), languageCode VARCHAR(6),
                          searchWord VARCHAR(32), duration INT );
CREATE INDEX rankings_pageurl_index ON Rankings (pageURL);
CREATE INDEX uservisits_visitdate_index ON UserVisits (visitDate);
```

Figure 7 The schema of the database used in the experiments.

The configuration of HDFS and MapReduce is based on the defaults with some variations described in Figure 6. Specifically, in HDFS, the maximum number of connections was set to be 4096 (default 256) in order to respond to multiple page accesses requested from the DBMSs. In MapReduce, the speculative execution flag was set to false in order to avoid running a task in more than one slave node simultaneously, and the maximum task parameter was set to 1 in order to avoid running more than one task in a single slave node simultaneously. We use Hbase version 0.94.7. Hbase is also configured using the defaults.

We implemented the Database Connector, the Data Loader, and the Catalog of HadoopDB as described in Abouzeid et al. [1]. The Database Connector is common to both PARADISE and HadoopDB while the Data Loader and the Catalog are used only for HadoopDB. The SMS planner for HadoopDB was implemented as specified by Abouzeid et al. [1], and that for PARADISE was implemented as explained in Section 3.2. All the subsystems listed above was implemented in Java.

In the experiments, we have used the synthetic data generated by Pavlo et al. [27].[11] Its schema is similar to that in Example 3. In Example 3, we have normalized the original UserVisits table to show a three-way join scenario; here, we use the original table itself without normalization. There are 37 million tuples for the Rankings tables and 155 million tuples for the UserVisits table. The schema of the database is described in Figure 7. An index is created on the pageURL attribute of the Rankings table, and one on the visitDate attribute of the UserVisits table. For Hbase, we store each value in a tuple by using a combination of a row key, column key, and value. In other words, to store a value, we assign an integer tuple identifier to each tuple using it as the row key and use the attribute name of the value as the column key. Since Hbase does not have DBMS functionality, we cannot use certain features in Hbase. For example, we cannot create an index in Hbase since it does not support a secondary index. In addition, the clustering of data in Hbase is fixed on the row key (the tuple identifier), while we can cluster tuples of a table on an arbitrary attribute in HadoopDB or PARADISE so that we can observe the effect of clustering.

[11] This data set was used by Abouzeid et al. [1] to show the performance of HadoopDB.

```
Scan (grep): SELECT * FROM UserVisits
               WHERE destURL LIKE '%foo%';
Aggregation: SELECT sourceIP, SUM(adRevenue)
               FROM UserVisits GROUP BY sourceIP;
Selection: SELECT sourceIP, adRevenue, visitDate
             FROM UserVisits WHERE visitDate
             BETWEEN '20000110' AND '20000125';
Join: SELECT sourceIP, AVG(pageRank), SUM(adRevenue)
      FROM Rankings AS R, UserVisits AS UV
      WHERE R.pageURL = UV.destURL AND
      UV.visitDate BETWEEN '20000115' AND '20000122'
      GROUP BY sourceIP;
```

Figure 8 The SQL queries used in the experiments

To experiment with HadoopDB, we partition and load the data in the DFS to eight local databases. As was done by Abouzeid et al [1], we partition the UserVisit table based on the destURL attribute and the Rankings table based on the pageURL attribute.[12] Figure 8 shows the SQL queries[13] processed by both PARADISE and HadoopDB for the experiments. We also hand-wrote the equivalent MapReduce programs for the Hadoop system. Queries used by Abouzeid et al. [1] have two aggregation queries: large and small, representing the number of groups in the results of the queries. In the paper, however, we present the results of only large queries since two results have a similar tendency.

The query processing time is the elapsed time of each MapReduce job, which can be measured using the MapReduce Administration Tool. We average the elapsed times of five identical executions of each query. In order to obtain consistent results, we flush the DBMS buffers, O/S file buffers, and disk buffers before executing each query. In other words, we conduct all the experiments in *cold start*. To represent the performance of PARADISE compared with that of HadoopDB, we define the overhead as shown in Equation (1).

$$overhead = \frac{T(PARADISE) - T(HadoopDB)}{T(HadoopDB)} \qquad T(X): \ elapsed\ time\ of\ the\ system\ X. \tag{1}$$

4.2 Performance results

4.2.1 Results of scan (grep) and aggregation queries

Figure 9 shows the performance results of scan (grep) and aggregation queries. The queries are based on the UserVisits table. Scan and aggregation read the entire data. For HadoopDB and PARADISE, we choose the destURL attribute to cluster the UserVisits table. The results are shown in Figure 9. 'PARADISE (w/o locality mapping)' in the figure indicates the performance of PARADISE that does not utilize the locality mapping feature described in Section 3.2.2; 'PARADISE' in the figure utilizes locality mapping. We can show the effectiveness of locality mapping in the experiments. PARADISE(w/o locality mapping) is 38∼46 % slower than HadoopDB for scan and aggregation queries due to overheads of

[12] In scan (grep), aggregation, and selection queries, whichever attribute is used for partitioning does not affect the results. However, in a join query, data should be partitioned based on the join attribute, e.g., the attributes destURL and pageURL in Figure 9.

[13] This SQL queries are the same as used by Abouzeid et al. [1].

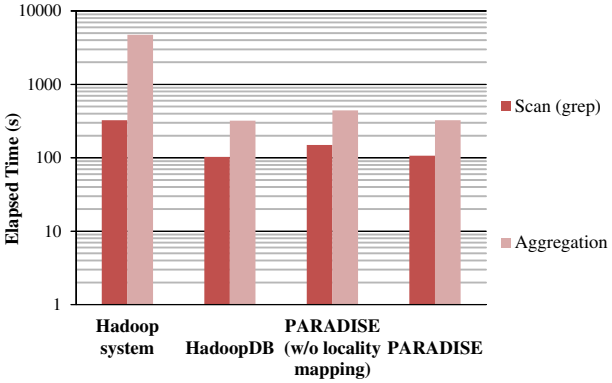


Figure 9 Results of scan (grep), aggregation queries

accessing data through the DFS as explained in Section 3.4. We will analyze these overheads incurred during sequential access in detail in Section 4.3.1. However, in PARADISE, scan and aggregation queries show almost no degradation compared to HadoopDB since locality mapping allows reading all the data needed from the local DFS DataNode, obviating all the overheads of PARADISE over HadoopDB. We also showed that the Hadoop system is 3.1∼14.7 times slower than HadoopDB and 3.0∼14.6 times slower than PARADISE. The reason for this slowdown is due to the columnar storage of Hbase incurring random disk access during scan.

4.2.2 Results of the selection query

Figure 10 shows the performance results of selection queries using the UserVisits table. The experiments consider two cases: the selection attribute (visitDate) is (1) a clustering attribute, (2) a non-clustering attribute. The results are as follows. (1) In the former, we do not find any notable performance difference between PARADISE and HadoopDB since all the access to data is done sequentially. We omitted experiments for the Hadoop system since it cannot cluster data on the attribute desired. (2) In the latter, PARADISE incurs 123 % overhead compared to HadoopDB since random access to data causes the overheads discussed in Section 3.4. We will analyze these overheads incurred during random access in detail in Section 4.3.2. We also showed that the Hadoop system is 13.6 times slower than HadoopDB due to lack of secondary indexes in the Hadoop system, incurring a full data scan.

4.2.3 Results of the join query

Figure 11 shows the performance results of the join query. We use the same join algorithm for PARADISE and HadoopDB, i.e., a nested-loop join algorithm that is supported by Odysseus DBMS. To compose an experiment with a reasonable processing time, we assume that the outer table of the join query has a selection predicate. Thus, the join query used in the experiment has a join predicate and a selection predicate for the outer table as shown in Figure 8. Thus, the query processor first evaluates the selection predicate for the outer table and, for each qualified tuple in the outer table, traverses the tuples of the inner table that match the join attribute value. We have performed the following two experiments: (1) the

clustering case and (2) the non-clustering case. For the former, the outer table is clustered on the selection attribute, and the inner table is clustered on the join attribute. For example, for the join query described in Figure 8, the UserVisits table is clustered on the visitDate attribute, and the Rankings table on the pageURL attribute. We omitted experiments in the Hadoop system in this case since it cannot cluster data on the attribute desired. For the latter, each table that participates in the join query is clustered neither on the join attribute nor on the selection attribute. In the experiments for HadoopDB, which need prior partitioning of the entire database into local databases, data are (should be) partitioned on the join attributes.

We observe the following from the experiments:

- Figure 11 shows that the overhead of PARADISE in the join query is 87 % for the clustering case and 69 % for the non-clustering case. We observe that the overhead of PARADISE in the join query for the clustering case (87 %) is much larger than that of the selection query for the clustering case (close to 0 %) in Figure 10. We have this phenomenon since we cannot take advantage of sequential access in the inner table due to the nested-loop join used; i.e., random access occurs in the inner table for both PARADISE and HadoopDB, and PARADISE suffers more in performance in random access as shown in Figure 10. In contrast, the overhead of the join query in the non-clustering case (69 %) is smaller than that of the selection query in the non-clustering case (123 %) in Figure 10. This overhead reduction is due to the buffering effect. That is, in processing the join query, if the outer table finds the tuples from the inner table that have already been retrieved before, the buffering becomes effective.
- Figure 11 also shows that the Hadoop system is 13.5 times slower than HadoopDB. Since the Hadoop system cannot create a secondary index, a full data scan is needed when processing join.

4.2.4 Join performance of HadoopDB with re-load

When processing join queries, in cases where HadoopDB cannot use the current snapshot of the database in the local databases, its performance degrades significantly since re-loading is required. The elapsed time for re-loading consists of (1) the time for sorting & partitioning the original data, (2) the time for loading the partitioned data to local databases, and (3) the time for index creation. In our experiments, it takes 2,760 seconds for step (1), 193 seconds for step (2), and 127 seconds for step (3). In total, the elapsed time for re-loading is 3,080

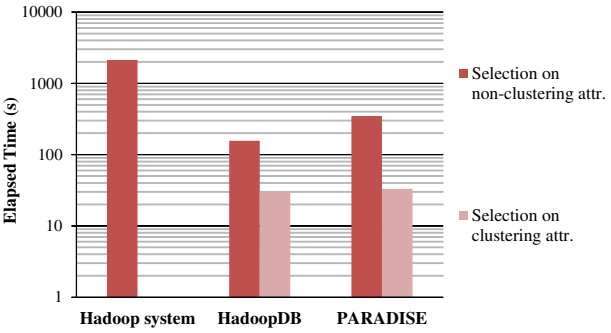


Figure 10 Results of selection queries

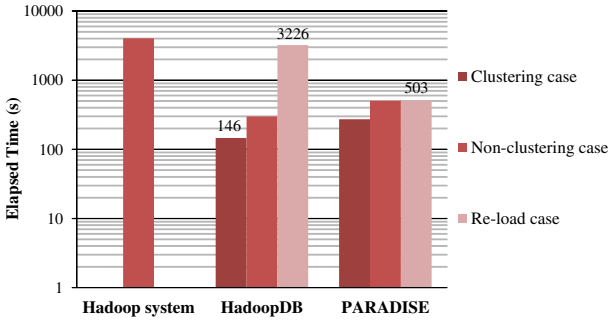


Figure 11 Results of join queries

seconds. In addition, the elapsed time for join query processing in the clustering case is 146 seconds as shown in Figure 11. Hence, the total processing time of the join query with re-loading in HadoopDB is 3,080+146 = 3,226 seconds as shown in Figure 11. As we observe in this experiment, re-loading in HadoopDB is a very time-consuming operation, taking 95 % of the total processing time. This indicates that the need for re-loading is one of the most significant drawbacks of HadoopDB.

In contrast, re-loading is not required in PARADISE. Even in the worst case where PARADISE cannot take advantage of clustering, the join query processing takes 503 seconds as shown in Figure 11. Therefore, PARADISE outperforms HadoopDB in processing the join query by up to 6.41 times when re-loading is required for HadoopDB, and this advantage gets bigger as the size of the database grows.

4.3 Analysis of performance overhead

We analyze the overheads incurred in PARADISE by performing queries in a controlled workload. Here, we analyze the overheads in the case of pure sequential access[14] and in the case of pure random access. We run a simple counting query using the UserVisits table (i.e., '`SELECT COUNT(*) FROM UserVisits;`') for sequential access; we run a selection query on a non-clustering attribute using the UserVisits table (i.e., '`SELECT * FROM UserVisits WHERE visitDate BETWEEN '20000110' AND '20000125';`' for random access.

The usual setting of the experiment would incur all three types of overheads that we discussed in Section 3.4: disk arm contention overhead (simply, $OH_{disk-arm}$), network transfer overhead (simply, $OH_{net-transfer}$), and network bottleneck overhead (simply, $OH_{net-bottleneck}$). Here, we assume that the effect of the three types of overheads are mutually independent. In order to delineate the impact of each type of overhead, we control the setting as follows. First, to remove the disk arm contention overhead, the entire query is processed in a single slave node (simply, *in 1-node*). Second, to remove the network bottleneck overhead, we employ O/S level throttling[15] of the disk transfer rate. We perform

[14] Here, to observe the specified overheads, we do not utilize locality mapping features.

[15] To throttle disk transfer rate at the O/S level, type to shell: echo "253:3 52428800" > /sys/fs/cgroup/blkio/blkio.throttle.read_bps_device. Here, 253:3 means the "major:minor" number of the device in the UNIX system, and 52428800 means the maximum limit of the disk transfer rate in bytes/sec.

Workloads / Overheads	Controlled Workloads (selection on a non-clustering attribute)		
	(A): No throttle	(B): In 1-node, no throttle	(C): In 1-node, throttle
Network transfer overhead ($OH_{net\text{-}transfer}$)	O	O	O
Network bottleneck overhead ($OH_{net\text{-}bottleneck}$)	O	O	X
Disk arm contention overhead ($OH_{disk\text{-}arm}$)	O	X	X
Overhead Composition	$OH_{net\text{-}transfer}$ $\times OH_{net\text{-}bottleneck}$ $\times OH_{disk\text{-}arm}$	$OH_{net\text{-}transfer}$ $\times OH_{net\text{-}bottleneck}$	$OH_{net\text{-}transfer}$

Figure 12 Overhead analysis using controlled workloads

experiments in both throttle and no throttle cases. With the throttle set, we artificially control the disk transfer rate so that the network speed is sufficiently faster than the disk transfer rate. Hence, the experiments with the throttle set are to test the query without the network bottleneck overhead while those without are to test the query with the overhead.

Figure 12 represents the overhead types that actually occur in each controlled workload. In workload (A), all the three types of overhead occur; thus, the overhead for workload (A) is obtained as $OH_{net-transfer} \times OH_{net-bottleneck} \times OH_{disk-arm}$. For workload (B), since the query is processed in 1-node, $OH_{disk-arm}$ does not occur; thus, the overhead for workload (B) is obtained as $OH_{net-transfer} \times OH_{net-bottleneck}$. For workload (C), since the query is processed in 1-node under the throttle mode, $OH_{net-bottleneck}$ and $OH_{disk-arm}$ do not occur; thus, the overhead for workload (C) is $OH_{net-transfer}$. As a result, we obtain the following equations for computing the overheads.

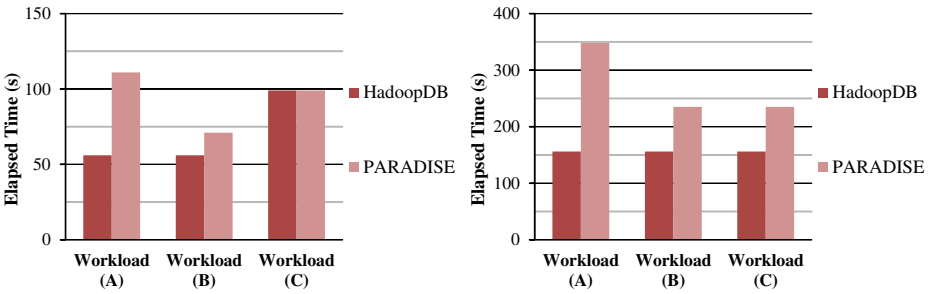
$$\bullet OH_{net-transfer} = (C) \tag{2}$$

$$\begin{aligned} &\bullet\ OH_{net-bottleneck} \\ &= OH_{net-transfer} \times OH_{net-bottleneck} \ / \ OH_{net-transfer} \\ &= (B) \ / \ OH_{net-transfer} \end{aligned} \tag{3}$$

$$\begin{aligned} &\bullet\ OH_{disk-arm} \\ &= \frac{OH_{net-transfer} \times OH_{net-bottleneck} \times OH_{disk-arm}}{OH_{net-transfer} \times OH_{net-bottleneck}} \\ &= (A) \ / \ OH_{net-transfer} \ / \ OH_{net-bottleneck} \end{aligned} \tag{4}$$

4.3.1 Pure sequential access

In Figure 13a, we show the 'non-controlled' performance of the simple counting query, representing the workload (A) defined in Figure 12; we also show controlled performance of the same query in the <1-node, no throttle> and <1-node, throttle> cases representing workloads (B) and (C) defined in Figure 12, respectively. Using (2) ∼ (4), we obtain each type of overhead from the experimental results shown in Figure 13a.



(a) Pure sequential access (b) Pure random access

Figure 13 Results of controlled workloads for analyzing overheads

- The overhead in the workload (C) is almost 0 %[16] from Figure 13(a). Therefore, the network transfer overhead is 0 % from (2), and we conclude that simple counting (i.e., pure sequential access) incurs almost no network transfer overhead. This result is obvious since sequential access incurs only a few DFS I/O requests constituting network transfer overhead.
- The overhead in the workload (B) is 26.8 % as in Figure 13a. Thus, the network bottleneck overhead is 26.8 % (i.e., 1.268/1.0 = 1.268) from (3). As we mentioned earlier in Section 3.4, in sequential access, the network speed is slower than the disk speed, incurring a certain amount of network bottleneck overhead.
- The overall overhead of the simple counting (i.e., the workload (A)) is 98.2 % as shown in Figure 13a. Thus, the disk arm contention overhead is 56.3 % (i.e., 1.982/1.0/1.268 = 1.563) from (4). This overhead depends on the number of subqueries that are concurrently processed in the same slave node and the number of disks (i.e., the number of disk arms) in the slave node.

4.3.2 Pure random access

In Figure 13b, we show the 'non-controlled' performance of the selection query on a non-clustering attribute, representing the workload (A) defined in Figure 12; we also show controlled performance of the same query in the <1-node, no throttle> and <1-node, throttle> cases representing workloads (B) and (C) defined in Figure 12, respectively. Using (2) ∼ (4), we obtain each type of overhead from the experimental results shown in Figure 13b.

- The overhead in the workload (C) is 50.6 % from Figure 13(b). Therefore, the network transfer overhead is 50.6 % from (2).
- The overhead in the workload (B) is 50.6 % as in Figure 13b. Thus, the network bottleneck overhead is almost 0 % (i.e., 1.506/1.506 = 1.0) from (3), and we conclude that selection on a non-clustering attribute (i.e., pure random access) incurs almost no network bottleneck overhead. This result is obvious since the data rate retrieved from disk for random access is far less than the network speed.

[16]The results are measured in a unit of a second. Therefore, in this case, we cannot find out the network transfer overhead effectively because the delay incurred by the network transfer is much less than a second.

– The overall overhead of the selection query on a non-clustering attribute (i.e., the overhead in the workload (A)) is 123 % as shown in Figure 13b. Thus, the disk arm contention overhead is 48 % (i.e., 2.23/1.506/1.0 = 1.48) from (4).

4.4 Comparison of performance

According to the experiments, PARADISE is much more efficient than HadoopDB in the case of join queries with re-load. Specifically, in this case, the performance of PARADISE outperforms HadoopDB by up to 6.41 times. This performance improvement would be much larger as the database size grows since the time for data loading linearly increases in the database size while the time for query processing increases logarithmically thanks to the use of the DBMS indexes.

When HadoopDB does not need re-loading, the performance of PARADISE is degraded by 0 % ∼ 123 % compared to HadoopDB due to the three types of overheads described in Section 3.4. Specifically, for the scan query, aggregation query, or selection query on the clustering attribute where sequential access is prevalent, PARADISE does not have any notable overheads compared to HadoopDB. For the selection query on the non-clustering attribute where random access is prevalent, PARADISE has 123 % of the overhead compared to HadoopDB; for the join query in the clustering and non-clustering cases, PARADISE has 87 % and 69 % of the overheads, respectively, compared to HadoopDB. Nevertheless, we claim that these overheads are modest in the light of excellent advantages—sharability, no need for re-loading, support of complex query types including three-way joins—of PARADISE that stem from the integrated database described in Section 3.3.

To delineate each type of overhead, we have analyzed the performance of the selection query on a non-clustering attribute. As a result, we have obtained that the network transfer overhead is 50.6 %; the network bottleneck overhead 0 %; the disk arm contention overhead 48 % in the selection query. In contrast, in the case of a selection query on a clustering attribute, we note that there is no overhead. We also note that the disk arm contention overhead can be reduced by assigning more disks in each slave effectively.

5 Conclusions

In this paper, we have proposed a new parallel processing approach for big data analytics, PARADISE, that uses an integrated database in the DFS. PARADISE uses the DFS-integrated DBMS as a base storage to support sharability of the entire data. The contributions of the paper are as follows. First, we have identified drawbacks of HadoopDB, which to date is the only method that directly uses the DBMS for big data analytics, and have shown that how PARADISE effectively resolves them. Specifically, (1) PARADISE outperforms the query performance of HadoopDB by up to 6.41 times when re-loading is required, and the advantage gets bigger as the database size grows; (2) PARADISE supports more complex query types, such as 3-way join and Cartesian product queries, than HadoopDB does. Second, we have proposed *logical splitting* as the job splitting method when using the DFS-integrated DBMS. Logical splitting enables efficient parallel query processing in an integrated database. Furthermore, we have proposed the notion of locality mapping for further optimization of logical splitting. Third, we have analyzed three types of performance overheads of PARADISE compared to HadoopDB through extensive experiments: (1) disk arm contention overhead, (2) network transfer overhead, and (3) network

bottleneck overhead. We note that our method of analyzing the performance can be applied not only to PARADISE but also to any other applications that concurrently access the DFS in multiple nodes.

Acknowledgments This work was supported by the National Research Foundation of Korea (NRF) grant funded by Korean Government (MSIP) (No. 2012R1A2A1A05026326).

References

1. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Rasin, A., Silberschatz, A.: HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads, In Proceedings of 35th Int'l Conf. on Very Large Data Bases (VLDB), pp. 922–933, Lyon, France (2009)
2. Blanas, S., Patel, J., Ercegovac, V., Rao, J., Shekita, E., Tian, Y.: A Comparison of Join Algorithms for Log Processing in MapReduce," In Proc. 2010 ACM Int'l Conf. on Management of Data (SIGMOD), pp. 975–986, Indianapolis, Indiana (2010)
3. Brantner, M., Florescu, D., Graf, D., Kossmann, D., Kraska, T.: Building a database on S3," In Proc. 2008 A C M Int'l Conf. on Management of Data (SIGMOD) pp. 251–264, Vancouver, Canada (2008)
4. Beyer, M., Feinberg, D., Adrian, M., Edjlali, R.: Magic Quadrant for Data Warehouse Database Management Systems, Gartner Reports (2012)
5. Chaiken, R., Jenkins, B., Larson, P., Ramsey, B., Shakib, D., Weaver, S., Zhou, J.: SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets In Proc. 34th Int'l Conf. on Very Large Data Bases (VLDB), pp. 1265–1276 Auckland, New Zealand (2008)
6. Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A., Gruber, R.: BigTable: A Distributed Storage System for Structured Data, In Proceedings of 6th Symposium on Operating Systems Design and Implementation (OSDI), pp. 205–218, Seattle, Washington (2006)
7. Chattopadhyay, B., et al.: Tenzing – A SQL Implementation On The MapReduce Framework, In Proceedings of 37th Int'l Conf. on Very Large Data Bases (VLDB), pp. 1318–1327, Seattle, Washington, Aug.–Sept. (2011)
8. Cooper, B., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H., Puz, N., Weaver, D., Yerneni, R.: PNUTS: Yahoo!'s Hosted Data Serving Platform, In Proceedings of 34th Int'l Conf. on Very Large Data Bases (VLDB), pp. 1277–1288, Auckland, New Zealand (2008)
9. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, In Proceedings of 4th Symposium on Operating Systems Design and Implementation (OSDI), pp. 137–150, San Francisco, California (2004)
10. DeWitt, D., Gray, J.: Parallel Database Systems: The Future of High-Performance Database Systems. Commun. ACM **35**(6), 85–98 (1992)
11. The Digital Universe. http://www.emc.com/leadership/programs/digital-universe.htm
12. Quiane-Ruiz, J., Jindal, A., Kargin, Y., Setty, V., Schad, J.: Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing)," In Proc. 36th Int'l Conf. on Very Large Data Bases (VLDB), pp. 515–529, Singapore, Sept. (2010)
13. Dittrich, J., Quiane-Ruiz, J., Richter, S., Schuh, S., Jindal, A., Schad, J.: Only Aggressive Elephants Are Fast Elephants, In Proceeidngs 38th Int'l Conf. on Very Large Data Bases (VLDB), pp. 1591–1692, Istanbul, Turkey (2012)
14. Friedman, E., Pawlowski, P., Cieslewicz, J.: SQL/MapReduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions, In Proceedings 35th Int'l Conf. on Very Large Data Bases (VLDB), pp. 1402–1413, Lyon, France (2009)
15. Gantz, J., Reinsel, D.: Extracting Value from Chaos, IDC iView (2011)
16. Ghemawat, S., Gobioff, H., Leung, S.: The Google File System, In Proceedings 19th ACM Symposium on Operating Systems Principles(SOSP), pp. 29–43, BoltonLanding, New York (2003)
17. Hadoop, M.apReduce. http://hadoop.apache.org
18. Hadoop, P.roject. http://hadoop.apache.org
19. Han, J., Kamber, M.: Data Mining: Concepts and Techniques. Morgan Kaufmann (2006)
20. HDFS. http://hadoop.apache.org
21. Herdotou, H., Babu, S.: Profiling, Whatif Analysis, and Costbased Optimization of MapReduce Programs, In Proceedings 37th Int'l Conf. on Very Large Data Bases (VLDB), pp. 1111–1122, Seattle, Washington (2011)

22. Jahani, E., Cafarella, M., Re, C.: Automatic Optimization for MapReduce Programs, In Proceedings 37th Int'l Conf. on Very Large Data Bases (VLDB), pp. 385–396, Seattle, Washington (2011)
23. Kim, J., Whang, K., Kwon, H., Song, I.: Odysseus/DFS: Integration of DBMS and the Distributed File System for Transaction Processing on Big Data, CoRR Technical Report (CS.DB/arXiv:1406.0435) (2014)
24. Lymna, P., Varian, H.: How Much Information?, Project Report, School of Information Management and Systems, University California at Berkeley (2003). http://www.sims.berkeley.edu/research/projects/how-much-info-2003
25. Morgan, T.: Can network architectures break the speed limit?, Enterprise Tech. (2011). http://www.theregister.co.uk/2011/10/10/network_architecture
26. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: "Pig Latin: A Not-So-Foreign Language for Data Processing," In Proc. 2008 ACM Int'l Conf. on Management of Data (SIGMOD), pp. 1099–1110, Vancouver, Canada (2008)
27. Pavlo, A., Paulson, E., Rasin, A., Abadi, D., DeWitt, D., Madden, S., Stonebraker, M.: A Comparison of Approaches to Large-Scale Data Analysis, In Proceedings 2009 ACM Int'l Conf. on Management of Data (SIGMOD), pp. 165–178, Providence, Rhode Island (2009)
28. Shute, J., et al.: F1: A Distributed SQL Database That Scales, In Proceedings of the 39th Int'l Conf. on Very Large Data Bases (VLDB), pp. 1068–1079, Riva del Garda, Italy (2013)
29. Stonebraker, M., Abadi, D., DeWitt, D., Madden, S., Paulson, E., Pavlo, A., Rasin, A.: MapReduce and Parallel DBMSs:Friends or Foes? Commun. ACM **53**, 64–71 (2010)
30. Thusoo, A., Sarma, J., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive - A Warehousing Solution Over a Map-Reduce Framework, In Proceedings 35th Int'l Conf. on Very Large Data Bases (VLDB), pp. 1626–1629, Lyon, France (2009)
31. Whang, K., Lee, M., Lee, J., Kim, M., Han, W.: Odysseus: a High-Performance ORDBMS Tightly-Coupled with IR Features, In Proceedings 21st IEEE Int'l Conf. on Data Engineering (ICDE), pp. 1104–1105, Tokyo, Japan. This paper received the Best Demonstration Award (2005)
32. Whang, K., Lee, J., Kim, M., Lee, M., Lee, K.: Odysseus: a High-Performance ORDBMS Tightly-Coupled with Spatial Database Features, In Proceedings 23rd IEEE Int'l Conf. on Data Engineering (ICDE), pp. 1493–1494, Istanbul, Turkey (2007)
33. 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 2013 ACM Int'l Conf. on Management of Data (SIGMOD), pp. 313–324, New York, New York (2013)
34. Whang, K., Lee, J., Lee, M., Han, W., Kim, M., Kim, J.: DB-IR integration using tight-coupling in the Odysseus DBMS, World Wide Web (2013). doi:10.1007/s11280-013-0264-y
35. Woligroski, D.: Gigabit Ethernet: Dude, Where's My Bandwidth?, Bestofmedia Group (2009). http://www.tomshardware.com/reviews/gigabit-ethernet-bandwidth,2321.html