Distrib Parallel Databases
DOI 10.1007/s10619-014-7153-y

Locality-aware allocation of multi-dimensional correlated files on the cloud platform

Xiaofei Zhang · Yongxin Tong · Lei Chen · Min Wang · Shicong Feng

Abstract The effective management of enormous data volumes on the Cloud platform has attracted devoting research efforts. In this paper, we study the problem of allocating files with multidimensional correlations on the Cloud platform, such that files can be retrieved and processed more efficiently. Currently, most prevailing Cloud file systems allocate data following the principles of fault tolerance and availability, while inter-file correlations, i.e. files correlated with each other, are often neglected. As a matter of fact, data files are commonly correlated in various ways in real practices. And correlated files are most likely to be involved in the same computation process. Therefore, it raises a new challenge of allocating files with multi-dimensional correlations with the "*subspace locality*" taken into consideration to improve the system throughput. We propose two allocation methods for multi-dimensional correlated files stored on the Cloud platform, such that the I/O efficiency and data access locality are improved in the MapReduce processing paradigm, without hurting the fault tolerance and availability properties of the underlying file systems. Different from the techniques

X. Zhang · Y. Tong · L. Chen (✉)
HKUST, Hong Kong, Hong Kong
e-mail: leichen@cse.ust.hk

X. Zhang
e-mail: zhangxf@cse.ust.hk

Y. Tong
e-mail: yxtong@cse.ust.hk

M. Wang
Google Research USA, New York, NY, USA
e-mail: minwang@google.com

S. Feng
Miao Zhen Company, Beijing, People's Republic of China
e-mail: fengshicong@gmail.com

Published online: 06 August 2014

proposed in [1,2], which quickly map the locations of desired data for a given query $\mathcal{Q}$, we focus on improving the system throughput for batch jobs over correlated data files. We clearly formulate the problem and study a series of solutions on HDFS [9]. Evaluations with real application scenarios prove the effectiveness of our proposals: significant I/O and network costs can be saved during the data retrieval and processing. Especially for batch OLAP jobs, our solution demonstrates well balanced workload among distributed computing nodes.

Keywords Distributed data allocation · Cloud storage · Multi-dimensional correlation · Subspace locality

1 Introduction

As a booming computing infrastructure, Cloud promises enormous storage capability. And the effective management over massive data has been attracting devoting efforts. Experiences from traditional parallel RDBMS [24] suggest that significant I/O savings can be achieved by exploiting the data locality. However, correlation-aware file allocation is often neglected or supported in representative Cloud file systems, e.g., Google File System [12], Amazon Simple Storage Service [3]. Mainly there are two reasons: (1) *Availability, Scalability*, and *Fault Tolerance* are the primary concerns of Cloud file systems; (2) there is a trade-off between data locality and the scale of job parallelism. Although distributing data randomly is expected to achieve the best parallelism, however, such a method may lead to degraded user experiences for introducing extra costs on large volume of remote accesses, especially for many applications that are featured with data locality, e.g., context-aware search, subspace oriented aggregation queries, and etc.

1.1 Motivation

Data correlation is common in the real world. And data are often correlated in multi-dimensions. For instance, object trajectory data correlated in time and space locations [6]. Gene expression patterns are often correlated, which is the base for gene co-expression analysis [5]. In social networks, two persons can be correlated in different attributes of their profiles, i.e., locations and hobbies. Here we use two scenarios to elaborate the benefits of taking file correlations into consideration during the file allocation.

Context-aware analytic. Context-aware analytics [19] is crucial to improve user experience. The primary step of context-aware analytic is to acquire enough related contexts for further learning and processing. For example, for a given input text, we can enrich the content of this text by referring to an existing corpus to facilitate the further processing, like disambiguation or classification, etc.

Figure1 gives such an example of the context-aware analytic system, which takes application dependent context information, i.e. *key words, time, location*, into consideration to determine the closeness between corpus files and the input query. For a query with key words "Yao Ming, Game" issued at time T_1 and location L_4, the

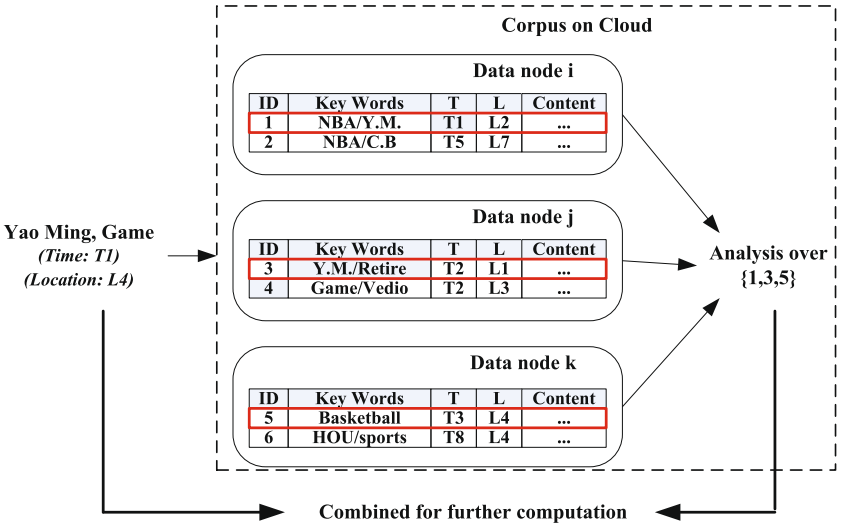


Fig. 1 An example of context-aware analytic application

context-aware analytic system will retrieve the documents that have closely matched key words or time and location context. In this example, file 1, 3, 5 are closely related with the input query in *key words*, *time* and *location* respectively. Therefore, it is necessary to retrieve them together for contextual analysis. A problem is that , since no data correlation is considered during the file allocation, these files are much likely to be distributed far away from each other in the storage network. Therefore, high network cost is inevitable during the data retrieval and further analytical processing.

Subspace aggregation query. For multi-dimensional data, aggregation queries are usually performed on some "*hot*" subspaces instead of the entire dimension space, e.g., using the "*Groupby*" function to filter out as many undesired data as possible. Considering a music video browsing application, users may specify certain criteria, like music style, producing time, singers' names, and have all the satisfied videos be played successively. If these videos are allocated with the consideration of correlations, thus a user's order may only involve a few nodes to conduct the video streaming.

1.2 Challenges and opportunities

In this work, we try to allocate files on the Cloud with the consideration of multi-dimensional correlations between files. Two steps are included. First, we need to partition a set of files such that the closely correlated files are grouped together. Second, we need to make sure files from the same partition group are collocated on the Cloud. Therefore, the partition strategy of a set of files and the file placement strategy are the main concerns of this work. Since files are represented by a number of data blocks on the Cloud; therefore, unless stated otherwise we use file placement and data placement on the Cloud interchangeable for the rest of the paper.

In traditional RDBMS, the data locality is generally considered as a partition problem. By learning workload patterns, database manager can partition tables horizontally or vertically to group the data which are most likely to be queried together. However, the same partition techniques cannot be directly applied to Cloud for two reasons. First, in traditional RDBMS, all the queries are answered by the operations defined on tables. Therefore, partition is in the context of tables. However, there is no support for queries conducting random data access on the Cloud file system. Queries are simply answered by scanning and processing on the entire set of files. Moreover, in traditional RDBMS, a table partition usually resides on a consecutive physical storage to preserve data locality. However, files management on Cloud only interfaces a conceptual abstraction of disks, e.g. data blocks, to promise the system scalability. Given a file, or a partition of files, once its size exceeds the predefined system block size, it will be split into blocks and distributed to different storage nodes according to the system's current workload. Therefore, simply partition cannot preserve the data locality on Cloud. For instance, Hive [28] is designed as a data warehouse solution on Cloud. However, a table partition in Hive still maps to randomly distributed data blocks on HDFS.

Exploiting the locality correlation for high dimensional data itself is a research problem that has been studied for years. Although existing dimension reduction techniques, like PCA [17] or feature selection [23], can help reduce the dimensionality of the data objects, two important factors are ignored. First, query patterns, which play a vital role in subspace selection, are not taken into consideration. Second, "extreme" locality can severely hurt the job parallelism. Therefore, in this work, we would like to study how to exploit the subspace locality correlations for a set of files under the guidance of query patterns to improve the throughput without hurting the massive parallelism property of Cloud. Furthermore, due to the frequent update of data and the evolution of query patterns, efficient solutions are needed to evaluate updates and perform the re-allocation when it is necessary.

For *availability* and *fault tolerance* considerations, most Cloud file systems, e.g. HDFS, usually employ naive data placement strategies, which are not optimized for the system throughput. As studied in CoHadoop [11], a more sophisticated data placement strategy can help preserve the data locality, without risking more data loss in general. Therefore, in this work, in addition to studying locality preserving partition techniques for a set of files that have multi-dimensional correlations, we also investigate the locality-aware data placement strategy on Cloud, aiming to improve the system's throughput performance.

1.3 Our proposal

Different from recent works on the indexing of multi-dimensional data on Cloud, which aim at providing quick look-up service, our work targets at facilitating the retrieval and processing of multi-dimensional correlated files on the file system level. To achieve this target, firstly, based on query patterns, we can identify several "*hot*" subspaces. Then, we study the data correlation properties under these selected subspaces. By taking two realistic factors into consideration, the heterogeneity of the storage network and

the scale of parallelism, we develop a cost metric to evaluate a data placement plan. Guided by the cost metric, we propose an algorithm to derive a subspace locality-aware allocation plan for files correlated in multi-dimensions. Furthermore, for new coming data, we can use the existing knowledge to immediately determine its allocation in the Cloud file system. Identified as an important issue and interesting future work, we find that it is necessary to have a control on the volume of data movement in case of the evolution of query patterns or even network partitions.

1.4 Contributions

To summarize, we made the following contributions in this work:

- We introduce a novel research problem: locality-aware allocation of multi-dimensional correlated files on Cloud platforms, which cannot be solved by simply adopting traditional partition techniques. We formalize the problem and present an effective solution.
- We develop a cost metric to quantify the degree that the file locality is satisfied under different placement strategies.
- We study a real application case, which validates the importance of sophisticated allocation of correlated files on Cloud, as well as the effectiveness of our solution.

The rest of the paper is structured as follows. For clear illustration purpose, in Sect. 2 we briefly introduce the HDFS and MapReduce framework. We formally define our problem in Sect. 3 and prove the problem is NP hard. Section 4 elaborates the cost model guided file allocation strategy and algorithm analysis. We present more system implementation details in Sect. 5. In Sect. 6, we demonstrate the evaluation results from both synthetic and real data sets. We survey and discuss the most recent related work on Hadoop extensions in Sect. 7 and conclude the paper in Sect. 8.

2 Preliminary

In this section, we briefly introduce the underlying file system of Hadoop and the MapReduce computing paradigm.

HDFS [27] is a component of Hadoop [26], which is an open source and Java-based implementation of the MapReduce framework. HDFS implements a distributed file system similar to Google File System [12]. It can be used to process vast amounts of data in parallel on large clusters in a reliable and fault-tolerant fashion. The architecture of HDFS is described in Fig. 2.

As shown in Fig. 2, HDFS manages files in a *master-slave* fashion. Users can only interact with the system through the *master*, namely the NameNode, which maintains all files' meta data on the storage system. HDFS manages all files on the block level. Each file is split into a number of blocks of certain size (64MB by default),[1] and every block is replicated over the storage network. By default, each data block has three copies on HDFS. The replica placement policy is rack-aware. It creates one copy on

[1] Small files will be placed in the same block until a block is full.

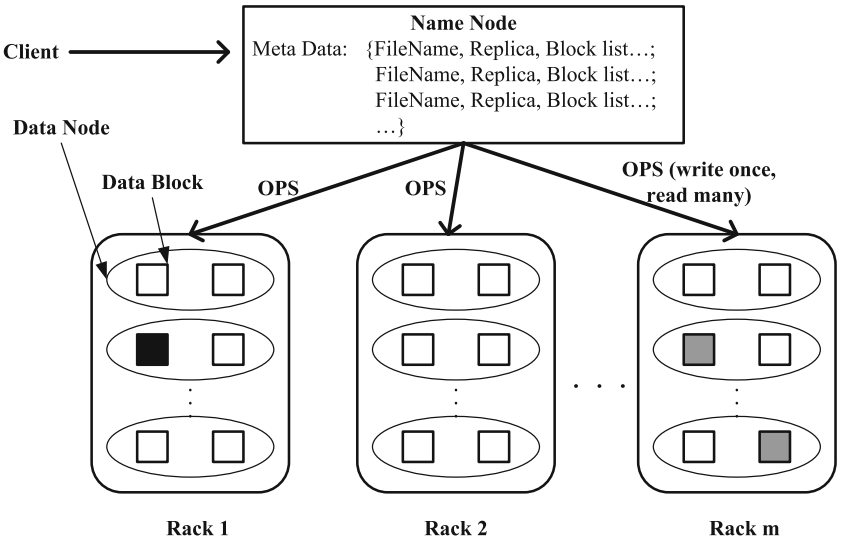


Fig. 2 HDFS architecture and data placement policy

the local disk (the black square in Fig. 2), and places two other copies on two different remote nodes, which reside in a same rack (the gray squares in Fig. 2). This placement strategy guarantees balanced storage and reduces the probability to lose data permanently. Although big files being distributed over the network increases the computation parallelism, however, the huge network traffic volume could be introduced when there is data copying among a large number of computing nodes.

A plain MapReduce job works as the follow. A *Master* node invokes Map tasks for each computing node that possesses a part of input, which guarantees the locality of computation. Map tasks transform the input $(key, value)$ pair (k^1, v^1) to, e.g., n new pairs: (k_1^2, v_1^2), (k_2^2, v_2^2), …, (k_n^2, v_n^2). The outputs of Map tasks are then by default hash-partitioned to different Reduce tasks with respect to k_i^2. Reduce tasks receive $(key, value)$ pairs grouped by k_i^2, and perform the user-specified computations on all the $values$ of each key, then write results back to the storage. In real practices, it turns out that the bottleneck of MapReduce performance lies in the cost of data copying over the network. Performances can dramatically degrade in case of considerable communication cost between a large group of computing nodes. Therefore, distributing jobs to a large number of computing nodes may not always be a good idea. The same observation is made in work [9].

3 Problem statement

In this work, we consider the files to be deployed or allocated on the Cloud platform are plain files with meta data. For example, a video clique with its meta data describing its creation time, length, producer, tags and etc. Apparently, files' meta-data describe data properties, *a.k.a.* features, like creation time, size and etc., which imply potential file

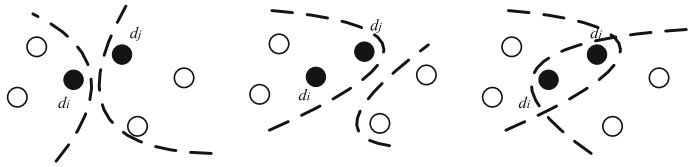


Fig. 3 File partition without query pattern taken into consideration

correlations in the context of feature value closeness. Correlated files are more likely to be involved in the same computation. For instance, transaction files that belong to the same user ID are usually processed together in the log processing [11]; videos of the same topic or with similar tags are usually the inputs for similarity detections. By allocating correlated files together, significant I/O savings can be achieved on reducing the huge cost of random data access over the entire distributed storage network. Currently, most Cloud file systems distribute files over storage nodes without taking file correlations into consideration. Although the latest release of Hadoop-0.21.0 supports user specified data placement in HDFS, the real challenge remains how to exploit the locality of files in the context of multi-dimensional feature space, such that a locality-aware allocation strategy can be derived to preserve the file locality in different feature dimensions as much as possible. We believe this is a fundamental research problem concerning the Cloud platform's throughput performance.

3.1 Subspace locality

Files described with multiple features are naturally correlated. For example, files can be sequenced according to modification time, or structured basing on some ontology model. Considering each feature as a file representing dimension, then files are points distributed in the multi-dimensional feature space. Note that due to the dimensionality curse, files are considered equally far away from each other. However, there must be several application-dependent *hot subspaces*, under which files are frequently being processed. Thus, the subspace locality problem is to efficiently group files that are closely correlated, i.e., most often being accessed at the same time in the subspace.

Intuitively, given a subspace, we can group files by employing the techniques introduced in multi-dimensional data clustering or dimension reduction. Such techniques generally employ some heuristic methodology, which cannot guarantee to produce a deterministic file grouping solution. For example, in Fig. 3, based on the current data clustering techniques, three different partition solutions are all reasonable. However, it is more reasonable to determine a cluster or partition based on query patterns. Assume for 90% queries, file d_i and d_j are accessed together, then they should definitely be placed into the same group.

Note the third partition solution in Fig. 3 introduces duplications. Considering the promising storage capability of Cloud, such a redundancy may bring potential benefits. There have been abundant research efforts on availability and throughput improvement by taking the advantage of more duplicates, like work [18,22,30]. However, duplication determination is orthogonal to the correlation based file grouping prob-

Table 1 Notation summarization

Notation	Meaning
$\mathcal{D}$	A set of files
$\mathcal{S}$	A set of subspaces
s_i	Subspace i
$\mathcal{P}_i$	File set partition strategy for s_i
k_i	The number of partition groups in $\mathcal{P}_i$
$g_{i,j}$	jth partition group in $\mathcal{P}_i$

lem. Therefore, solutions from these works can be directly adopted to further improve the system performance. Thus, we only consider the mutually exclusive partition of a set of files in this work. Note that such a partition solution not only serves as data allocation guidance on the distributed storage, but also reveals potential opportunities on system performance optimization. For example, after data is partitioned, we may find that certain partition is larger or accessed more frequently than other data partitions. Then, optimization techniques can be applied accordingly to improve the overall system performance.

With application dependent "*hot*" subspaces identified, we can have multiple file set partition strategies by incorporating query patterns, where each partition strategy serves the locality correlations in a certain feature subspace. The problem is how to find a compromised partition solution to well serve the file correlations of different feature subspaces as much as possible.

3.2 Problem definition

Considering m hot subspaces, which determine m file set partition strategies. In this work, we mainly focus on the mapping from the m locality-aware partition strategies to one partition solution. We try to make this final partition strategy satisfy the m different locality correlations as much as possible, in terms of minimizing the value of a proposed cost metric. Table 1 summarize the notations employed in the problem definition.

Given a set of files $\mathcal{D}$ stored on HDFS, each file $d_i \in \mathcal{D}$ is described with a set of features $\mathcal{F} = \{f_0, f_1, \ldots f_r\}$. Assume some features commonly being involved together in queries, it gives us a set of frequently queried subspaces, denoted as $\mathcal{S}$=$\{s_i | s_i \subset \mathcal{F}\}$. Given a feature subspace $s_i \in \mathcal{S}$, we can have $\mathcal{D}$ be partitioned into k_i groups, such that files in the same partition group are *closely correlated*[1] with each other in s_i. Let $\mathcal{P}_i$ denote the partition of $\mathcal{D}$ in s_i, where $\mathcal{P}_i$=$\{g_{i,j} | g_{i,j} \subset \mathcal{D}, 0 \leq j < k_i\}$, we have the following partition property always satisfied: 1) $\mathcal{D} = \bigcup_{j=0}^{k_i-1} g_{i,j}, g_{i,j} \in \mathcal{P}_i$; 2) $\forall g_{i,j},\ g_{i,k} \in \mathcal{P}_i, g_{i,j} \cap g_{i,k} = \emptyset$, where $j \neq k$. Assuming $|\mathcal{S}|$=m, which implies m

[1] We believe the closeness measurement is application dependent and consider it as a predefined metric.

different partition strategies, we want another partition solution which preserves the locality in m different subspaces as much as possible.

Let $\mathcal{P}^*$=$\{g_{*,i}|g_{*,i} \subset \mathcal{D}, 0 \leq i < n\}$ gives a mutually exclusive n-group partition of $\mathcal{D}$. Given $\mathcal{P}_i$ determined by s_i, we measure the cost of using $\mathcal{P}^*$ to satisfy the locality correlations implied in $\mathcal{P}_i$. Given $g_{i,j} \in \mathcal{P}_i$, we may need more than one file group in $\mathcal{P}^*$ to cover $g_{i,j}$. Therefore, the cost of covering $g_{i,j}$ with $\mathcal{P}^*$ can be defined as follows:

$$c(\mathcal{P}^*, g_{i,j}) = c_{\text{rf}} + \alpha_i \times c_{\text{rg}} \tag{1}$$

where c_{rg} is the extra file groups from $\mathcal{P}^*$ to cover $g_{i,j}$, and c_{rf} is the number of redundant files involved to cover $g_{i,j}$. α_i is a constant number. We shall elaborate the determination of α_i in Sect. 4. Let $MSC(\mathcal{P}^*, g_{i,j})$ denote the minimal set of file groups in $\mathcal{P}^*$ to cover all the files in $g_{i,j}$. Thus,

$$c(\mathcal{P}^*, g_{i,j}) = |\bigcup u - g_{i,j}| + \alpha_i \times (|MSC(\mathcal{P}^*, g_{i,j})| - 1),\ u \in MSC(\mathcal{P}^*, g_{i,j}) \tag{2}$$

where $|\cdot|$ is the cardinality function. Then, the cost of using $\mathcal{P}^*$ to preserve the locality implied by s_i, denoted as $C_i(\mathcal{P})$, can be measured as the following summation function[2]:

$$C_i(\mathcal{P}^*) = \sum_{j=0}^{k_i} c(\mathcal{P}^*, g_{i,j}) \tag{3}$$

Note that C_i is the summation of two cost factors. Intuitively, the former factor abstracts the cost on storage imbalance while the latter one represents the cost for involving more storage nodes into computation. For example, if too many files are grouped together, the imbalance cost would raise and degrade the scale of job parallelism; if files are partitioned into too many small groups, data copying traffic across storage nodes would increase. Since Cloud file systems usually treat files as data blocks, we treat a big file as a number of block-sized files. For a small file, the minimal I/O cost is one data block. Therefore, it makes C_i a reasonable metric which is essentially computed using the number of involved files. By aggregating the costs from all m different subspaces, we can compute the degree that $\mathcal{P}^*$ serves the locality correlations of given $\mathcal{S}$, denoted as $\mathcal{C}^* = \frac{1}{m}\sum_{i=0}^{m-1} C_i$. Now we can formally define our problem as the following:

Problem Definition 3.1 *Given a set of files $\mathcal{D}$, a set of feature subspaces $\mathcal{S}$ and a set of corresponding partition strategies $\{\mathcal{P}_0, \ldots, \mathcal{P}_{m-1}\}$, find a partition strategy $\mathcal{P}_{opt}$ of $\mathcal{D}$ such that $\mathcal{C}^*$ is minimized.*

3.3 Problem hardness

Intuitively, Problem 3.1 is a NP hard problem. Because given any partition strategy, there is no way to determine whether this strategy is optimal or in polynomial time.

[2] We consider each partition group in $\mathcal{P}_i$ is equally important. So is the m different feature subspaces.

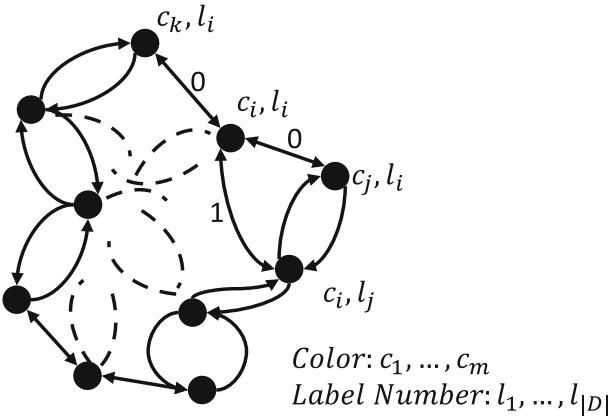


Fig. 4 Graph construction example

To prove this problem is NP hard, we need to prove that it is at least as hard as one of the NP complete problems.

Theorem 1 *Problem 3.1 is NP hard.*

Proof We employ traveling salesman problem to prove this theorem. Considering a weighted labeled directed graph $G =< V, E, L, W >$, where $|V| = m \times |\mathcal{D}|$. L is the set of vertex labels. A vertex label is a unique value pair (*color*,*label id*). In total, there are m different colors. *label id* is a value from 1 to $|\mathcal{D}|$. Thus, for a *label id* l_i, there are m vertices labeled with l_i but differ in colors. The edge between any two directly connected vertices is bi-directed. However, edge weight W is not always symmetric. The graph is constructed as the follows. Vertices with the same *label id* are connected to each other with edge weight that equals to 0. For any two vertices i and j of the same color, they are either symmetrically connected to each other with edge weight $w < i, j >$ that equals to 1 or value $c_i + k_j$, where c_i is the number of vertices connected to node i with weight that equals to 1, and k_j is the number of vertex groups of node j's color, where each group of vertices are strictly connected with edge weight that equals to 1. And vice versa we can compute $w < j, i >= c_j + k_i$. Figure 4 elaborate this construction process.

Considering the traveling salesman's problem, for any traversal path we take, it consists a number of edges with weight 0, 1 and certain w. We treat nodes connected with weight 0 as one file, nodes symmetrically connected with weight 1 as a file group, then we can have a partition strategy of file set $\mathcal{D}$. And if the traversal path is proven to be the optimal path, it directly mapped to an optimal solution of Problem 3.1 by Definition and the locality preserving cost metrics (Equ.1 to 3). Thus Problem 3.1 is at least as hard as traveling salesman problem, which is a well-known NP complete problem; therefore, Problem 3.1 is NP hard. □

Intuitively, a proper solution for traveling salesman problem (TSP) can be transformed to solve Problem 3.1. However, according to the reduction process, the constructed graph is asymmetric and does not satisfy the triangle property. Abundant

literatures on solving TSP with approximation bounds are derived based upon the triangle property; however, these solutions are not applicable in this scenario. Therefore, instead of taking the liberty of simple greedy and random heuristics, we propose a solution that finds a near optimal partition strategy by starting from a sub-optimal solution.

4 Allocation solution

A brute-force solution of Problem 3.1 is to enumerate all the possible partitions of $\mathcal{D}$ and choose the optimal one which gives the minimum $\mathcal{C}^*$. However, it is infeasible to check exponential number of partition strategies. Instead, our solution is to start from a sub-optimal solution and employ some heuristics to derive a near optimal partition with as less cost as possible. To elaborate, our approach includes three steps. First, among all the existing partition strategies $\{\mathcal{P}_0, \ldots, \mathcal{P}_{m-1}\}$, we find the one, denoted as $\mathcal{P}_b$, that gives the minimum $\mathcal{C}^*$. Intuitively, $\mathcal{P}_b$ better preserves the locality correlations of $\mathcal{D}$ comparing to all the other subspaces. Second, we generate a set of tasks $\{t_i, \ldots, t_j\}$ to update $\mathcal{P}_b$. Correspondingly, a set of candidate partition strategies can be derived after applying the updates to $\mathcal{P}_b$. To avoid generating too many candidate partition strategies, we employ some heuristics to reduce the cardinality of $\{t_i, \ldots, t_j\}$. Finally, we evaluate all the candidate partition strategies and select the one that gives the minimum value of $\mathcal{C}^*$.

4.1 Finding a near optimal solution

As indicated in the cost model, $\mathcal{P}^*$ preserves the locality correlation of each subspace by introducing the cost of involving uncorrelated files and access to multiple partition groups. Intuitively, there are two natural partition strategies $\mathcal{P}_{all}$ and $\mathcal{P}_{single}$. $\mathcal{P}_{all}$ treats the entire data set $\mathcal{D}$ as one partition, implying that all the files are closely correlated. On the contrary, $\mathcal{P}_{single}$ treats every single file as one partition, implying that no two files are closely correlated. Considering the data placement policy of HDFS (summarized in Sect. 2), the default data placement can well serve these two partition strategies. Thus, the data placements on HDFS for $\mathcal{P}_{all}$ and $\mathcal{P}_{single}$ are the same. Therefore, to serve the locality correlations defined by $\mathcal{S}$ by employing $\mathcal{P}_{all}$ or $\mathcal{P}_{single}$ would results in the same cost. Given a subspace $s_i \in \mathcal{S}, C_i(\mathcal{P}_{all}) = (k_i - 1)|\mathcal{D}|$, $C_i(\mathcal{P}_{single})=\alpha_i\ (|\mathcal{D}| - k_i)$. Since k_i is usually an order of magnitude smaller than $|\mathcal{D}|$, we treat α_i as a constant value of $k_i - 1$. Having α_i for each subspace s_i being determined, we can quantify the degree that a given partition strategy serves the locality correlations specified by $\mathcal{S}$.

For clear illustration purpose, we shall first elaborate our algorithm for finding a near optimal partition strategy $\mathcal{P}^*$ as well as the solution to handle new coming files. Meanwhile, we analyze some factors that could contribute significantly to the algorithm efficiency.

Let $\mathcal{P}_{opt}$ be the optimal solution. We try to reach $\mathcal{P}_{opt}$ by conducting a set of "modification tasks" on an existing partition strategy $\mathcal{P}_i$. Here we define two types of "modification tasks": *split* and *merge*. One split divides a file group into two partitions;

$D = \{d_1,...,d_{10}\}$

$P_1 : g_{1,1} = \{d_1,d_2,d_3,d_4,d_5\}; g_{1,2} = \{d_6,d_7,d_8\}; g_{1,3} = \{d_9,d_{10}\}$

$P_2 : g_{2,1} = \{d_1,d_4,d_7,d_9\}; g_{2,2} = \{d_5,d_6,d_{10}\}; g_{2,3} = \{d_2,d_3,d_8\}$

$P_3 : g_{3,1} = \{d_1,d_3\}; g_{3,2} = \{d_5,d_7,d_9\}; g_{3,3} = \{d_2,d_4\}; g_{3,4} = \{d_6,d_8,d_{10}\}$

C(Pi,Pj)	P1	P2	P3
P1	0	28	24
P2	27	0	29
P3	17	20	0

Fig. 5 A file set partition example

while one merge combines two partition groups into one file group. Intuitively, among all the m given partition strategies from different feature subspaces, there is a partition best describing the locality property of all the other $m-1$ strategies. We call such a partition the base partition and denote it as $\mathcal{P}_b$. Intuitively, $\mathcal{P}_b$ is the closest to $\mathcal{P}_{opt}$. Algorithm 1 describes the computation of $\mathcal{P}_b$. Instead of computing m^2 pair-wise costs and selecting the minimum accumulative cost, as shown in line 6, Algorithm 1 always selects a partition strategy that currently has the minimum accumulative cost to be the candidate of $\mathcal{P}_b$, and proceeds the computation. Line 7 guarantees that the final returned partition strategy does have the minimum accumulative cost to serve all the other partition strategies. Obviously, Algorithm 1 gives $\mathcal{O}(m^2)$ running time complexity in the worst case. However, in practice, the algorithm is expected to be much faster as it filters out many unnecessary computations.

Algorithm 1 Finding $\mathcal{P}_b$

Input: Partition strategies $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{m-1}$; Minimal heap I
Output: $\mathcal{P}_b$
1: **for** $i = 0 : m-1$ **do**
2: $< C(\mathcal{P}_i, \mathcal{P}_0),\ i,\ 0 > \rightarrow I$ //$C(\mathcal{P}_i, \mathcal{P}_0)$ is the key
3: **end for**
4: $k = 0$
5: **while** $k \neq m-1$ **do**
6: return the root $< C',\ i,\ j >$ from I
7: $k \leftarrow j$
8: $< C(\mathcal{P}_i, \mathcal{P}_{k+1}) + C',\ i,\ k+1 > \rightarrow I$
9: **end while**
10: return the root $< C',\ i,\ j >$ from I
11: $\mathcal{P}_b \leftarrow \mathcal{P}_i$

Intuitively, a desired near optimal solution $\mathcal{P}^*$ is even more closer to the other $m-1$ partition strategies than $\mathcal{P}_b$. Thus, by taking the "*suggestions*" from other partition strategies, i.e., comparing the differences between $\mathcal{P}_b$ and $\mathcal{P}_i$ $(i \neq b)$, we can derive a set of suggested "modification tasks" to update $\mathcal{P}_b$ such that $\mathcal{P}^*$ can be reached. We shall use the example shown in Fig. 5 to elaborate our solution.

We choose $\mathcal{P}_3$ as $\mathcal{P}_b$. Then, to satisfy $\mathcal{P}_1$ and $\mathcal{P}_2$ with $\mathcal{P}_b$, we can obtain some *suggested* "modification tasks". For example, to satisfy $g_{1,1}$ with $\mathcal{P}_b$, there are three suggested tasks: 1) split $g_{3,2}$ into $\{d_5\}$, $\{d_7, d_9\}$; 2) merge $g_{3,1}$ and $g_{3,3}$; 3) after 1) and 2) are done, merge $\{d_1, d_2, d_3, d_4\}$ and $\{d_5\}$. Thus, three new partition strategies are suggested. Recursively, by obtaining *suggested* tasks from other partition groups in different subspaces, numbers of new partition strategies are derived. We can always find a strategy which minimizes the cost $\mathcal{C}^*$. However, this procedure still involves an exponential number of partition strategies to evaluate. In the above example, we can do either only 1) or 2), or both 1) and 2), or 3). In fact, we can refine the solution by filtering out as many unnecessary tasks as possible. We always consider split before merge, which guarantees the finest refinement granularity. Thus, the following lemma holds:

Lemma 1 *If applying merge task t_i gives negative results δ, assume that task t_j depends on t_i and other p tasks, then t_j should not be considered if $|\delta| \geq p \times \sum_{y=0}^{m-1} k_y$.*

Proof t_j is a merge task. Merge takes the risk of introducing uncorrelated file readings to trade off the cost of accessing multiple partition groups, which is upper bounded with $p \times \sum_{y=0}^{m-1} k_y$. Thus, if $|\delta| \geq p \times \sum_{y=0}^{m-1} k_y$, then t_j, which depends on t_i will only bring in more overheads of uncorrelated file readings. Thus, t_j should not be considered. □

In fact, the above Lemma implies a heuristic trick that if merge task t_i is bad, then any merge task t_j depends on it should not be considered. Because in real practices, the number of files is orders of magnitude larger than the number of partition groups. Therefore, $p \times \sum_{y=0}^{m-1} k_y$ is much more likely to be smaller than the overheads of introducing uncorrelated file readings. And experiments demonstrate the effectiveness of this heuristic rule.

Lemma 2 *If a number of orthogonal tasks*[3] *individually gives positive results, then applying all these tasks together still gives positive results.*

Lemma 4.2 can be proven by definition. It also implies an important heuristic to compute $\mathcal{P}^*$: each time we should apply as many positive orthogonal tasks as possible. Algorithm 2 describes the computation of $\mathcal{P}^*$ starting from $\mathcal{P}_b$.

To elaborate Algorithm 2, we consider the example shown in Fig. 5. From line 1 to line 3, a *suggested* modification task set $\mathcal{MTS}$ is generated. Tasks are enumerated in the first two columns of Table 2. In the table, "E." denotes the evaluation score of a task, computed by $\mathcal{C}^*(\mathcal{P}_b) - \mathcal{C}^*(\mathcal{P}_{tmp})$. "D." denotes the final decision on that task. A task ID, tid, is presented as $t_i\{g_{b,k}\}$, where $g_{b,k}$ is the partition group which is going to be updated in task t_i. For example, $t_1\{g_{3,1}, g_{3,3}\}$ is a task that updates partition groups $g_{3,1}$ and $g_{3,3}$. For space saving, we also use one task ID to denote all the tasks that depend on any two or above number of tasks from set $\{t_i\}$, where $|\{t_i\}| > 2$. We use $\vdash$ to denote the dependency relationship. Therefore, $t_{12} \vdash \{t_2, t_3, t_6\}$ in the table actually includes four tasks, $\vdash (t_2, t_3), \vdash (t_2, t_6), \vdash (t_3, t_6)$ and $\vdash (t_2, t_3, t_6)$.

[3] two tasks are orthogonal if they are not performed on the same partition group of $\mathcal{P}_b$.

Algorithm 2 Finding a near optimal partition strategy $\mathcal{P}^*$ starting from $\mathcal{P}_b$

Input: Partition strategies $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{m-1}$; $\mathcal{P}_b$; Suggested modification task set $\mathcal{MTS} = \emptyset$
Output: $\mathcal{P}^*$
1: **for** $i = 0 : m-1$ and $\mathcal{P}_i \neq \mathcal{P}_b$ **do**
2: Task($\mathcal{P}_b, \mathcal{P}_i$)$\rightarrow \mathcal{MTS}$ // Generate suggested tasks in reference of $\mathcal{P}_i$ and put into $\mathcal{MTS}$
3: **end for**
4: $\forall t_i \in \mathcal{MTS}$, t_i.evaluation$\leftarrow 0$
5: **while** $\exists t_i \in \mathcal{MTS}$ not evaluated **do**
6: $\mathcal{P}_{tmp} \leftarrow$ apply t_i to $\mathcal{P}_b$
7: t_i.evaluation$\leftarrow \mathcal{C}^*(\mathcal{P}_b) - \mathcal{C}^*(\mathcal{P}_{tmp})$ // $\mathcal{C}$ gives overall cost of using a partition strategy to describe the locality defined by all other m subspaces
8: **if** t_i is a merge and t_i.evaluation is negative **then**
9: $\mathcal{MTS} \leftarrow \mathcal{MTS}$-{task depend on t_i}
10: **end if**
11: **end while**
12: return a set of orthogonal tasks T from $\mathcal{MTS}$ that gives the maximal accumulative evaluation score
13: $\mathcal{P}^* \leftarrow$ apply T to $\mathcal{P}_b$

Table 2 The generated $\mathcal{MTS}$ and its according evaluations

tid	Task	E.	D.
$t_1\{g_{3,1}, g_{3,3}\}$	$\{1,3\}, \{2,4\} \rightarrow \{1,2,3,4\}$	$37-35=2$	$\surd$
$t_2\{g_{3,1}\}$	$\{1,3\} \rightarrow \{1\}, \{3\}$	$37-40=-3$	$\times$
$t_3\{g_{3,2}\}$	$\{5,7,9\} \rightarrow \{5\}, \{7,9\}$	$37-33=4$	$\surd$
$t_4\{g_{3,2}\}$	$\{5,7,9\} \rightarrow \{7\}, \{5,9\}$	$37-36=1$	$\times$
$t_5\{g_{3,2}\}$	$\{5,7,9\} \rightarrow \{9\}, \{5,7\}$	$37-39=-2$	$\times$
$t_6\{g_{3,3}\}$	$\{2,4\} \rightarrow \{2\}, \{4\}$	$37-30=-3$	$\times$
$t_7\{g_{3,4}\}$	$\{6,8,10\} \rightarrow \{10\}, \{6,8\}$	$37-38=-1$	$\times$
$t_8\{g_{3,4}\}$	$\{6,8,10\} \rightarrow \{8\}, \{6,10\}$	$37-35=2$	$\surd$
$t_9 \vdash (t_1, t_3)$	$\{1,2,3,4\}, \{5\} \rightarrow \{1,2,3,4,5\}$	$37-42=-5$	$\times$
$t_{10} \vdash (t_4, t_7)$	$\{7\}, \{6,8\} \rightarrow \{6,7,8\}$	–	$\times$
$t_{11} \vdash (t_5, t_7)$	$\{9\}, \{10\} \rightarrow \{9,10\}$	–	$\times$
$t_{12} \vdash \{t_2, t_3, t_6\}$	...	–	$\times$
$t_{13} \vdash (t_3, t_8)$	$\{5\}, \{6,10\} \rightarrow \{5,6,10\}$	$37-43=-6$	$\times$
$t_{14} \vdash \{t_2, t_6, t_8\}$	...	–	$\times$

During the evaluation of each task, we can effectively prune t_{10}, t_{11}, t_{12} and t_{14} (line 8 to 10 in Algorithm 2). Although t_4 has a positive evaluation, a conflicting task t_3 obviously brings more benefits. Therefore, t_4 is not reported. Moreover, even though t_9 and t_{13} cannot be pruned, their evaluations are negative and therefore not be reported. Finally, only t_1, t_3 and t_8 are reported and applied to $\mathcal{P}_b$, resulting a partition strategy of {1,2,3,4}, {5}, {7,9}, {6,10} and {8}.

Theorem 2 *Algorithm 2 deterministically finds the best near optimal partition solution that we could have by starting from $\mathcal{P}_b$.*

Proof Let $\mathcal{P}'$ denote the other "optimal" solution that could be obtained from $\mathcal{P}_b$. $\mathcal{P}'$ itself defines a set of tasks T' that can change $\mathcal{P}_b$ to $\mathcal{P}'$. Now we argue that $\forall t_i \in T'$ is

included in the modification task set $\mathcal{MTS}$ defined in Algorithm 2. Assume $\exists t_i \in T'$ and $t_i \notin \mathcal{MTS}$, as $\mathcal{P}'$ is a better solution, it is beneficial to apply t_i to $\mathcal{P}_b$. However, any possible task that may have $\mathcal{P}_b$ get "closer" to the optimal solution are in $\mathcal{MTS}$ for examination. Therefore, t_i must be included in $\mathcal{MTS}$. If T returned by the end of Algorithm 2 is more beneficial than T', then $\mathcal{P}'$ is not the best near optimal solution. However, if T gives a fewer accumulative evaluation score than T' does, then T would not be returned. Therefore, T and T' gives the same accumulative evaluation score. Thus, $\mathcal{P}^*$ is for sure the best near optimal solution we can have by starting from $\mathcal{P}_b$. □

4.2 Handling updates

So far, we have presented our problem definition on locality-aware allocation of files with multi-dimensional correlations on Cloud, studied the hardness of the problem and described our heuristic solution. However, for a file allocation strategy, it is crucial to handle new coming files both efficiently and effectively. To keep the simplicity of the allocation solution, we take the following new file allocation policy. We collocate a new file d_{new} to a group containing the file d' which is closely correlated with d_{new} in the most number of subspaces. To elaborate, for d_{new}, we figure out the files that frequently co-appear with it in the same partition group under different subspaces. We can sort these files in a descending order according to the times of a file's co-appearance with d_{new} in all subspaces, and select the first file as d'. Then d_{new} is assigned to the partition group contains d'.

In our discussion, we treat the locality property under different subspaces equally. However, the query pattern can evolve over time, which implies new locality demands. Then adaptive allocation solution is necessary to meet this challenge, which is considered as an interesting future work.

4.3 Data collocation

For data placement policy, we mainly adopt the method introduced in [11]. In work [11], the essential idea is a "*best effort*" data placement, which shall place files of the same group to the same set of Datanodes until all storage space on those Datanodes are consumed. [11] proved that this method gives the perfect data collocation as long as the Datanodes' storage is big enough. However, in real practice, also noted in [11], a Datanode's storage capability is very limited. Then, files of the same group are not guaranteed to be placed on the same set of Datanodes. To be specific, only a subset of the correlated files is grouped on the same set of Datanodes. Therefore, [11]'s proposal makes the final data distribution unpredictable in real practice. This is because the later file allocation depends on the initial file uploading sequence.

A noticeable difference of our data placement policy comparing to [11] is that we have different scenario setting. As files are correlated in multi-dimensions, the allocation process for the original file set cannot be constructed along with sequential uploading. The partition strategy is pre-computed, and we have the statistics for each partition group. Therefore, we do not employ the "*best effort*" data placement. Instead, we can explicitly collocate files to the same group of Datanodes, with a storage con-

Table 3 Mapping from *fid* to *gid*

fid	*gid* vector of size $m+1$
d_i	$< G(\mathcal{P}_0, d_i), G(\mathcal{P}_1, d_i), \ldots, G(\mathcal{P}_{m-1}, d_i), G(\mathcal{P}^*, d_i) >$

Table 4 Mapping from *gid* to *fid*

gid	*fid count*	a set of *fid*
g_i	cnt_i	$\{d_i, d_j, \ldots\}$

sumption threshold guaranteed on all the Datanodes. Therefore, when new files are coming, there is still available storage to make correlated files collocated together. The storage consumption threshold is a tuning parameter to justify the trade-off between strong data locality and strong load balance in terms of system's throughput.

5 Implementation

We use Hadoop-0.21.0, which provides a API for customized data placement policy, to implement the system. We elaborate our system design, including essential data structures and the MapReduce implementation of the proposed algorithms from Sect. 4.

5.1 System design

Our application scenario is that for a given set of files, we apply locality-aware partition algorithm to partition the files into groups, and upload files of the same group to the same set of Datanodes. Therefore, for a file from the original set, we can explicitly identify its location in the system. Accordingly, the file-block mapping will be automatically generated and maintained at the Namenode. With respect to fault tolerance, a copy of this meta data will also be stored on the Secondary node by default.

Moreover, we employ two more data structures to mainly serve two purposes. One is to determine the allocation of new coming files; the other is to facilitate further adjustment of file partition and allocation strategy according the evolution of query patterns and file correlation statistics. Although the second point is considered as a future work, we expect our current solution framework to be adaptive and extensible.

The two data structures are the mappings from files to partition groups and vice versa, as shown in Tables 3 and 4.

gid refers to a group ID, and $G(\mathcal{P}_i, d_i)$ gives the *gid* of d_i in the $\mathcal{P}_i$. cnt_i represents the number of files assigned to group g_i. With the help of the above two data structures, a new coming file's allocation can be easily determined. For example, for a new coming file, d_{new}, according to m different subspace partition strategies, we can obtain its *fid* to *gid* information and append it to the tail of the entire *fid* to *gid* data structure. Then by

checking the first m elements of the *gid* vector of d_{new}, we can easily obtain the most co-appearing *fid* of d_{new}, for instance fid_i. Thus, according to our new file allocation policy, d_{new} shall be allocated to the group $G(\mathcal{P}^*, d_{new})$. After the allocation, we also update the mapping from *gid* to *fid*. To preserve persistence, the above data structures are also maintained in the Secondary node and updated periodically.

5.2 MapReduce implementation

To take the advantage of the enormous computing capacity promised by Cloud, we implement Algorithm 1 and Algorithm 2 in the MapRedue paradigm; such that our solution can scale up with the problem's input size. In fact, Algorithm 1 and Algorithm 2 can be easily parallelized for better efficiency. The detailed steps are described in Algorithms 3 and 4, respectively. Algorithm 3 and Algorithm 4 describe the pseudo code for MapReduce jobs. For Algorithm 3, it identifies $\mathcal{P}_b$ by distributing $1 \rightarrow m$ computations to m Map tasks. Reduce key is $\mathcal{P}_i$ itself, then with one phase of scan in Reduce task, $\mathcal{P}_b$ can be chosen.

Algorithm 4 implements Algorithm 2 in parallel. After $\mathcal{P}_b$ is chosen, it is distributed to other $m-1$ Map tasks, where each Map task shall generate a set of modification suggested tasks by comparing $\mathcal{P}_b$ and $\mathcal{P}_i$ ($i \neq b$) and compute the evaluation. The key issue is that we make each partition group of $\mathcal{P}_b$ the hash key from Map to Reduce. Thus, in the Reduce task, we can easily choose a number of orthogonal tasks with the maximal benefits.

Algorithm 3 MapReduce implementation of Algorithm 1

Input: $\mathcal{P}_0, \mathcal{P}_1,\ldots,\mathcal{P}_{m-1}$;
Output: $\mathcal{P}_b$
Map(Key, Context)
for all $\mathcal{P}_i$ from $\{\mathcal{P}_0, \mathcal{P}_1,\ldots,\mathcal{P}_{m-1}\}$-$\{\mathcal{P}_j\}$ **do**
  $key \leftarrow \mathcal{P}_j$
  $value \leftarrow C(\mathcal{P}_j, \mathcal{P}_i)$
  $Context.write(key, value)$
end for
Reduce(Key, Context)
for all val from $< Values >$ **do**
  scan for the minimum val
end for
Report $\mathcal{P}_b$
$Context.write(key, val)$

6 Evaluation

We validate our solution with extensive experiments over both real and synthetic data sets. In this section, we shall first elaborate the configuration of the testbed, as well as the data sets we employed in the experiments. Then, we present the evaluation results.

Algorithm 4 MapReduce implementation of Algorithm 2

Input: $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{m-1}$; $\mathcal{P}_i$, $\mathcal{P}_b$; Suggested modification task set $\mathcal{MTS} = \emptyset$
Output: $\mathcal{P}_*$
Map(Key, Context)
for all $g_{b,j}$ from $\mathcal{P}_b$ **do**
  Generate modification task t according to $\mathcal{P}_i$ and put it into $\mathcal{MTS}$
  $key \leftarrow g_{b,j}$
  for all task $t \in \mathcal{MTS}$ **do**
    $\mathcal{P}_{tmp} \leftarrow$ apply t to $\mathcal{P}_b$
    $t.evaluation \leftarrow \mathcal{C}(\mathcal{P}_b) - \mathcal{C}(\mathcal{P}_{tmp})$
    if t is a merge and t.evaluation is negative **then**
      $\mathcal{MTS} \leftarrow \mathcal{MTS}$-{task depend on t}
    end if
    $value \leftarrow (t.\text{evaluation}, t)$
    $Context.write(key, value)$
  end for
end for
Reduce(Key, Context)
for all val from $< Values >$ **do**
  scan for the maximum val //t.evaluation
end for
Report t **for each** $g_{b,j}$ **that gives the maximal value**
$Context.write(key, val)$

Table 5 Hadoop parameter configuration

Parameter name	Default	Set to
$fs.bloksize$	64 MB	64 MB
$io.sort.mb$	100 M	512 MB
$io.sort.record.percentage$	0.05	0.1
$io.sort.spill.percentage$	0.8	0.9
$io.sort.factor$	100	300
$dfs.replication$	3	3

6.1 Experiments setup

We run all the experiments on a cluster of 28 computing nodes. Each node has 2 CPUs of 3.06 GHz and 2 GB memory, 200 GB disk storage attached, running 2.6.35-22-server #35-Ubuntu SMP. The settings of some major Hadoop parameters are given in Table 5, which follows the setting suggested by [15]. We used the *TestDFSIO* program to test the I/O performance of the system, found that the system performance is stable, with average writing rate 2.98MB/sec and reading rate 18.17 MB/s. We run each experiment job 20 times and report the average execution time.

Our evaluation is composed of two parts. Firstly, we shall evaluate the scalability of our algorithms to compute $\mathcal{P}^*$, i.e. Algorithm 2 and Algorithm 4. As we discussed in Sect. 4, the efficiency of Algorithm 2 is not only subjected to the number of data files and number of chosen subspaces (i.e. *subspace count*, denoted as SubC), but is also affected by the underlying data file distribution among partition groups. We validate this point with four synthetic data sets, as summarized in Table 6. In the table, FC

Table 6 Synthetic data sets

FC	SubC	PGC	FD
$10^4/10^5/10^6/10^7$	2/4/8/16	$U \sim (10, 20)/N \sim (15, 2)$	$U/FN * N \sim (1, 0.3)$

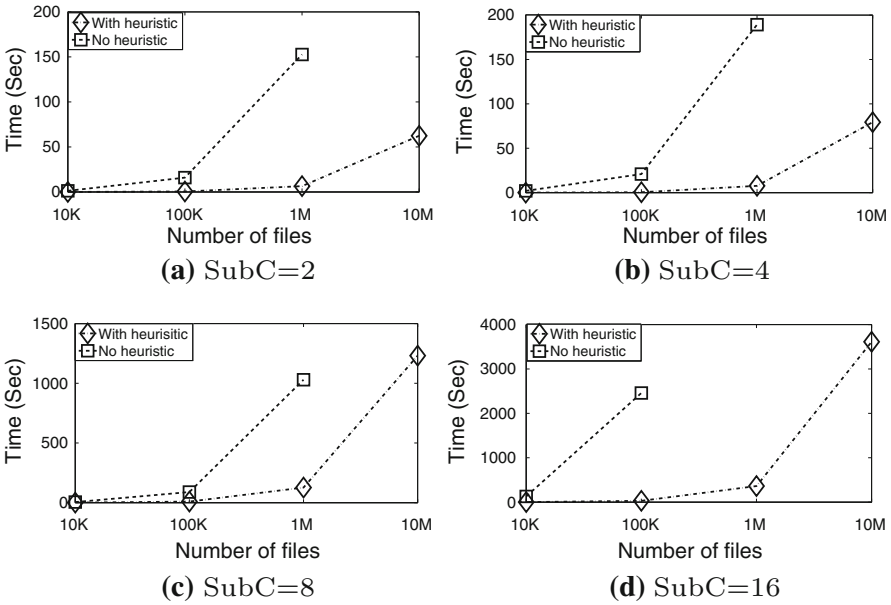


Fig. 6 Computing $\mathcal{P}^*$ with and without heuristics

refers to the total file count. PGC is the count of partition groups and FD gives the file distribution among partition groups. For each partition strategy, we chose PGC in two different ways: unified selection between 10 and 20, and selection following a normal distribution N∼(15,2). After a partition strategy and PGC are decided, we distributed files to each partition group following two different distributions: unified distribution and normal distribution.

Secondly, we consider a video retrieval application. We use a real world data set from IMDb [14], which is a public data set of videos, including over 2 million videos' meta data, describing a video's rating, year, type and etc. In our experiments, we use a sampled data set from IMDb containing 10,000 movies, and manually create data files representing the real video. In the data set, each movie has over 50 attributes. For each attribute, the data set can be partitioned according to the attribute value. The sizes of generated video files have a mean of 100(MB) and variance of 5(MB). We compare the data uploading time between our locality-aware allocation and the plain Hadoop solution. Given a set of video selection & retrieval queries, we compare the retrieval efficiency of our method against the CoHadoop and naive Hadoop solution.

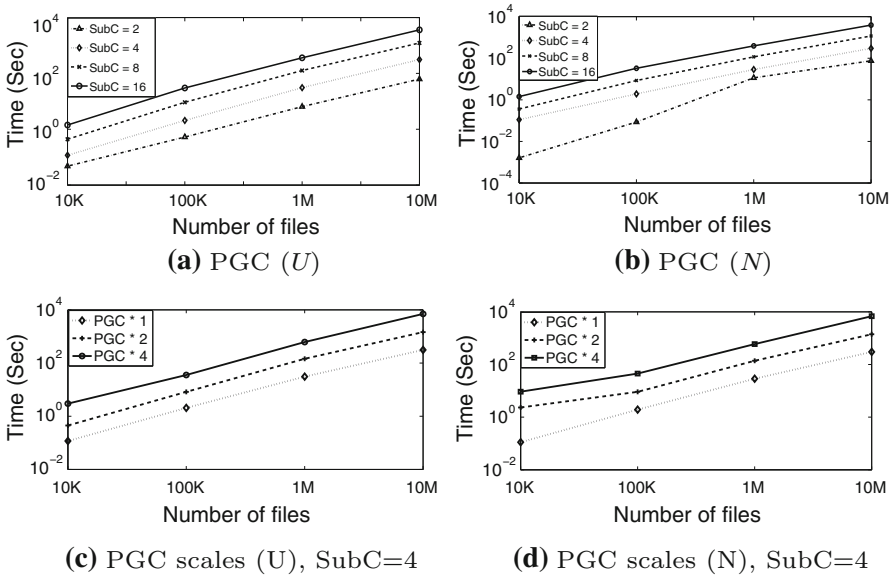


Fig. 7 Computing $\mathcal{P}^*$ with files uniformly distributed among partition groups

6.2 Computing $\mathcal{P}^*$

In this section we first study how our heuristics would work on computing $\mathcal{P}^*$; then we validate the scalability of Algorithm 2 and Algorithm 4, as well as how the underlying distribution of data files affect the efficiency of the algorithm.

We first evaluate Algorithm 1 and 2. Figure 6 shows the time differences of computing $\mathcal{P}^*$ with and without heuristics. In the figure, some running time results of the naive method are not reported because it goes far beyond the time frame comparing with the heuristic method. Apparently, with our proposed heuristics, the computing time of $\mathcal{P}^*$ can be significantly reduced. Without clear specification, the following experiment results are all reported with heuristics employed.

Figure 7 shows the experimental results when files are uniformly distributed among different partition groups. In the figure, (U) and (N) are adopted to represent uniform distribution and normal distribution respectively. Form Fig. 7a and b, we can tell that Algorithm 2 scales well with respect to the increasing number of files and SubC. However, the distribution of PGC (Partition Group Count) among different partition strategies, contributes little to the computation efficiency of $\mathcal{P}^*$. Figure 7c and d demonstrate how Algorithm 2 reacts to the growth of PGC. In experiments, we set SubC=4 and double the original PGC. As shown in the figure, the time to compute $\mathcal{P}^*$ grows significantly when PGC grows.

Similarly, Fig. 8 validates the scalability of Algorithm 2 when files are distributed among partition groups that follow a normal distribution. Both Fig. 8a and b demonstrate the similar scalability performance as shown in Fig. 7. The difference, however, lies in the time efficiency. Apparently, under the same SubC and PGC settings, $\mathcal{P}^*$ can

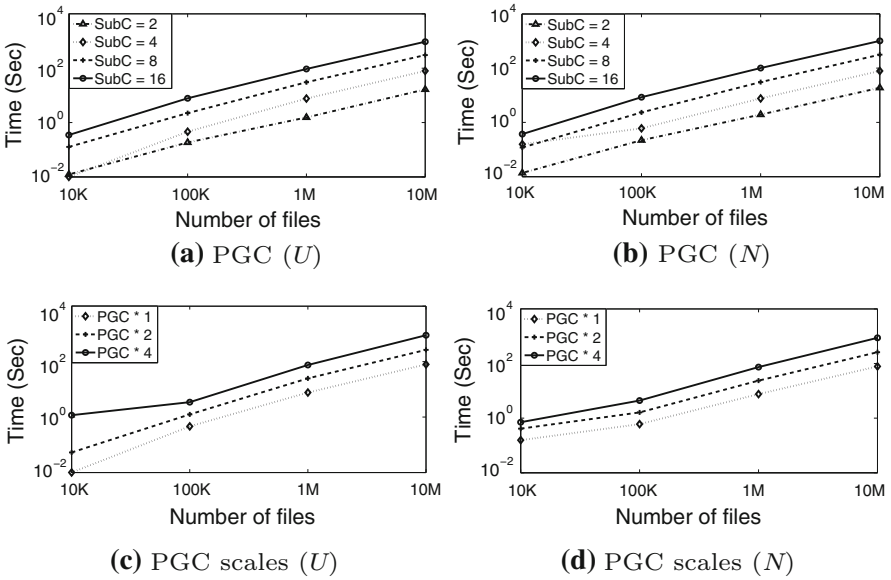


Fig. 8 Computing $\mathcal{P}^*$ with files distributed to partition groups in normal distribution

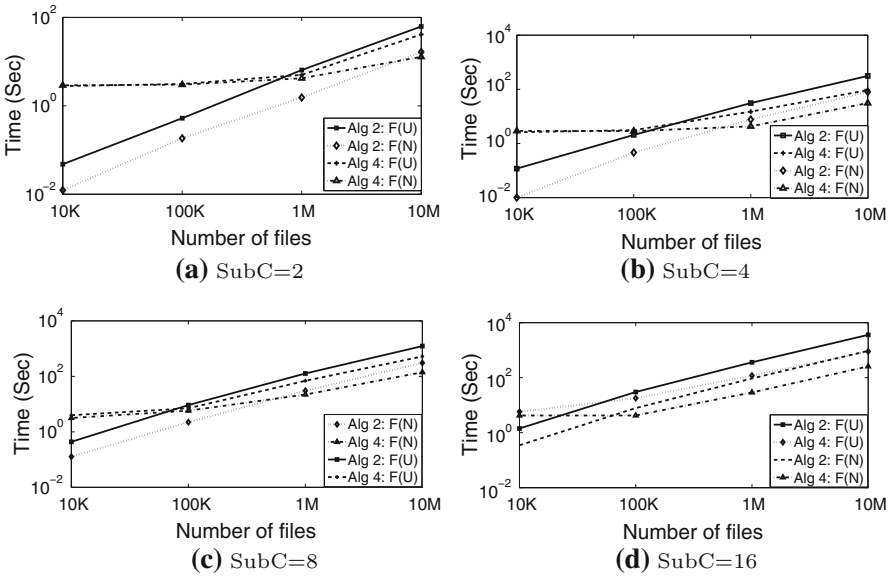


Fig. 9 The speedup of Algorithm 4 in different settings

Table 7 Statistics of the video data sets

Subspace	PGC	Mean	Std.Var.
s_1	98	102	2.3
s_2	135	74	24.4
s_3	57	175	12.1
s_4	14	714	15.7

be computed much faster (in terms of orders of magnitude) when files are normally distributed among partition groups.

Figure 9 demonstrates how Algorithm 4 scales in different settings. We set the number of Map tasks equal to SubC and only one Reduce task is employed. Apparently, when it scales to a large number of files, Algorithm 4 provides much better efficiency since all the computation is divided into several Map tasks. However, MapReduce job takes extra cost on job initialization and data copying over the network. Therefore, we observe that Algorithm 4 is suitable for the scenarios of large volume of files. In addition, we can observe the computation cost goes up when SubC increases. Thus, for a high SubC value, Algorithm 4 is a great help for better time efficiency.

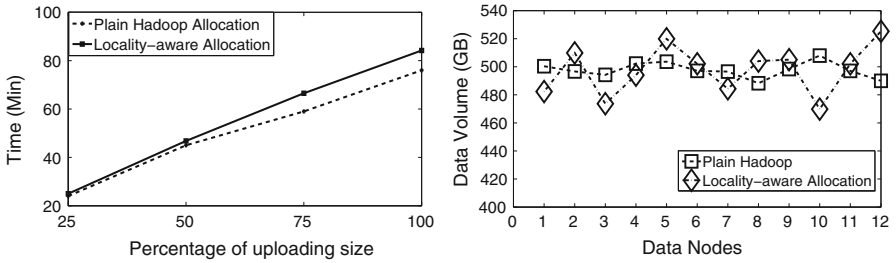
6.3 Video retrieval and analysis

As reported in the experiments' setup section, we generate synthetic data files to represent the real video files. We select ten features (attributes) for each movie: producing year, length, clarity, language, type, rating, region, producer, award and director. We choose four different subspaces, which are intuitively being queried the most: s_1(rating, director), s_2(rating, type), s_3(year, region, award) and s_4(length, clarity, language). After clustering the videos in each subspace, we have the following statistics on the number of partition groups and file distributions, as shown in Table 7. In the four chosen subspaces, only S_1 has somehow a uniform-like distribution of files among partition groups. Files are extremely uneven distributed in three other subspaces. Although the partition group number is large, due to the total number of files is relatively small, we employed Algorithm 2 to yield better efficiency.

We validate our locality-aware file allocation with two common applications, video retrieval and similar video fragment detection. We compare our solution with both plain Hadoop and CoHadoop, which only considers files being correlated in one dimension. In the experiments, we use subspace s_1's partition strategy to be the CoHadoop's file-locator mapping strategy, because it gives the minimum value of $\mathcal{C}^*$ (defined in Sect. 3). First of all, we shall examine the data uploading cost and the imbalanced data placement.

6.3.1 Uploading cost

By applying Algorithm 4, we find a partition strategy for the data set, which includes 198 partition groups and the file distribution among all the groups are more even,



(a) Locality aware data uploading v.s. plain Hadoop

(b) Data volume distribution among Data nodes

Fig. 10 Locality-aware data uploading and load distribution among Datanodes

with standard variance less than 10. We report the data uploading time in Fig. 10a. Compare to plain Hadoop, the data uploading time is slightly longer for locality-aware allocation. Since our underlying implementation adopts CoHadoop's data distribution methodology, that the files of the same partition group, as well as their replicas, will be distributed to the same group of Datanodes. However, during the uploading, we observe more I/O failures than uploading data to plain Hadoop. We believe that too many I/O requests concentrate on a small group of Datanodes within a small period of time may be the problem. However, this uploading cost is acceptable, since it can be amortized in future computations.

When the uploading is complete, the data volume distribution among Datanodes is shown in Fig. 10b. We have the same observation made in [11], that plain Hadoop has more balanced data load. In the experiments, the standard variance of data volume for plain Hadoop is 4.3 (GB), while for locality-aware allocation it is 16.6 (GB). In fact, this imbalance is closely related to multiple factors, like file distribution among partition groups, the order of files being uploaded and etc. [11]. However, our data retrieval and analysis upon such a data placement suggest that imbalance data load is an inevitable cost to achieve high throughput performance.

6.3.2 Retrieval

Note that we are not only querying the meta data of movies, we would like to have them retrieved (or scanned) from disk as some context analysis applications would require. The simplest query is given as follows: "Retrieve all the movies that are comedy, rating above or equal to 5 and produced in the 80s". We manually write a bench of such queries of different selectivity to test how our partition solution would improve the overall I/O performance. Figure 11 shows the time saving of select & retrieval queries concerning different subspaces and have different selectivity. The figure reports an aggregated time saving of all subspaces.

As shown in the figure, by applying locality-aware file allocation, some select & retrieval requests can be served more efficiently. The reason that our solution only works well for a select & retrieval query that concerns a small percentage of the entire data set, is that the collocated data are better only if it is faster to read a set of data

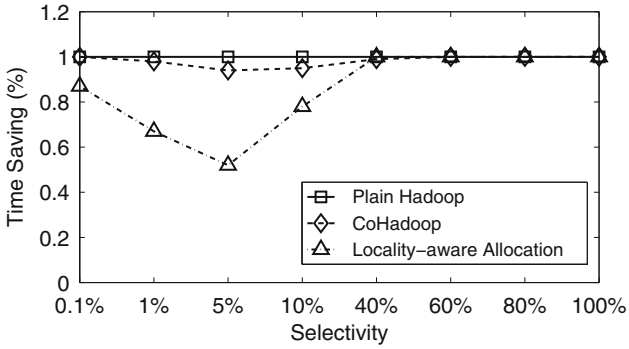


Fig. 11 Retrieval time saving for different selectivity

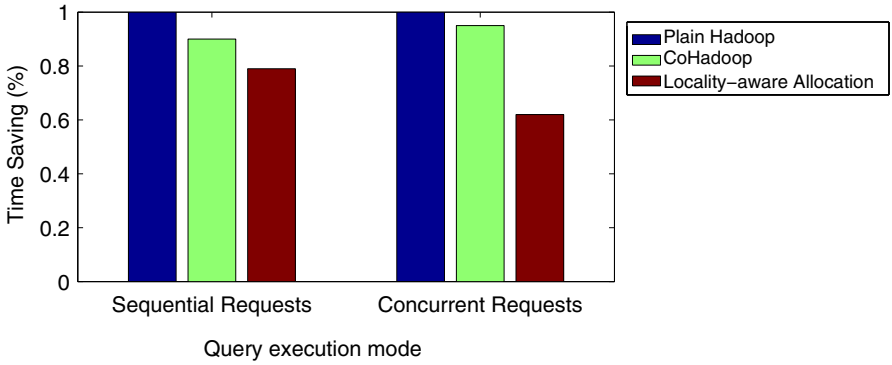


Fig. 12 Time saving for sequential and concurrent retrieval requests

from a few disks other than from almost all the Datanodes. When the selectivity is extremely high (very little data files will be retrieved), then no matter what the file placement strategy is, these files only locate on a very limited number of Datanodes. Therefore, the retrieval efficiency is no different. On the contrary, if the selectivity is low (large percentage of files will be retrieved), then the files to be retrieved will be located on a large percentage of the Datanodes, therefore, no retrieval efficiency is observed.

Since collocated data files best serve the requests that fit the feature of file partition, we report some convincing experiment results in Fig. 12. We report the time saving in two different query execution modes. One is to execute the select & retrieve query sequentially, the other one is to execute them concurrently. We can clearly observe that in the concurrent query execution mode, our file allocation strategy obtains significant time savings. The reason is that comparing to plain Hadoop and CoHadoop, the files being requested are more closely grouped into different sets of Datanodes. Therefore, each Datanode is expected to focus on limited data retrieval and transfer tasks.

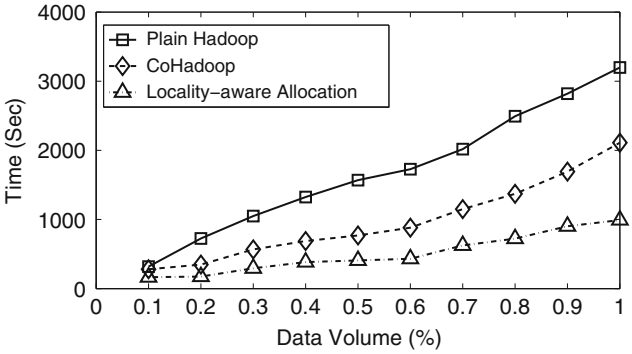


Fig. 13 Time cost of similar fragment detection under different file placement strategies

6.3.3 Similar fragment detection

Duplication detection is an important application in multimedia processing to protect copyrights. Since our data set is 60,000 different movies, there are no exact duplications. However, there are a lot of similar scenes, in our case similar fragments, that many movies may contain. Therefore, by analyzing the similar fragments, we can identify those hot scenes that usually appear in the movie, which can further help justify how similar two different movies are. Generally, we want to find out the hot or similar fragments for a category of movies, for example, the actions, probably fragments containing gun fire and car racing will be identified. Moreover, we want to find out the similar fragments for a set of movies that only take up a small subspace of movie attribute dimensions, where our locality-aware allocation strategy can be directly applied.

We conducted the similar fragment detection using different file allocation strategies. In the experiments, we compute our locality-aware partition strategy and compare it with the CoHadoop's and plain Hadoop's data placement strategy. For CoHadoop, we use $\mathcal{P}_b$ to determine the mapping from a *file id* to a *locator id*. Results are shown in Fig. 13. As shown in the figure, with multi-dimensional data correlation considered, our locality-aware allocation strategy is better than both CoHadoop and plain Hadoop. The main cost saving lies in the significant reduction of network volume. As closely related files are grouped together on a set of Datanodes, then the similar fragments among similar files can be well "combined" during the Map phase, and therefore less network volume will be introduced when inter-media data are copied from Map to Reducers.

7 Related work

Although Cloud promises enormous storage and computing capabilities, convincing studies [21,25] have pointed out the gaps between the MapReduce oriented Cloud computing paradigm and traditional parallel RDBMS in terms of relational operation supporting and efficiency. Extensive research efforts have been devoted to exploit MapReduce's performance [15] and enabling data indexing [29,32] and common

relational operations on Cloud, like join [2,4,16] and aggregation [7,8]. In this section, we briefly survey work [11] and some other most related work, which all take the advantage of grouping correlated files to achieve efficiency.

HadoopDB [1] tries to incorporate the advantages of both Hadoop and traditional RDBMS. Therefore, by building an overlay upon a group of RDBMS instances, HadoopDB promises efficient relational processing and a simple overview of the entire system. Since HadoopDB employs RDBMS to store data, traditional partition techniques are applicable to ensure the data locality. However, as HadoopDB changes the outlay of Hadoop architecture and may introduce heavy manual maintain cost of RDBMS instances, it sacrifices scalability and flexibility.

Hadoop++ [10] mainly targets on the operation over two correlated file, like join process. It introduces the "Trojans" to pre-process and index data files. Although it is completely built upon Hadoop, it is limited to predefined file correlations and may pay a significant cost when new files keep coming in.

The collocation of file on Cloud is first proposed in [11] in the context of log processing; however, it only considers the situations where files are correlated in one dimension. For any coming file, the system assigns it a "locator" number. And all the files with the same "locator" number shall be put into the same set of storage nodes following a best effort strategy. Both theoretical and empirical studies state their data placement policy keeps the fault tolerance property. Although CoHadoop is the first effort to collocate files on HDFS, it leaves the file allocation problem under multi-dimensional correlations not discussed.

Rimma Nehme et al. [20] study the traditional parallel database partition problem under a massive parallel processing (MPP) paradigm. Its focus is the way to decide which table should be replicated (to every computing node), and be replicated according to which specific column or attribute. On the contrary, our solution on Hadoop does not taking replication into consideration. We take the replication process is transparent and self-maintained by HDFS itself. Data allocation is slightly touched in work [13]. However, no data correlation-aware partition method is discussed. Ceph [31] actually considered correlation-aware file partitions by defining a data placement function "CRUSH". And data files are not only grouped, but also be partitioned. Being a sophisticated distributed system, Ceph has more information or control over data placement. However, for Cloud file systems, the underlying services are always transparent to application developers and users. Therefore, our work aims at exploring the possible improvement of file allocation strategy on Cloud platform from the perspective of application developers.

8 Conclusion

In this work, we identify a novel research problem of allocating files with multi-dimensional correlations on the Cloud storage, where traditional partition and collocation cannot be directly applied. We clearly formalize the problem and prove that it is NP-hard. We propose a heuristic algorithm to help us reach a near optimal file set partition solution. Extensive experiments on both synthetic and real data sets validate that our solution can significantly improve the execution efficiency of subspace-aware

data retrieval and analysis. We also identify several research challenges, which are tended to be addressed in our next step work.

References

1. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., Rasin, A.: HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In: Proceedings of VLDB Endow, pp. 922–933 (2009)
2. Afrati, F.N., Ullman, J.D.: Optimizing joins in a map-reduce environment. In: Proceedings of EDBT, pp. 99–110 (2010)
3. Amazon Web Service. http://s3.amazonaws.com
4. Blanas, S., Patel, J.M., Ercegovac, V., Rao, J., Shekita, E.J., Tian, Y.: A comparison of join algorithms for log processing in MaPreduce. In: Proceedings of SIGMOD, pp. 975–986 (2010)
5. Brunet, J., Tamayo, P., Golub, T.R., Mesirov, J.P.: Metagenes and molecular pattern discovery using matrix factorization. PNAS **101**(12), 4164–4169 (2004)
6. Chen, Y., Wang, W., Du, X., Zhou, X.: Continuously monitoring the correlations of massive discrete streams. In: Proceedings of CIKM, pp. 1571–1576 (2011)
7. Condie, T., Conway, N., Alvaro, P., Hellerstein, J.M., Elmeleegy, K., Sears, R.: MapReduce online. In: Proceedings of NSDI, pp. 313–328 (2010)
8. Condie, T., Conway, N., Alvaro, P., Hellerstein, J.M., Gerth, J., Talbot, J., Elmeleegy, K., Sears, R.: Online aggregation and continuous query support in MapReduce. In: Proceedings of SIGMOD, pp. 1115–1118 (2010)
9. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Proceedings of OSDI, pp. 137–150 (2004)
10. Dittrich, J., Quiané-Ruiz, J., Jindal, A., Kargin, Y., Setty, V., Schad, J.: Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). In: Proceedings of VLDB Endow, pp. 515–529 (2010)
11. Eltabakh, M.Y., Tian, Y., Özcan, F., Gemulla, R., Krettek, A., McPherson, J.: CoHadoop: flexible data placement and its exploitation in Hadoop. In: Proceedings of VLDB Endow, pp. 575–585 (2011)
12. Ghemawat, S., Gobioff, H., Leung, S.: The Google file system. In: Proceedings of SOSP, pp. 29–43 (2003)
13. Huang, J., Abadi, D.J., Ren, K.: Scalable SPARQL querying of large RDF graphs. Proc. VLDB **4**(11), 1123–1134 (2011)
14. IMDb. http://www.imdb.com/interfacesplain
15. Jiang, D., Ooi, B.C., Shi, L., Wu, S.: The performance of MapReduce: an in-depth study. In: Proceedings of VLDB Endow, pp. 472–483 (2010)
16. Jiang, D., Tung, A.K.H., Chen, G.: MAP-JOIN-REDUCE toward scalable and efficient data analysis on large clusters. IEEE Trans. Knowl. Data Eng. **23**(9), 1299–1311 (2011)
17. Jolliffe, I.T.: Principal Component Analysis. Springer, New York (2002)
18. Lei, M., Vrbsky, S.V., Hong, X.: An on-line replication strategy to increase availability in Data Grids. J. Futur. Gener. Comput. Syst. **24**(2), 85–98 (2008)
19. Lieberman, H., Selker, T.: Out of context: computer systems that adapt to, and learn from, context. IBM Syst. J. **39**(3–4), 617–632 (2000)
20. Nehme, R., Bruno, N.: Automated partitioning design in parallel database systems. In: Proceedings of SIGMOD, pp. 1137–1148 (2011)
21. Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., DeWitt, D.J., Madden, S., Stonebraker, M.: A comparison of approaches to large-scale data analysis. In: Proceedings of SIGMOD, pp. 165–178 (2009)
22. Ranganathan, K., Iamnitchi, A., Foster, I.: Improving data availability through dynamic model-driven replication in large peer-to-peer communities. In: Proceedings of CCGRID, pp. 376–381 (2002)
23. Samet, H.: Foundations of Multi-dimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling). Morgan Kaufmann Publishers Inc. (2005)
24. Silberschatz, A., Korth, H., Sudarshan, S.: Database Systems Concepts, 5th edn. MCGraw-Hill Inc. (2006)
25. Stonebraker, M., Abadi, D., DeWitt, D.J., Madden, S., Paulson, E., Pavlo, A., Rasin, A.: MapReduce and parallel DBMSs: friends or foes? Commun. ACM **53**(1), 64–71 (2010)
26. The Apache Software Foundation. Hadoop. http://hadoop.apache.org/

27. The Apache Software Foundation. HDFS architecture guide. https://hadoop.apache.org/hdfs/docs/current/hdfs_design.html
28. The Apache Software Foundation. Hive. http://hive.apache.org/
29. Wang, J., Wu, S., Gao, H., Li, J., Ooi, B.C.: Indexing multi-dimensional data in a cloud system. In: Proceedings of SIGMOD, pp. 591–602 (2010)
30. Wang, J., Jea, K.: A near-optimal database allocation for reducing the average waiting time in the grid computing environment. J. Inf. Sci. **179**(21), 3772–3790 (2009)
31. Weil, S.A., Brandt, S.A., Miller, E.L., Long, D.D.E., Maltzahn, C.: Ceph: a scalable, high-performance distributed file system. In: Proceedings of OSDI, pp. 307–320 (2006)
32. Zhang, X., Ai, J., Wang, Z., Lu, J., Meng, X.: An efficient multi-dimensional index for cloud data management. In: Proceedings of CloudDB, pp. 17–24 (2009)