WATSON: A Workflow-based Data Storage Optimizer for Analytics

Jia Zou, Ming Zhao
Computer Science and Engineering
Arizona State University
Tempe, USA
{jia.zou, mingzhao}@asu.edu

Juwei Shi
Search Technology Center Asia
Microsoft
Beijing, China
juwei.shi@microsoft.com

Chen Wang
School of Software, EIRI
Tsinghua University
National Engineering Lab for Big Data Software
Beijing, China
wang_chen@tsinghua.edu.cn

***Abstract*—This paper studies the automatic optimization of data placement parameters for the inter-job *write once read many* (WORM) scenario where data is first materialized to storage by a producer job, and then accessed for many times by one or more consumer jobs. Such scenario is ubiquitous in Big Data analytics applications but existing Big Data auto-tuning techniques are often focused on single job performance.**

To address the shortcomings in existing works, this paper investigates data placement parameters regarding blocking, partitioning and replication and models the trade-offs caused by different configurations of these parameters through a producer-consumer model. We then present a novel cross-layer solution, WATSON, which can automatically predict future workloads' data access patterns and tune data placement parameters accordingly to optimize the performance for an inter-job WORM scenario. WATSON can achieve up to eight times performance speedup on various analytics workloads.

***Index Terms*—storage, data placement, auto-tuning, parameter optimization, Big Data analytics**

I. Introduction

Data analytics applications based on MapReduce [33] (e.g. Hadoop [35]), distributed dataflow computing (e.g. Spark [36], and distributed relational systems (e.g. Spark SQL [7]) are widely deployed today for daily operations. However, it is observed that enterprise IT professionals and data scientists tend to use default values to configure storage parameters such as replication factor, block size, and partitioning of the underlying distributed file systems, and most of them do not understand the importance and the art of tuning these parameters. This causes slow interfacing between storage and computation and leads to under-utilized storage and computation resources. Tuning Big Data storage can significantly improve performance, but it is also a time-consuming task for IT professional and data scientists. Therefore to offer timely and cost-effective analytics processing with least effort is a key for success in many businesses, such as banking, telecommunication and so on.

In this paper, we mainly focus on the problem of automatic storage parameter tuning for an inter-job **write once read many** (WORM) scenario. In such scenario, once a producer job outputs data to a distributed file system (abbreviated as DFS), the data will be processed by one or more consumer jobs repeatedly. The inter-job WORM scenario is ubiquitous in many popular applications deployed in enterprise and cloud, and we just list a few as follows:

- **Data Warehousing.** Data warehouses such as Spark-SQL [34] and Hive [34] are playing critical roles in daily operations of enterprises [9]. Once data is persisted to DFS, the data may be repeatedly processed by a large number of different jobs for various purposes, e.g. queries and pre-processings.
- **Iterative Analytics.** A lot of machine learning algorithms are iterative-converging in nature. In such iterative analytics workloads, the same input data (i.e. training samples) needs to be repeatedly processed.

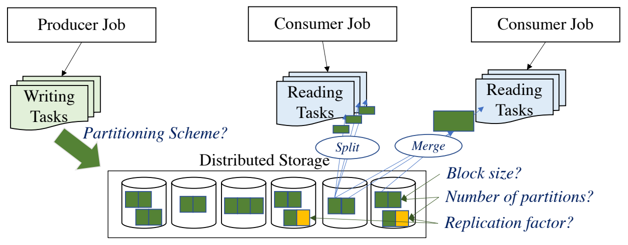


Fig. 1. Abstraction of inter-job WORM scenario.

The WORM pattern is prevalent in real-world cloud traces, such as the Cloudera traces [9] and Yahoo cluster traces [1]. For example, in the publicly available Yahoo cluster traces, we observe that, the total size of files that have been read for at least once is 2.02 peta bytes, and 72% of these bytes are accessed for more than 10 time after being written, and 28% of these bytes are accessed for more than 100 times.

As shown in Fig. 1, in an inter-job WORM scenario, we use *producer* to represent the analytics job that writes data to DFS, and use *consumer* to represent an job that reads the data for processing. WORM scenarios not only exist in workflows of multiple applications, where the producer and consumer are belonging to two standalone applications, but also exist in applications that run multiple job stages, where the producer and consumer are simply two different job stages inside the same application.

Existing auto-tuning tools, such as MRTuner [33], What-if Engine [16], CDBTune [37] and QTune [22], cannot really optimize the overall performance for an inter-job WORM scenario, because they focus on the performance of a single job. When applied to optimize the producer's performance, they are not aware of consumers and do not consider the inter-job trade-offs between the producer and consumers. However we find that data placement parameters for the producer's output, such as block size, partitioning, and replication factor, are controlling performance trade-offs between the producer and consumers in an inter-job WORM scenario. These parameters are not only determining the producer's speed and volume for writing to DFS, but also critical to the consumers' data locality. For example, replication factor and partitioning determine data distribution across cluster, and skewed data distribution may cause difficulty in scheduling tasks on local data. In addition, non-alignment between block size and consumer's split size will cause a map split, the input data of a map task, to span multiple nodes, and incur unnecessary network transfer, while setting split size to a default block size like 128MB can also hurt performance [33].

To address the shortcomings of existing works, we establish a novel Producer-Consumer model to formalize the total overhead dependent on data placement parameters for all jobs. We model DFS writing overhead of the producer as aggregation of per-block overhead caused by disk seek and metadata management, and per-byte overhead caused by writing block data to DFS. We also model data transfer overhead caused by split spanning, task scheduling, and re-partition respectively for a consumer's job stage. Based on the model, we can determine the optimal data placement parameters and minimize total overhead. However, there are two challenges in solving above optimization problem:

1) Due to the gap that exists between DFS and computation in current practice, the DFS writing process is unaware of workloads' data access pattern and it is difficult to obtain consumer information for the model.
2) Due to vast search space for data placement parameters, it is difficult to solve the optimization problem online using a brute-force approach.

To reduce the search space, based on a high-level abstraction of analytics job scheduling policy, we mathematically prove deductions that can be summarized as: a consumer's data transfer overhead can be minimized, if and only if block size is aligned with split size, data is proportionally distributed to each node's maximum parallelism, and the partitioning scheme is mostly consistent with consumers re-partitioning plans. Then we develop an optimization approach based on greedy strategy to automatically optimize the data placement parameters.

To bridge the gap between DFS and computation, and to provide the greedy algorithm with consumer information, we propose a novel cross-layer solution, WATSON, to enable workload-aware data storage. WATSON can automatically collect runtime behavior of previous jobs. Thus, when a new job is launched, WATSON can automatically predict consumer information if the same type of jobs have been executed before. Then WATSON applies the greedy algorithm to compute optimal data placement parameters. WATSON is not only applicable for tuning of the storage parameters of the data that connects multiple job stages inside a single application, but also working well for data that connects multiple standalone applications.

The contributions of this work are as follows:

1) We identify the key role of storage parameters in controlling inter-job trade-offs, and to model total overheads dependent on these parameters for all jobs in an inter-job WORM scenario. We also mathematically prove the deductions that lead to an efficient greedy algorithm and significant reduction of the search space for the optimization problem.
2) We present an online cross-layer solution, WATSON, to automatically predict consumer information, tune storage parameters and optimize the overall performance for inter-job WORM scenarios in Big Data analytics applications.
3) We implement WATSON on top of PlinyCompute [38], a distributed object-oriented relational database using Pangea [39], [40] as its storage, and Hadoop. We also conduct an experimental evaluation using various analytics workloads. The experiment results show that WATSON can achieve significant speedup for analytics workloads such as TPC-H, linear algebra operation like matrix multiplication, PageRank analytics flow, TeraSort, k-means, NutchIndex and so on.

The remainder of this paper is organized as follows. We analyze the problem in Section II. Then we discuss the Producer-Consumer model and deductions in Section III. After that, we describe the architecture of WATSON in Sec. IV. We discuss evaluation results in Sec. V. Finally, we discuss related works and conclude the paper.

II. Problem Analysis

A. Distributed File System

Big Data analytics such as MapReduce systems, distributed dataflow systems, and distributed relational database systems, can employ mainly three different types of storage abstractions: i) block-based distributed file systems, such as GFS [14], HDFS [2], and GPFS [15]; ii) distributed object stores, such as Amazon S3 [5] and SWIFT [30]; and iii) distributed in-memory file systems, such as Tachyon [23]. This paper focuses on the block-based distributed file systems, which is the most popular storage for Big Data analytics.

For block-based distributed file systems (abbreviated as DFS), its communication with upper-layer systems include following aspects:

- **Chunking.** It is to chunk data into blocks and use block as the unit of data allocation and transfer to save disk seek time. The *block size* parameter can be configured for each analytics job output.
- **Partitioning.** For an analytics job that outputs data to DFS, each writing task of the job will generate a file partition in local node. Each partition will consist of

one or more blocks. Therefore the *number of partitions* for output is controlled by the *number of writing tasks*. Writing tasks can be reducers or mappers for map-only job. In addition, the upper layer system can control how data is grouped into blocks and how blocks are grouped to a partition based on the selection of partitioning key(s). For example, the upper layer system can write to storage in a way that all data having the same value of partitioning keys must reside in one partition, and should not live in more than one partitions.

- **Replication.** For each block in a partition, DFS will replicate it to a few nodes across the cluster for resilience. The *replication factor* parameter can be configured for each job output to control the number of replicas per block.

B. Analysis about Inter-Job WORM Scenario

In an inter-job WORM scenario as shown in Fig. 1, we observe that the data placement parameters such as block size, number of partitions (i.e. number of writing tasks), partitioning schemes and replication factor, are double-edged swords for the performance of producer and consumers. In addition, those parameters will involve knowledge gaps between computations and DFS, if computation is unaware of underlying DFS read/write principles or DFS is unaware of upper-layer job characteristics.

In Big Data analaytics systems, an analytics job can have many stages, and each stage reads in input data from memory/storage, process it (through a set of homogeneous parallel tasks with each task executing a series of pipelined operators), and the stages are inter-connected by the shuffling operator that repartitions data to prepare for the next stage. For block-based distributed file systems (abbreviated as DFS), the job scheduling models inside a job stage can fall into two categories: (1) Short-living-Task-based scheduling, where each time the scheduler gets a split of input data and schedules a task as a short-living thread/process for processing the split, such as Spark and Hadoop; (2) Long-living-Process-based scheduling, where each long-living process iteratively gets a split and processes it until all blocks have been processed, such as Pliny-Compute [38]. Although the two scheduling strategies slightly differ in task scheduling overhead and resource utilization, we find their interplay with distributed storage is similar.

We identify following data placement parameters that represent the key performance trade-offs and cross-layer gaps under both job scheduling models.

1) **Block Size:** In our work, we find that it is critical to tune block size to make the memory footprint of all concurrent tasks fit available memory, and make all CPU cores fully utilized. Now we give several examples.

n-gram is a language modeling approach widely used in text classification. Given a text ”Alice was beginning to get very tired”, if n is set to 3, a 3-gram step will produce shingles ”alice beginning”, ”alice beginning get”, ”beginning get”, ”beginning get very”, ”get very”, ”get very tired”, and ”very tired”.

In the n-gram workload, the size of the output, which will exponentially increase with the parameter n, could be significantly larger than the size of the input text (e.g. the output size is more than five times larger than the input size when $n = 3$, and more than nine times larger when $n = 4$).

In skip-gram, which is another popular language model, words do not need to be consecutive, so the output will be even larger in size than n-gram for the same input.

For such workloads, using a default block size like 256 megabytes and using block size as task split size as widely adopted is sub-optimal, because the output for all concurrently running tasks may not fit the available memory. In such situation, we need use much smaller task split size to avoid disk spilling.

Workloads that invoke computational intensive UDFs (e.g. matrix multiplication, dynamic time warping, and etc.) will benefit from splitting even a small input dataset into a large number of task splits so that each CPU core in a distributed cluster can be allocated with a task to run on it, and then all computational resource can be fully utilized to reduce the latency.

In contrary, workloads that do very simple computations and have small memory footprint, such as top-k, count, merge-sort and so on, will run more efficiently by using large task split size to reduce the number of waves of tasks and thus reduce the overhead for setting up and cleaning up tasks.

For the producer, chunking the output data into fewer blocks will lead to less metadata management, smaller metadata size, and smaller total disk seek latency.

For a consumer, a job divides the input to fixed-size splits and creates a map task to handle each split, while for underlying storage, block is the basic unit that can be guaranteed to be stored in a single node. When a split spans two or multiple blocks which are possibly stored in different nodes, part of this split may have to be transferred. Hadoop guideline [35] suggests to set size of input split to the block size to avoid this nonalignment problem, but it overlooks the fact that setting split size to a default block size like 64MB or 128MB can hurt performance for a lot of applications [33].

Gap 1. If block size is set unaware of consumer split size, the gap between block size and split size causes the non-alignment problem.

2) **Number of Partitions:** For the producer, the number of writing tasks determines the number of partitions for the output data. It also determines the parallelism and pipelines used to write data to DFS, which influences the writing speed.

For a consumer, partition number can influence the distribution of its input data. If the distribution of data is disproportional with the distribution of consumer map tasks, unnecessary data transfer will be incurred. For example, the producer writing process can cause that a node executing more consumer tasks may not have enough data to process, while a node executing fewer consumer map tasks may have too much data. Thus, data has to be transferred to nodes executing more map tasks.

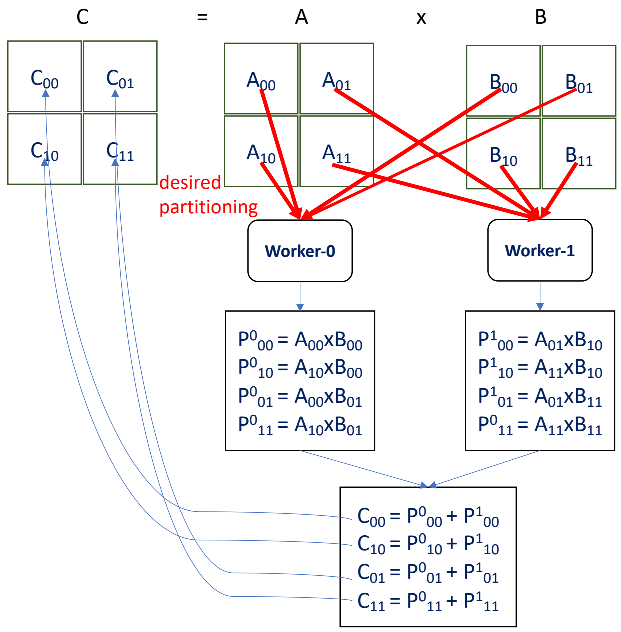


Fig. 2. Illustration of distributed matrix multiplication.

Gap 2. If the number of writing tasks is set unaware of the distribution of consumer tasks, disproportional data distribution may cause additional data transfer overhead for consumers.

3) **Partitioning Scheme:** For the producer, adopting different partitioning schemes such as random partitioning, range partitioning, or hash partitioning will incur varying amount of overheads. For example, range partitioning requires sorting objects; and hash partitioning requires computing the hash function for each object. On the other side, the partitioning scheme of the data can significantly impact the consumer job's performance: range partitioning can accelerate consuming workloads that involve range-based filtering operations by avoiding unnecessary scanning overhead; and hash partitioning of two correlated datasets based on the join key can speedup corresponding join consumers by getting rid of the re-partitioning (i.e. shuffling) overhead. For example, as illustrated in Fig. 2, consider two producers create two $N \times N$ matrices, A and B, and a consumer performs distributed multiplication of the two matrices. Then if the producers partition A and B, so that A[i][j] co-locates with B[j][k] for any i, j, k, e.g. the j-th column of A and the j-th row of B both get dispatched to the same node $j\%nh$, where nh represents the number of hosts in the cluster. Then the matrix multiplication consumer can avoid the re-partitioning overhead for computing the multiplications of all pairs of A[i][j] and B[j][k].

Gap 3. If the partitioning scheme is set unaware of the consumer filtering and joining operations, undesirable data partitioning scheme may cause additional scanning or shuffling overhead for consumers.

4) **Replication Factor:** For the producer, increasing replication factor will increase the volume of data to write to DFS.

For a consumer, increasing the replication factor of the input data can enable a task easier to be scheduled with data co-located on the same node, thus improving the data locality.

Gap 4. If the number of replication factor is set unaware of how different replication factors impact consumer's locality, overall performance may not be optimal.

C. Problem Statement

This paper is focused on following problem: to automatically determine the best data placement parameters, such as block size, number of partitions, partitioning schemes, and replication factor, for the output data of a given producer, so that the overall performance including the producer and consumer(s) in an inter-job WORM scenario can be optimized.

All applications including the inter-job WORM scenario, such as data warehouse applications, analytic flows, and etc., should benefit from such optimization.

III. Producer-Consumer Model

A. Model and Notations

We propose a Producer-Consumer model, shown in Fig. 1. The producer writes output data to DFS, while the data will be processed by consumers.

To achieve the optimal overall performance, we seek to minimize the total overhead (described in EQ(1)) involved in this Producer-Consumer model: the overhead of writing data to storage in the producer job stage $O_{producer}$ and the overhead of transferring data in all consumers' job stages. We group consumers having similar run-time behavior as a consumer type. We let $O_{consumer_i}$ denote the overhead caused by the i-th type of consumers, and let f_i denote the number of occurrences for the i-th type of consumer.

$$O_{total} = O_{producer} + \sum_{i=1}^{n} f_i \times O_{consumer_i} \tag{1}$$

We integrate the Short-living-Task-based scheduling and Long-living-Processed-based scheduling by abstracting a higher-level scheduling model, in which the total number of input splits that can be processed concurrently (i.e. the parallelism) are defined as ns, which is a concept similar to the number of slots in Hadoop [35] and Spark [36], and close to the number of long-living processes in PlinyCompute [38]. Then each time when a slot is available, or a long-living process is idle, an input split will be fetched to be processed by a newly created short-living thread as in Hadoop and Spark, or an idle long-living process as in PlinyCompute. All tasks that run concurrently form a wave.

We denote the data output from the producer as a set of objects, as $D = \{x\}$, the number of objects as $\|D\|$, and the cluster as a set of hosts, as $H = \{h\}$. The replication factor is denoted as r. We also assume the total size (in bytes) of D is denoted as S, and the block size is denoted as B. In addition, we assume the producer uses k threads to shuffle data into p partitions following a partitioning function denoted as $f : x \rightarrow h$.

At the consumer side, the split size for the i-th consumer is represented as s_i, and the desired partitioning scheme of the input data is denoted as a function $g_i : x \rightarrow h$.

B. Producer Overhead Modeling

The producer overhead falls into two parts: the partitioning overhead denoted as $O_{partitioning}$ that the partitioning threads take to shuffle objects and the writing overhead denoted as $O_{writing}$ that the writing threads take to write partitions to disk.

For a single object x, its communication overhead involved in partitioning can be illustrated in EQ(2), where $host(x)$ represents the host node where x resides before being partitioned, and o_p represents the unit communication overhead (in terms of latency) associated with each object that is partitioned to a remote node.

$$o_{partition}(x) = \begin{cases} o_p & host(x) \neq f(x) \\ 0 & host(x) = f(x) \end{cases} \tag{2}$$

Then the total communication overhead involved in partitioning can be described as $\sum_{x \in D} o_{partition}(x)$, which is to sum the overhead for all objects.

In addition, the computation overhead involved in partitioning can be represented as $o_f \times \frac{ns \times S}{k}$, where o_f represents the unit computation overhead for running partition function f on unit size of data.

Then, the total partitioning overhead can be represented as EQ(3), where $o_f \times \frac{ns \times S}{k}$ models the per wave overhead, and $\lceil \frac{k}{ns} \rceil$ represents the number of waves.

$$O_{partition} = \sum_{x \in D} o_{partition}(x) + o_f \times \frac{ns \times S}{k} \times \lceil \frac{k}{ns} \rceil \tag{3}$$

The overhead involved in producer writing process falls into two categories: one is per-block overhead, including overhead about meta data management and disk seek, and we let o_i denote its unit overhead; and the other one is per-byte overhead, such as writing data to storage, and we let o_w denote unit per-byte ovehead. o_w is relevant with disk speed and disk number.

The total per-block and per-byte overhead for each writing task then can be represented as $\frac{o_i \times \frac{S}{B} \times r + o_w \times S \times r}{p}$. In addition, the number of waves can be computed as $nw = \lceil \frac{p}{ns} \rceil$.

Because writing tasks in a wave can execute in parallel, according to Amdahl's law [6], $O_{writing}$ can be computed as EQ(4).

$$O_{writing} = \frac{o_i \times \frac{S}{B} \times r + o_w \times S \times r}{p} \times \lceil \frac{p}{ns} \rceil \tag{4}$$

Therefore, we can further formulate producer overhead as described in EQ(5).

$$O_{producer} = O_{partition} + O_{writing} \tag{5}$$

C. Consumer Overhead Modeling

The placement of the input data can also impact the performance of consumer jobs and incur two types of overhead. One is the re-partitioning overhead if the partitioning scheme is not desirable (i.e. $f \neq g_i$). The other is the reading overhead due to the inconsistency between split size and block size (i.e. $B \neq s_i$), and non-proportional data distribution (i.e. $p \% ns \neq 0$).

The re-partitioning overhead for each object in the i-th consumer job can be modeled as EQ(6).

$$o_{repartition_i}(x) = \begin{cases} o_p & f(x) \neq g_i(x) \\ 0 & f(x) = g_i(x) \end{cases} \tag{6}$$

Then the overall re-partition overhead aggregated for all objects in the i-th consumer job can be represented as EQ(7).

$$O_{repartition_i} = \sum_{x \in D} o_{repartition_i}(x) \tag{7}$$

In addition, if o_t is used to represent the average per-byte overhead of transferring data from remote nodes to form input splits, and p_j is used to denote the expected percentage of split data locating on remote nodes for the j-th task, then the expectation of data transfer overhead for each task in the i-th consumer, denoted as o_{input_i}, can be described as EQ(8).

$$o_{input_i} = \frac{\sum_{j=1}^{\frac{S}{s_i}} o_t \times (p_j \times s_i)}{\frac{S}{s_i}} \tag{8}$$

Because the number of waves can be computed as $nw = \lceil \frac{\frac{\|D\|}{s_i}}{ns} \rceil$, the overhead for forming input splits in the i-th consumer, O_{input_i} can be modeled as EQ(9).

$$O_{input_i} = o_{input_i} \times nw = \frac{\sum_{j=1}^{\frac{S}{s_i}} o_t \times (p_j \times s_i)}{\frac{S}{s_i}} \times \lceil \frac{\frac{S}{s_i}}{ns} \rceil \tag{9}$$

Then, the total overhead caused by data placement in the i-th consumer can be represented by EQ(10).

$$O_{consumer_i} = O_{repartition_i} + O_{input_i} \tag{10}$$

D. Useful Rules Deducted

Based on the Producer-Consumer Model, we can derive four rules through deductions. (Please refer to the Appendix for more detailed proofs.)

1. **Consistent Partitioning Scheme Rule.** $O_{consumer_i}$ can be minimized only if the partitioning scheme of the input data is consistent with the desired partitioning scheme of the i-th consumer job. (i.e. $f = g_i$).

If the data is already partitioned as desired, each object resides on the right host, and $O_{repartition_i} = 0$.

2. **Aligned Split and Block Size Rule.** $O_{consumer_i}$ can be minimized only if block size is equal to split size (i.e. $s_i = B$).

Intuitively, if block size is smaller than input split size, then a split consists of two or more blocks that are possibly stored in different nodes, and part of this split may have to be transferred. If block size is larger, a block is going to be processed by two or more tasks that can possibly run on different nodes. The latter case also generates unnecessary network transfer.

3. **Proportional Data Distribution Rule.** $O_{consumer_i}$ can be minimized only if the size of data stored in each node is proportional to the number of slots in each node (i.e. $\forall$node $j, ns_j/ns = nb_j/nb$, where nb represents total number of blocks and nb_j represents number of blocks on the j-th node).

This rule is to ensure that the data distribution should be proportional to the resource distribution. A node with more computation power should be stored with more data.

4. **Invariant Replication Rule.** If for a consumer job, the storage block size is aligned with task split size and input data is distributed proportionally to slot number for each node in the cluster (i.e. $s_i = B\ and\ \forall$node $j,\ ns_j/ns = nb_j/nb$), there will be $O_{inpupt_i} = 0$, regardless of the replication factor. Therefore, tuning replication factor can not further improve the performance of this particular consumer.

IV. WATSON SYSTEM OVERVIEW

A. Data Placement Parameter Tuning

Based on the Producer-Consumer Model and the four deductions introduced in Section III-D, we propose an automatic parameter tuning approach based on a greedy strategy [41] for data placement in WORM scenario.

The main idea is to optimize for the dominant consumer type. The dominant consumer type (abbreviated as DCT) refers to the consumer type that has the highest occurrences in all consumer types. Once DCT is identified, we optimize its performance by setting the partitioning scheme to be the desired partitioning scheme of DCT, based on Rule 1; setting data block size to be equal to DCT's split size, according to Rule 2; and then setting the number of partitions to be proportional to slot number for each node, according to Rule 3. Thus we transform the problem to a searching problem with only single parameter, which is the optimal replication factor r in a user specified range $[r_{default}, r_{max}]$. Usually, we set $r_{default} = 3$ and $r_{max} = 5$. The algorithm is illustrated in Alg. 1, where mb represents the maximum memory resources available for each slot, and other notations are explained in Sec. III-A.

During the search process, we simplify $O_{partition}$ and $O_{repartition_i}$ to be zero, as these parts of overhead are irrelevant with the value of r. The formulation of p_i can be found in Eq(19) in Appendix.

If heterogeneous replication [39] is applied with a replication factor of r, we can extend the above algorithm to select top-r DCTs, and each replica can have data placement parameters tuned to meet the requirements of one of r DCTs.

While the algorithm is straight-forward, the challenge is that the greedy algorithm requires workload-level predictions, such as the frequency, desired partitioning scheme, and desired split size as well as platform information, such as the desired number of slots on each cluster node, to compute the optimal data placement parameters. While it is relatively easier to obtain platform-related information that is usually static [16], [33], it is difficult to predict information regarding future workloads (i.e. consumers). In this paper, we mainly discuss how to obtain workload-related information as listed in TABLE I.

Algorithm 1 Greedy Strategy with Multi-way Block-level Replication.

```
Output: Block Size: $B$; Partition Number: $np$; Partitioning Scheme: $f$; Replication factor: $r$
 1: $dct \leftarrow searchIndexOfMostFrequentCustomerType()$
 2: if $s_{dct} \times ns > S$ then
 3:     $B \leftarrow S/ns$
 4: else
 5:     $B \leftarrow s_{dct}$
 6: end if
 7: $f \leftarrow g_{dct}$
 8: $nw \leftarrow \frac{S}{mb \times ns}$
 9: $np \leftarrow nw \times ns$
10: $O_{\min} \leftarrow \infty$
11: $r_{optimal} \leftarrow r_{default}$
12: if $n > 1$ then
13:     for each $r_{default} \leq r \leq r_{max}$ do
14:         $O \leftarrow O_{producer} + \sum_{i=1}^{n} f_i \cdot O_{consumer_i}$
15:         if $O < O_{\min}$ then
16:             $O_{\min} \leftarrow O$
17:             $r_{optimal} \leftarrow r$
18:         end if
19:     end for
20:     $r \leftarrow r_{optimal}$
21: end if
22: return
```

TABLE I
WORKLOAD INFORMATION REQUIRED BY THE GREEDY ALGORITHM

Notation	Description
C	Set of consumer job types $\{consumer_i\}$
f_i	Occurences of the i-th consumer job type
g_i	partitioning scheme of the i-th consumer job type
s_i	Split size of the i-th consumer job type

B. Prediction of Workload-level Information

It is observed in real production deployments in Facebook, Cloudera, and Microsoft that a significant portion of running jobs will re-execute in the future [9], [19]. This implies that it is possible to predict runtime behavior of a job based on its historical runs.

So the main idea is that when a job is going to write data to storage, we can predict the potential consumers of the data, based on historical producer-consumer patterns. That is, if a historical job once consumed data written by a similar producer, we assume the producer-consumer workflow will reoccur, and the job may execute again to consume the data that is being written now. Then we can obtain a set of predicted consumer jobs through historical workflow analysis.

Thus, the two challenging questions are:

(1) *How to know the current running job is a re-execution of a historical run of the same job?* In SQL-based systems, it can be easily achieved by matching the relational algebra representation. However, it is not so easy for applications that heavily use User Defined Functions (UDFs).

(2) *How to efficiently analyze workflows to predict consumer job information?* Here, a workflow represents a set of jobs that have producer-consumer relationships, where a job consumes outputs of other jobs.

We address these questions in the next two sections.

1) Job Matching: Existing works such as MRTuner [33] and Foreseer [41] propose to concatenate job names and parameter values for classifying jobs. Such approach is insufficient for cloud environment, since different executions of the same job submitted by different users could use different job names. Other existing works such as ProfSpan [42] use frequent instruction sequences profiled from hardware sample data to classify jobs, which is still not ideal, because sometimes slight changes (e.g. one line change in join predicate) could bring significant impacts to workload behavior. Another approach is to cluster jobs by performance metrics such as CPU utilization, memory utilization, and I/O activities captured in time series. However, it does not work well in our case, because the same job running with different sizes of data must have very different performance metrics and resource utilization, while actually such executions should be classified as belonging to the same job. Another approach is to abstract a job as a computational graph based on its intermediate representation (IR), where each node is an operator that is customized by UDFs (such as map, reduce, join, aggregate, groupBy, and flatten), and each edge represents the data. Then the job matching problem is modeled as a graph isomorphism problem [13]. We propose to use this approach in most cases, but to use job identifiers and frequent instruction sequences to match jobs on platforms where IR is not available for analysis.

2) Consumer Prediction.: Given a producer with specific output data path, we can predict the consumers of the output data as job instances belonging to any of following two sets:

- **Set of Historical Job Successors.** A historical job successor is a historical job instance whose input data is the output data of another historical job instance that matches current producer's IR.
- **Set of Historical Data Successors.** A historical data successor is a historical job instance and its input data path is the same with the producer's output data path.

Given the consumers predicted for the data being written by the given producer, we can further predict the occurrences, task split size, and desired partitioning scheme of each consumer.

1. Occurrence Prediction. For each type of consumer jobs, we track its occurrences, and use the inverse of its most recent reference distance (i.e. the difference of the two most recent access time) to predict its frequency.

2. Desired Split Size Prediction. If the task output size becomes larger than memory buffer size, disk spills will occur and cause additional latency. So the best practice to tune the split size is to keep the output bytes generated bounded by the memory buffer size [33]. Therefore, given memory buffer size for each task not changed, the optimal split size can be regarded as fixed for a job type, no matter how data input size changes. In this work, for each job type, we will first analyze its historical execution performance counters to derive the selectivity ratio of each job stage, which is defined as the ratio of the size of output data to the size of the input data. This selectivity ratio (denoted as sr) is then used to estimate the desired input split size of the job stage. We consider two cases with different memory architectures. In the first case, each thread reads input from disk and uses a thread-local buffer to cache the output, which is then written to disk or shuffled over network, while disk files can be cached in OS buffer cache. Hadoop is an example system of this case. In the second case, all threads/processes on the same worker share a centralized cache that can be used to cache both input and output in memory. Both Spark and PlinyCompute fall in the second case. For both situations, the goal of tuning split size is to avoid disk spills by caching all data required for a wave, including input and output, in the main memory. But there exists a special situation where input data size is small and cannot fully utilize the memory resource. In this case, we choose to fully utilize the CPU resource by distributing the data to all threads on each worker. So the desired split size can be estimated based on Eq. 11.

$$s_{desired} = \min(\frac{avail\_mem\_size}{(1+sr)\times ns}, S/ns) \qquad (11)$$

3. Desired Partitioning Scheme Prediction. In relational systems where the interface is based on SQL, such as Hive [34], SparkSQL [7], it is easy to infer the desired partitioning scheme of a consumer query. For example, for a join query such as *"Select employee.id, department.code From employee, department, Where employee.depart == department.name"*, it is easy to infer that the desired partitioning scheme of the employee table should be hash partitioning on the depart field; and the desired scheme of the department table should be hash partitioning on the name field, so that, the join can avoid the shuffling overhead. In Spark, as all join interface is based on key-value pairs, we can derive from the IR that a join consumer's input data should be hash partitioned using the function that extracts the join keys. In PlinyCompute [38], we developed a domain-specific language (DSL) for programmers to specify the details of join predicates in form of lambda calculus [8], [27], so that a system optimizer software can infer the desired partitioning scheme. Due to space limitation, we leave the detailed description of IR matching and the discovery of desired partitioners to future works.

C. WATSON *System Design*

In this section, we propose the design of a cross-layer solution: WATSON, which is processing between computational framework and storage to enable workload-aware data storage. WATSON is responsible for collecting historical information,

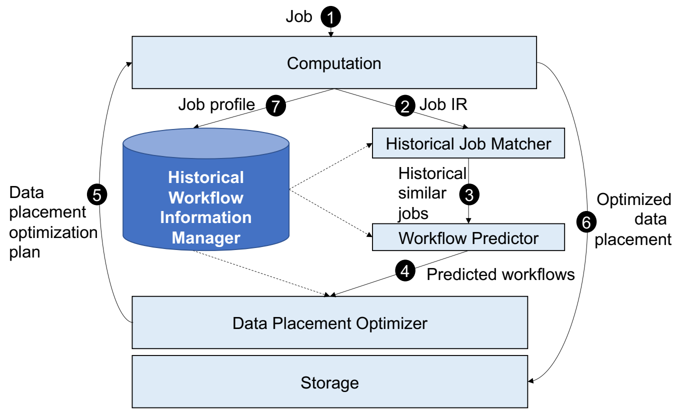
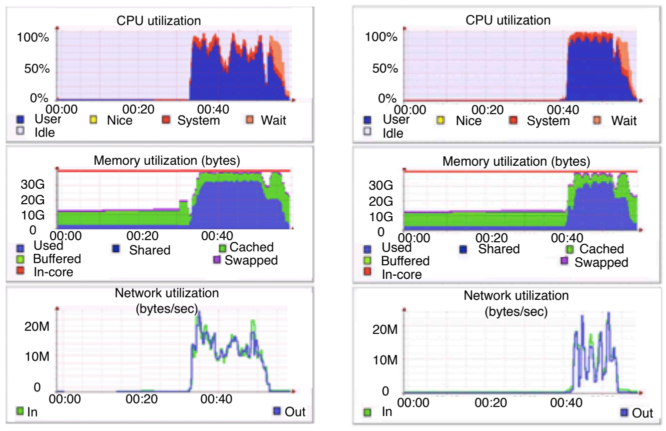


Fig. 3. WATSON Architecture Overview.



(a) Before Tuning (b) After Tuning

Fig. 4. Profiling results for CPU, memory and network before and after tuning block size and number of partition

predicting workload information, and optimizing data placement parameters using the greedy strategy as described in Sec. IV-A.

As shown in Fig. 3, WATSON has the following components:

1) **Historical Workload Information Collector:** This component collects and manages historical workload-level information, such as job IR, job input path, job output path, job input data size, and job output data size. It also collects and manages platform-related information, such as number of CPU cores and memory size on each cluster node. The collected information is stored in a SQLite database.

2) **Online Consumer Behavior Predictor:** Given a producer going to write data to storage, this component analyzes the producer's job IR and output path. Then it predicts consumer information as listed in TABLE I based on historical workflow analysis as described previously.

3) **Online Data Placement Optimizer:** Given the information predicted by the Online Workload Behavior Predictor, the Data Placement Optimizer is responsible for computing the optimal block size, number of partitions, partitioning scheme, and replication factor, following Alg. 1, to configure the storage of data that is to be output by producer.

V. Evaluation

A. Experiment Environment Setup

We implemented the WATSON system on top of PlinyCompute [38] and Hadoop 1.1.1. For Hadoop, we support the tuning of only block size, number of partitions, and replication factor, and on Hadoop we compare WATSON to existing automatic parameter tuning tool such as MRTuner. For PlinyCompute, we not only tune these parameters, but also tune the partitioning schemes by analyzing PlinyCompute's Lambda Calculus IR at runtime.

In this section, we evaluate the performance improvement gained by WATSON for Big Data analytics workloads on these platforms. We deploy PlinyCompute on a cluster consists of ten AWS r4.2xlarge instances on AWS. Each instance consists of eight cores, 61GB memory, and 100GB CPU cores. For PlinyCompute, we tune the number of slots to eight, and the buffer pool size to fifty gigabytes. For Hadoop, we deploy it on a cluster that consists of ten HS21 blade servers connected by 10Gb Ethernet. Each of the nodes has four single core 3.6GHz processor, 4GB memory, and 130GB disk drive. The slot number is set to four, and all other parameters are configured by MRTuner [33].

B. Hadoop Platform

On the Hadoop Platform, we only tune the block size, number of partitions, and replication factor. Due to the unavailability of an analyzable IR to unobscure user defined `map` and `reduce` functions, we cannot tune the partitioning schemes on this platform.

1) Model Verification: Because the rules obtained in Section III are fundamental to Algorithm 1, we first examine whether experiment results are consistent with some of these rules.

Block Size and Number of Partitions We test the HiBench Terasort benchmark [18] with 50GB data generated by Teragen and processed by Terasort. All the run-time parameters are tuned by MRTuner.

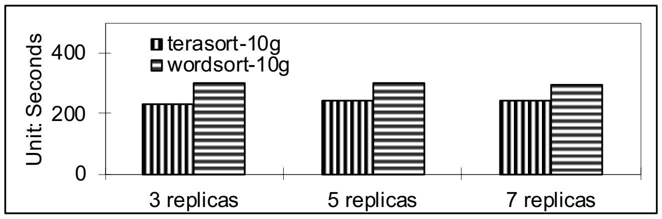
As illustrated in Fig. 4(a) captured by Ganglia [26], we observe significant network transfer overhead and CPU overhead incurred in the map setup phase, and the total elapsed time measured is 1539 seconds.

We manually tune the data placement parameters for the input data so that block size is aligned with the Terasort runtime split size (232MB) and tune the number of partitions from the default value of 96 to 38.

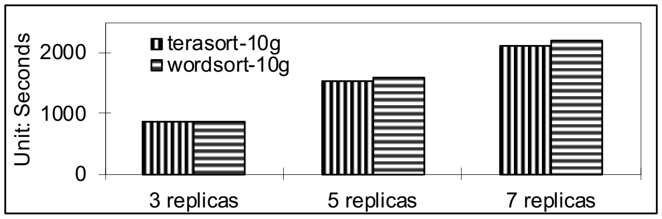
Then as shown in Fig. 4(b) captured by Ganglia, we observe significant reduction in network transfer overhead and CPU overhead at the map setup phase. The tested total elapsed time is 1096 seconds, which shows 28.8% improvement.

Above results are consistent with Deduction 2 & 3.

Replication Factor. We generate 10GB data with varying replication factors using Teragen and Wordgen respectively for



(a) Consumer performance



(b) Producer performance

Fig. 5. Replication factor's performance impacts to consumer and producer

Terasort and WordCount to process. We tune data placement parameters for both workloads, so that data are uniformly distributed and block size is aligned with split size.

As shown in Fig. 5(a), for both workloads, increase of the replication factor can not improve consumer performance. This observation is consistent with Deduction 4.

In addition, for both workloads, the data writing latency increases almost linearly with the number of replicas as shown in Fig. 5(b). This is consistent with the linear relationship between the writing overhead and replication factor, as used in our model.

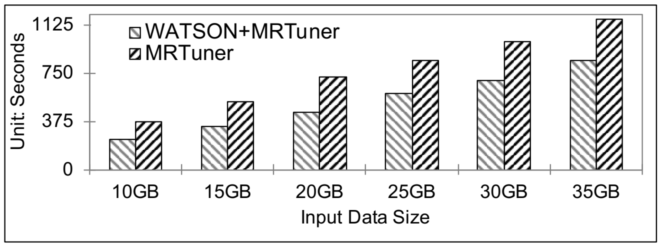
2) Performance Comparison: We also compared WATSON to MRTuner, an existing history based automatic parameter tuning tool. Because MRTuner is implemented on Hadoop v 1.1.1, so we also implemented WATSON on the same platform. We evaluated both tools using three representative benchmarks from the HiBench benchmark suite [18]: TeraSort, k-means, and NutchIndex.

We first ran these workloads in training runs, and use the Historical Workload Information Collector of WATSON to parse the Hadoop configuration files and counter files generated by those training runs and store extracted information into a SQLite database.

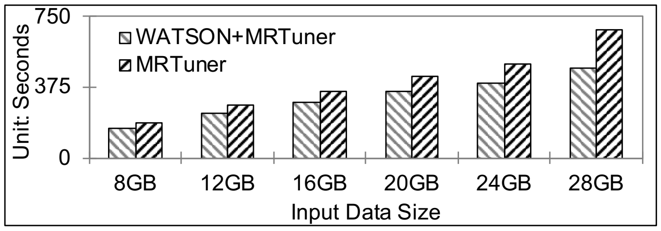
In the training step, the job configurations for all job instances are tuned for optimal performance using MRTuner [33]. TABLE II also lists the tuned split size for the consumer in each workload. The second step is the testing step, wherein for each workload, we ran it against six input datasets with varying sizes as shown in TABLE III. Among them, three datasets' sizes are exactly the same with the training datasets' sizes (denoted as Trained1, Trained2 and Trained3); three datasets' sizes are different with any of the training datasets (denoted as NotTrained1, NotTrained2 and NotTrained3).

For each dataset of each workload, we compare the consumers' performance results for using MRTuner and using both WATSON and MRTuner. For each test run where WATSON is applied, once the producer is submitted to run, WATSON APIs will be invoked immediately to predict potential consumer and consumer's behavior, to compute optimal data placement parameters, and to apply those parameters.

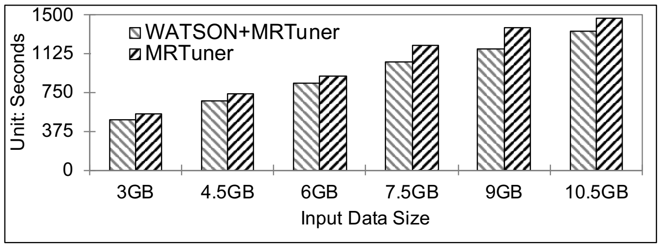
We measure the total elapsed time in each workload. The results for consumers are shown in Fig. 6. For Terasort, KMeans, and NutchIndex, WATSON can reduce the total elapsed time by up to 37%, 30%, and 15%, respectively.



(a) Terasort



(b) KMeans



(c) NutchIndex

Fig. 6. Performance comparison with MRTuner on the Hadoop platform

C. PlinyCompute Platform

PlinyCompute is a recent distributed object-oriented relational system [38]. We configure it to use eight long-living processes to compute and process input splits, which match the number of CPU cores. Different with Hadoop's memory architecture that uses thread-local buffers, PlinyCompute deploys a cache that is shared by all processes on each worker, which caches both input splits and output data in memory. Therefore, we use Eq. 11 to estimate the split size. In addition, it also provides a lambda calculus DSL, which can be used to describe UDFs, so that UDFs are not opaque to the system any more. For example, utilizing lambda calculus DSL, a join consumer that has its join selection UDF returning "*employee.getSupervisorName() == supervisor.getName()*" will be compiled to a tree of lambda terms, so that WATSON can easily traverse the tree to understand the desired partitioning schemes and extract partitioning functions for the employee dataset, and the supervisor dataset, respectively. In the above example, the desired partition key

TABLE II
INPUT DATA SIZE AND CONSUMER JOB SPLIT SIZE FOR TRAINING STEP

Benchmark	Dataset1 Size	Dataset2 Size	Dataset3 Size	Tuned Split Size
Terasort	10GB	20GB	30GB	232MB
KMeans	8GB	16GB	24GB	211MB
NutchIndex	3GB	6GB	9GB	146MB
Bayesian	10GB	20GB	30GB	4MB

TABLE III
INPUT DATA SIZE FOR TESTING STEP

Benchmark	Trained1	Trained2	Trained3	NotTrained1	NotTrained2	NotTrained3
Terasort	10GB	20GB	30GB	15GB	25GB	35GB
KMeans	8GB	16GB	24GB	12GB	20GB	28GB
NutchIndex	3GB	6GB	9GB	4.5GB	7.5GB	10.5GB
Bayesian	10GB	20GB	30GB	15GB	25GB	35GB

extraction function for hash-partitioning the supervisor dataset is supervisor.getName(), and for the employee dataset is employee.getSupervisorName(). Then in each node, the employee objects and supervisor objects having the same values of join keys are co-located together and a distributed join can be simplified as local joins on each node, which avoids significant amount of the re-partitioning/shuffling overhead. With the IR capability of PlinyCompute, we can tune the partitioning schemes for different workloads.

On the PlinyCompute platform, we selected four representative applications: matrix multiplication, sparse matrix multiplication, an iterative PageRank analytics flow, and a dataware house applications with TPC-H queries. All of these applications extensively involve join operations which can benefit from tuning of partitioning schemes. For these applications, we mainly compare the consumers' performance of using WATSON to the case of using all default data placement parameters of round-robin partitioning, 256MB block size and split size, and 3 replicas.

To test these workflows with WATSON, we first ran each workflow once to form history, which is stored in WATSON's historical information collector as described in Sec. IV-B.

1) Dense Matrix Multiplication: Matrix multiplication is a computational intensive application, which means to multiply even small sizes of matrices may take significant computational costs. For example, if distribute a $1,000 \times 1,000,000$ matrix (stored as set of $1,000$ $1,000 \times 1,000$ blocks) to ten workers, the data on each worker is merely hundreds of megabytes and cannot fully utilize the memory resources. In addition, there are only four blocks on each worker that needs to be computed by eight processes, so four processes will be idle and cannot fully utilize the CPU resources. By simply tuning the aligned block and split size to around 100MB to guarantee that every process has at least one split to process, we can observe about $1.2\times$ performance speedup for dense matrix multiplication of a $1,000 \times 1,000,000$ matrix and a $1,000,000 \times 1,000$ matrix.

In addition, the distributed matrix multiplication involves an expensive join operation, as illustrated in Fig. 2. Therefore we observe significant performance speedup when multiply a $1,000 \times m$ matrix and a $m \times 1,000$ matrix with varying m from $1,000,000$ to $10,000,000$ with WATSON applied to optimize the partitioning scheme as well as other parameters. The results are illustrated in Fig. 7. Due to the significant fluctuations of latency measured in AWS, normalized latency is used here.

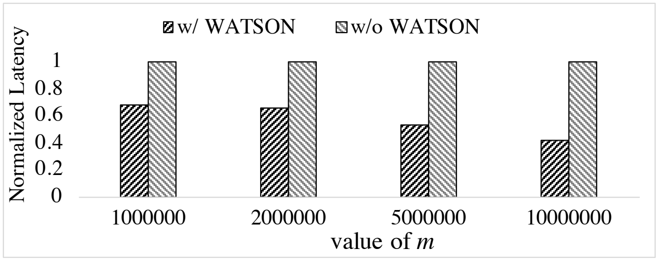
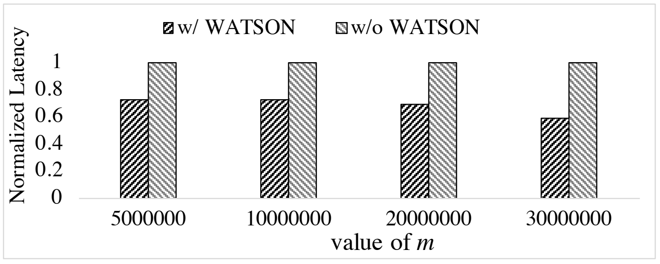
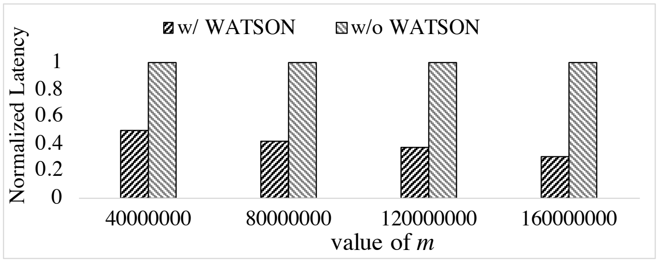


Fig. 7. Performance comparison of dense matrix multiplication.



(a) sparsity=0.001



(b) sparsity=0.000001

Fig. 8. Performance comparison of sparse matrix multiplication.

2) Sparse matrix multiplication: In this application, we first created a sparse matrix as a set of $1,000$ by $1,000$ sparse matrix blocks represented in compressed sparse row (CSR) format using Intel MKL library. Similar with the dense matrix multiplication application, the first matrix is in the shape of

$1,000 \times m$ and the second matrix is in the shape of $m \times 1,000$. We ran the sparse matrix multiplication operator with varying value of m and sparsity. The results are illustrated in Fig. 8(a).

3) PageRank: PageRank is a graph analytics algorithm that assigns a weight to each of a linked set of documents, such as web pages to measure and rank the importance of pages. The producer randomly generates a varying number of web pages and each page has five links in average with a probability of 0.85 for user continuing to click a link at each step (damping factor). We use five iterations for each test. The performance speedups are illustrated in Fig. 9.

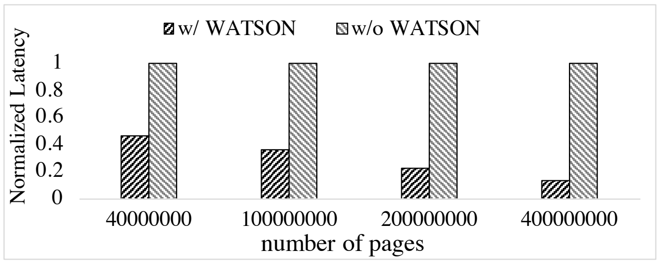


Fig. 9. Performance comparison of PageRank.

4) TPC-H Queries: We tested WATSON using TPC-H data warehouse workload that has 40GB randomly generated data including seven linked tables for storing information related with lineitems, orders, suppliers, customers, parts, partsupp, nations and so on. We ran ten TPC-H standard queries and the results are illustrated in Fig. 10.

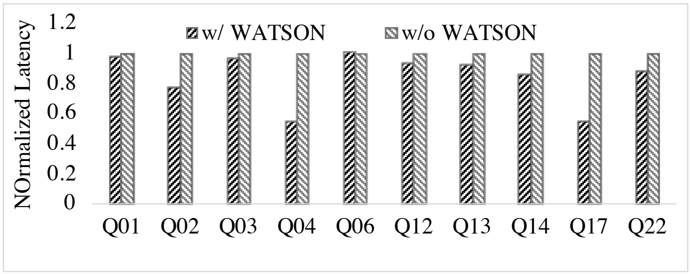


Fig. 10. Performance evaluation of TPC-H queries.

D. Further Discussion

Compared to PlinyCompute, WATSON achieves less performance gain on Hadoop, mainly because of two reasons. First, due to the lack of an IR or similar mechanisms to reason about UDFs on Hadoop, the implementation of WATSON on Hadoop, as well as MRTuner, can not tune the partitioning scheme, which is fully tuned in PlinyCompute and contributes to a major portion of performance gain achieved for join workloads on PlinyCompute. Second, on Hadoop, we are comparing to MRTuner that has tuned related parameters, such as the input split size, for each workload. Above results show that on Hadoop, WATSON can work as a complement to existing workload-based parameter tuning tool such as MRTuner and similar tools to further optimize storage parameters for workflows, where producer workloads write data to storage, which is then processed by consumer workloads.

E. Spark Platform

While we did not implement WATSON for Spark, we manually tuned the block size on Spark following the Aligned Split and Block Size Rules to demonstrate the potential of our approach on this platform. We first computed the optimal input split size by dividing the total available memory to store the output data by the number of slots, as described in Alg. 1 for a Spark job stage, and then set the block size to be consistent with the split size. We implemented a Top-k query workload on denormalized TPC-H data on Spark, where each customer object contains a list of order objects, each order object contains a list of lineitem objects, and each lineitem object contains a part object and a supplier object. Given a list of parts, the Top-k query returns the k closest customers whose list of ordered parts are the most similar to the given list.

We deployed Spark on the same eleven-node cluster of PlinyCompute, as mentioned in in Sec. V-A. For Spark, we tuned the number of cores to eight, and set the executor memory size to 50 gigabytes.

We generated 9.6 millions of denormalized customer objects, which is about 160GB data size in total. Additionally, we configured k to be between 1,024 and 10,240. We compared two cases: in the first case, we use default HDFS block size of 128MB; and in the second case, we compute and configure the optimal split size to be 2GB. The optimal split size is computed as 2GB, because for Top-k job, the output data (k top elements) is much smaller in size than input data. Therefore, using large partition size can fully utilize the memory and significantly reduce the number of tasks to avoid the overhead for launching unnecessary tasks. We configured the optimal split size by setting ”spark.hadoop.mapreduce.input.fileinputformat.split.minsize= 2147483648” and ”spark.hadoop.mapreduce.input. fileinputformat.split.maxsize=2147483648” in the Spark job configuration. We also configured the block size correspondingly. As a result, the end-to-end latency is reduced from 181 seconds to 59 seconds, which is three times speedup for this workload.

This result demonstrates that automatically tuning of data placement parameters in Spark is promising, which we will explore in more detail in the future.

VI. Related Works

A. MapReduce Auto-Tuner and Workflow Optimizer

MRTuner [33] and What-if Engine [16], are parameter auto-tuning tools for single MapReduce job, and are not considering the inter-job producer-consumer relationship.

Starfish [17] provides a vision about workflow-level optimization that mentions producer-consumer relationship, as well as a simple discussion regarding data layout optimization. However, their solution is based on What-if Engine [16], which is a cost model for a single job. As a result, they overlook many important inter-job trade-offs such as number of partitions and the alignment of block size and split size, which are the two main factors for the performance speedup achieved in this paper.

Other workflow optimizers like Stubby [24] and YSmart [21] are targeting at optimization of job execution plan, and are not tuning data placement parameters. In addition, their cost model or rules are focused on performance comparison of different job execution plans, and can not describe the trade-offs caused by data placement parameters in an inter-job WORM scenario.

Recent works apply reinforcement learning to the parameter tuning in relational database, such as CDBTune [37] and QTune [22]. However the tuning of data placement parameters in standalone relational database systems is very different to Big Data frameworks built on top of distributed file systems.

B. Performance Improvement for WORM Scenario

Tachyon [23] could be used to improve performance of inter-job WORM scenario, however it proposes fundamental changes to MapReduce read/write principles, while our work is based on the cases where MapReduce directly reads from and writes to block-based distributed file system. In addition, our work enables a mathematical method to automatically tune data placement parameters and achieve performance improvement independent of job execution sequence.

CoHadoop [12] can improve the performance for the inter-job WORM scenario that includes a consumer doing join-like operations, by co-locating datasets with the same join key. HadoopDB [3] replaces HDFS using relational databases. Then it allows user to specify the partition key for each table to partition data at loading time. Hadoop++ [11] proposes a co-partitioned join operator called Trojan join and assumes application programmer understand the data schema and workloads and trust them to use Trojan join properly.

Different with these works, WATSON is not limited to join-like consumers, because we are focused on data placement parameters which are generic to any inter-job WORM scenario. In addition, we provide an automatic parameter tuning method, while CoHadoop relies on information provided by human.

C. Physical Database Design for Relational Databases.

There are a lot of works that recommonds physical database design schemes, including partitioning schemes, indexing schemes, materialization schemes and so on, for relational database, including IBM DB2 Partition Advisor [31], Microsoft's AutoAdmin for SQLServer [4] MESA [28], Legobase [32], AdaptDB [25], Schism [10], Sword [20], Horticulture [29], etc.. However, these works are not designed for the Big Data analytics and NoSQL data, and cannot be applied to tune data placement parameters on distributed file systems.

VII. Conclusion

This paper presents WATSON, a novel cross-layer solution to address the auto-tuning problem of storage parameters for the inter-job WORM scenario. It employs a novel Producer-Consumer model to describe the inter-job performance trade-offs caused by data placement parameters. Based on this model, it then takes an efficient greedy strategy to greatly reduce the search space for optimal data placement parameters, by utilizing the rules derived from the model. In addition, WATSON also includes a workload prediction mechanism to help bridge the gap between storage and computation and to inform the storage writing process with workload-level information. The experimental evaluation demonstrates the effectiveness of WATSON.

VIII. Acknowledgements

Thanks to ASU graduate students Pradeep Jampani and Srihari Jayakumar for helping with the implementation of the matrix multiplication benchmark, and Rice graduate student Dimitrije Jankov for helping with the implementation of the page rank benchmark. Thanks to Dr. Tongping Liu for his insightful feedbacks on an earlier version of this paper. This work has been supported by the DARPA MUSE award No. FA8750-14-2-0270, AWS Cloud Credits for Research program, National Science Foundation CAREER award CNS-1619653 and awards CNS-1562837 and CNS-1629888.

Appendix

The *consistent partitioning scheme rule* is quite intuitive (when $g_i = f$, $O_{repartition_i} = 0$), so we omit the proof here and focus on the remaining three rules. Because here we only consider a particular consumer, we use $consumer$ to replace $consumer_i$, and split size s to replace s_i.

A. **Proof of the Block and Split Alignment Rule**

$O_{consumer}$ can be minimized only if block size is equal to split size (i.e. $s = B$).

Proof:
Suppose input split for the i-th mapper has m subsplits, each of them on a different node. We assume that SS_i is a reverse-order array that contains sizes of all subsplits, $SS_i = \{ss_{i_0}, ..., ss_{i_{m-1}}, 0\}$, where ss_{i_0} is the largest size. We further denote the probability that the i-th task is scheduled to the node having the j-th subsplit ss_{i_j} as q_{ij}. For all situations that this task is scheduled to a node without any subsplit, we denote them as ss_{i_m}, with a probability q_{im}, and all data (S) has to be copied from other nodes. When this task is scheduled to a node with a subsplit size ss_{i_j}, with the probability q_{ij}, other subsplits (with a total size $s - ss_{i_j}$) have to be copied to this node. Thus, for the i-th map task, p_i–the expected percentage of data that has to be transferred from remote nodes–can be illustrated as EQ.(12):

$$p_i = \frac{\sum_{j=0}^{m} (s - ss_{i_j}) \cdot q_{ij}}{s} \quad \text{where} \sum_{j=0}^{m} q_{ij} = 1 \tag{12}$$

If block size is equal to input split size, then we will have $s = B$. Since a block is the smallest unit for data storage and a block can only be at one node, there is only one subsplit, with $m = 1$ and $SS_i = \{s, 0\}$. Based on EQ(12), p_i can be simplified as EQ(13), where we add single quote mark to p_i

and q_{i1} to distinguish those symbols with their counterparts in EQ(12).

$$p'_i = \frac{s \cdot q'_{i1}}{s} = q'_{i1} \tag{13}$$

Based on our assumption of the FIFO scheduling policy, we will have the following two relations:

- **Relation 1:** Since subsplits are transparent to the current scheduler, the probability that a task will be scheduled on the split with the largest size will be the same, which means $q'_{i0} = q_{i0}$. Thus, when block size is equal to input split size, we will have a relation described as EQ(14), based on the the condition of EQ(12).

$$q'_{i1} = 1 - q'_{i0} = 1 - q_{i0} = \sum_{j=1}^{m} q_{ij} \tag{14}$$

- **Relation 2:** Since Hadoop defaulted FIFO scheduler at first tries to schedule a task to the node with the largest subsplit, we will have $q_{i0} \geq q_{ij}$(if $j > 0$).

Based on these two relations, we will have $p_i \geq p'_i$ in the end. To prove this, at first we expand EQ(12) as follows.

$$p_i = \frac{(s - ss_{i_0})}{s} \cdot q_{i0} + \frac{(s - ss_{i_1})}{s} \cdot q_{i1} + ... \frac{(S - ss_{i_m})}{S} \cdot q_{im} \tag{15}$$

When we combine EQ(13) and EQ(14) in Relation 1, we have $p'_i = q'_{i1} = q_{i1} + q_{i2} + ... + q_{im}$.

Thus, the subtraction of p_i and p'_i can be shown as follows.

$$p_i - p'_i = \frac{(s - ss_{i_0})}{s} \cdot q_{i0} - \frac{ss_{i_1}}{s} \cdot q_{i1} - ... - \frac{ss_{i_m}}{s} \cdot q_{im} \tag{16}$$

Because $\frac{(s-ss_{i_0})}{S} \cdot q_{i0}$ can be expressed as $\frac{ss_{i_1}+ss_{i_2}+...+ss_{i_m}}{s} \cdot q_{i0}$, EQ.(16) can be transfromed to the following one.

$$p_i - p'_i = \frac{ss_{i_1}}{s} \cdot (q_{i0} - q_{i1}) + \frac{ss_{0_2}}{s} \cdot (q_{i0} - q_{i2}) + ... + \frac{ss_{0_m}}{s} \cdot (q_{i0} - q_{im}) \tag{17}$$

Based on Relation 2, since every term of EQ.(17) is larger than 0, thus we have $p_i - p'_i \geq 0$.

Based on EQ.(8), we have $O_{input} - O'_{input} \geq 0$, where O'_{input} represents data transfer overhead of consumers when block size is equal to split size, i.e. $s = B$.

Now **Rule 2** has been proved successfully.

B. Proof of the Proportional Data Distribution Rule

$O_{consumer}$ can be minimized only if size of data stored in each node is proportional to the number of slots in each node. (i.e. $\forall$node $j, ns_j/ns = nb_j/nb$, where nb represents total number of blocks and nb_j represents number of blocks on the j-th node).

Proof.
We first consider the case when block size is equal to split size, i.e. $s = B$. We let q_{i0} denote the probability of scheduling the i-th task to the node storing the input split. Because we assume the scheduler will try to schedule a task to a data local node having available slot first, we model q_{i0} as the ratio of the total number of tasks that have their splits co-located with the the i-th task's split, to the total number of available slots through all waves of map tasks.

Because replication factor can improve the data locality, we consider the lower-bound case where replication factor is one. We use $y(i)$ to represent the host of the input split data for the i-th task. In addition, we use $ns_{y(i)}$ to represent the number of slots on the node storing the input split for the i-th task, and $nb_{y(i)}$ to represent the number of blocks on the same node. We use n to denote the total number of nodes in the cluster.

Then, q_{i0} can be represented as $q_{i0} = \min\{\frac{ns_{y(i)} \cdot nw}{nb_{y(i)}}, 1\}$, where nw represents the number of waves for map tasks.

To maximize the average data-local probability for all tasks, we have $q = \overline{q_{i0}}$ as shown in EQ. 18.

$$\begin{aligned} &\text{maximize } \overline{q_{i0}} \ (0 \leq i \leq \frac{S}{s}) \\ &\text{where } \ q_{i0} = \min\{\frac{ns_{y(i)} \cdot nw}{nb_{y(i)}}, 1\} \end{aligned} \tag{18}$$

It is easy to see that if $\forall$node $j, ns_j/ns = nb_j/nb$, then there will be $q_{i0} = \min\{\frac{ns}{nb} \cdot nw, 1\}$. The total number of map tasks can be expressed as $nm = \frac{S}{s}$. Then nw can be represented as $nw = \lceil \frac{nm}{ns} \rceil = \lceil \frac{\frac{S}{s}}{ns} \rceil$. Because $s = B$, there will be $nw = \lceil \frac{\frac{S}{B}}{ns} \rceil = \lceil \frac{nb}{ns} \rceil$.

If number of blocks is larger than number of slots ($nb > ns$), then $q_{i0} = \min\{\frac{ns}{nb} \cdot \lceil \frac{nb}{ns} \rceil, 1\} = 1$. Otherwise, $nw = 1$, then $\frac{ns}{nb} \geq 1$, and there will be $q_{i0} = \min\{\frac{ns}{nb}, 1\} = 1$.

Therefore, Rule 3 is proved for the case with $s = B$.

Because subsplit is transparent to scheduler, we can extend above conclusion to the case when $s \neq B$. We omit the proof for the case due to limit of space.

C. Proof of the Invariant Replication Factor Rule

If block size is aligned with split size and data is distributed proportionally to slot number for each node in the cluster (i.e. $s = B \ and \ \forall$node $j, \ ns_j/ns = nb_j/nb$), there will be $O_{consumer} = 0$, regardless with the replication factor.

Proof. When $s = B$ and $\forall$node $j, ns_j/ns = nb_j/nb$, there will be $\forall$task $i, \ q_{i0} = 1$, then based on EQ. 8 and EQ. 12, we can obtain $O_{consumer} = 0$.

Therefore, Rule 1 is proved.

However, when block size is not aligned with split size, a split consists of multiple subsplits with each subsplit being from a different block. Increasing replication factor can increase the expected number of blocks which are co-located on the node executing this task.

For such case, the probability of the j-th subsplit of the i-th task to be scheduled in a data-local node, denoted by u_{ij}, can be approximated as the probability that it has at least one replica co-located with the host, which will be $\frac{r}{n}$. Then the portion of data need to be transferred for the i-th task can be formulated as in EQ. 19.

$$p_i = \sum_{j=1}^{m-1} ss_{ij} \cdot (1 - u_{ij}) = (s - ss_{i0}) \cdot (1 - \frac{r}{n}) \qquad (19)$$

References

[1] Yahoo cloud trace. https://webscope.sandbox.yahoo.com/catalog.php?datatype=s.

[2] Hdfs architecture guide. 2013. http://hadoop.apache.org/docs/stable/hdfs_design.html.

[3] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.

[4] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 359–370. ACM, 2004.

[5] Amazon. Amazon simple storage service developer guide. http://docs.aws.amazon.com/AmazonS3/latest/dev/s3-dg.pdf, March 2006.

[6] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[7] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[8] W. Chen, M. Kifer, and D. S. Warren. Hilog: A foundation for higher-order logic programming. *The Journal of Logic Programming*, 15(3):187–230, 1993.

[9] Y. Chen and et al. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proceedings of the VLDB Endowment*, 5(12):1802–1813, 2012.

[10] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010.

[11] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). *Proceedings of the VLDB Endowment*, 3(1-2):515–529, 2010.

[12] M. Eltabakh and et al. Cohadoop: flexible data placement and its exploitation in hadoop. *Proceedings of the VLDB Endowment*, 4(9):575–585, 2011.

[13] S. Fortin. The graph isomorphism problem. 1996.

[14] S. Ghemawat and et al. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[15] K. Gupta and et al. Gpfs-snc: An enterprise storage framework for virtual-machine clouds. *IBM Journal of Research and Development*, 55(6):2–1, 2011.

[16] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proc. of the VLDB Endowment*, 4(11):1111–1122, 2011.

[17] H. Herodotou and et al. Starfish: A self-tuning system for big data analytics. In *CIDR*, volume 11, pages 261–272, 2011.

[18] S. Huang and et al. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *ICDEW*, pages 41–51, 2010.

[19] A. Jindal, S. Qiao, H. Patel, Z. Yin, J. Di, M. Bag, M. Friedman, Y. Lin, K. Karanasos, and S. Rao. Computation reuse in analytics job service at microsoft. In *Proceedings of the 2018 International Conference on Management of Data*, pages 191–203. ACM, 2018.

[20] K. A. Kumar, A. Quamar, A. Deshpande, and S. Khuller. Sword: workload-aware data placement and replica selection for cloud data management systems. *The VLDB Journal*, 23(6):845–870, 2014.

[21] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang. Ysmart: Yet another sql-to-mapreduce translator. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 25–36. IEEE, 2011.

[22] G. Li, X. Zhou, S. Li, and B. Gao. Qtune: a query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment*, 12(12):2118–2130, 2019.

[23] H. Li and et al. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *SOCC*, pages 1–15. ACM, 2014.

[24] H. Lim, H. Herodotou, and S. Babu. Stubby: A transformation-based optimizer for mapreduce workflows. *Proceedings of the VLDB Endowment*, 5(11):1196–1207, 2012.

[25] Y. Lu, A. Shanbhag, A. Jindal, and S. Madden. Adaptdb: adaptive partitioning for distributed joins. *Proceedings of the VLDB Endowment*, 10(5):589–600, 2017.

[26] M. Massie and et al. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[27] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of logic and computation*, 1(4):497–536, 1991.

[28] R. Nehme and N. Bruno. Automated partitioning design in parallel database systems. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1137–1148. ACM, 2011.

[29] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 61–72. ACM, 2012.

[30] Rackspace. Welcome to swift's documentation. http://docs.openstack.org/developer/swift/, 2014.

[31] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 558–569. ACM, 2002.

[32] A. Shaikhha, Y. Klonatos, and C. Koch. Building efficient query engines in a high-level language. *ACM Transactions on Database Systems (TODS)*, 43(1):4, 2018.

[33] J. Shi, J. Zou, J. Lu, Z. Cao, S. Li, and C. Wang. Mrtuner: a toolkit to enable holistic optimization for mapreduce jobs. *Proceedings of the VLDB Endowment*, 7(13):1319–1330, 2014.

[34] A. Thusoo and et al. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[35] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2012.

[36] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*. USENIX Association, 2010.

[37] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, pages 415–432. ACM, 2019.

[38] J. Zou, R. M. Barnett, T. Lorido-Botran, S. Luo, C. Monroy, S. Sikdar, K. Teymourian, B. Yuan, and C. Jermaine. Plinycompute: A platform for high-performance, distributed, data-intensive tool development. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1189–1204. ACM, 2018.

[39] J. Zou, A. Iyengar, and C. Jermaine. Pangea: Monolithic distributed storage for data analytics. *PVLDB*, 12(6):681–694, 2019.

[40] J. Zou, A. Iyengar, and C. Jermaine. Architecture of a distributed storage that combines file system, memory and computation in a single layer. *The VLDB Journal*, pages 1–25, 2020.

[41] J. Zou, J. Shi, T. Liu, Z. Cao, and C. Wang. Foreseer: Workload-aware data storage for mapreduce. In *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*, pages 746–747. IEEE, 2015.

[42] J. Zou, J. Xiao, R. Hou, and Y. Wang. Frequent instruction sequential pattern mining in hardware sample data. In *2010 IEEE International Conference on Data Mining*, pages 1205–1210. IEEE, 2010.