Data-intensive Scheduling

by

Xiao Meng

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2019

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

In many modern data management scenarios, we encounter tasks, operations or computational phases that are data-intensive where the sheer volume of data proves to be overwhelming to handle and becomes a performance bottleneck. For data-intensive tasks, the bottleneck is data loading, where the cost of loading data into memory is more significant than the cost of actual computation. For data-intensive shuffling, the bottleneck is data transfer, where intermediate data are scattered and shuffled for further processing. This thesis addresses two data-intensive scheduling problems: (1) multiprocessor scheduling for data-intensive tasks to reduce redundant data loading; (2) reducer scheduling for data-intensive shuffling to reduce redundant data communication.

For data-intensive tasks, we focus on workloads with precedence constraints of data dependencies, which are common in various applications such as data analytics and ETL processing. These workloads are often known in advance, are presented as directed acyclic graphs (DAG), and are data-intensive and sensitive to cache misses. We solve the problem of scheduling DAGs of data-intensive tasks on multiple processors or machines, in order to minimize execution time. To do so, we propose scheduling algorithms that take cache misses into account. Simulations and an experimental evaluation using a Spark cluster demonstrate the advantages of our solutions in terms of workload completion time.

For data-intensive shuffling, we focus on MapReduce-style processing. Communication overhead is incurred in the Shuffle stage which sends intermediate results from mappers to reducers. We solve this problem: given a collection of mapper outputs (intermediate key-value pairs) and a partitioning of this collection among the reducers, which node should each reducer run on to minimize data transfer? We reduce two natural formulations of this problem to optimization problems for which polynomial solutions exist. We show that our techniques can cut communication costs by 50 percent or more compared to Hadoop's default reducer placement, which leads to lower network utilization and faster MapReduce job runtimes.

Acknowledgements

First and foremost, I would like to express my sincerest gratitude to my advisor Prof. Lukasz Golab, who has offered me unwavering support at every step of this journey. His great advice on both research and career has given me invaluable guidance for the future.

I also want to thank my thesis readers for sharing their exceptional research experience and giving expertise feedback on my thesis.

Last but not least, none of these could have happened without unfailing support from my wonderful friends and family. I would like to acknowledge them for inspiring and helping me throughout my graduate studies in Waterloo, the good times and memories we have shared together, and their continued support in the years to come.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

In the era of Big Data, many modern data management scenarios consist of tasks, operations and phases that are data-intensive. By data-intensive, we mean that for such tasks or operations, the sheer volume of data proves to be overwhelming to handle and becomes a performance bottleneck. Data-intensive operations can be of different granularities and levels in a modern data processing paradigm, and in this thesis we study two important aspects of them.

One important aspect is data-intensive tasks. These tasks can be very costly and commonly observed in OLAP workload, ETL process and hybrid transactional and analytical processing [51, 64]. The major bottleneck here is data loading, where the cost of loading data into memory is more significant than the cost of actual computation. More importantly, these tasks often emerge with data dependencies and precedence constraints, making them more complex to manage. Therefore, scheduling such DAGs of tasks focuses on reducing redundant data loading.

Now we consider data-intensive scheduling from a higher level of data processing frameworks and paradigms. Take MapReduce-style system for example, they have become the standard for big data processing in distributed environments. Flowing through a MapReduce-style paradigm, data shuffling becomes the most data-intensive operational phase in such modern frameworks, where intermediate data are scattered among geographically distributed servers and need to be shuffled to the corresponding servers for further processing (e.g. Reduce function). This data shuffling can lead to network

congestion and poor performance. Therefore, scheduling of data-intensive shuffling aims at reducing data communication.

With these problems in mind, this thesis addresses the scheduling problems in data-intensive task as well as data-intensive shuffling, with the goal to alleviate the impact of big data. Next we introduce and motivate the two problems separately in more details.

1.1.1 Multi-processor Scheduling of Data-intensive Tasks

In modern data management scenarios, we encounter tasks that are data-intensive, which means that their data loading cost (i.e., time spent to bring data into memory) is higher than the subsequent computation cost (i.e., time spent for the actual processing). This has been observed and documented in the literature in recent years [51, 64] for OLAP workloads, ETL processes, and hybrid transactional and analytical processing.

For a concrete example, we present the results of a Spark SQL test for cold and hot runs of three queries (we describe the experimental setup in detail in Section 2.4). For cold runs, a data-intensive query directly reads base tables, where data has to be loaded from disk (HDFS) into memory; for hot runs, the same query works on materialized views (RDD) that are just created and loaded, so hot runs have the required data in memory. Of the three queries, Q1 is a COUNT(*) query, Q2 is a INSERT INTO query to fetch data into a new table, and Q3 is a more complex data-intensive query Q21 from TPC-DS (shown in the snippet below), all using a TPC-DS benchmark dataset. Q3 computes the percentage change in inventory in a specified time period for items whose price was changed on a given date. As shown in Table 1.1, the hot vs. cold gap can range from around 10x to 700x. This performance gap can be attributed to disk I/O and de-serialization [58], highlighting the data-intensive nature of data analytics tasks.

Listing 1.1: Snippet: TPC-DS Query Q21

```
select * from
(select w_warehouse_name, i_item_id,
    sum(case when (cast(d_date as date) <
    cast ('1998-04-08' as date))
    then inv_quantity_on_hand else 0 end) as inv_before,
    sum(case when (cast(d_date as date) >=
    cast ('1998-04-08' as date))
    then inv_quantity_on_hand else 0 end) as inv_after
    from inventory, warehouse, item, date_dim
```

```
    where i_current_price between 0.99 and 1.49
    and i_item_sk = inv_item_sk
    and inv_warehouse_sk = w_warehouse_sk
    and inv_date_sk = d_date_sk
    and d_date between date1 and date2
    group by w_warehouse_name, i_item_id) x
where (case when inv_before > 0
    then inv_after / inv_before else null end)
    between 2.0/3.0 and 3.0/2.0
order by w_warehouse_name, i_item_id;
```

Queries	Hot	Cold
Q1	*210.866s*	*0.358s*
Q2	*324.412s*	*35.682s*
Q3	*705.451s*	*62.124s*

Table 1.1: Data-intensive queries: hot vs. cold runtimes.

Moreover, we often encounter workloads consisting of a set of data-intensive tasks that are known in advance. For instance in Extract-Transform-Load (ETL) pipelines, before loading a dataset into database, data passes through a predefined workflow of operations for data cleansing and normalization. Another example is in data warehousing and data analytics, where view maintenance involves updating a hierarchy of views which are defined beforehand.

The above use cases follow a pattern: new data arrives and needs to be ingested periodically, and a predefined set of operations are re-executed to process the new batch of data (e.g., to refresh materialized views or to pre-process raw data). It is critical to finish these DAGs (directed acyclic graphs) of queries/tasks as soon as possible so that the system will be able to accommodate the next batch of data. Sequencing such tasks then becomes a significant problem because some sequences may incur more cache misses than others, leading to a longer execution time[1]. Therefore, by optimizing the scheduling of such data-intensive tasks, we aim to minimize completion time, which consequently translates into reducing the possibility of cache misses - we identify this problem as the bottleneck for these data-intensive workloads.

[1]In this thesis, we use the word cache to refer to different levels of in-memory storage for different use cases, not exclusively SRAM cache memory. In this case, we are referring to RAM memory for multi-core and multiprocessor servers.

Traditional scheduling strategies often assume that the execution times (or estimates) of tasks are known in advance. However, in our problem, it is not practical to assume the knowledge of execution times. In modern data centers with multiple tenants and shared resource allocation, it may not be possible to determine what is in the cache at any given moment in time. Instead, in this work, we only assume a LRU-based caching strategy where the longer the wait, the slimmer the chance of data remaining in the cache, but we do not assume the knowledge of execution times. Therefore, it becomes infeasible to apply existing multiprocessor DAG scheduling strategies that require task runtimes.

Furthermore, the availability of multi-core processors in modern data analytics and ETL systems also present challenges, aside from the computation capacity boost. In this setting, each core can take up one task, with shared memory and CPU cache (L2 and above) available. Therefore, how to load balance among the tasks becomes a problem, and it is not obvious which ordering of the tasks among the cores will strike a balance between load and locality to reduce the completion time of the workload.

There has been previous work on data-intensive scheduling to produce a serial schedule for a single thread or machine [8]. We call this problem Serial Data-Intensive Scheduling (SDIS). Given a DAG of tasks with data dependencies, SDIS finds an ordering of the tasks that obeys the precedence constraints given by the DAG and reduces the possibility of cache misses. However, it remains unanswered how to solve this scheduling problem in the modern distributed multi-core setting since modern data management infrastructure is not serial. That is the problem we address in this thesis - the Multi-Processor Data-Intensive Scheduling (MPDIS) problem for a DAG of data-intensive tasks. The added complexity comes from (1) the additional search space for a schedule because of additional processors/cores; (2) the load balancing requirement that is omitted from SDIS.

Example: Consider the DAG of tasks in Figure 1.1, with edges showing data dependencies (for example, the data output of task zero becomes the data input to task 3). Assume each task produces an output of unit size. Suppose for each of the six tasks, the computation cost (hot runtime) is one time unit while the loading cost of one data unit is ten time units. Assume the cache can hold up to two data items at the same time. Figure 1.2 shows two possible schedules for two processors (PU1 and PU2) and plots them on a time axis. The figure also shows the contents of the cache at various points in time.

For both schedules, there will be cache misses for items 0 and 1 since the cache is initially empty. For the first schedule, S1, at timestamp 11, tasks 0 and 1 finish, and both of their outputs are in the cache. Task 2 is started on processor 2. Task 3 has to wait for task 2 to finish because its input is the output of task 2. Task 3 takes one time unit

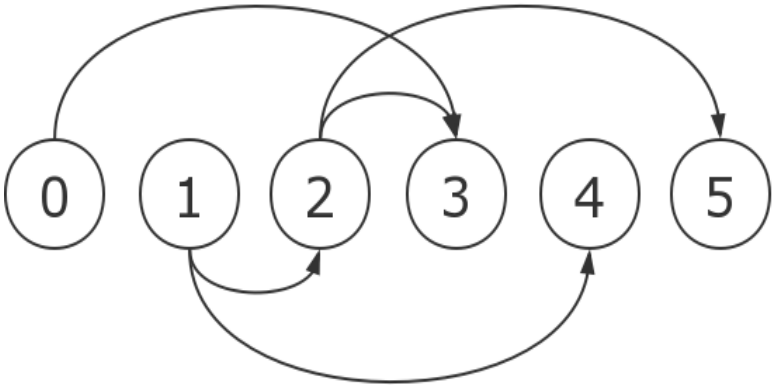


Figure 1.1: Example DAG 1.

because the data needed for task 2, namely the data output of task 1, is in the cache. At timestamp 12, when task 2 is finished, task 3 and task 5 are started. The output of task 2 is in the cache, evicting the output of task 0 according to the LRU policy. Thus, now the cache holds the outputs of tasks 1 and 2. This means that task 3 causes a cache miss for item 0, and finishes at time 23, while task 5 finishes earlier at time 13 (because its input, which was the output of task 2, was in the cache). At this time, the cache holds the outputs of tasks 0 and 2. Thus, task 4 causes a cache miss for item 1, and therefore finishes at time 34.

In schedule S2, when tasks 0 and 1 terminate, tasks 2 and 4 can run hot because their input (the output of task 1) is in the cache. When tasks 2 and 4 are done, the cache now contains the output of task 2 (which evicts the output of 0) and the output of task 1 (note that task 4 does not produce any output for use by subsequent tasks). This means that task 5 runs hot, but task 3 incurs a cache miss because it requires the output of task 0. Note that schedule S2 incurs fewer cache misses and has a shorter completion time, highlighting the need for a scheduling strategy for data intensive workloads.

In this thesis, we present scheduling techniques for the MPDIS problem for a DAG of tasks. Assuming a LRU-style cache, the intuition behind our solutions is: the longer a data item stays in the cache, the more likely it is to be evicted; so, when a data item is fetched into the cache, we should run tasks that require this data item as soon as possible. In other words, we want to minimize the distance in the schedule (i.e., the number of other tasks scheduled) between tasks sharing the same data input.

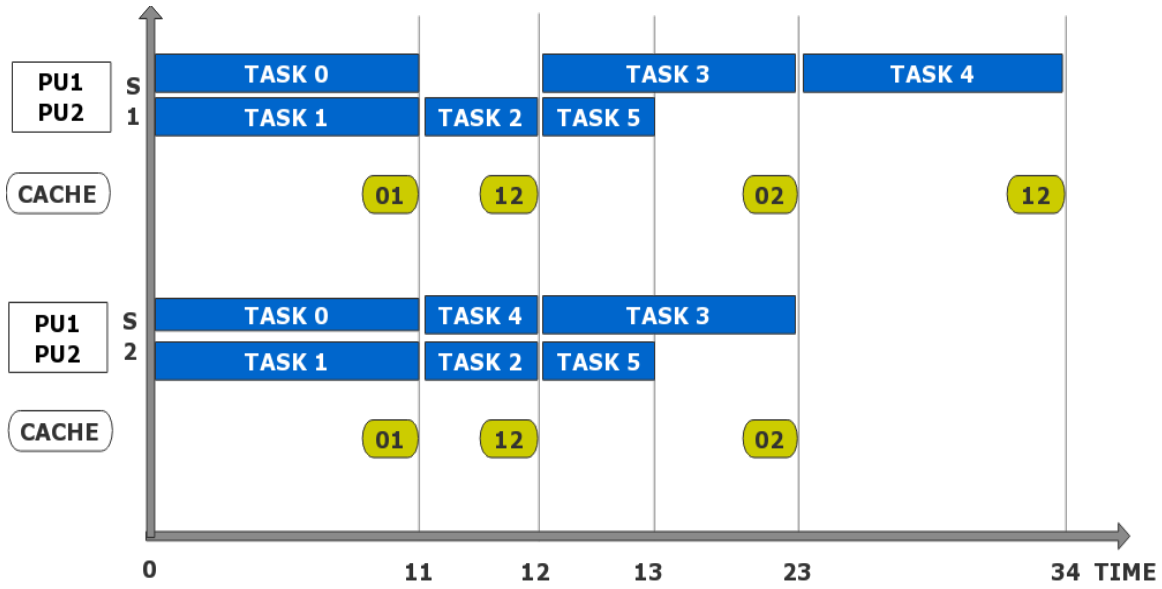


Figure 1.2: Two schedules for Example DAG 1 on 2 processors and cache states (e.g. 01 means the double-unit cache holds item 0 and 1).

1.1.2 Reducer Scheduling for Data-intensive Shuffling

With the explosion of data volume, velocity and variety, MapReduce has become the standard for big data processing in distributed environments, with Apache Hadoop [1] being the most popular implementation. Hadoop achieves parallelism by scheduling Map tasks (mappers) and Reduce tasks (reducers) over distributed data storage across a cluster of servers.

In MapReduce-style processing, the Map stage locally produces intermediate key-value pairs on each processing node. Next, a *partitioner* module assigns intermediate keys to reducers for processing, a task we call *key partitioning*. Evenly distributing intermediate results, and thus the processing load, among the reducers corresponds to the NP-hard problem of *Minimum Makespan Scheduling* [30]. Much of the previous work on Hadoop task scheduling focuses on load balancing during the Reduce stage [68, 31, 69, 25, 55, 45].

However, key-value pairs assigned to a particular reducer may be scattered across some or all processing nodes and need to be transferred to the node on which this reducer will run. Thus, *data shuffling* between the Map stage and the Reduce stage can be the bottleneck, especially in commodity computing environments with limited network

bandwidth. Moreover, in many applications, the intermediate result (total mapper output) size is at least equal to the input size [33, 37]. This leads to the following problems.

1. Network congestion. As Chowdhury et al. [19] point out, modern data centres often suffer from network oversubscription and congestion.

2. Job execution time. As CPU and GPU technologies advance, network bandwidth for data shuffling can become the bottleneck in data-intensive distributed computing [52, 7]. Analysis of a week-long trace from Facebook's Hadoop cluster containing 188,000 MapReduce jobs shows that data transfer accounts for 33% of the running time on average, and over half the running time for 26% of the jobs [19].

Thus, as data volumes increase, minimizing data transfer during the shuffle stage of MapReduce is becoming increasingly important. This is exactly the problem we study in this thesis. We assume that a key partitioning has been computed, e.g., by the Hadoop Partitioner, and we answer the following question: which node should each reducer run on to minimize data transfer? We call this problem *reducer placement*.

While key partitioning to balance load is NP-hard, it turns out that our problem, in which we compute an optimal reducer placement for a given key partitioning, is solvable in polynomial time.

1.2 Contributions

The contributions of this thesis are as follows. For multi-processor scheduling of data-intensive tasks:

1. We define the MPDIS problem of scheduling tasks in data-intensive workloads to optimize cache usage in multi-core settings.

2. We propose algorithms for solving the MPDIS problem using cache metrics from the Programming Language and Compiler Research literature.

3. We experimentally show the effectiveness of proposed algorithms against baseline algorithms using real-world based DAGs and the TPC-DS decision support benchmark.

We make the following contributions for reducer scheduling for data-intensive shuffling:

1. We introduce the problem of minimizing data transfer in MapReduce-style computation: the *reducer placement* problem. We formulate two practical versions of this problem: 1) minimizing total data transfer to reduce network congestion and 2) minimizing the maximum data transfer to any one reducer to mitigate shuffle skew.

2. We reduce both versions of our problem to optimization problems with existing polynomial-time solutions. We show that version 1) corresponds to Linear Sum Assignment and version 2) corresponds to Linear Bottleneck Assignment.

3. We empirically validate our approach on real datasets and real workloads, showing over 50 percent improvements over Hadoop's default reducer placement [63] in terms of total data transfer (network utilization). The improvement in maximum data transfer was less pronounced and much more sensitive to the key distribution and key partitioning strategy, ranging between 10 and 50 percent. This leads to a similar improvement in the runtimes of the shuffle phase.

1.3 Organization

The remainder of this thesis is organized as follows. In Chapter 2, we present our work on multi-processor scheduling for data-intensive tasks. In Section 2.1, we review related work. We formulate our scheduling problem in Section 2.2 and propose solutions in Section 2.3. We present experimental results in Section 2.4.

In Chapter 3, we present our work on reducer scheduling for data-intensive shuffling. In Section 3.1, we review the stages of MapReduce computation and related work on optimizing them. We formulate our reducer placement problems in Section 3.2 and solve them in Section 3.3. We present experimental results in Section 3.4.

We conclude this thesis in Chapter 4.

Chapter 2

Multi-processor Scheduling of Data-intensive Tasks

2.1 Related Work

DAG scheduling and multiprocessor scheduling are well-studied problems in the scheduling literature. Scheduling DAGs on a single processor (or single-thread DAG scheduling) has been studied in [17], however it does not provide a solution for minimizing the distance between related tasks that require the same data item(s), which is our target in this thesis. Multiprocessor DAG scheduling is an NP-Complete problem with only a few exceptions. Therefore, many heuristics have been proposed to strike a balance between load balancing and cache affinity. [16] compared these heuristics empirically and finds that Heterogeneous Earliest Finish Time (or HEFT) is among the best heuristics for multiprocessor DAG scheduling, so we adopted a modified HEFT as a baseline for comparison.

Scheduling with sequence-dependent setup times is a related topic, where the execution time of each task includes a certain setup time that is dependent on all the tasks that have been executed up to now. However, the MPDIS problem is more complex because the sequence of tasks executed up to now may not be sufficient to determine the contents of the cache.

Another related topic is cache metrics in the programming language/compiler literature, used as a metric to approximate cache misses for monitoring and analyzing concurrent programs: (LRU) Stack Distance (SD) [20, 11, 46] and Reference Distance (RD)

[54, 10]. Total Maximum Bandwidth (TMB) [8] has also been proposed, which is a modified version of RD. While our solutions are independent of the cache metric, we will use SD in the remainder of this thesis (details in Section 2.2).

Data-intensive scheduling has been studied in the database literature, mostly in the context of sharing scans to maximize data sharing [73, 44, 53]. For example, the cooperative scans strategy coordinates multiple scan requests to maximize I/O bandwidth reuse for queries that perform concurrent (clustered index) scans [73]; and the throttling strategy uses adaptive throttling of scan speeds of concurrent queries in order to keep scans using the same index closer together [44]. Query optimization strategies have also been proposed for this problem, such as Simultaneous Pipelining, which shares intermediate results of common sub-plans [53]. However, this line of research focuses on simple/shallow dependencies where a single base table is shared by many queries, while our solutions are designed for arbitrary DAGs.

The authors of [8] studied the SDIS problem, and proposed an optimal A* search algorithm as well as several heuristics. However, the MPDIS problem was not considered.

Scheduling in Apache Spark, on the other hand, has different considerations. Given that Resilient Distributed Datasets (RDD) are a fundamental data structure of Spark that supports in-memory processing, previous work focuses on optimizing RDDs. [66] employs fine-grained memory caching of RDD partitions; [70] considers RDD eviction policies and proposes an algorithm for multi-stage workloads to cache the most valuable intermediate datasets that can be reused in the future; [24, 48] consider RDD/cache reuse and leverage information on cached data to schedule together tasks that share data. However, these caching-related strategies do not consider dependency graphs and relationships. On the other hand, [36] mimics the classic shortest job first scheduling policy without knowing the job sizes in advance for MapReduce-style systems; [22] proposes a distributed approximation of the classic Least Attained Service (LAS) scheduling policy; and [18] tackles scheduling problems in Spark Streaming by dynamically scheduling parallel micro-batch jobs. However, these scheduling strategies do not exploit RDD caching opportunities.

2.2 Problem Definition

We consider tasks with precedence constraints corresponding to data dependencies, scheduled onto n processors. Precedence constraints are presented in the form of a directed acyclic graph (DAG) G = (V, E), where each node v $\in$ V represents a task and each

directed edge e = (u, v) $\in$ E represents a precedence constraint. An edge in the DAG denotes that the data output of one task is the data input to another. A precedence constraint of (u, v) requires that task u has to be scheduled before task v. Optional input could include the size of the data output of each task. In addition to fulfilling the precedence constraints represented in the DAG, we will impose optimization goals on the generated ordering to avoid cache misses. Note that there could be other types of constraints that are not focused on input/output relationships (e.g., a concurrency constraint that two tasks have to run simultaneously). In this thesis, we focus on data-intensive tasks, which is why we are modelling input/output relationships and precedence constraints. However, these workflow constraints should also be checked to make sure a schedule is valid. We leave the modelling of more constraints for future work.

We make the following assumptions:

1. We assume a shared-everything architecture, in which each processor can execute one task at a time (we use the terms multi-core and multiprocessor interchangeably), and all n processors have access to a shared cache (e.g., shared RAM). We assume homogeneous processors for simplicity.

2. We assume an LRU-based caching policy, which is realistic given that it is the basis for caching policies in most server operating systems (e.g., Linux servers). However, we do not assume the knowledge of (available) cache size. We assume that the input to at least one task fits in the cache to avoid external memory or out-of-core computation.

3. We assume the tasks are data-intensive, which means that the bottleneck is loading the data into the cache rather than the subsequent processing. The tasks may be SQL queries, ETL jobs or user-defined functions. We also view tasks as atomic, which means they run to completion and cannot be paused or migrated. In other words, we assume non-preemptive scheduling. We also assume that each task produces its output as a whole and that a task may start when all of its input is available. That is, we focus on scheduling tasks rather than scheduling the flow of data streams throughout the DAG.

4. As mentioned in Section 1.1, we assume a storage hierarchy with significant speed gaps between different levels, e.g., a disk/RAM hierarchy or a HDFS/RDD hierarchy. We use the term cache more generally, referring to RAM/RDD-in-memory.

In our problem, a precedence constraint (u,v) indicates that the output of u is the input to v. The intuition behind our scheduling objective is to schedule v as soon as pos-

sible after u. The longer we wait, the more likely it is that other tasks will be scheduled, which may require other data inputs. Thus, the longer we wait, the more likely it is that the output of u will be evicted from an LRU cache, causing a cache miss when v runs.

To formalize this intuition, we use the notion of *Stack Distance* [20, 11, 46] (SD) from the programming languages literature as a metric for data locality. The stack distance between two accesses of a data item counts the number of other data items that were accessed in between. We aim to minimize stack distance to maximize the chance that the data item will still be in the cache when it is accessed again.

For example, recall the DAG in Figure 1.1 and assume the following schedule: [0, 1, 2, 3, 4, 5]. The output of task 1 becomes the input to tasks 2 and 4. Thus, the output of task 1 is referenced three times in the schedule: by task 1 at creation time, by task 2, and by task 4. The stack distance between the first and second reference is zero: no other tasks ran in between. The stack distance between the second and the third reference is two: task 3 ran in between and it accessed the outputs of task 0 and 2. Thus, it is more likely that the output of task 1 was evicted from the cache before it is needed by task 4.

Next, we define the stack distance of a schedule as the sum of the stack distances between every pair of consecutive references to the same data item, with reference denoting producing the item as output or consuming the item as input. If a task references more than one output, then we sequence these accesses in lexicographic order for computation (e.g., in Figure 1.1, we assume that task 3 first accesses the output of task 0 and then the output of task 3). Returning to Figure 1.1:

- task 0 produces output that is referenced once by task 3. In between, task 1 produced output referenced by task 2, giving a stack distance of one.
- task 1 produces output that is referenced twice (becomes the input to two downstream tasks), giving stack distances of zero and two, respectively, as calculated above.
- task 2 also produces output that is referenced twice, with the corresponding stack distances of zero (nothing runs between tasks 2 and 3), and two (task 3 additionally references the output of task 0 and task 4 requires the output of task 1).

This gives a stack distance of $1 + 0 + 2 + 0 + 2 = 5$ for the schedule [0, 1, 2, 3, 4, 5].

Problem 1: Single-thread Data-intensive Scheduling (SDIS). Given a DAG of tasks with precedence constraints, produce a serial schedule that obeys the precedence constraints with the smallest stack distance.

A version of SDIS was studied in [8], which only counts the stack distance between the first and the last access of each data item. This version of SDIS was shown to be polynomially solvable.

In this thesis, we solve the following novel problem:

Problem 2: Multi-processor Data-intensive Scheduling (MPDIS). Given a DAG of tasks with precedence constraints and n processors in a shared-everything architecture, produce a parallel schedule across the n processors that obeys the precedence constraints, with the smallest stack distance over a serialized representation of the parallel schedule according to task start times (we compute stack distance over this serialized representation since all processors access the same cache). Note that to break ties for tasks with same start times, we can follow the ordering of processor IDs.

For example, we compute SD for the complete schedules in Figure 1.2 below. S1 [0, 1, 2, 3, 5, 4] costs $1 + 2 + 1 = 4$ and S2 [0, 1, 4, 2, 3, 5] costs $1 + 0 + 1 = 2$. Note that S2 has a smaller stack distance and a shorter completion time.

We remark that there exists a straightforward weighted version of Problem 2, where instead of counting the number of other data items accessed between two references of some data item, we count the total size of the other data items accessed. Data item sizes can be given as edge weights in the precedence DAG.

2.3 Solutions

In this section we propose two solutions to the MPDIS problem and we discuss the baselines used in our experimental comparison. Our solution are online, meaning that tasks are scheduled on-the-fly rather than being statically assigned to different processors.

2.3.1 Parallel SDIS (PS)

The first solution, Parallel SDIS, is a straightforward extension of the SDIS solution from [8], modified to produce a static serial schedule that minimizes stack distance.

The algorithm works as follows. First, we generate a single-threaded schedule S using the existing SDIS solution. Then, whenever a processor is available, we schedule the next task from S, call it t, on this processor. Note that if t is not schedulable at this time (i.e., all the tasks it depends on have not yet terminated), then the processor is idle until t becomes schedulable.

Task	Loading(time unit)	Computation(time unit)
0	50	1
1	10	1
2	10	1
3	60	1
4	10	1
5	10	10

Table 2.1: Runtime of tasks for example DAG.

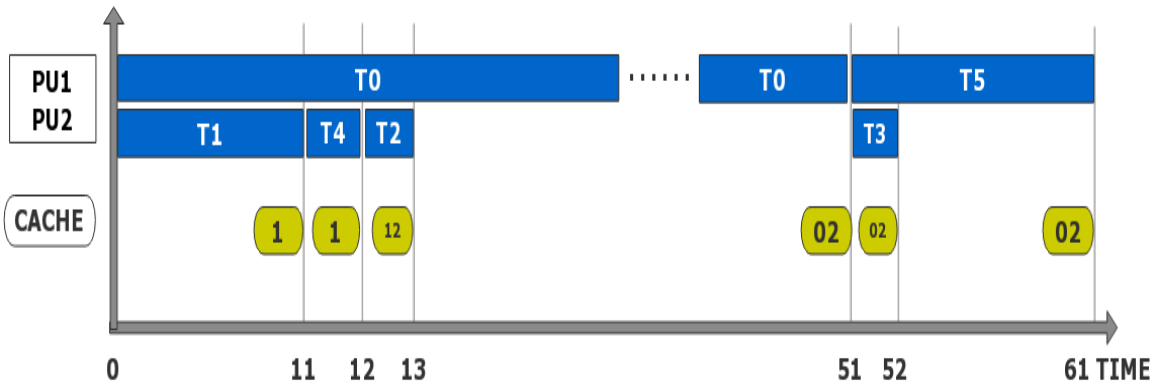


Figure 2.1: DAG 1 schedule with Algorithm PS.

Figure 2.1 shows an example of Parallel SDIS on two processors using the DAG from Figure 1.1 with computation and loading times listed in Table 2.1, and assuming the cache can hold two data items. Here, an optimal SDIS schedule turns out to be [0, 1, 4, 2, 3, 5]. First, task 0 is scheduled on PU1 and runs cold for 51 time units. At the same time, task 1 is scheduled on PU2 and runs cold for 11 time units. When task 1 terminates, the next task in the SDIS schedule is task 4, which is now scheduled on PU2. Task 4 is now schedulable (because it only relies on task 1, which just terminated), and runs hot until time 12. At this time, the cache holds the outputs of task 1. Next up in the SDIS schedule is task 2, which is scheduled on PU2 and runs hot until time 13. At this time, the cache holds the output of tasks 1 and 2. Next, task 3 is scheduled on PU2, but it must wait until task 0 terminates. Thus, task 3 begins running only at time 51 and terminates at time 52. Also, when task 0 terminates at time 51, the last task is task 5, which is now scheduled on PU1. Task 5 runs hot for 10 time units, terminating at time 61. The makespan (overall completion time) of the workload in this example is thus 61.

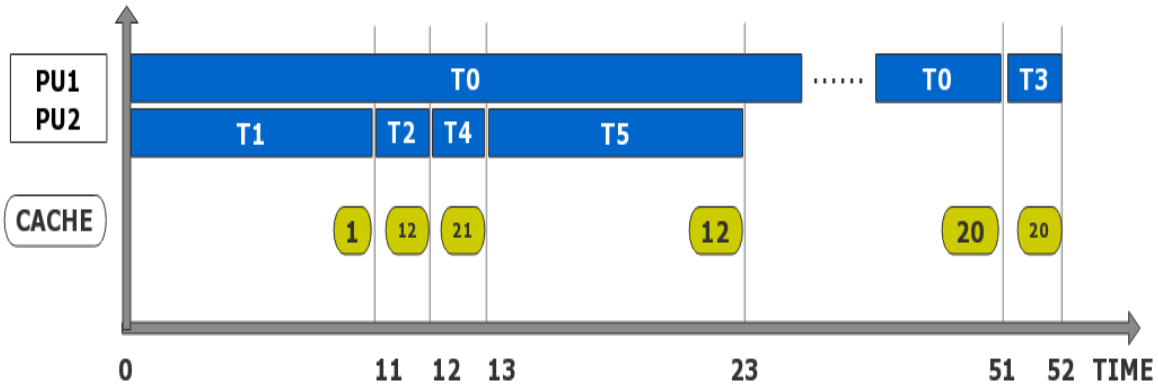


Figure 2.2: DAG 1 schedule with Algorithm OG.

2.3.2 Online Greedy (OG)

Notice a potential problem with the Parallel SDIS algorithm: since it uses a single-threaded sequencing as a seed, the next task in the schedule may not yet be schedulable in parallel with another task that is currently running. This causes some processors to be idle (e.g., PU2 in Figure 2.1 is idle from time 13 to time 51). To address this problem, we present an Online Greedy (OG) algorithm. OG does not compute a single-threaded schedule beforehand. Instead, whenever a processor becomes available, OG chooses the next *schedulable* task that yields the smallest SD when added to the current partial schedule (with ties broken arbitrarily).

Figure 2.2 shows an example of OG on two processors using the DAG from Figure 1.1, with computation and loading times listed in Table 2.1, and assuming the cache can hold two data items. First, the only schedulable tasks are 0 and 1. Breaking ties randomly, we assign task 0 to PU1 and task 1 to PU2, and both tasks run cold. When task 1 finishes at time 11 and PU2 becomes free, there are now two schedulable tasks: task 2 and task 4. To decide which task to schedule on PU2 at this time, we compute the SD of the following partial schedules: [0,1,2] and [0,1,4]. Both are zero, so we break ties randomly. Let task 2 run on PU2.

Next, task 2 terminates at time 12 (it ran hot because the output of task 1 is in the cache). At this time, tasks 4 and 5 are schedulable, so we compute the SD of the following partial schedules: [0,1,2,4] equals SD of [0,1,2,5]. Both are again zero, so we break ties randomly. Let task 4 run on PU2 (it runs hot because the output of task 2 is in the cache), finishing at time 13. Now, task 0 is still running on PU1, so the only schedulable task is task 5. Thus, we run task 5 on PU2. It runs hot because the output of task 2 is in the

cache, terminating at time 23. At this time, there are no schedulable tasks, so PU2 is idle. When task 0 terminates at time 51, the only remaining task is task 3, which runs hot until time 52.

2.3.3 Algorithm Baselines

In our experimental evaluation, we use the following two baselines:

- B1: random scheduling, which chooses a random schedulable task whenever a processor is available.
- B2: the HEFT algorithm, shown in [16] to be effective for scheduling DAGs of related tasks. The idea behind HEFT is to prioritize tasks based on the runtimes of all the tasks depending on them. In our case, we set task priorities based on the sum of the sizes of the data inputs of the tasks depending on them.

2.4 Experimental Evaluation

In this section, we present experimental results comparing our solutions and the baselines from Section 2.3. We start with simulation results and then present results using an Apache Spark cluster.

2.4.1 Experiment I: Data-intensive Scheduling Simulation

Experimental Setup

We first identified three DAGs from real-world applications [12, 2, 8], and concatenated them to create our workload, referred to as DAG1 and illustrated in Figure 2.3. The three DAGs correspond to a network monitoring workflow [8], a NASA/IPAC image stitching workflow called Montage [2], and an earthquake analysis workflow called CyberShake [2]. Notice that there are no dependencies across tasks from the three concatenated DAGs; we are modelling a workload with three independent DAGs of tasks.

In addition to DAG1, we created larger versions of it, referred to as DAG1 v2 through DAG1 v6 in the Appendix A. DAG1 v2 in Figure A.1 horizontally duplicates DAG1; DAG1

v3 in Figure A.2 vertically "grows" DAG1 by duplicating each level of tasks; DAG1 v4 in Figure A.3 is similar to DAG1 v3 but adds more data dependencies among the tasks; DAG1 v5 in Figure A.4 combines horizontal and vertical duplication; and DAG1 v6 in Figure A.5 adds more data dependencies to DAG1 v5. The number of tasks in each DAG is presented in Table 2.2. Increasing the number of tasks correspondingly increases the number of schedulable tasks at any point in time.

DAG)	**DAG1**	**v2**	**v3**	**v4**	**v5**	**v6**
Size of $\|V\|$	*62*	*113*	*113*	*113*	*145*	*145*
Size of $\|E\|$	*96*	*192*	*205*	*240*	*224*	*237*

Table 2.2: Number of tasks/constraints for each DAG in simulation.

Our simulation environment, implemented in Python, has two components: cache simulation and schedule simulation. For cache simulation, we adopted Pylru [3] for facilitating a LRU cache. By keeping the key of the data item in the LRU cache through a dictionary data structure, we simulate the cache contents at any given moment. The eviction handling is managed by Pylru. Schedule simulation uses a scheduler module, where we have implemented various scheduling algorithms for comparison. The input parameters include the DAG, with edge weights corresponding to data sizes, the cold and hot runtimes (defined as functions of the input data size), the cache size, and the number of cores/processors. The simulator then schedules the tasks as prescribed by the given scheduling algorithm, and keeps track of statistics such as the simulated disk I/O, processor idle percentage, and the *makespan*, i.e., the overall completion time of the workload.

Cache Pressure Experiments

We start by considering different cache pressure points. To do so, we set the cache size to 20GB and we vary the input size to each task. Tasks at the first level of the DAG are set to be ten times slower than other tasks to simulate base table updates and materialized views over aggregated data. Based on the results of a 120GB input simulation shown in Table 2.3, we notice that varying the number of threads does not exhibit significant difference in terms of performance improvement of OG over the baselines, which could be because of the limited dataset size. Therefore, for more cache pressure, we increase the size of the data from 120GB to 840GB.

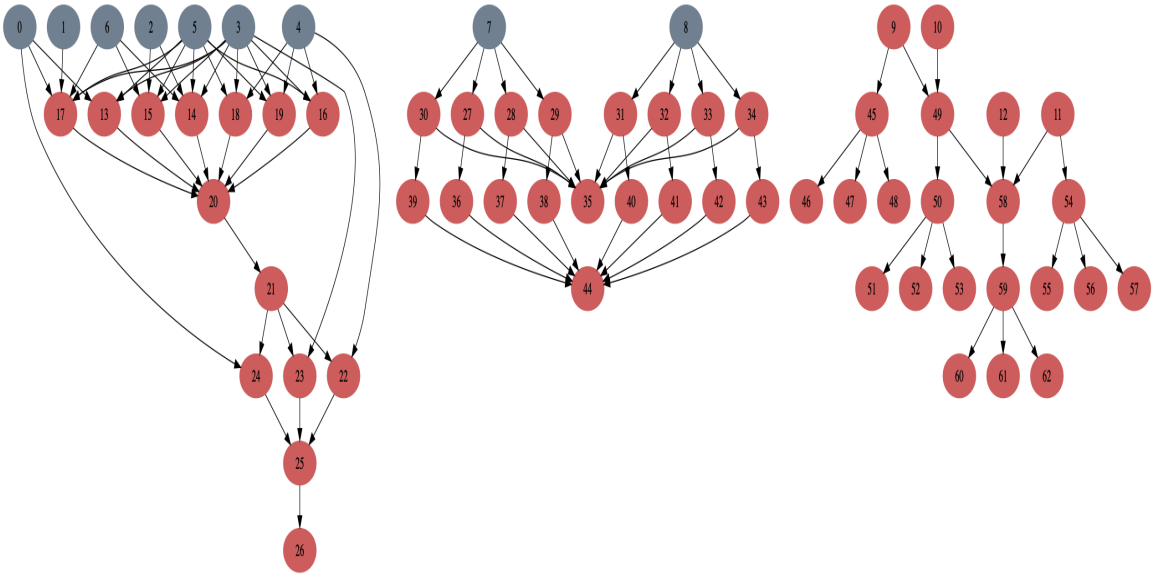


Figure 2.3: DAG1 based on real workloads.

We report job makespan for the original DAG1 in Figure 2.4, DAG1-v2 in Figure 2.5, DAG1-v3 in Figure 2.6, for DAG1-v4 in Figure 2.7, DAG1-v5 in Figure 2.8 and DAG1-v6 in Figure 2.9. Additionally, for the 840GB data size, we show disk I/O transfer and the percentage of CPU idle time for each scheduling algorithm in Table 2.4.

Number of threads	**1**	**2**	**4**	**8**
Performance Gap (B1/OG)	*1.95x*	*1.83x*	*1.84x*	*1.72x*

Table 2.3: Performance gap (120GB).

Observation 1 As shown in Figure 2.4 to Figure 2.9, both baselines, B1 and B2, give schedules with similar runtimes.

Observation 2 For DAG1, as cache pressure increases, the improvement of OG over the baselines stabilizes around 2.3x (as shown in Figure 2.4); for DAG1-v2, the gap stabilizes at 3.1x (Figure 2.5). For DAG1-v3, the gap is at 2.9x; for DAG1-v4, the gap stabilizes around 3.2x (similar to v2); DAG1-v5 significantly increases the gap to 4.5x; DAG1-v6 stabilizes at 4.3x. The disk I/O reported in Table 2.4 corroborated with the trend.

Observation 3 From the reported CPU idle time in Table 2.4, we observe that PS causes more CPU idleness than OG.

Conclusion 1 In our simulations, OG improves the runtimes of DAGs up to 4.5x over Baselines because OG generally produced schedules with less SD; the more complex the DAG, the more potential for higher improvement.

I/O (GB)	DAG1
B1	*1740.5*
B2	*1597.4*
PS	*1206.5*
OG	*921.4*

Idle (%)	DAG1
B1	*12.5*
B2	*13.7*
PS	*12.8*
OG	*7.9*

Table 2.4: I/O transfer and CPU idle time percentage (840GB).

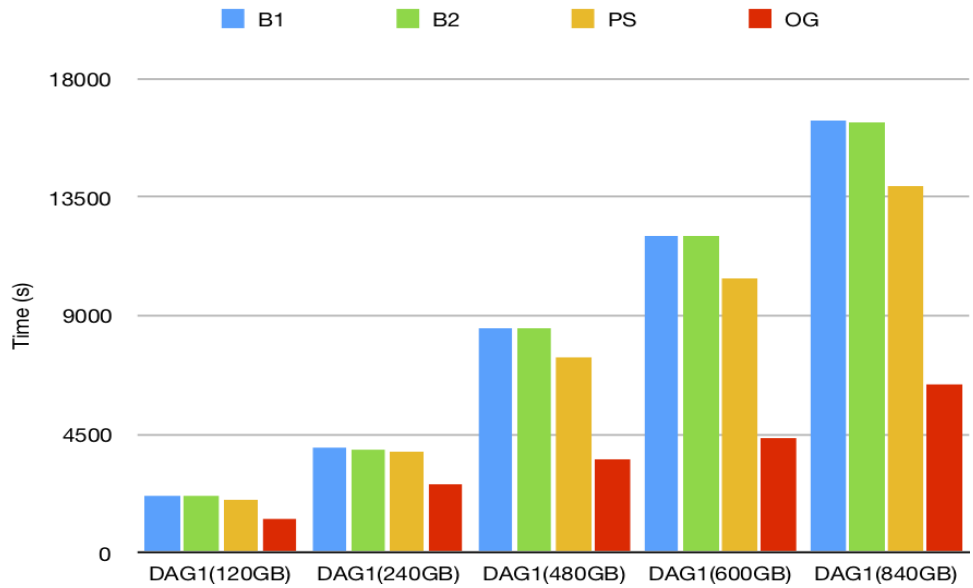


Figure 2.4: DAG1 experiments (x-axis: data size; y-axis: runtime; number of threads: 4.).

2.4.2 Experiment II: Data-intensive Scheduling in Apache Spark

Experimental Setup

For Spark experiments, we used a private cluster of 8 nodes (as well as a subset of 4 nodes from this cluster) running Ubuntu. Each node is equipped with 4 Intel Xeon E5-

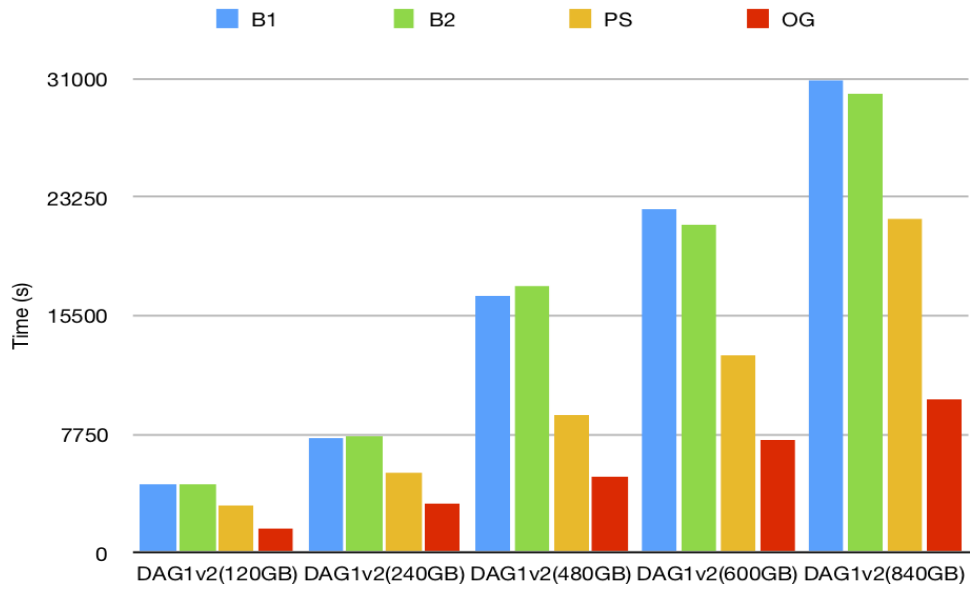


Figure 2.5: DAG1-v2 experiments (x-axis: data size; y-axis: runtime; number of threads: 4.).

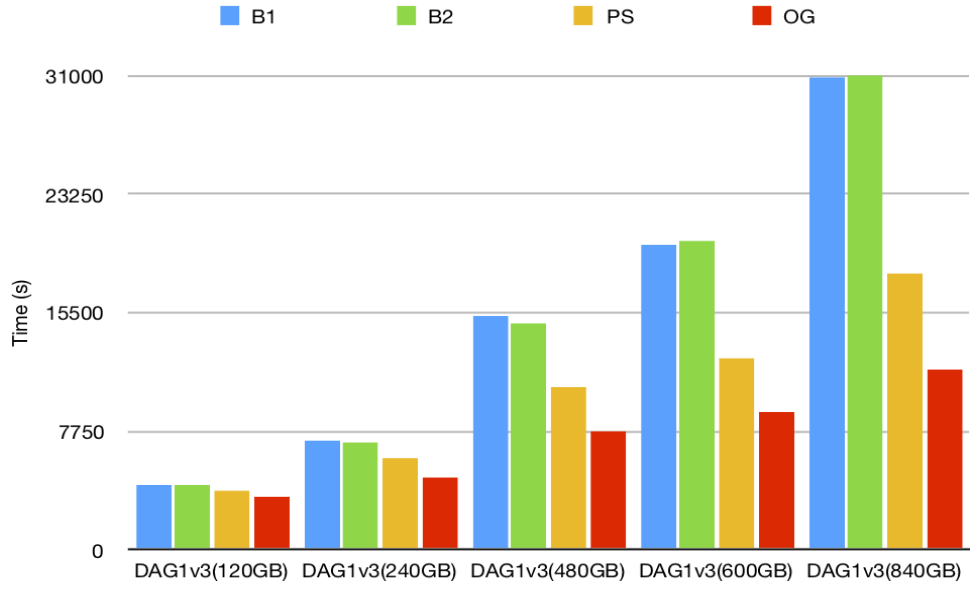


Figure 2.6: DAG1-v3 experiments (x-axis: data size; y-axis: runtime; number of threads: 4.).

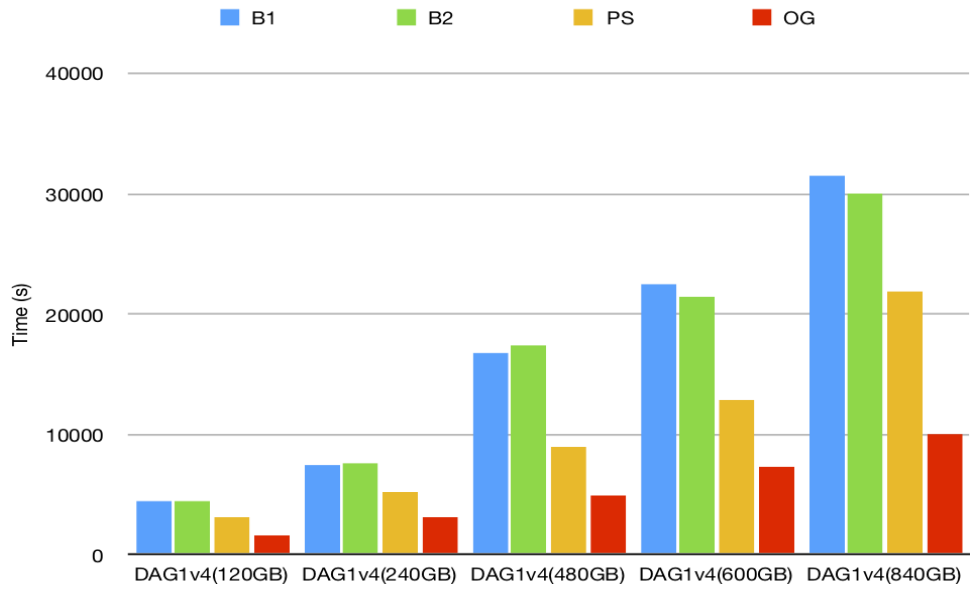


Figure 2.7: DAG1-v4 experiments (x-axis: data size; y-axis: runtime; number of threads: 4.).

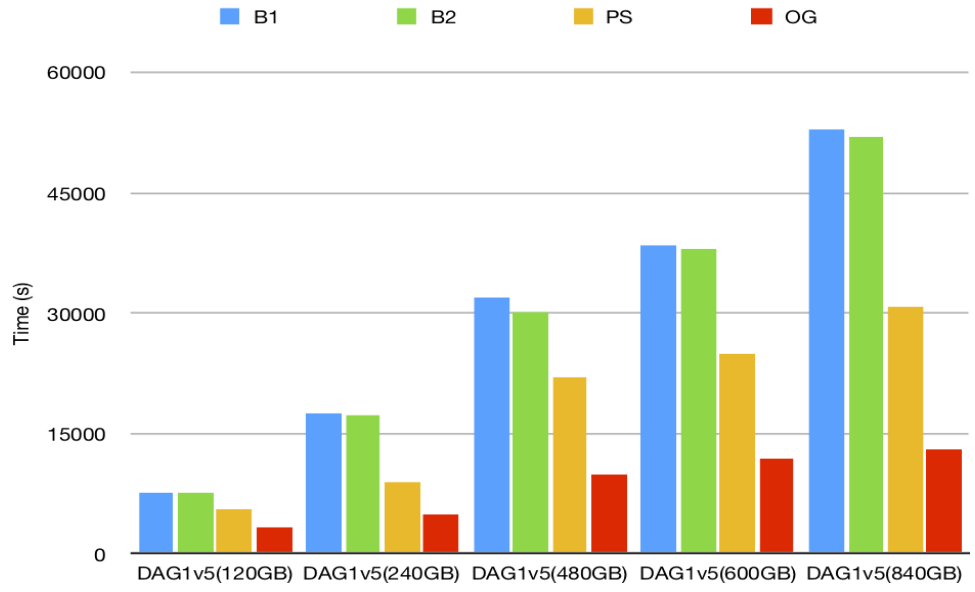


Figure 2.8: DAG1-v5 experiments (x-axis: data size; y-axis: runtime; number of threads: 4.).

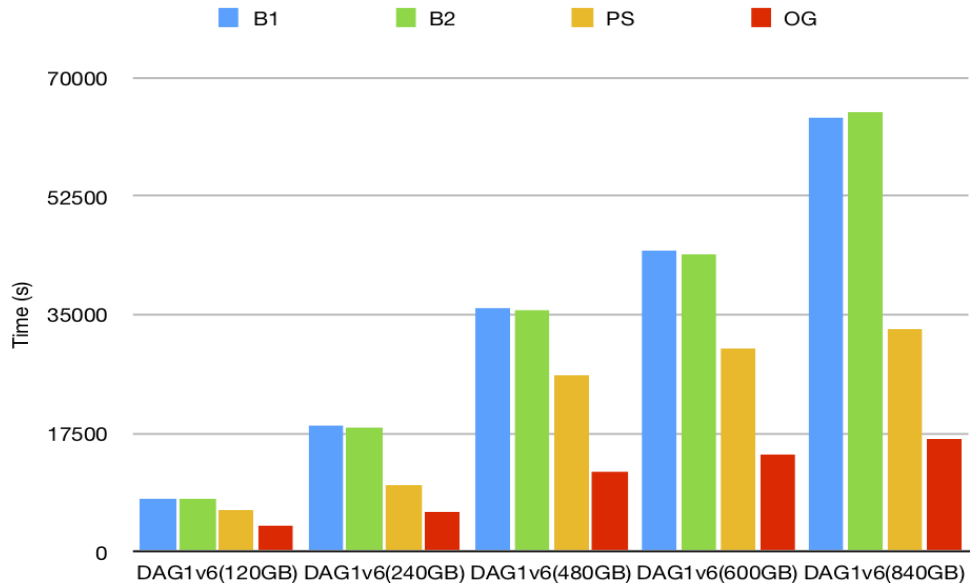


Figure 2.9: DAG1-v6 experiments (x-axis: data size; y-axis: runtime; number of threads: 4.).

2620 2.10 GHz 6-core CPUs, 64 GB of DDR3 RAM and 2.7 TB of local storage. The cluster runs Apache Spark 2.3.1 and Apache Hadoop 2.6 for HDFS support.

We use the TPC-DS benchmark as the dataset generator [49] with 200GB data. We use data-intensive queries from TPC-DS benchmark identified by [8], as well as inner join and cross join queries on top of the base tables from TPC-DS (example query in the snippet below which calculate the average quantity on hand for each product name, brand, class and category). We then insert these tasks into DAG1 from Section 2.4.1, which is the integrated DAG from three real-world applications. We use the Spark Standalone mode to simplify the setup and avoid the impact of cluster managers such as YARN [4].

Listing 2.1: Snippet: TPC-DS Query Q22

```
select i_product_name, i_brand, i_class, i_category,
avg(inv_quantity_on_hand) qoh
from inventory, date_dim, item, warehouse
where inv_date_sk = d_date_sk
and inv_item_sk = i_item_sk
and inv_warehouse_sk = w_warehouse_sk
and d_month_seq between 1176 and 1176 + 11
group by i_product_name, i_brand, i_class, i_category
```

```
order by qoh, i_product_name, i_brand, i_class, i_category;
```

We use a default Spark configuration and each executor is given all the cores available on a worker by default. We experiment with two setups: in the 4-node setup, we use four nodes in the cluster and we limit the number of concurrent tasks to 4; in the 8-node setup, we use all 8 nodes and we limit the number of concurrent tasks to 8. We implemented the workload as a packaged application, and included the scheduling algorithms as callable routines in the code.

Experimental Results

We compared our algorithms, PS and OG, with the baselines, and the results are presented in Figure 2.10. For PS/OG over B1/B2, we observe an improvement of up to 1.8x: 1.31x with 2 nodes, 1.65 x with 4 nodes, and 1.81x with 8 nodes. We observe a 1.2x improvement of OG over PS. Also OG outperforms PS not as much as in simulation. This can happen because the actual finishing order of tasks is not as FIFO as we assume in simulation and system noises also contribute to common overhead.

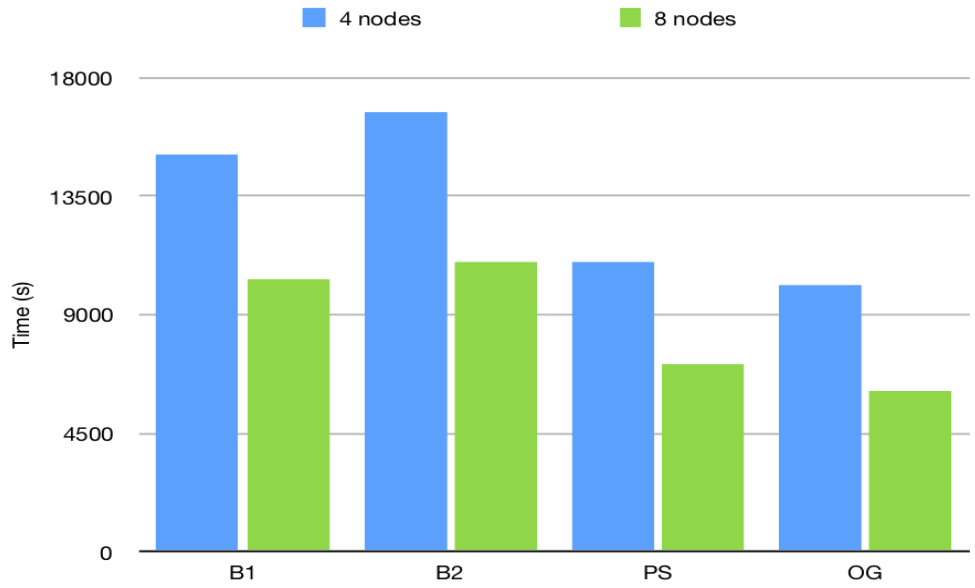


Figure 2.10: Spark tests on various number of nodes (x-axis: algorithm; y-axis: runtime).

We also monitor the total cache misses when running different schedules. To measure this, we collect the cache misses using Linux tools on each server first, and then

we aggregate the statistics for a final total cache misses of the system. In Figure 2.11, we observe that PS and OG generally produce fewer cache misses than B1 and B2, for both 4-node and 8-node Spark setups. This validates our conclusions from the simulation experiments that our algorithms minimize cache misses and therefore decrease the completion time of DAGs of data-intensive tasks.

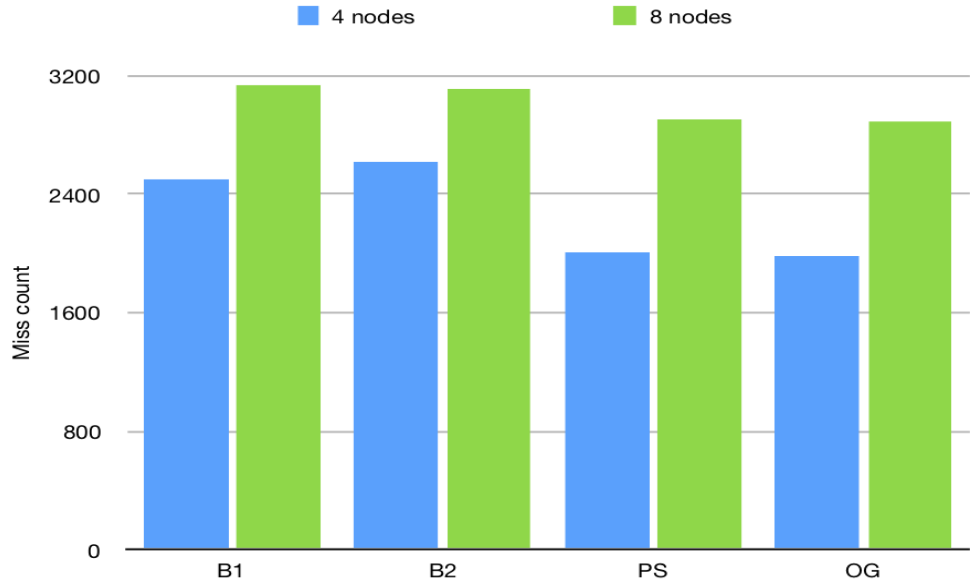


Figure 2.11: Parallel Spark tests: total cache misses (x-axis: algorithm; y-axis: miss count).

Conclusion 2 PS and OG outperform the Baselines in our Spark experiments, with OG giving slightly better schedules than PS. This is what we expected based on our simulation, and we explained the results in the observations above.

Chapter 3

Reducer Scheduling for Data-intensive Shuffling

3.1 Related Work

A MapReduce job is automatically divided into a group of map tasks and a group of reduce tasks. When computation starts, it enters the Map stage. Many map tasks (mappers) run in parallel, each one consuming data residing locally on the node on which it is running. Mappers generate intermediate key-value pairs and write them to local storage. Partitioners then perform key partitioning, i.e., assigning a group of intermediate keys to each reducer for processing. Next, in the Shuffle stage, intermediate results, i.e., key-value pairs generated by the mappers, are sent to the corresponding reducers. Finally, reduce tasks run in parallel, applying reducer code to the intermediate results assigned to them to produce the final output.

We now discuss related work on optimizing the MapReduce framework. While there are various possible optimization objectives (e.g., fairness, response time, availability, energy efficiency) [63], we focus on work involving data locality, which directly impacts data transfer. By data locality, we refer to the proximity of computation to data source. Instead of moving large data to computation which causes data transfer, we want to move the computation close to where the required data is.

Once a MapReduce job is submitted, user-level and job-level scheduling algorithms (e.g., Hadoop default, Fair and Capacity schedulers) perform resource allocation and management for this user and job. There are two schedulers that take data locality into

account. Delay scheduling delays some users' jobs if the nodes which are currently idle do not have the data required by these jobs [71]. Quincy [38] is another attempt to balance data locality and fairness for concurrent jobs by representing jobs as flow networks and finding the minimum flow cost. In this thesis, we consider different objectives and different granularity (reduce tasks of the same job vs. concurrent jobs from different users).

Next, map tasks are generated and scheduled. Similar to Delay scheduling, Match-Making [34] may delay map tasks until the nodes that have the required data are idle. BAlance-Reduce (BAR) [40] considers data locality while optimizing for completion time. Locality for mapper placement has also been studied in [32] and modelled as a linear assignment problem. We do not address mapper scheduling in this thesis.

The next stage concerns reduce task scheduling, which consists of two problems we mentioned earlier: key partitioning to assign groups of intermediate keys to reducers, and reducer placement to decide which reducer will run on which node. The default Hadoop partitioner uses hashing, but users may also write customized partitioner code. Furthermore, there is prior work on approximate algorithms for the NP-hard problem of optimal key partitioning to balance reducer load; see, e.g., [68, 55, 45, 25].

Given a key partitioning, the next step is to place the reducers on the nodes in the cluster, which is the problem we want to solve. In Hadoop, reducers are placed randomly on lightly-utilized nodes without considering data locality. The closest work to ours is the Center-of-gravity (CoGRS) algorithm [33]. When a node becomes free, this algorithm gives preference to reduce tasks for which most of the key-value pairs are already on this node. While this strategy reduces data transfer, we formalize and optimally solve the problem of minimizing data transfer.

3.2 Problem Definition

Suppose we have a cluster of n servers performing MapReduce-style processing. Suppose the Map stage has terminated and let K_k^j be the number of intermediate key-value pairs for key k generated by the mapper(s) running on server j. Suppose some key partitioning algorithm produced a partition of the intermediate key space consisting of n key groups: $G_1, G_2, \ldots, G_n$. Let G_i be the key group assigned to reducer i.

A single server may run multiple reducers, perhaps as many as the number of cores. In other words, any key group G_i can be assigned to multiple reducers running in parallel on the same server. Since we are interested in minimizing data transfer, all we need to

know is which keys will be processed at which server. Thus, in this thesis, key groups are assigned to servers and the number of "reducers" implicitly equals n.

Let C be an $n \times n$ data communication *cost matrix*, whose (i, j)th entry, denoted c_{ij}, is the communication cost assuming we place the ith reducer (which is responsible for key group G_i) on the jth server. In the simplest case, we can count the number of key-value pairs required by the ith reducer, namely those corresponding to the keys in G_i, which are not already on the jth server[1]. Formally:

$$c_{ij} = \sum_{k \in G_i, m \neq j} K_k^m$$

In this thesis, we want to find optimal reducer placements. We represent a reducer placement using a binary $n \times n$ matrix X, whose (i, j)th entry, denoted x_{ij}, is defined as follows:

$$x_{ij} = \left\{ \begin{array}{cc} 1 & \text{if reducer i is assigned to server j} \\ 0 & \text{otherwise} \end{array} \right\}$$

The first version of the reducer placement problem is to minimize the total data communication cost to reduce network congestion. Formally, we want to compute a reducer placement matrix X to minimize

$$\sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}$$

such that each reducer is placed on exactly one server and each server hosts exactly one reducer:

$$\sum_{j=1}^{n} x_{ij} = 1 \ (i = 1, 2, ..., n)$$

$$\sum_{i=1}^{n} x_{ij} = 1 \ (j = 1, 2, ..., n)$$

$$x_{ij} \in 0, 1 \ (i, j = 1, 2, ..., n).$$

The second version is to minimize the maximum data communication cost for any one server to mitigate shuffle skew. Formally, we want to compute a reducer placement matrix X to minimize

$$\max_i \sum_{j=1}^{n} c_{ij} x_{ij}$$

[1]For datacenters with heterogeneous network architectures, the number of key-value pairs that need to be transferred can be weighted by the distance or link speed between the two machines.

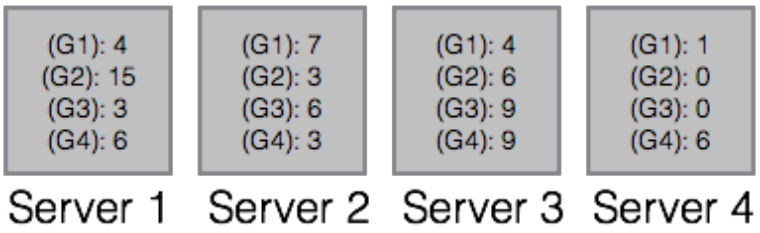


Figure 3.1: An example of an intermediate key distribution over four servers.

subject to the same constraints as above.

We conclude this section with a simple example. Figure 3.1 shows a distribution of intermediate keys generated by the Map stage on four servers. We use the notation "G1: N" to say that N key-value pairs from key group G1 were generated on a given server. The corresponding cost matrix is shown below.

$$\begin{bmatrix} 12 & 9 & 12 & 15 \\ 9 & 21 & 18 & 24 \\ 15 & 12 & 9 & 18 \\ 18 & 21 & 15 & 18 \end{bmatrix}$$

For example, $c_{11} = 12$ is the data communication cost of processing key group G_1 on server 1. Four key-value pairs for G_1 are already on server 1, but the remaining 12 must be transferred to server 1 from the other three servers.

Consider the reducer placement $[1, 3, 4, 2]$, i.e., reducer 1 is placed on server 1, reducer 2 is placed on server 3, reducer 4 is placed on server 3 and reducer 2 is placed on server 4. The total data communication cost is $c_{11} + c_{23} + c_{34} + c_{42} = 12 + 18 + 18 + 21 = 69$. The maximum communication cost per-reducer is 21. On the other hand, the reducer placement $[3, 1, 2, 4]$ has a lower total communication cost of 51 and a maximum communication cost per-reducer of 18.

Note: computing optimal reducer placements is important only if there is skew in the distribution of key groups after the Map tasks are finished, i.e., if the numbers in a particular row of the cost matrix are different. In the above example, if it were true that $c_{11} = c_{12} = c_{13} = c_{14}$, then the communication cost of reducer 1 would be there same regardless of where it was placed.

3.3 Solutions

This section presents our solutions to the two reducer placement problems defined in Section 3.2. We require an $n \times n$ communication cost matrix C as input, meaning that we need to know the intermediate key distribution on each of the n servers. Fortunately, a variety of keyspace summarization techniques for MapReduce-like systems exist, using sampling [55, 62], histograms [31, 39], sketches [68, 69], etc. The output is a reducer placement matrix X.

3.3.1 DataSum: Minimizing Total Data Transfer

Our first problem corresponds to Linear Sum Assignment which can be solved optimally in polynomial time using the so-called Hungarian algorithm [14], with a time complexity of $\mathcal{O}(n^3)$. Our solution, named *DataSum*, is shown in Algorithm 1; however, there exist more time-efficient algorithms for the linear sum assignment problem that may be used instead [26, 27]. We adopted the Hungarian algorithm since it is a classic approach upon which many newer algorithms are based, and it is a lightweight implementation. Given that in practice the number of reducers should be near the number of nodes multiplied by the maximum number of containers per node, it is small enough that the Hungarian algorithm suffices.

The DataSum algorithm exploits the following property: if a number is added to or subtracted from all entries of any one row or column in C, then an optimal solution for the modified matrix is the same as that for the original cost matrix C. To obtain a solution, the algorithm keeps adding or subtracting entries in C until all the zeros can be covered by n straight horizontal or vertical lines. When that happens, an optimal solution consists of n zero-cells in the modified matrix, such that no two zeros lie in the same row or column, and is returned in line 11 of the algorithm.

We give a worked example of *DataSum* using the following cost matrix:

$$\begin{bmatrix} 4 & 3 & 4 & 5 \\ 3 & 7 & 6 & 8 \\ 5 & 4 & 3 & 6 \\ 6 & 7 & 5 & 6 \end{bmatrix}$$

First, *DataSum* subtracts 3 from all entries in row 1, 3 from all entries in row 2, 3 from

```
Input: an n × n cost matrix C_ij;
foreach i in [1, n] (i.e. for each row of C) do
    Subtract the smallest entry in the ith row of C from all entries in this row;
end
foreach j in [1, n] (i.e. for each column of C) do
    Subtract the smallest entry in the jth column of C from all entries in this
     column;
end
while true do
    Cover the zero cost entries in C with L straight (horizontal or vertical) lines;
    if L == n then
        return an optimal assignment in C;
    p = smallest c_ij not covered by any line;
    Subtract p from each row that is not yet covered by any straight line;
    Add p to each column covered by a straight line;
end
```

Algorithm 1: *DataSum*.

all entries in row 3, and 5 from all entries in row 4. This gives the following matrix:

$$\begin{bmatrix} 1 & 0 & 1 & 2 \\ 0 & 4 & 3 & 5 \\ 2 & 1 & 0 & 3 \\ 1 & 2 & 0 & 1 \end{bmatrix}$$

Next, the algorithm subtracts 0 from all entries in column 1, 2 and 3, and 1 from all entries in column 4, giving the following matrix:

$$\begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 4 & 3 & 4 \\ 2 & 1 & 0 & 2 \\ 1 & 2 & 0 & 0 \end{bmatrix}$$

We can now cover all zeros in this matrix with four straight lines: one horizontal line through each row, or equivalently, one vertical line through each column. The only possible set of four zero-cells in this matrix, no two of which lie in the same row or column, is: c_{12}, c_{21}, c_{33} and c_{44}. This corresponds to an optimal reducer placement of $[2, 1, 3, 4]$, with a total data transfer of $3 + 3 + 3 + 6 = 15$.

3.3.2 DataMax: Minimizing Maximum Data Transfer

Our second problem corresponds to Linear Bottleneck Assignment which can also be solved in polynomial time by a variety of existing algorithms. Our solution, referred to as *DataMax* is shown in Algorithm 2.

We use a simple threshold algorithm [15] which consists of two stages (see [23] for a more efficient augmenting-path algorithm). We adopt this algorithm because it is more lightweight and straightforward, and for a practical number of reducers in a Hadoop system, it is sufficiently efficient. In the first stage, a threshold cost value c^* is chosen. In the second stage, a threshold cost matrix $\bar{C}[c^*]$ is defined for that threshold based on the cost matrix C provided as input. The (i, j)th entry of $\bar{C}[c^*]$, denoted $\bar{c}_{ij}$, is defined as follows.

$$\bar{c}_{ij} = \left\{ \begin{array}{ll} 1, & \text{if } c_{ij} > c^* \\ 0, & \text{otherwise} \end{array} \right\}$$

The algorithm then checks whether a reducer placement with zero total cost exists for $\bar{C}[c^*]$. To do so, the insight is that linear assignment problems are instances of *bipartite graph perfect matching*, where for graph $G(V, E)$, a *perfect matching* $PM \subset E$ exists if every vertex is incident to exactly one edge in PM. Thus, the algorithm defines a bipartite graph $G[c^*](V, E)$ which has an edge $[i, j] \in E$ iff $c_{ij} \leq c^*$, for a given threshold cost value c^*. A perfect matching in this graph implies that a zero-cost reducer placement exists in $\bar{C}[c^*]$, which in turn gives an optimal solution for the original cost matrix C.

The time complexity of *DataMax* is $\mathcal{O}(T(n) \log n)$ where $T(n)$ is the time complexity of checking for a perfect match. In our implementation, we use the Hopcroft-Karp algorithm [35] with time complexity of $\mathcal{O}(n^{2.5})$ for dense graphs (common in MapReduce jobs whose intermediate results are scattered across the cluster, leaving the cost matrix with few zeros).

We now give a worked example of *DataMax* on the same cost matrix as the one used in the worked example of *DataSum*. We show the matrix again below for convenience.

$$\begin{bmatrix} 4 & 3 & 4 & 5 \\ 3 & 7 & 6 & 8 \\ 5 & 4 & 3 & 6 \\ 6 & 7 & 5 & 6 \end{bmatrix}$$

At the beginning, we have $c_0^* = 3$ and $c_1^* = 8$. The median of all c_{ij} within the range $[c_0^*, c_1^*]$ is $c^* = 6$. This gives the following threshold matrix, $\bar{C}[6]$, which translates to the bipartite graph in Figure 3.2.

```
1  Input: an n × n cost matrix C;
2  c_0^* ← min c_ij, c_1^* ← max c_ij;
3  if c_0^* ≠ c_1^* then
4  |   while C^* = {c_ij | c_0^* < c_ij < c_1^*} ≠ ∅ do
5  |   |   c^* ← median of C^*;
6  |   |   if a perfect matching exists in G[c^*] then
7  |   |   |   c_1^* ← c^*
8  |   |   else
9  |   |   |   c_0^* ← c^*
10 |   |   end
11 |   end
12 |   if G[c_0^*] not checked for perfect matching then
13 |   |   if a perfect matching exists in G[c_0^*] then
14 |   |   |   c_1^* ← c^*
15 |   return a perfect matching in G[c_1^*];
16 else
17 |   Any reducer placement is optimal;
18 end
```

Algorithm 2: *DataMax*.

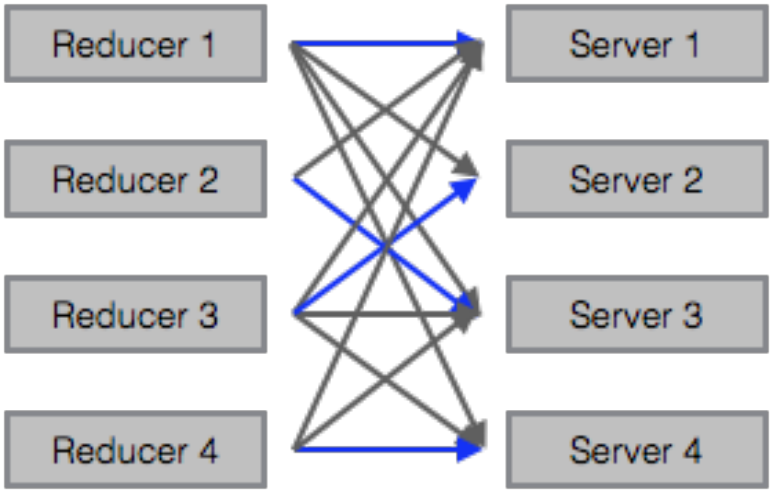


Figure 3.2: Bipartite graph for $\bar{C}[6]$.

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

There is a perfect match in this graph (highlighted in blue) so we update $c_1^* \leftarrow c^* = 6$. The new c^* is 5, from which we generate a new threshold matrix, $\bar{C}[5]$, shown below and the corresponding bipartite graph shown in Figure 3.3.

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

Again, there exists a perfect matching (highlighted in blue), and we update $c_1^* \leftarrow c^* = 5$. The new c^* is 4 which leads to $\bar{C}[4]$ and the corresponding bipartite graph in Figure 3.4.

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

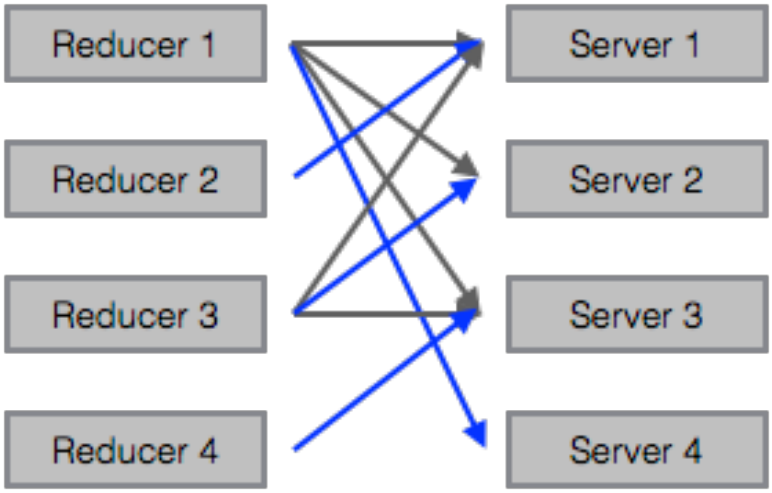


Figure 3.3: Bipartite graph for $\bar{C}[5]$.

Table 3.1: Dataset details.

Dataset	Source	Size	Key
New York Times	UCI ML Repo.	958MB	docID
PubMed	UCI ML Repo.	7.3GB	docID

There is no perfect matching in Figure 3.4, so we update $c_0^* \leftarrow c^* = 4$. For $c_0^* = 4$ and $c_1^* = 5$, there is no value that falls in this range (that is, $C^* = \{c_{ij} | c_0^* < c_{ij} < c_1^*\} = \varnothing$), so the while-loop terminates. The final assignment uses a threshold of 5, corresponding to the perfect matching illustrated in Figure 3.3. Thus, we get the following reducer placement: [4,1,2,3]. The maximum data transfer per reducer is 5.

3.4 Experimental Evaluation

This section presents our experimental results using a private cluster of 16 nodes (as well as a subset of 8 nodes from this cluster) running CentOS 6.4. Each node is equipped with 4 Intel Xeon E5-2620 2.10 GHz 6-core CPUs, 64 GB of DDR3 RAM and 2.7 TB of local storage. The cluster runs Apache Hadoop 1.2.1.

To test *DataSum* and *DataMax* against native Hadoop strategies, we use the TeraSort benchmark from the native Hadoop distribution, available at *sortbenchmark.org*. TeraSort

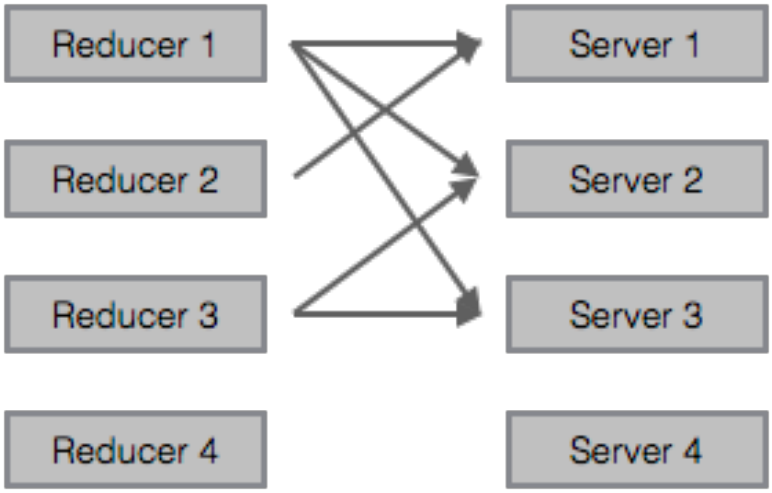


Figure 3.4: Bipartite graph for $\bar{C}[4]$.

is an external sort and is known to be shuffle-intensive. We use two real datasets described in Table 3.1, New York Times (NYT) and PubMed, and available at [5]. Both of these datasets consist of documents, and, for sorting, we use document IDs (docIDs) as intermiediate keys. Document IDs are not primary keys: there may be many records with the same docID. Both datasets have skewed distribution of the docIDs; i.e., some docIDs are much more frequent than others.

Since our cluster resides on a single rack and all the network links have the same speed, we calculate cost matrices using the simple method mentioned in Section 3.2, i.e., by counting the number of key-value pairs that need to be transferred.

3.4.1 Experiment I: Minimizing total data transfer

For this set of experiments, our methodology is as follows. First, we load the datasets into the Hadoop Distributed File System (HDFS) and compute the distribution of keys locally stored on each server. We then run the Map stage of the TeraSort benchmark followed by key partitioning, which is done by the custom sampling-based tree partitioner included in TeraSort. At this point, we have the key distribution for each server and the key groups, which allows us to compute the cost matrix. Next, we proceed to the Reduce stage and extract the reducer placement used by standard Hadoop from the Hadoop log file. Finally, we compute our reducer placements using *DataSum* and *DataMax*, and we calculate data transfer incurred by each strategy from the cost matrix. Thus, we are not

calculating the actual data transfer that took place during the MapReduce job, but rather we are estimating it based on the actual key distribution and the actual key partitioning.

We use the New York Times and PubMed datasets, and we use the full 16-node cluster and an 8-node subset. Figures 3.5a and 3.5b show the results for New York Times using 8 and 16 servers, respectively. Figures 3.5c and 3.5d show the results for PubMed. The Y axis shows the total data transfer across all the reducers. We include all three tested algorithms, keeping in mind that only *DataSum* optimizes for total data transfer.

We conclude that:

- *DataSum reduces the total data transfer by at least 50 percent in all tested scenarios compared to Hadoop. Notably, proper key partitioning alone (performed by the custom TeraSort partitioner) can still lead to high data transfer costs, which can be minimized by applying our reducer placement techniques.*
- *Not surprisingly, DataMax gives reducer placements that have higher total data transfer than those of DataSum, but is still better than Hadoop.*

3.4.2 Experiment II: Minimizing max. data transfer per-server

Here, we again use the New York Times and PubMed datasets and the same methodology as in the previous experiment. Figures 3.6a and 3.6b show the results for New York Times, and Figures 3.6c and 3.6d show the results for PubMed (8 and 16 servers, respectively). In this experiment, the Y axis shows the maximum data transfer per server.

Interestingly, *DataMax* reduced the maximum data transfer per server compared to Hadoop only by 10-12 percent. Upon further inspection, we found that the combination of key distribution across the servers and key partitioning was to blame. Many key groups created by the TeraSort partitioner contained a small number of frequent keys which were localized to a small number of servers (between one and three). However, three key groups included key-value pairs that were scattered nearly-uniformly across nearly every server. Thus, no matter which nodes were assigned these three key groups for processing, the data transfer costs were approximately equal since there was no single node that locally stored a majority of the required key-value pairs. As a result, one of these three key groups always had nearly the same maximum per-reducer data transfer (within 10 or so percent), regardless of reducer placement. This explains the results in Figures 3.6a through 3.6d.

We hypothesized that *DataMax* would perform better than Hadoop on the TeraSort benchmark if the key distribution or the partitioner were different. For example, if each key group had a "preferred" server and no key group had key-value pairs uniformly scattered across all servers, then *DataMax* would be much more likely than standard Hadoop to find those preferred servers. To test this hypothesis, we modified the PubMed dataset by deleting the keys which were previously scattered uniformly across all servers.

Figure 3.7a shows the total data transfer and Figure 3.7b shows the maximum per-server data transfer of each of the three tested techniques using the modified datasets. *DataMax* computes similar reducer placements as *DataSum*, and both of our techniques reduce the total and maximum data transfer by about 50 percent compared to Hadoop. This is likely the best-case scenario for our algorithms given the PubMed dataset and the TeraSort benchmark (including the custom TeraSort partitioner).

Based on the results in this section, we conclude that:

- *The relative improvement of DataMax versus Hadoop in terms of maximum data transfer per-server is sensitive to the distribution of intermediate keys and the key partitioning. With the default data placement on HDFS and the TeraSort partitioner, the improvement was only 10 percent. With a modified PubMed dataset, the best-case improvement was about 50 percent.*

3.4.3 Experiment III: Job runtimes in a Hadoop cluster

In this experiment, we implement our schedulers in Hadoop 1.2.1, and study how they impact MapReduce runtimes. We use the default TeraSort benchmark from Hadoop distribution, over both the original and a modified PubMed, all tested on 8 nodes of the cluster. We define *job completion time* as the total runtime and *shuffle time* as the time between the finish time of the last shuffle and the beginning of the first reduce task. We compare *DataSum* and *DataMax* with the default Hadoop scheduler (JobQueueTaskScheduler, essentially a FIFO random strategy). Cost matrices are computed offline in this experiment.

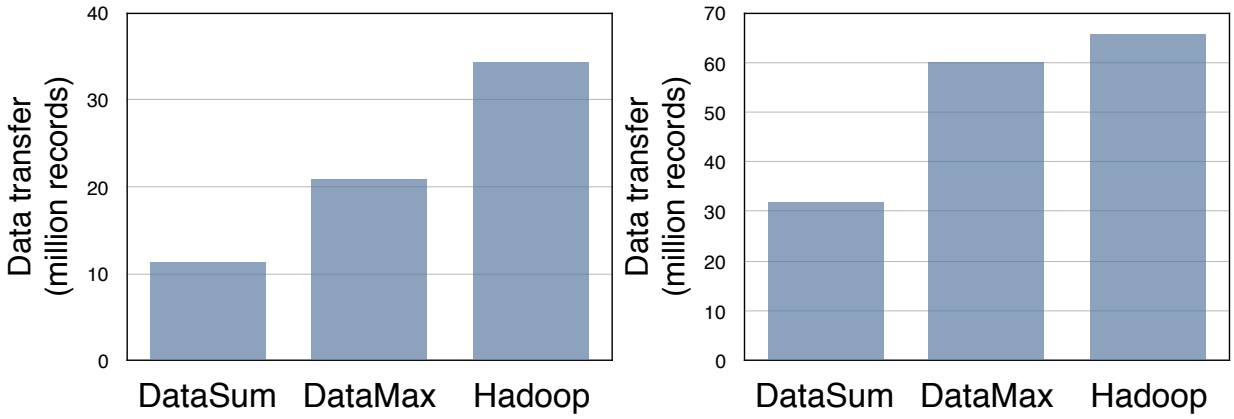
Figure 3.8a and Figure 3.8b plot the TeraSort completion times (blue bars) and shuffle times (green bars) on original PubMed and modified PubMed, respectively. We observe that:

- *For the original PubMed, our techniques improve shuffle runtimes by around 10 percent and job completion times by about 6 percent over standard Hadoop. This is consistent with the 10 percent improvement in maximum data transfer per server from Section 3.4.2.*
- *For the modified more skewed PubMed from Section 3.4.3, our techniques can further improve the shuffle time by around 24 percent. Thus, DataSum and DataMax can substantially reduce network utilization and also reduce job runtimes (depending on the skew in the key distribution).*

3.4.4 Experiment IV: Efficiency and scalability of computing optimal reducer placements

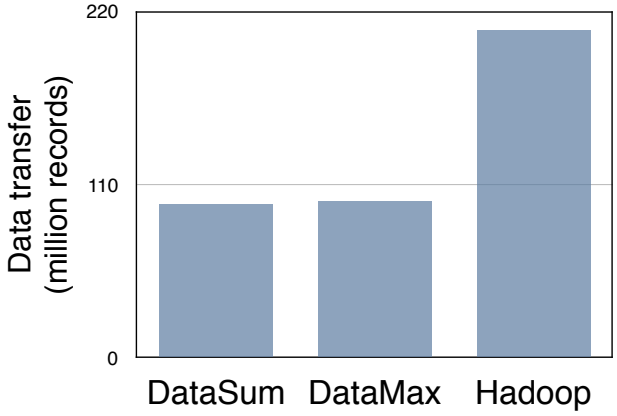
The cost of the improvements in network utilization and job runtimes, of course, is that we need to keep track of the key distribution and run our algorithms. In the final experiment, we compute the running time of *DataSum* and *DataMax* on random cost matrices of various sizes. We implemented stand-alone versions of these algorithms in Java, and ran them on the login node of the cluster. Figure 3.9a and Figure 3.9b show the average running time (over three runs) as a function of n, the number of servers and therefore also the number rows and columns in the cost matrices. *DataMax* is simpler and therefore faster than *DataSum*, but both can compute optimal reducer placements for thousands of servers within a second.

We reiterate that there are several ways to improve performance which we will investigate in future work: by optimizing our implementations, or by using more efficient and/or distributed algorithms for the corresponding linear assignment problems. Additionally, we can estimate cost matrices early, even before the Map stage terminates, to further reduce the impact of computing an optimal reducer placement on job completion time.

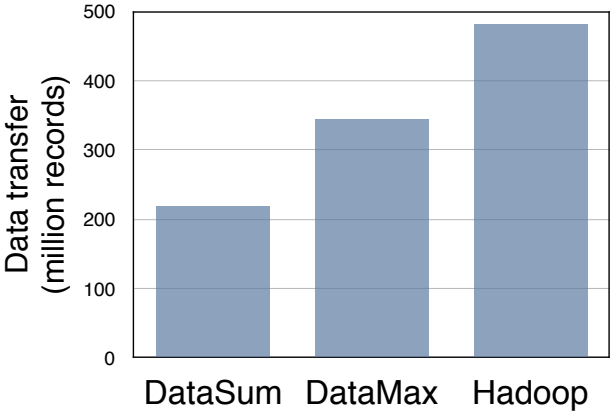


(a) NYT, 8 nodes

(b) NYT, 16 nodes

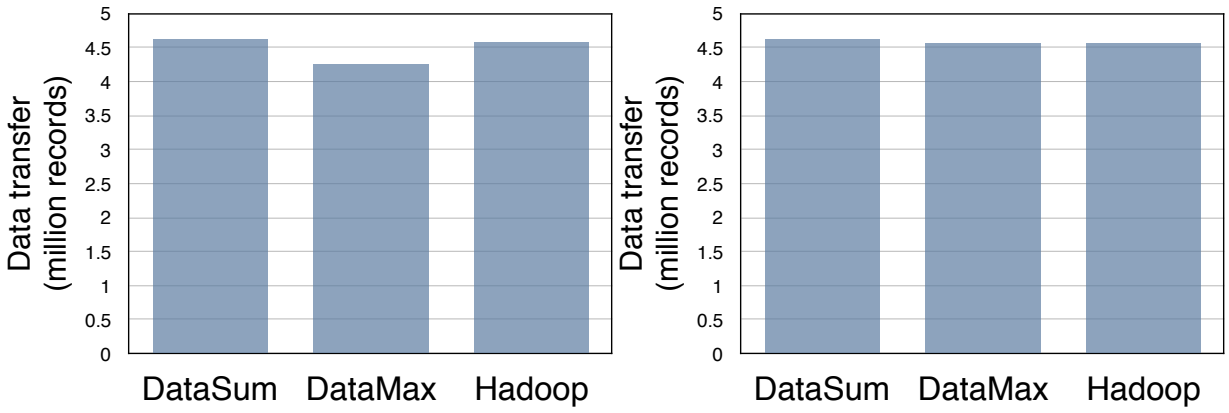


(c) PubMed, 8 nodes



(d) PubMed, 16 nodes

Figure 3.5: Total data transfer for *DataSum*, *DataMax* and Native Hadoop.



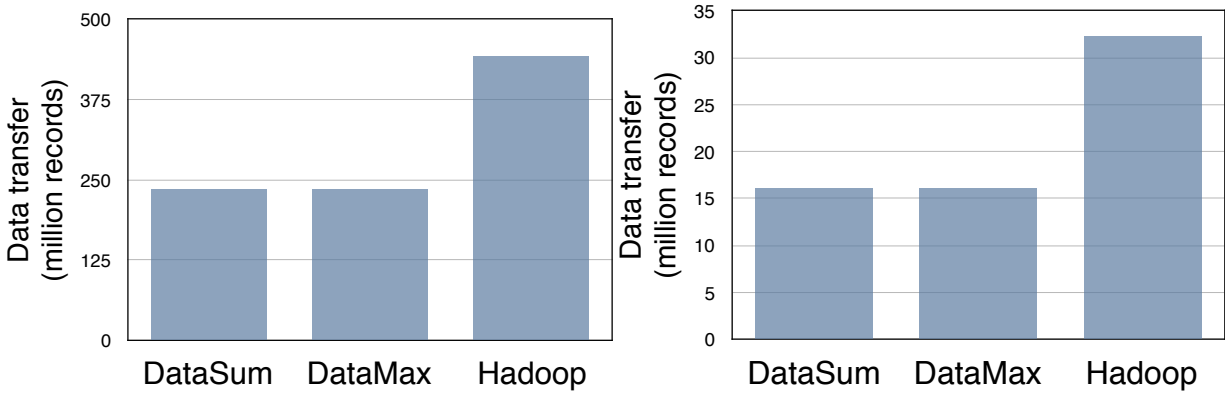
(a) NYT, 8 nodes

(b) NYT, 16 nodes

(c) PubMed, 8 nodes

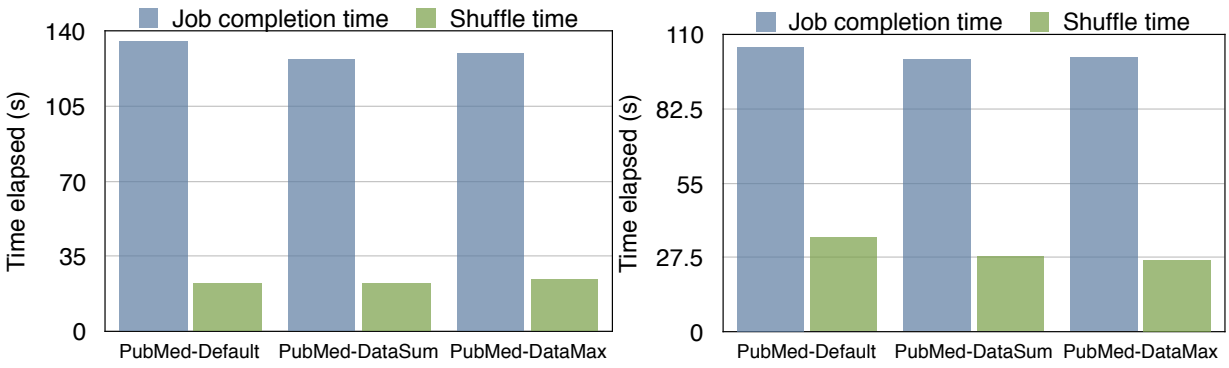
(d) PubMed, 16 nodes

Figure 3.6: Maximum data transfer per server for *DataSum*, *DataMax* and Native Hadoop.



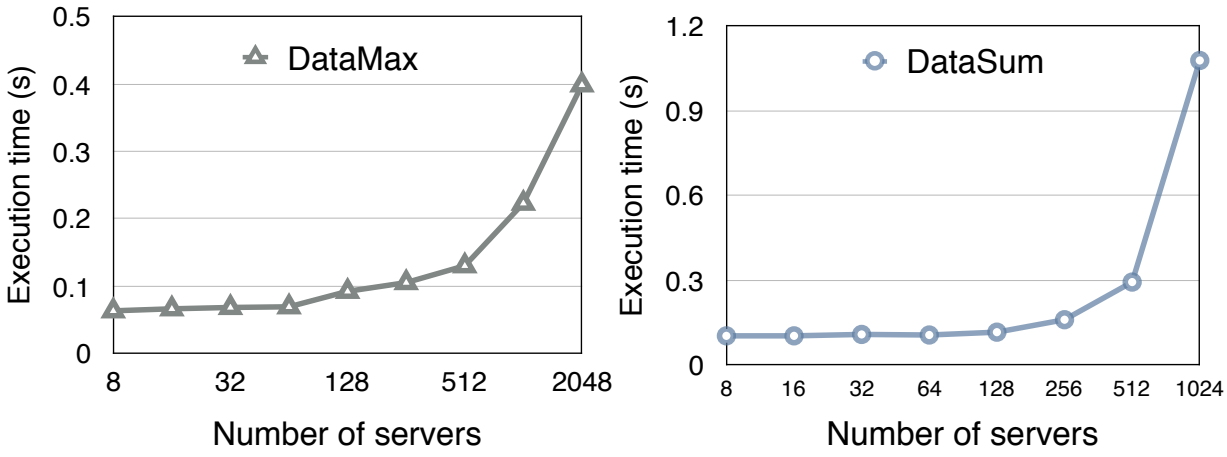
(a) Total data transfer (b) Maximum data transfer

Figure 3.7: Total and maximum data transfer for the modified dataset, 16 nodes.



(a) original PubMed (b) skewed PubMed

Figure 3.8: Hadoop Job completion and shuffle times, 8 nodes.



(a) *DataMax* (b) *DataSum*

Figure 3.9: *DataMax* and *DataSum* execution times for different numbers of servers.

Chapter 4

Conclusions

In this thesis, we address the problem of data-intensive scheduling. More specifically, we tackle two problems with data-intensive characteristics: multi-processor scheduling for data-intensive tasks, and reducer scheduling for data-intensive shuffling.

In Chapter 2, we defined the MPDIS problem of scheduling a DAG of data-intensive tasks on multiple processors. We defined the problem based on the notion of stack distance from the programming languages literature. We proposed and experimentally evaluated heuristic algorithms to optimize for less cache misses and smaller completion time.

In Chapter 3, we solved a scheduling problem in the context of MapReduce-style processing. We showed how to assign reducers to processing servers in a way that 1) minimizes the total data transfer or 2) minimizes the maximum data transfer per-reducer. We provided optimal polynomial-time solutions for these problems by reducing them to existing optimization problems: Linear Sum Assignment and Linear Bottleneck Assignment, respectively. Our experimental results showed over 50-percent improvements in network utilization (total data transfer) over standard Hadoop. The improvement in maximum data transfer per-server was less pronounced and more sensitive to the key distribution and key partitioning, and ranged from 10 percent to a best-case of 50 percent; this led to a similar improvement in shuffle runtimes.

We suggest three directions for future work. First, for data-intensive task scheduling, we assumed a shared-everything architecture, in which multiple processors share a cache. In future work, we will study different versions of the MPDIS problem for shared-nothing settings. The additional complexity will be to partition the workload in a way that allows each partition to schedule its workload in a cache-friendly way. Second, for

data-intensive shuffling scheduling, we can combine key partitioning and reducer placement and investigate pareto-optimal solutions with respect to their load balancing and communication cost. Finally, we plan to study reducer placement for workflows of multiple MapReduce jobs, where the key-value pairs required by a reducer of some job may have already been computed on some node during a previous job (e.g., a maintenance job that is executed periodically).

References

[1] Apache hadoop. http://hadoop.apache.org/. Accessed: 2019-08-05.

[2] Pegasus workflow generator. https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator. Accessed: 2019-08-05.

[3] Pylru 1.2.0. https://pypi.org/project/pylru/. Accessed: 2019-08-05.

[4] Spark standalone. https://spark.apache.org/docs/latest/spark-standalone.html. Accessed: 2019-08-05.

[5] Uci machine learning repository. http://archive.ics.uci.edu/ml/datasets.php. Accessed: 2019-08-05.

[6] Foto N Afrati, Nikos Stasinopoulos, Jeffrey D Ullman, and Angelos Vassilakopoulos. Sharesskew: An algorithm to handle skew for joins in mapreduce. *Information Systems*, 77:129–150, 2018.

[7] Shivnath Babu, Herodotos Herodotou, et al. Massively parallel databases and mapreduce systems. *Foundations and Trends® in Databases*, 5(1):1–104, 2013.

[8] Arian Bär, Lukasz Golab, Stefan Ruehrup, Mirko Schiavone, and Pedro Casas. Cache-oblivious scheduling of shared workloads. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 855–866. IEEE, 2015.

[9] Paul Beame, Paraschos Koutris, and Dan Suciu. Skew in parallel query processing. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 212–223. ACM, 2014.

[10] Erik Berg and Erik Hagersten. Statcache: a probabilistic approach to efficient and accurate data locality analysis. In *IEEE International Symposium on-ISPASS Performance Analysis of Systems and Software, 2004.*, pages 20–27. IEEE, 2004.

[11] Kristof Beyls and Erik DâĂŹHollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*, volume 14, pages 350–360, 2001.

[12] Shishir Bharathi, Ann Chervenak, Ewa Deelman, Gaurang Mehta, Mei-Hui Su, and Karan Vahi. Characterization of scientific workflows. In *2008 third workshop on workflows in support of large-scale science*, pages 1–10. IEEE, 2008.

[13] Nicolas Bruno, YongChul Kwon, and Ming-Chuan Wu. Advanced join strategies for large-scale distributed computation. *Proceedings of the VLDB Endowment*, 7(13):1484–1495, 2014.

[14] Rainer E Burkard and Eranda Cela. Linear assignment problems and extensions. In *Handbook of combinatorial optimization*, pages 75–149. Springer, 1999.

[15] Rainer E Burkard, Mauro Dell'Amico, and Silvano Martello. *Assignment problems*. Springer, 2009.

[16] Louis-Claude Canon, Emmanuel Jeannot, Rizos Sakellariou, and Wei Zheng. Comparative evaluation of the robustness of dag scheduling heuristics. In *Grid Computing*, pages 73–84. Springer, 2008.

[17] Chandra Chekuri and Rajeev Motwani. Precedence constrained scheduling to minimize sum of weighted completion times on a single machine. *Discrete Applied Mathematics*, 98(1-2):29–38, 1999.

[18] Dazhao Cheng, Yuan Chen, Xiaobo Zhou, Daniel Gmach, and Dejan Milojicic. Adaptive scheduling of parallel jobs in spark streaming. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[19] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. *ACM SIGCOMM Computer Communication Review*, 41(4):98–109, 2011.

[20] Edward Grady Coffman and Peter J Denning. *Operating systems theory*, volume 973. prentice-Hall Englewood Cliffs, NJ, 1973.

[21] Pierluigi Crescenzi and Viggo Kann. A compendium of np optimization problems (1998). *ftp://ftp.nada.kth.se/Theory/Viggo-Kann/compendium.pdf*.

[22] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Kairos: Preemptive data center scheduling without runtime estimates. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*, pages 135–148, 2018.

[23] Ulrich Derigs and Uwe Zimmermann. An augmenting path method for solving linear bottleneck assignment problems. *Computing*, 19(4):285–295, 1978.

[24] Francis Deslauriers, Peter McCormick, George Amvrosiadis, Ashvin Goel, and Angela Demke Brown. Quartet: Harmonizing task scheduling and caching for cluster computing. In *8th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[25] Prateek Dhawalia, Sriram Kailasam, and Dharanipragada Janakiram. Chisel++: handling partitioning skew in mapreduce framework using efficient range partitioning technique. In *Proceedings of the sixth international workshop on Data intensive distributed computing*, pages 21–28. ACM, 2014.

[26] Harold Gabow and Robert Tarjan. Almost-optimum speed-ups of algorithms for bipartite matching and related problems. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 514–527. ACM, 1988.

[27] Harold N Gabow and Robert E Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5):1013–1036, 1989.

[28] Carlos Garcia-Alvarado, Venkatesh Raghavan, Sivaramakrishnan Narayanan, and Florian M Waas. Automatic data placement in mpp databases. In *2012 IEEE 28th International Conference on Data Engineering Workshops*, pages 322–327. IEEE, 2012.

[29] Lukasz Golab, Marios Hadjieleftheriou, Howard Karloff, and Barna Saha. Distributed data placement to minimize communication costs via graph partitioning. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, page 20. ACM, 2014.

[30] Ronald L Graham, Eugene L Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. In *Annals of discrete mathematics*, volume 5, pages 287–326. Elsevier, 1979.

[31] Benjamin Gufler, Nikolaus Augsten, Angelika Reiser, and Alfons Kemper. Load balancing in mapreduce based on scalable cardinality estimates. In *2012 IEEE 28th International Conference on Data Engineering*, pages 522–533. IEEE, 2012.

[32] Zhenhua Guo, Geoffrey Fox, and Mo Zhou. Investigation of data locality in mapreduce. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 419–426. IEEE Computer Society, 2012.

[33] Mohammad Hammoud, M Suhail Rehman, and Majd F Sakr. Center-of-gravity reduce task scheduling to lower mapreduce network traffic. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 49–58. IEEE, 2012.

[34] Chen He, Ying Lu, and David Swanson. Matchmaking: A new mapreduce scheduling technique. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 40–47. IEEE, 2011.

[35] John E Hopcroft and Richard M Karp. An nˆ5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231, 1973.

[36] Zhiming Hu, Baochun Li, Zheng Qin, and Rick Siow Mong Goh. Job scheduling without prior information in big data processing systems. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 572–582. IEEE, 2017.

[37] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 41–51. IEEE, 2010.

[38] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276. ACM, 2009.

[39] Jeffrey Jestes, Ke Yi, and Feifei Li. Building wavelet histograms on large data in mapreduce. *Proceedings of the VLDB Endowment*, 5(2):109–120, 2011.

[40] Jiahui Jin, Junzhou Luo, Aibo Song, Fang Dong, and Runqun Xiong. Bar: An efficient data locality driven task scheduling algorithm for cloud computing. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 295–304. IEEE, 2011.

[41] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

[42] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 25–36. ACM, 2012.

[43] YongChul Kwon, Kai Ren, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Managing skew in hadoop. *IEEE Data Eng. Bull.*, 36(1):24–33, 2013.

[44] Christian A Lang, Bishwaranjan Bhattacharjee, Tim Malkemus, Sriram Padmanabhan, and Kwai Wong. Increasing buffer-locality for multiple relational table scans through grouping and throttling. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 1136–1145. IEEE, 2007.

[45] Yanfang Le, Jiangchuan Liu, Funda Ergün, and Dan Wang. Online load balancing for mapreduce with skewed data input. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*, pages 2004–2012. IEEE, 2014.

[46] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[47] Xiao Meng and Lukasz Golab. Optimal reducer placement to minimize data transfer in mapreduce-style processing. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 339–346. IEEE, 2017.

[48] Pietro Michiardi, Damiano Carra, and Sara Migliorini. Cache-based multi-query optimization for data-intensive scalable computing frameworks. *arXiv preprint arXiv:1805.08650*, 2018.

[49] Raghunath Othayoth Nambiar and Meikel Poess. The making of tpc-ds. In *Proceedings of the 32nd international conference on Very large data bases*, pages 1049–1058. VLDB Endowment, 2006.

[50] Rimma Nehme and Nicolas 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.

[51] Ilia Pietri and Rizos Sakellariou. Scheduling data-intensive scientific workflows with reduced communication. In *Proceedings of the 30th International Conference on Scientific and Statistical Database Management*, page 25. ACM, 2018.

[52] Orestis Polychroniou, Rajkumar Sen, and Kenneth A Ross. Track join: distributed joins with minimal network traffic. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1483–1494. ACM, 2014.

[53] Iraklis Psaroudakis, Manos Athanassoulis, and Anastasia Ailamaki. Sharing data and work across concurrent analytical queries. *Proceedings of the VLDB Endowment*, 6(9):637–648, 2013.

[54] Changwoo Pyo, Kyung-Woo Lee, Hye-Kyung Han, and Gyungho Lee. Reference distance as a metric for data locality. In *Proceedings High Performance Computing on the Information Superhighway. HPC Asia'97*, pages 151–156. IEEE, 1997.

[55] Smriti R Ramakrishnan, Garret Swart, and Aleksey Urmanov. Balancing reducer skew in mapreduce workloads using progressive sampling. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 16. ACM, 2012.

[56] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy 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.

[57] Wolf Rödiger, Sam Idicula, Alfons Kemper, and Thomas Neumann. Flow-join: Adaptive skew handling for distributed joins over high-speed networks. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 1194–1205. IEEE, 2016.

[58] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proceedings of the VLDB Endowment*, 8(13):2110–2121, 2015.

[59] Alkis Simitsis, Panos Vassiliadis, Umeshwar Dayal, Anastasios Karagiannis, and Vasiliki Tziovara. Benchmarking etl workflows. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 199–220. Springer, 2009.

[60] John A Stankovic, Marco Spuri, Krithi Ramamritham, and Giorgio C Buttazzo. *Deadline scheduling for real-time systems: EDF and related algorithms*, volume 460. Springer Science & Business Media, 2012.

[61] Thomas Stöhr, Holger Märtens, and Erhard Rahm. Multi-dimensional database allocation for parallel data warehouses. In *VLDB*, volume 2000, pages 273–284, 2000.

[62] Zhuo Tang, Wen Ma, Kenli Li, and Keqin Li. A data skew oriented reduce placement algorithm based on sampling. *IEEE Transactions on Cloud Computing*, 2016.

[63] Nidhi Tiwari, Santonu Sarkar, Umesh Bellur, and Maria Indrawan. Classification framework of mapreduce scheduling algorithms. *ACM Computing Surveys (CSUR)*, 47(3):49, 2015.

[64] Pinar Tözün and Helena Kotthaus. Scheduling data-intensive tasks on heterogeneous many cores. *IEEE Data Eng. Bull.*, 42(1):61–72, 2019.

[65] Aleksandar Vitorovic, Mohammed Elseidy, and Christoph Koch. Load balancing and skew resilience for parallel joins. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 313–324. Ieee, 2016.

[66] Erci Xu, Mohit Saxena, and Lawrence Chiu. Neutrino: Revisiting memory caching for iterative data analytics. In *8th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[67] Yu Xu, Pekka Kostamaa, Xin Zhou, and Liang Chen. Handling data skew in parallel joins in shared-nothing systems. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1043–1052. ACM, 2008.

[68] Wei Yan, Yuan Xue, and Bradley Malin. Scalable and robust key group size estimation for reducer load balancing in mapreduce. In *2013 IEEE International Conference on Big Data*, pages 156–162. IEEE, 2013.

[69] Wei Yan, Yuan Xue, and Bradley Malin. Scalable load balancing for mapreduce-based record linkage. In *2013 IEEE 32nd International Performance Computing and Communications Conference (IPCCC)*, pages 1–10. IEEE, 2013.

[70] Zhengyu Yang, Danlin Jia, Stratis Ioannidis, Ningfang Mi, and Bo Sheng. Intermediate data caching optimization for multi-stage and parallel big data frameworks. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 277–284. IEEE, 2018.

[71] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278. ACM, 2010.

[72] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J Freedman. Riffle: optimized shuffle service for large-scale data analytics. In *Proceedings of the Thirteenth EuroSys Conference*, page 43. ACM, 2018.

[73] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. Cooperative scans: dynamic bandwidth sharing in a dbms. In *Proceedings of the 33rd international conference on Very large data bases*, pages 723–734. VLDB Endowment, 2007.

APPENDICES

Appendix A

Simulation DAGs

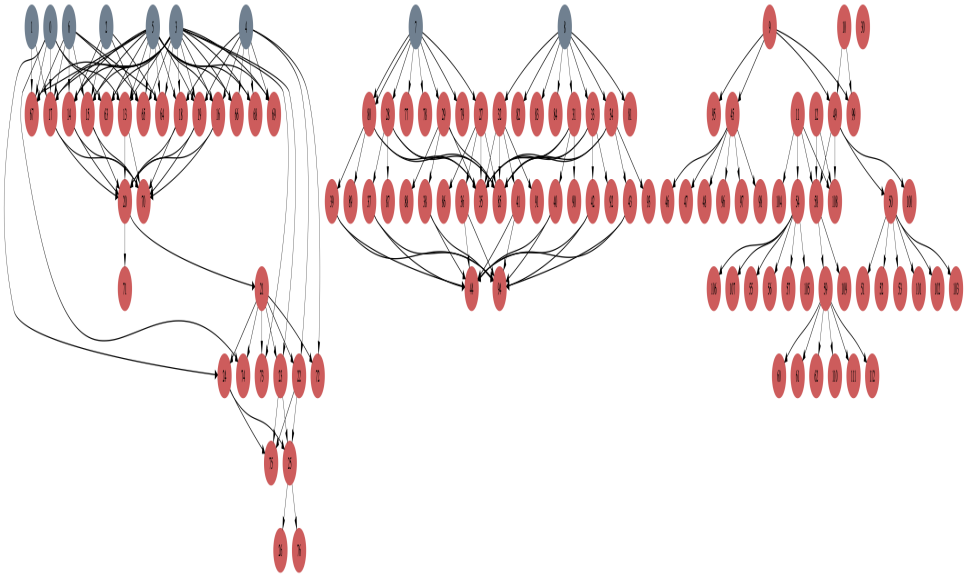


Figure A.1: DAG1-v2 based on expanded DAG1 (horizontal expansion).

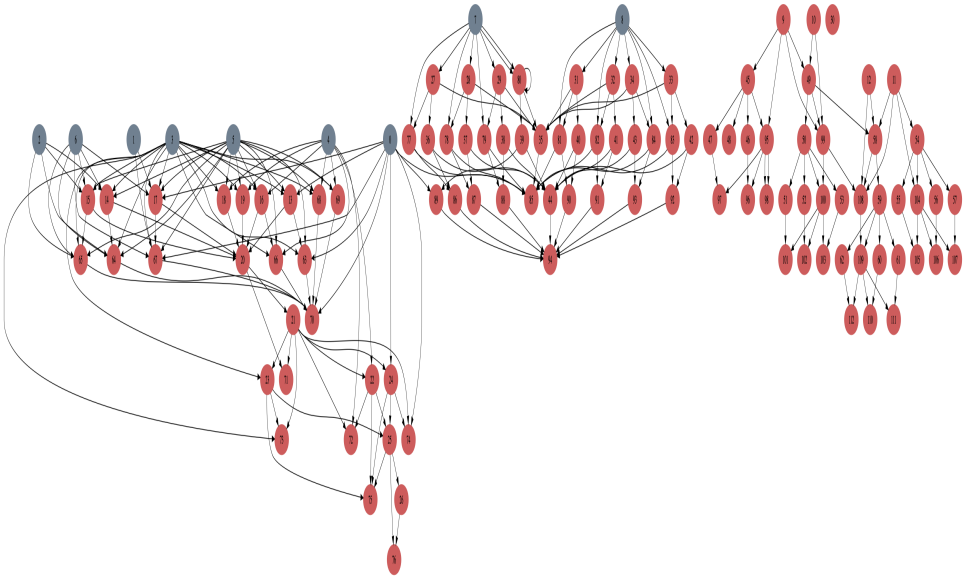


Figure A.2: DAG1-v3 based on expanded DAG1 (vertical expansion).

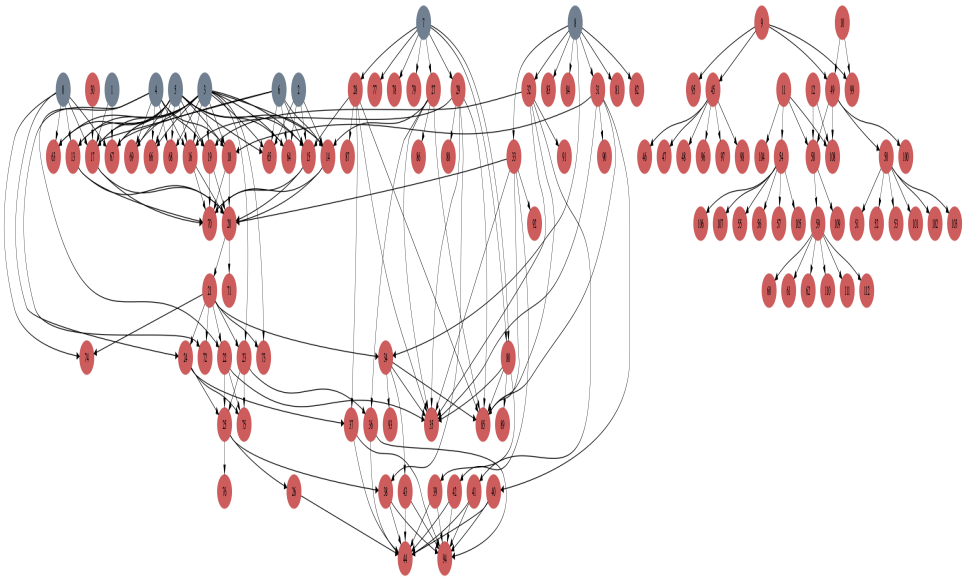


Figure A.3: DAG1-v4 based on expanded DAG1 (horizontal expansion with cross edges).

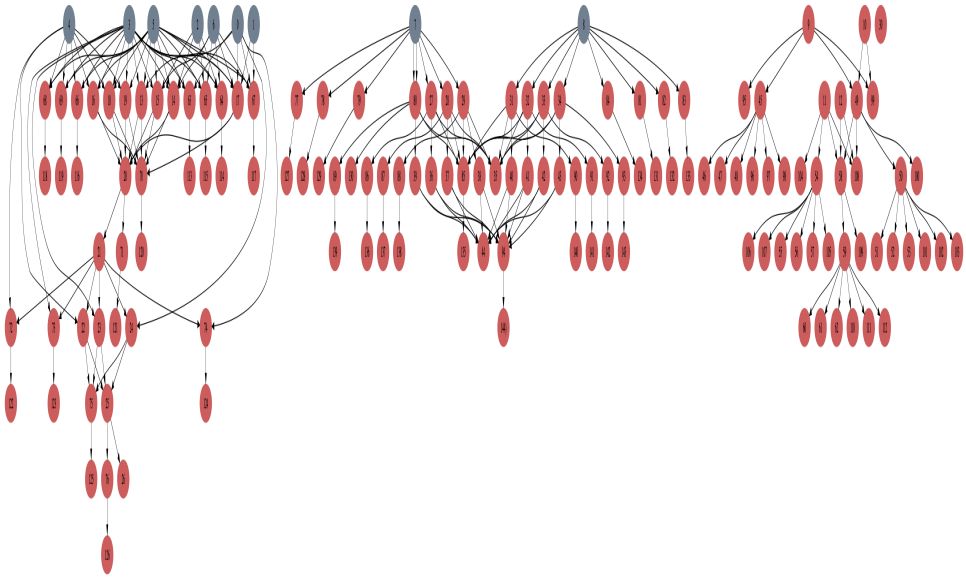


Figure A.4: DAG1-v5 based on expanded DAG1 (vertical + horizontal expansion).

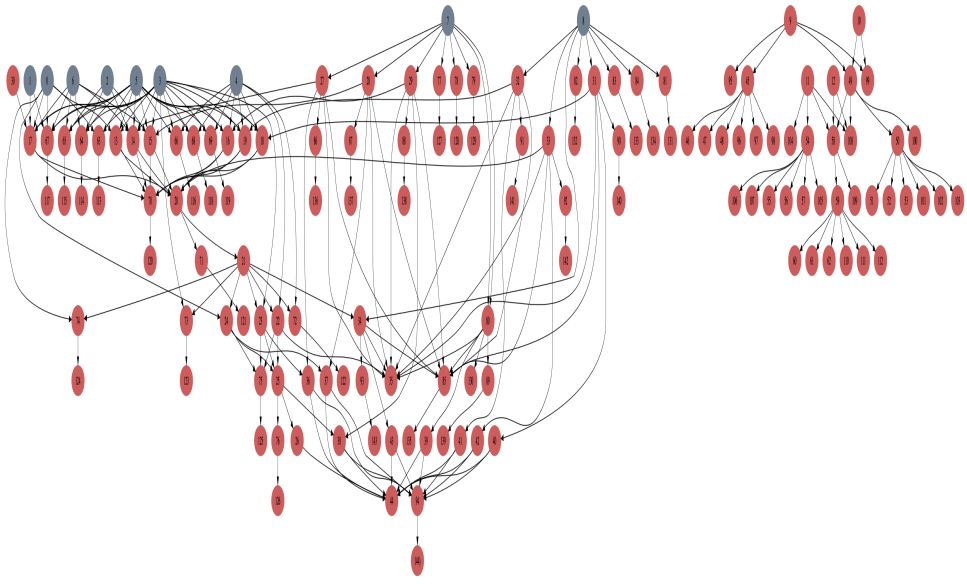


Figure A.5: DAG1-v6 based on expanded DAG1 (vertical + horizontal with cross edges).