DOT: A Matrix Model for Analyzing, Optimizing and Deploying Software for Big Data Analytics in Distributed Systems

Yin Huai[1] Rubao Lee[1] Simon Zhang[2] Cathy H. Xia[3] Xiaodong Zhang[1]

[1,3]Department of Computer Science and Engineering, The Ohio State University
[2]Department of Computer Science, Cornell University

[1]{huai,liru,zhang}@cse.ohio-state.edu [2]sz235@cornell.edu [3]xia.52@osu.edu

ABSTRACT

Traditional parallel processing models, such as BSP, are "scale up" based, aiming to achieve high performance by increasing computing power, interconnection network bandwidth, and memory/storage capacity within dedicated systems, while big data analytics tasks aiming for high throughput demand that large distributed systems "scale out" by continuously adding computing and storage resources through networks. Each one of the "scale up" model and "scale out" model has a different set of performance requirements and system bottlenecks. In this paper, we develop a general model that abstracts critical computation and communication behavior and computation-communication interactions for big data analytics in a scalable and fault-tolerant manner. Our model is called DOT, represented by three matrices for data sets (D), concurrent data processing operations (O), and data transformations (T), respectively. With the DOT model, any big data analytics job execution in various software frameworks can be represented by a specific or non-specific number of *elementary/composite DOT blocks*, each of which performs operations on the data sets, stores intermediate results, makes necessary data transfers, and performs data transformations in the end. The DOT model achieves the goals of scalability and fault-tolerance by enforcing a data-dependency-free relationship among concurrent tasks. Under the DOT model, we provide a set of optimization guidelines, which are framework and implementation independent, and applicable to a wide variety of big data analytics jobs. Finally, we demonstrate the effectiveness of the DOT model through several case studies.

Categories and Subject Descriptors

H.1 [**MODELS AND PRINCIPLES**]: Miscellaneous; H.3.4 [**INFORMATION STORAGE AND RETRIEVAL**]: Systems and Software—*Distributed systems*

General Terms

Design, Performance

Keywords

Big Data Analytics, Distributed Systems, System Modeling, System Scalability

1. INTRODUCTION

The data explosion has been accelerated by the prevalence of Internet, e-commerce and digital communication. With the rapid growth of "big data", the need for quickly and efficiently manipulating the datasets in a scalable and reliable way is unprecedentedly high. Big data analytics has become critical for industries and organizations to extract useful information from huge and chaotic data sets to support their core operations in many business and scientific applications. Meanwhile, the computing speed of commodity computers and the capacity of storage systems continue to improve while their unit prices continue to decrease. Nowadays, it is a common practice to deploy a large scale cluster with commodity computers as nodes for big data analytics.

In response to the high demand of big data analytics, several software frameworks on large and distributed cluster systems have been proposed and implemented. Representative systems include Google MapReduce [11], Hadoop [1], Dryad [17] and Pregel [22]. These system frameworks and implementations share two common goals: (1) for distributed applications, to provide a scalable and fault-tolerant system infrastructure and supporting environment; and (2) for software developers and application practitioners, to provide an easy-to-use programming model that hides the technical details of parallelization and fault-tolerance. Although the above mentioned systems have been operational to provide several major Internet services and prior studies have been conducted to improve the performance of software frameworks of big data analytics, e.g. [15] and [20], the following three issues to be addressed demand more basic and fundamental research efforts.

Behavior Abstraction: The "scale out" model of big data analytics mainly concerns two issues:

1. how to maintain the scalability, namely to ensure a proportional increase of data processing throughput as the size of the data and the number of computing nodes increase; and
2. how to provide a strong fault-tolerance mechanism in underlying distributed systems, namely to be able to quickly recover processing activities as some service nodes crash.

Currently, several software frameworks are either claimed or experimentally demonstrated that they are scalable and fault-tolerant by case studies. However, the basis and principles that jobs can be executed with scalability and fault-tolerance is not well studied. To address this issue, it is desirable to use a general model to accurately abstract the job execution behavior, because it is the most critical factor for scalability and fault-tolerance. The job execution behavior is reflected by the computation and communication behavior and computation-communication interactions (called *processing paradigm* in the rest of the paper) when the job is running on a large scale cluster.

Application Optimization: Current practice on application optimization for big data analytics jobs is underlying software framework dependent, so that optimization opportunities are only applicable to a specific software framework or a specific system implementation. Several projects have focused on this type of optimizations, e.g. [12]. A bridging model between applications and underlying software frameworks would enable us to gain opportunities of software framework and implementation independent optimization, which can enhance performance and productivity without impairing scalability and fault tolerance. With this bridging model, system designers and application practitioners can focus on a set of general optimization rules regardless of the structures of software frameworks and underlying infrastructures.

System Comparison, Simulation and Migration: The diverse requirements of various big data analytics applications cause the needs of system comparison and application migration among existing and/or new designed software frameworks. However, without a general abstract model for the processing paradigm of various software frameworks for big data analytics, it is hard to fairly compare different frameworks in several critical aspects, including scalability, fault-tolerance and framework functionality. Additionally, a general model can provide guide to building software framework simulators that are greatly desirable when designing new frameworks or customizing existing frameworks for certain big data analytics applications. Moreover, since a bridging model between applications and various underlying software frameworks is not available, application migration from one software framework to another depends strongly on programmers' special knowledge of both frameworks and is hard to do in an efficient way. Thus, it is desirable to have guidance for designing automatic tools used for application migration from one software framework to another.

All of above three issues demand a general model that bridges applications and various underlying software frameworks for big data analytics. In this paper, we propose a candidate for the general model, called DOT, which characterizes the basic behavior of big data analytics and identifies its critical issues.The DOT model also serves as a powerful tool for analyzing, optimizing and deploying software for big data analytics. Three symbols "D", "O", and "T" are three matrix representations for distributed data sets, concurrent data processing operations, and data transformations, respectively. Specifically, in the DOT model, the dataflow of a big data analytics job is represented by a *DOT expression* containing multiple root building blocks, called *elementary DOT blocks*, or their extensions, called *composite DOT blocks*. For every elementary DOT block, a matrix representation is used to abstract basic behavior of computing and communications for a big data analytics job. The DOT model eliminates the data dependency among concurrent tasks executed by concurrent data processing units (called "workers" in the rest of the paper), which is a critical requirement for the purpose of achieving scalability and fault-tolerance of a large distributed system.

We highlight our contributions in this paper as follows.

- We develop a general purpose model for analyzing, optimizing and deploying software for big data analytics in distributed systems in a scalable and fault-tolerant manner. In a concise and organized way, the model is represented by matrices that characterize basic operations and communication patterns along with interactions between computing and data transmissions during job execution.
- We show that the processing paradigm abstracted by the DOT model is scalable and fault-tolerant for big data analytics applications. Using MapReduce and Dryad as two representative software frameworks, we analyze their scalability and fault-tolerance by the DOT model. The DOT model also provides basic principles for designing scalable and fault-tolerant software frameworks for big data analytics.
- Under the DOT model, we provide a set of optimization guidelines, which are framework and implementation independent, and effective for a large scope of data processing applications. Also, we show the effectiveness of these optimization rules for complex analytical queries.

The rest part of this paper is organized as follows. Our model and its properties are introduced in Section 2. Section 3 shows that the processing paradigm of the DOT model is scalable and fault-tolerant. In Section 4, we identify optimization opportunities provided by the DOT model. Section 5 demonstrates the effectiveness of the DOT model by several case studies. Section 6 introduces related work, and Section 7 concludes the paper.

2. THE DOT MODEL

The DOT model consists of three major components to describe a big data analytics job: (1) a root building block, called *an elementary DOT block*, (2) an extended building block, called *a composite DOT block*, that is organized by a group of independent elementary DOT blocks and (3) a method that is used for building the dataflow of a big data analytics job with elementary/composite DOT blocks.

2.1 An Elementary DOT Block

An elementary DOT block is the root building block in the DOT model. It is defined as interactions of the following three entities that are supported by both hardware and software.

1. A *big data (multi-)set* that is distributed among storage nodes in a distributed system;
2. A set of *workers*, i.e. concurrent data processing units, each of which can be a computing node to process and store data; and
3. *Mechanisms* that regulate the processing paradigm of workers to interact the big data (multi-)set in two steps. First, the big data (multi-)set is processed by a number of workers concurrently. Each worker processes a part of the data and stores the output as the intermediate result. Moreover, there is no dependency among workers involved in this step. Second, all intermediate results are collected by a worker. After that, this single worker performs the last-stage data transformations based on intermediate results and stores the output as the final result.

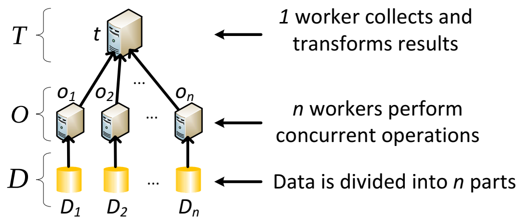
T
O
D
t
o_1 o_2 ... o_n
D_1 D_2 ... D_n
1 worker collects and transforms results
n workers perform concurrent operations
Data is divided into n parts

Figure 1: The illustration of the elementary DOT block

An elementary DOT block is illustrated by Figure 1 with a three-layer structure. The bottom layer (D-layer) represents the big data (multi-)set. A big data (multi-)set is divided into n parts (from D_1 to D_n) in a distributed system, where each part is a sub-dataset (called a *chunk* in the rest of the paper). In the middle layer (O-layer), n workers directly process the data (multi-)set and o_i is the data-processing operator associated with the ith worker. Each worker only processes a chunk (as shown by the arrow from D_i to o_i) and stores intermediate results. At the top layer (T-layer), a single worker with operator t collects all intermediate results (as shown by the arrows from o_i to t, $i = 1, \ldots, n$), then performs the last-stage data transformations based on intermediate results, and finally outputs the ending result. What must be noticed is that as shown in Figure 1, the basic rule of an elementary DOT block is that n workers in the first step are prohibited from communicating with each other and the only communication in the block is intermediate results collection shown by the arrows from o_i to t, $i = 1, \ldots, n$.

A simple example using one elementary DOT block is to calculate the sum of a large collection of integers. In this example, this large collection of integers is partitioned to n chunks. These n partitions are stored on n workers. Firstly, each worker of the n workers storing integers calculates the local sum of integers it has and stores the local sum as an intermediate result. Thus, all of operators o_1 to o_n are summation.Then, a single worker will collect all of intermediate results from n workers. Finally, this single worker will calculate the sum of all intermediate results and generate the final result. Thus, operator t is summation.

2.2 A Composite DOT Block

In an elementary DOT block, there is only one worker performing last-stage transformations to update results on intermediate results. It is natural to use multiple workers to collect intermediate results and perform last-stage data transformation, because either intermediate results tend to be huge, or the cardinality of the set of categories of intermediate results is greater than one.Thus, a group of independent elementary DOT blocks is needed, which we define as a composite DOT block, the extension of the elementary DOT block. A composite DOT block is organized by a group of independent elementary DOT blocks, which have the identical worker set of the O-layer and share the same big data (multi-)set as input divided in an identical way. Suppose that a composite DOT block is organized by m elementary DOT blocks, each of which has n workers in the O-layer. This composite DOT block will combine these elementary DOT blocks (trees) and then form a forest structure shown in Figure 2. Each worker in the O-layer will have m operators, and operator $o_{i,j}$ means that this operator originally belongs to the worker i of the jth elementary DOT block. The T-layer of this composite DOT block will have m workers and operator t_j means that it is the operator for last-stage transformations of the jth elementary DOT block.

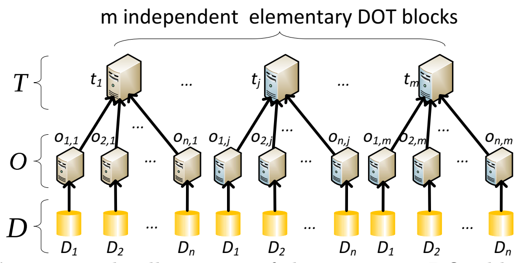
m independent elementary DOT blocks
T
O
D
t_1 ... t_j ... t_m
$o_{1,1}$ $o_{2,1}$ ... $o_{n,1}$ $o_{1,j}$ $o_{2,j}$ ... $o_{n,j}$ $o_{1,m}$ $o_{2,m}$ ... $o_{n,m}$
D_1 D_2 ... D_n D_1 D_2 ... D_n D_1 D_2 ... D_n

Figure 2: The illustration of the composite DOT block

Based on the definitions of the composite DOT block, there are three restrictions on communications among workers:

1. workers in the O-layer cannot communicate with each other;
2. workers in the T-layer cannot communicate with each other; and
3. intermediate data transfers from workers in the O-layer to their corresponding workers in the T-layer are the only communications occurring in a composite DOT block.

An example using one composite DOT block is a job used to calculate the sum of even numbers and odd numbers from a large collection of integers. Similar to the example shown in Section 2.1, this large collection of integers is partitioned to n chunks. Two elementary DOT blocks can be used to finish this job, one elementary DOT block for calculating the sum of even numbers and another for calculating the sum of odd numbers. In the elementary DOT block for calculating the sum of even/odd numbers, each worker in the O-layer will first filter out odd/even numbers and calculate the local sum of even/odd numbers as the intermediate result; then, a single worker will collect all intermediate results and calculate the sum of even/odd numbers. A composite DOT block is organized by these two elementary DOT blocks. In the T-layer of this composite DOT block, there are 2 workers. Operator t_1 can generate the sum of even numbers, while t_2 can generate the sum of odd numbers.

In a composite DOT block, the execution of its m elementary DOT blocks is flexible. For any two elementary DOT blocks of those m elementary DOT blocks, these two elementary DOT blocks can be executed concurrently or sequentially, depending on specific system implementations.

2.3 Big Data Analytics Jobs

In the DOT model, a big data analytics job is described by its dataflow, global information and halting conditions.

Dataflow of a Job: The dataflow of a big data analytics job is represented by a specific or non-specific number of elementary/composite DOT blocks. For the dataflow of a big data analytics job, any two elementary/composite DOT blocks are either dependent or independent. For two elementary/composite DOT blocks, if the result generated by a DOT block is directly or indirectly consumed by another DOT block, i.e. one DOT block must be finished before another, they are dependent, otherwise they are independent. Independent elementary/composite DOT blocks can be executed concurrently.

Global Information: Workers in an elementary/composite DOT block may need to access some lightweight global information, e.g. system configurations. In the DOT model, the global information is available in a common place, such as the coordinator or the global master of the distributed systems. Every worker in an elementary/composite DOT block can access the global information at any time.

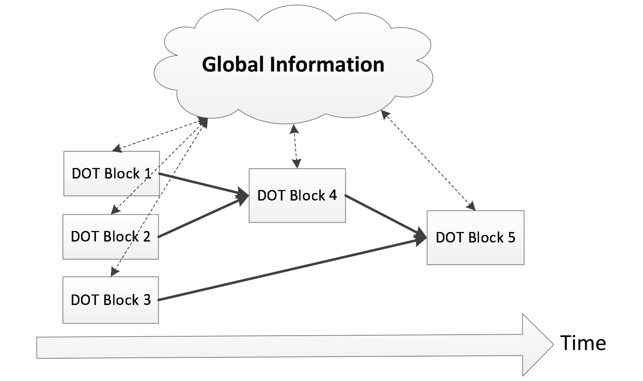

Figure 3: An example of big data analytics job described by the DOT model with five DOT blocks

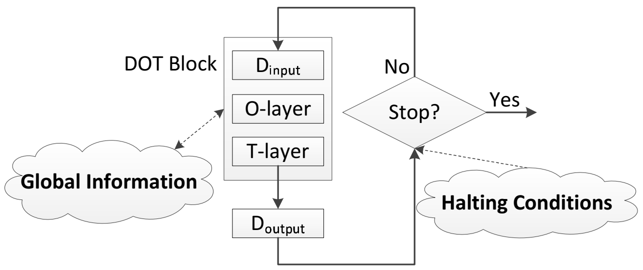

Figure 4: An iterative job described by the DOT model with non-specific number of DOT blocks

Halting Conditions: The halting conditions determine when or under what conditions a job will stop. If a job is represented by a specific number of elementary/composite DOT blocks, the job simply stops after finishing the given number of blocks. In this case, no specific halting condition is needed. For a job represented by a recurrence relation [2], one or multiple conditions must be given, so the application can determine if this job should stop. For example, convergence conditions and a maximum number of iterations are two commonly used halting conditions in iterative algorithms, such as PageRank [24] and the k-means algorithm [3].

Figure 3 shows an dataflow example described in the DOT model with five DOT blocks. In this example, DOT blocks 1, 2 and 3 process the input data first. Then, DOT block 4 will consume the results generated by DOT blocks 1 and 2. Finally, DOT block 5 will take results of DOT blocks 3 and 4 as its input and generate the final result. The global information can be accessed or updated by all of these five DOT blocks. Because this job will stop after the DOT block 5 stops, there is no halting condition needed.

Figure 4 shows an iterative job described in the DOT model with a non-specific number of DOT blocks. In this example, operators in the O-layer and T-layer of the DOT block in each iteration are the same. After every iteration, halting conditions will be evaluated. If all of the halting conditions are true, this job will stop. Similar to the previous example, the global information can be accessed or updated by the DOT block in this iterative job.

2.4 Formal Definitions

In the DOT model, the elementary/composite DOT block can be formally defined using a matrix representation. The dataflow of a big data analytic job involving a specific or non-specific number of DOT blocks is represented by an expression, called a DOT expression, which is defined by an algebra.

2.4.1 *The Elementary DOT Block*

In the DOT model, the elementary DOT block is formally defined in a matrix representation involving three matrices. The big data (multi-)set is represented by a *data vector* $\vec{D} = \begin{bmatrix} D_1 & \cdots & D_n \end{bmatrix}$, where D_i is a chunk. Symbol o_i denotes the operator of worker i in the O-layer (middle-layer of Figure 1) for processing D_i. Operators o_1 to o_n will form matrix O to represent n concurrent operations on n chunks.

The T-layer (top-layer of Figure 1) is represented by another matrix called T, which has one element representing operator t of the single worker for the last-stage transformations based on intermediate results. Note that t is an n-ary function with n inputs. The output of an elementary DOT block is still a data (multi-)set and the dimension of the data vector representing the output is 1×1. The matrix representation of the elementary DOT block is formulated as:

$$\begin{aligned}\vec{D}OT &= \begin{bmatrix} D_1 & \cdots & D_n \end{bmatrix} \begin{bmatrix} o_1 \\ o_2 \\ \vdots \\ o_n \end{bmatrix} \begin{bmatrix} t \end{bmatrix} = \left[\bigsqcup_{i=1}^{n} (o_i(D_i)) \right] \begin{bmatrix} t \end{bmatrix} \\ &= \begin{bmatrix} t(o_1(D_1), \cdots, o_n(D_n)) \end{bmatrix}.\end{aligned}$$

In the above matrix representation, matrix multiplication follows the row-column pair rule of the conventional matrix product. The multiplication of corresponding elements of the two matrices is defined as: (1) a multiplication between a data chunk D_i and an operator f (f can either be the operator in matrix O or the one in matrix T) means to apply the operator on the chunk, represented by $f(D_i)$; (2) multiplication between two operators (e.g. $f_1 \times f_2$) means to form a composition of operators (e.g., $f = f_2(f_1)$). In contrast to the original matrix summation, in the DOT model, the summation operator $\sum$ is replaced by a group operator $\bigsqcup$. The operation $\bigsqcup_{i=1}^{n} (f_i(D_i)) = (f_1(D_1), \cdots, f_n(D_n))$ means to compose a collection of data sets $f_1(D_1)$ to $f_n(D_n)$. It is not required that all elements of the collection locate in a single place.

2.4.2 *The Composite DOT Block*

Given m elementary DOT blocks $\vec{D}O_1T_1$ to $\vec{D}O_mT_m$, a composite DOT block $\vec{D}OT$ is formulated as:

$$\begin{aligned}&\biguplus_{j=1}^{m} (\vec{D}O_jT_j) = \vec{D}O_1T_1 \uplus \ldots \uplus \vec{D}O_mT_m = \vec{D}O_{composite}T_{composite} \\ &= \begin{bmatrix} D_1 & \cdots & D_n \end{bmatrix} \begin{bmatrix} o_{1,1} & o_{1,2} & \cdots & o_{1,m} \\ o_{2,1} & o_{2,2} & \cdots & o_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ o_{n,1} & o_{n,2} & \cdots & o_{n,m} \end{bmatrix} \begin{bmatrix} t_1 & 0 & \cdots & 0 \\ 0 & t_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & t_m \end{bmatrix} \\ &= \left[\bigsqcup_{i=1}^{n} (o_{i,1}(D_i)) \quad \cdots \quad \bigsqcup_{i=1}^{n} (o_{i,m}(D_i)) \right] \begin{bmatrix} t_1 & 0 & \cdots & 0 \\ 0 & t_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & t_m \end{bmatrix} \\ &= \begin{bmatrix} t_1(o_{1,1}(D_1), \cdots, o_{n,1}(D_n)) & \cdots & t_m(o_{1,m}(D_1), \cdots, o_{n,m}(D_n)) \end{bmatrix},\end{aligned}$$

where the operator $\biguplus$ means to construct a composite DOT block from m elementary DOT blocks by: (1) using one data vector that includes all chunks used by the m elementary DOT blocks and each chunk occurs exactly once; (2) putting matrix O_i (column vector) in the ith column of the matrix $O_{composite}$; and (3) putting the single element of matrix T_i in the place (i,i) of matrix $T_{composite}$. The output of a composite DOT block is still a data (multi-)set and the dimension of the data vector representing the output is $1 \times m$. Details about operator $\biguplus$ are given in section 2.4.3. Moreover, operator "0" will produce an empty set, i.e. given a chunk D, $0(D) = \emptyset$ and $D \bigsqcup \emptyset = D$.

2.4.3 An Algebra for Representing the Dataflow of Big Data Analytics Jobs

With the definition of elementary and composite DOT blocks, the dataflow of a big data analytics job can be systematically represented by a combination of multiple elementary and/or composite DOT blocks. We introduce an algebra among elementary and composite DOT blocks. With this algebra, a big data analytics job can be represented by an expression, called a *DOT expression*. For example, a job can be composed by three composite DOT blocks, $\vec{D}_1O_1T_1$, $\vec{D}_2O_2T_2$ and $\vec{D}_3O_3T_3$, where the results of $\vec{D}_1O_1T_1$ and $\vec{D}_2O_2T_2$ are input of $\vec{D}_3O_3T_3$. With the algebra defined in this section, the DOT expression of this job is

$$(\vec{D}_1O_1T_1 \oplus \vec{D}_2O_2T_2)O_3T_3,$$

where $\vec{D}_3 = (\vec{D}_1O_1T_1 \bigoplus \vec{D}_2O_2T_2)$.

In the DOT model, an elementary/composite DOT block can be viewed as a data vector. In the algebra for DOT expressions, operands are data vectors, elementary DOT blocks and composite DOT blocks. The algebra defines two basic interactions between two DOT blocks, as shown by the above simple example, the interaction between two independent DOT blocks and that between two dependent DOT blocks. If two DOT blocks do not have data dependency, they are independent DOT blocks; otherwise, they are dependent DOT blocks. There are two operations to define the interaction among independent DOT blocks for DOT expressions:

- $\bigoplus$: For two data vectors $\vec{D}_1 = [D_{1,1} \quad D_{1,2} \quad \cdots \quad D_{1,n}]$ and $\vec{D}_2 = [D_{2,1} \quad D_{2,2} \quad \cdots \quad D_{2,m}]$,

  $$\begin{aligned}&\vec{D}_1 \oplus \vec{D}_2 = \begin{bmatrix}\vec{D}_1 & \vec{D}_2\end{bmatrix}\\ &= \begin{bmatrix}D_{1,1} & D_{1,2} & \cdots & D_{1,n} & D_{2,1} & D_{2,2} & \cdots & D_{2,m}\end{bmatrix}.\end{aligned}$$

  The operator $\bigoplus$ simply collects all chunks and forms a new data vector of a higher dimension. For two elementary/composite DOT blocks DOT blocks $\vec{D}_1O_1T_1$ and $\vec{D}_2O_2T_2$,

  $$\vec{D}_1O_1T_1 \oplus \vec{D}_2O_2T_2 = \begin{bmatrix}\vec{D}_1 & \vec{D}_2\end{bmatrix}\begin{bmatrix}O_1 & 0\\ 0 & O_2\end{bmatrix}\begin{bmatrix}T_1 & 0\\ 0 & T_2\end{bmatrix}.$$

  For a data vector $\vec{D}_1$ and an elementary/composite DOT block $\vec{D}_2O_2T_2$,

  $$\vec{D}_1 \oplus \vec{D}_2O_2T_2 = \begin{bmatrix}\vec{D}_1 & \vec{D}_2\end{bmatrix}\begin{bmatrix}I & 0\\ 0 & O_2\end{bmatrix}\begin{bmatrix}I & 0\\ 0 & T_2\end{bmatrix},$$

  of which I is an identity matrix. Here, an identity matrix is a matrix with the operator "1"(s) on the main diagonal and the operator "0"(s) elsewhere. For a given chunk D_i, operator "1" is defined as $1(D_i) = D_i$. Thus, for a given $1 \times n$ data vector $\vec{D}$ and a $n \times n$ identity matrix I, $\vec{D}I = \vec{D}$.
- $\biguplus$: For two data vectors $\vec{D}_1 = [D_{1,1} \quad D_{1,2} \quad \cdots \quad D_{1,n}]$ and $\vec{D}_2 = [D_{2,1} \quad D_{2,2} \quad \cdots \quad D_{2,m}]$,

  $$\begin{aligned}&\vec{D}_1 \uplus \vec{D}_2\\ &= \begin{bmatrix}D_{1,1} & D_{1,2} & \cdots & D_{1,n} & D_{2,q_1} & D_{2,q_2} & \cdots & D_{2,q_k}\end{bmatrix},\end{aligned}$$

  where $k \leq m$, $1 \leq q_1 < q_2 < \ldots < q_k - 1 < q_k \leq m$ and for each i $(1 \leq i \leq k)$, $D_{2,q_i} \notin \bigcup_{1 \leq i \leq n} D1_i$. Like the operator $\bigoplus$, operator $\biguplus$ also combines multiple data vectors into a single data vector of higher dimension, and to combine multiple independent elementary/composite DOT blocks into one composite DOT block. However, after the operation of $\biguplus$, each common chunk used by multiple independent elementary/composite DOT blocks occurs exactly once. For matrices O and T in the new composite DOT block, operators will simply be assigned to match the new data vector. We use two examples to explain $\biguplus$ operators. Consider that the data vectors in these two examples are $\vec{D}_1 = [D_1 \quad D_2]$ and $\vec{D}_2 = [D_1 \quad D_3]$. For two data vectors $\vec{D}_1$ and $\vec{D}_2$, $\vec{D}_1 \biguplus \vec{D}_2 = [D_1 \quad D_2 \quad D_3]$. Consider two elementary/composite DOT blocks, $\vec{D}_1O_1T_1$ and $\vec{D}_2O_2T_2$, of which

  $$O_1 = \begin{bmatrix}o_{1,1,1} & o_{1,1,2}\\ o_{1,2,1} & o_{1,2,2}\end{bmatrix} \qquad T_1 = \begin{bmatrix}t_{1,1} & 0\\ 0 & t_{2,2}\end{bmatrix}$$

  $$O_2 = \begin{bmatrix}o_{2,1,1} & o_{2,1,2}\\ o_{2,2,1} & o_{2,2,2}\end{bmatrix} \qquad T_2 = \begin{bmatrix}t_{2,1} & 0\\ 0 & t_{2,2}\end{bmatrix}.$$

  The result of $\vec{D}_1O_1T_1 \biguplus \vec{D}_2O_2T_2$ is:

  $$\begin{aligned}&\vec{D}_1O_1T_1 \uplus \vec{D}_2O_2T_2\\ &= \begin{bmatrix}D_1 & D_2 & D_3\end{bmatrix}\begin{bmatrix}o_{1,1,1} & o_{1,1,2} & o_{2,1,1} & o_{2,1,2}\\ o_{1,2,1} & o_{1,2,1} & 0 & 0\\ 0 & 0 & o_{2,2,1} & o_{2,2,2}\end{bmatrix}\\ &\begin{bmatrix}t_{1,1} & 0 & 0 & 0\\ 0 & t_{1,2} & 0 & 0\\ 0 & 0 & t_{2,1} & 0\\ 0 & 0 & 0 & t_{2,2}\end{bmatrix}.\end{aligned}$$

For two dependent DOT blocks, these two DOT blocks will be chained together according to the data dependency, e.g. $(DO_1T_1)O_2T_2$. Two dependent DOT blocks can be merged into one DOT block in a certain condition, described by the property of *conditional associativity*.

PROPERTY 2.1. **conditional associativity:** *If matrix O in a composite DOT block is a diagonal matrix, this composite DOT block can be merged into matrix T of its preceding composite DOT block or into matrix O of its succeeding composite DOT block.*

For example, there are two dependent DOT blocks, described by

$$\begin{aligned}(DO_1T_1)O_2T_2 = (\begin{bmatrix}D_1 & D_2\end{bmatrix}\begin{bmatrix}o_{1,1,1} & o_{1,1,2}\\ o_{1,2,1} & o_{1,2,2}\end{bmatrix}\begin{bmatrix}t_{1,1} & 0\\ 0 & t_{2,2}\end{bmatrix})\\ \begin{bmatrix}o_{2,1,1} & 0\\ 0 & o_{2,2,2}\end{bmatrix}\begin{bmatrix}t_{2,1} & 0\\ 0 & t_{2,2}\end{bmatrix}.\end{aligned}$$

Based on the property of *conditional associativity*, these two DOT blocks can be merged into one DOT block, which is

$$\begin{bmatrix}D_1 & D_2\end{bmatrix}\begin{bmatrix}o_{1,1,1} & o_{1,1,2}\\ o_{1,2,1} & o_{1,2,2}\end{bmatrix}\begin{bmatrix}t_{2,1}(o_{2,1,1}(t_{1,1})) & 0\\ 0 & t_{2,2}(o_{2,2,2}(t_{2,2}))\end{bmatrix}.$$

With the above algebra, the dataflow of a big data analytics job can be described by a DOT expression composed by data vector, elementary/composite DOT block, operator $\bigoplus$ and/or operator $\biguplus$. A context-free grammar to derive a DOT expression is shown in Figure 5.

⟨DOTexpression⟩→⟨dataVector⟩
⟨dataVector⟩→⟨D⟩|(⟨D⟩)|⟨dataVector⟩⟨O⟩⟨T⟩|
(⟨dataVector⟩⟨O⟩⟨T⟩)|
⟨D⟩⊕⟨dataVector⟩|(⟨D⟩⊕⟨dataVector⟩)|
⟨D⟩⊎⟨dataVector⟩|(⟨D⟩⊎⟨dataVector⟩)|
⟨D⟩⟨O⟩⟨T⟩⊕⟨dataVector⟩|(⟨D⟩⟨O⟩⟨T⟩⊕⟨dataVector⟩)|
⟨D⟩⟨O⟩⟨T⟩⊎⟨dataVector⟩|(⟨D⟩⟨O⟩⟨T⟩⊎⟨dataVector⟩)
⟨D⟩→*a data vector* $\vec{D}$
⟨O⟩→*a O matrix*
⟨T⟩→*a T matrix*

Figure 5: The context-free grammar of the DOT expression

If the dataflow of a job needs a non-specific number of elementary/composite DOT blocks, such as an iterative algorithm for a PageRank evaluation [24], the dataflow should be described by a recurrence relation. The general format of a DOT expression representing a recurrence relation is: $\vec{D}(t) = \bigoplus_{k=i}^{j} \vec{D}(k)O(k+1)T(k+1), 0 \le i \le j < t$. For example, a recurrence relation $\vec{D}(t) = \vec{D}(t-1)O(t)T(t)$ means at time t, the result is generated by $\vec{D}(t-1)O(t)T(t)$, of which $\vec{D}(t-1)$ is the output at time $t-1$.

With the algebra used for representing the dataflow of a big data analytics job as a DOT expression, the job can be described by a DOT expression, global information and halting conditions.

2.5 Restrictions

To make the DOT model effective in practice, we add several restrictions.

The power of workers: In big data analytics, it is reasonable to assume there is no single worker that can store the entire data (multi-)set. Thus, similar to [18], we restrict that the storage capacity of a single worker is sublinear to the size of data. We do not set any restriction on the computation power of a single worker, which mainly determines the elapsed time of an operator on a single worker.

The number of workers: In big data analytics, it is reasonable to assume the total number of workers in matrix O or matrix T is much smaller than the size of the data. For matrices O and T in a DOT block, we assume that the total number of workers is sublinear to the size of data.

3. SCALABILITY AND FAULT-TOLERANCE

Scalability and fault-tolerance are two critical issues in big data analytics. In this section, we show that the processing paradigm of the DOT model is scalable and fault-tolerant.

3.1 Scalability

There are five concepts needed to be defined at first.

Definition 3.1. **A minimum chunk** *is defined as a chunk that cannot be further divided into two chunks. A minimum chunk represents a operator-specific collection of data that has to be processed by a single operator at a single place during a continuous time interval.*

Definition 3.2. **A basic concurrent data processing unit** *is defined as a worker which only processes a minimum chunk.*

Definition 3.3. **A processing step** *is defined as a set of data operations being executed on a fixed number of concurrent workers during a continuous time interval.*

Definition 3.4. **Scalability of a job:** *The scalability of a job is defined by two aspects:*

1. *with a fixed input data size N_0, the throughput of this job linearly increases as the number of workers involved in each processing step of this job linearly increases at ratio γ (an integer). This linear increase of the throughput stops when there exists a processing step, where each worker is reduced to a basic concurrent data processing unit; and*
2. *with an initial input data size N_0, as the input data size increases linearly at ratio ω and thus the input data size is ωN_0, the elapsed time of this job can be kept constant by increasing the number of workers involved in this job at ratio γ.*

Definition 3.5. **Scalability of a processing paradigm:** *Given a job class A that all job of this class satisfy two conditions: (1) the time complexity of operations on every worker is $\Theta(n)$, where n is the size of the input data; and (2) the input data of a processing step can be equally divided into multiple data sub-sets. Also, suppose that the point-to-point throughput of the network transfer between two workers will not drop when adding more workers.*

A processing paradigm is scalable if any job of the class A represented by this processing paradigm is scalable.

With the above five definitions, we will show that the processing paradigm of the DOT model is scalable.

Lemma 3.1. *The processing paradigm of the DOT model is scalable.*

Proof. Firstly, we prove that any job of the class A represented by a single DOT block is scalable. Consider a job represented by a single DOT block, which is

$$\vec{D}OT = \begin{bmatrix} D_1 & \cdots & D_n \end{bmatrix} \begin{bmatrix} o_{1,1} & \cdots & o_{1,m} \\ \vdots & \ddots & \vdots \\ o_{n,1} & \cdots & o_{n,m} \end{bmatrix} \begin{bmatrix} t_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & t_m \end{bmatrix},$$

where n and m are the initial number of workers in matrices O and T, respectively. Based on the definition of a processing step, there are two processing steps represented by matrices O and T, respectively. The throughput of this DOT block is represented as $Throughput = \frac{\omega N_0}{T_{elapsed}}$. The $T_{elapsed}$ is the elapsed time of this DOT block and

$$T_{elapsed} = \omega t_{O,n} + \omega t_{T,m} + f(\omega N_0, n, m),$$

of which $t_{O,n}$ is the elapsed time of matrix O with n workers (the longest elapsed time of operation execution among all workers in matrix O), $t_{T,m}$ is the elapsed time of matrix T with m workers (the longest elapsed time of operation execution among all workers in matrix T), and $f(\omega N_0, n, m)$ is the elapsed time of network transfer from workers in matrix O to workers in matrix T with the input data size ωN_0, n workers in O and m workers in T.

If the input data size is fixed as N_0, ω will be constant value 1. The linear increase of workers by a factor of γ means that matrix O will have γn workers and T will have γm workers. This increase of worker can be done as follows: for every 3-tuple $(D_i, o_{i,j}, t_j)$, where $D_i \in \vec{D}$, $o_{i,j} \in O$ and $t_j \in T$

- D_i in $\vec{D}$ will be replaced by $D_1^i, \cdots, D_\gamma^i$;

- $o_{i,j}$ in matrix O will be replaced by

$$\begin{matrix} o_{1,1}^{i,j} & \cdots & o_{1,\gamma}^{i,j} \\ \vdots & \ddots & \vdots \\ o_{\gamma,1}^{i,j} & \cdots & o_{\gamma,\gamma}^{i,j} \end{matrix};$$

- t_j in matrix T will be replaced by

$$\begin{matrix} t_1^j & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & t_\gamma^j \end{matrix}.$$

Here, for a given γ, D_i will be equally divided into $D_1^i, \cdots, D_\gamma^i$. For example, for a composite DOT block

$$\vec{D}OT = \begin{bmatrix} D_1 & D_2 \end{bmatrix} \begin{bmatrix} o_{1,1} & o_{1,2} \\ o_{2,1} & o_{2,2} \end{bmatrix} \begin{bmatrix} t_1 & 0 \\ 0 & t_2 \end{bmatrix},$$

with 2 workers in matrix O, 2 workers in matrix T, and a worker-increase factor γ of 2, the new DOT block $\vec{D}'O'T'$ representing 4 workers in O and 4 workers in T will be

$$\vec{D}'O'T' = \begin{bmatrix} D_1^1 & D_2^1 & D_1^2 & D_2^2 \end{bmatrix} \begin{bmatrix} o_{1,1}^{1,1} & o_{1,2}^{1,1} & o_{1,1}^{1,2} & o_{1,2}^{1,2} \\ o_{2,1}^{1,1} & o_{2,2}^{1,1} & o_{2,1}^{1,2} & o_{2,2}^{1,2} \\ o_{1,1}^{2,1} & o_{1,2}^{2,1} & o_{1,1}^{2,2} & o_{1,2}^{2,2} \\ o_{2,1}^{2,1} & o_{2,2}^{2,1} & o_{2,1}^{2,2} & o_{2,2}^{2,2} \end{bmatrix}$$

$$\begin{bmatrix} t_1^1 & 0 & 0 & 0 \\ 0 & t_2^1 & 0 & 0 \\ 0 & 0 & t_1^2 & 0 \\ 0 & 0 & 0 & t_2^2 \end{bmatrix}.$$

With this method, the elapsed time of matrix O, T and network transfer will decrease linearly, i.e. $t_{O,\gamma n} = \frac{t_{O,n}}{\gamma}$, $t_{T,\gamma n} = \frac{t_{T,n}}{\gamma}$ and $f(N_0, \gamma n, \gamma m) = \frac{f(N_0,n,m)}{\gamma}$. Thus, the elapsed time of this DOT block will be

$$T_{elapsed} = \frac{t_{O,n}}{\gamma} + \frac{t_{T,m}}{\gamma} + f(N_0, \gamma n, \gamma m) = \frac{1}{\gamma}C,$$

of which C is constant value to γ. So, the throughput of this DOT block is

$$Throughput = \frac{N_0}{C}\gamma,$$

where N_0 and C are constant values to γ. When every worker in either O or T only processes the minimum chunk defined by the application, i.e. every worker is reduced to a basic concurrent data processing unit, the number of workers of this DOT block cannot be further added to gain linear increase of the throughput. Thus, with a fixed input data size, the throughput of the DOT block linearly increases as the number of workers linearly increases until there is a processing step, each worker of which is reduced to a basic concurrent data processing unit.

If the the data size increases linearly, the number of workers in two matrices can be scaled with the same method provided above and the elapsed time of network transfer will be proportional to the ratio ω to γ, i.e. $f(\omega N_0, \gamma n, \gamma m) = \frac{\omega}{\gamma} f(N_0, n, m)$. So the elapsed time of the DOT block is

$$T_{elapsed} = \omega\frac{t_{O,n}}{\gamma} + \omega\frac{t_{T,m}}{\gamma} + f(\omega N_0, \gamma n, \gamma m) = \frac{\omega}{\gamma}C,$$

of which C is a constant value to ω and γ. Thus, if $\frac{\omega}{\gamma}$ is a constant, the $T_{elapsed}$ will be a constant value and then the job elapsed time can be kept constant by increasing the number of workers linearly.

For a big data analytics job of the class A expressed by a DOT expression, a matrix O or T will represent a processing step. The conclusion that a job of class A represented by a DOT expression is scalable is a corollary of the fact that a job of class A represented by a DOT block is scalable.

Thus, the processing paradigm of the DOT model is scalable. $\square$

3.2 Fault-Tolerance

Here are four basic concepts to be used in the rest of this sub-section.

Definition 3.6. **Initial input data** *is defined as the available data to be accessed in an archived storage at the start of a set of concurrent operations.*

Definition 3.7. **Runtime data** *is defined as the data generated during the runtime of a set of concurrent operations.*

Definition 3.8. **Fault-tolerance of a job:** *Assuming the initial input data of every operator is always available, a big data analytics job is fault-tolerant if it can finish and generate a correct result when worker failures happen. The result of a job involving worker failures is correct if and only if the result is identical to one of the possible results generated by this job without worker failures.*

Definition 3.9. **Fault-tolerance of a processing paradigm:** *A processing paradigm is fault-tolerant if any job represented by this processing paradigm is fault-tolerant.*

With the above four definitions, we will show that the processing paradigm of the DOT model is fault-tolerant.

Lemma 3.2. *The processing paradigm of the DOT model is fault-tolerant.*

Proof. Consider a job represented by a DOT block with n workers in matrix O and m workers in matrix T, and consider that there are k_1 $(1 \leq k_1 \leq n)$ failed workers in O and k_2 $(1 \leq k_2 \leq m)$ failed workers in T during the job execution. Because the DOT model enforces that there is no data dependency among peer workers in matrix O and those in matrix T, which is reflected by the interactions between matrices O and T, new workers to substitute the services of failed workers will only be started to re-execute operators running on the failed k_1 workers in O and k_2 workers in T. Since the initial input data of every operator is always available, there is no difference on functionalities between a substitute worker and an original worker. Thus, a DOT block is fault-tolerant.

For a big data analytics job expressed by a DOT expression composed by a specific or non-specific DOT blocks, because any DOT block of this DOT expression is fault-tolerant, the job expressed by this DOT expression is fault-tolerant.

Since any job represented by the processing paradigm of the DOT model is fault-tolerant, the processing paradigm of the DOT model is fault-tolerant. $\square$

4. OPTIMIZATION RULES

We identify three types of framework and implementation independent optimization rules under the DOT model.

These three types of rules can be applied on various frameworks to optimize the performance of big data analytics jobs. To be concise and without loss of generality, we use two composite DOT blocks as examples to explain these three types of optimization rules in this section. These two composite DOT blocks are defined as follows:

$$block_i = \vec{D}_i O_i T_i = \begin{bmatrix} D_{i,1} & D_{i,2} \end{bmatrix} \begin{bmatrix} o_{i,1,1} & o_{i,1,2} \\ o_{i,2,1} & o_{i,2,2} \end{bmatrix} \begin{bmatrix} t_{i,1} & 0 \\ 0 & t_{i,2} \end{bmatrix},$$

$i = 1,\ 2$.

4.1 Preliminary Definitions

Here are four definitions to be used in the rest of this section.

Definition 4.1. **Equivalence between two data vectors:** *Two data vectors* $\vec{D1} = \begin{bmatrix} D1_1 & \cdots & D1_n \end{bmatrix}$ *and* $\vec{D2} = \begin{bmatrix} D2_1 & \cdots & D2_m \end{bmatrix}$ *are equivalent, denoted as* $\vec{D1} \equiv \vec{D2}$, *if and only if* $\bigcup_{1 \le i \le n} D1_i = \bigcup_{1 \le i \le m} D2_i$.

Definition 4.2. **Equivalence between two DOT blocks:** *Two elementary/composite DOT blocks* $\vec{Do1} = \vec{D1} O_1 T_1$ *and* $\vec{Do2} = \vec{D2} O_2 T_2$ *are equivalent, denoted as* $\vec{D1} O_1 T_1 \equiv \vec{D2} O_2 T_2$, *if and only if* $\vec{D1}$ *and* $\vec{D2}$ *are equivalent, and* $\vec{Do1}$ *and* $\vec{Do2}$ *are equivalent.*

Definition 4.3. **Equivalence between two DOT expressions:** *Two DOT expressions* EXP_1 *and* EXP_2, *are equivalent, denoted as* $EXP_1 \equiv EXP_2$, *if and only if*

$$\bigoplus_{1 \le i \le n} \vec{D1}_{input_i} \equiv \bigoplus_{1 \le i \le m} \vec{D2}_{input_i} \text{ and}$$

$$\bigoplus_{1 \le i \le n} \vec{D1}_{output_i} \equiv \bigoplus_{1 \le i \le m} \vec{D2}_{output_i},$$

of which (1) $\vec{D1}_{input_i}$ $(i = 1, \ldots, n)$ *and* $\vec{D2}_{input_i}$ $(i = 1, \ldots, m)$ *are the original input of* EXP_1 *and* EXP_2, *respectively, and (2)* $\vec{D1}_{output_i}$ $(i = 1, \ldots, n)$ *and* $\vec{D2}_{output_i}$ $(i = 1, \ldots, m)$ *are the final output of* EXP_1 *and* EXP_2, *respectively.*

Definition 4.4. **Associative-decomposable operators:** *As defined in [28], an operator* H *is associative-decomposable if it can be decomposed into two operators* F *and* C *so that:*

1. $\forall D_1, D_2, H(D_1 \oplus D_2) = C(F(D_1) \oplus F(D_2))$;
2. F *and* C *are commutative, i.e.* $\forall D_1, D_2, F(D_1 \oplus D_2) = F(D_2 \oplus D_1), C(D_1 \oplus D_2) = C(D_2 \oplus D_1)$;*and*
3. C *is associative, i.e.* $\forall D_1, D_2, D_3, C(C(D_1 \oplus D_2) \oplus D_3) = C(D_1 \oplus C(D_2 \oplus D_3))$.

4.2 Substituting Expensive Remote Data Transfers with Low-Cost Local Computing

In the DOT model, there is no restriction on the amount of data transfers in the communication phase, because the amount of data transfers is application dependent. Some applications may need to transfer a large amount of data, while others may only need to transfer a small amount. However, considering the fact that "Computing is free" and "Network traffic is expensive" [14], application designers should minimize the amount of remote data transfers in practice.

In an elementary/composite DOT block, an associative-decomposable operator in matrix T can transfer partial operations to matrix O. Associative-decomposable operators which originally belong to the same elementary DOT block, i.e. these operators locate in the same column of the matrix O, can transfer common-partial operations to matrix T. With the definition of equivalence between two DOT blocks, an elementary/composite DOT block can be transformed to an equivalent one. For example, if in $block_1$, operator $t_{1,1}$ is associative-decomposable and can be decomposed into two operators F and C, the equivalent composite DOT block of $block_1$ will be:

$$block_1 \equiv block_1' = \begin{bmatrix} D_{1,1} & D_{1,2} \end{bmatrix} \begin{bmatrix} F(o_{1,1,1}) & o_{1,1,2} \\ F(o_{1,2,1}) & o_{1,2,2} \end{bmatrix} \begin{bmatrix} C & 0 \\ 0 & t_{1,2} \end{bmatrix}.$$

Moreover, if the amount of intermediate results generated by $F(o_{1,1,1})$ and $F(o_{1,2,1})$ is less than that generated by $o_{1,1,1}$ and $o_{1,2,1}$, the total amount of data transfered will decrease. Thus, the expensive remote data transfers are replaced by an additional low cost local computations F. For example, considering an application that counts the number of occurrences of each word in a large number of documents or web pages. A simple way to implement this application is to let operators in matrix O emit a pair *(word, 1)* for every word. Then, we use a hash partitioning function to decide which worker in matrix T a pair will be sent to. Finally, workers in matrix T calculate the number of occurrences for each word. There is a aggregation operator F that will accumulate the value for pairs with the same key, i.e. generate a pair *(word, local_count)* for each word. If this operator F is applied locally to all operators in matrix O, each worker in the new matrix O only emits pairs with accumulated number of occurrences for each word. By introducing the operator F, the total amount of intermediate results transferred through networks will be significantly reduced and thus the execution time of this job will decrease. In the MapReduce framework, this operator F can be implemented by the *Combiner Function* [11].

4.3 Exploiting Sharing Opportunities among Composite DOT Blocks

Exploiting sharing opportunities has been studied by the database community in different contexts, e.g. [21], [8] and [9].Based on the DOT model, we identify three types of sharing opportunities among two independent composite DOT blocks, $block_1$ and $block_2$, and introduce how to exploit these three types of sharing opportunities. In this sub-section, we consider that the dataflow of a job is $block_3 = block_1 \uplus block_2$.

Sharing Common Chunks: If $block_1$ and $block_2$ share a common chunk, e.g. suppose $D_{1,1} = D_{2,1}$, the result of $block_1 \uplus block_2$ will be:

$$\begin{aligned} block_3 &= block_1 \uplus block_2 \\ &= \begin{bmatrix} D_{1,1} & D_{1,2} & D_{2,2} \end{bmatrix} \begin{bmatrix} o_{1,1,1} & o_{1,1,2} & o_{2,1,1} & o_{2,1,2} \\ o_{1,2,1} & o_{1,2,2} & 0 & 0 \\ 0 & 0 & o_{2,2,1} & o_{2,2,2} \end{bmatrix} \\ &\quad \begin{bmatrix} t_{1,1} & 0 & 0 & 0 \\ 0 & t_{1,2} & 0 & 0 \\ 0 & 0 & t_{2,1} & 0 \\ 0 & 0 & 0 & t_{2,2} \end{bmatrix}. \end{aligned}$$

Because the decision of how to execute operators in a worker is flexible, when operators in a worker share a single scan of the chunk, sharing common chunks will reduce the local disk I/O.

Sharing Common Operations in Matrix O: After sharing the common chunks, if two operators in different columns of matrix O have common operations, these two columns can be merged. Suppose that $o_{1,1,1}$ and $o_{2,1,1}$ are common operations, so a new operator o' can represent both operators of $o_{1,1,1}$ and $o_{2,1,1}$. Thus, $block_3$ will be transformed to a new composite block $block_3'$, which is equivalent to $block_3$. The DOT block $block'_3$ will be

$$block'_3 = \begin{bmatrix} D_{1,1} & D_{1,2} & D_{2,2} \end{bmatrix} \begin{bmatrix} o' & o_{1,1,2} & o_{2,1,2} \\ o_{1,2,1} & o_{1,2,2} & 0 \\ o_{2,2,1} & 0 & o_{2,2,2} \end{bmatrix} \begin{bmatrix} (t_{1,1}, t_{2,1}) & 0 & \\ 0 & t_{1,2} & 0 \\ 0 & 0 & t_{2,2} \end{bmatrix},$$

of which operator $(t_{1,1}, t_{2,1})$ means to generate two outputs, one for operator $t_{1,1}$ and another for operator $t_{2,1}$. The benefit of sharing common operators in matrix O is to reduce redundant processing operations, which will decrease elapsed time.

Sharing Common Operations in Matrix T: After sharing common operations in matrix O, an equivalent composite DOT block $block'_3$ is created. If in the DOT block $block'_3$, $t_{1,1}$ and $t_{2,1}$ have common operations, a new operator t' can represent both operators $t_{1,1}$ and $t_{2,1}$. Thus, in $block'_3$, operator t' will replace operator $(t_{1,1}, t_{2,1})$.

4.4 Exploiting the Potential of Parallelism

In this sub-section, we consider that $block_1$ and $block_2$ are dependent and suppose that $block_1$ is the input of $block_2$, i.e. $\vec{D}_2 = \vec{D}_1 O_1 T_1$. If O_2 is a diagonal matrix, i.e. $o_{2,1,2} = 0$ and $o_{2,2,1} = 0$, the intermediate results of $block_2$ do not need to be redistributed in the network. According to Property 2.1 in Section 2.4.3, $O_2 T_2$ can be merged into matrix T_1 of $block_1$. Thus, the new $block_1$ is:

$$block'_1 = \vec{D}_1 O_1 T'_1 = \begin{bmatrix} D_{1,1} & D_{1,2} \end{bmatrix} \begin{bmatrix} o_{1,1,1} & o_{1,1,2} \\ o_{1,2,1} & o_{1,2,2} \end{bmatrix} \begin{bmatrix} t_{2,1}(o_{2,1,1}(t_{1,1})) & 0 \\ 0 & t_{2,2}(o_{2,2,2}(t_{1,2})) \end{bmatrix}.$$

With this optimization, for a job composed by $\vec{D}_1 O_1 T'_1$, once intermediate results generated by $\vec{D}_1 O_1$ are collected by workers in T'_1, each workers will process its data in parallel till the end of this job. This optimization will eliminate the time spent on launching the $block_2$, and storing and collecting intermediates results generated by workers in matrix O_2. Thus, the elapsed time of $block'_1 = \vec{D}_1 O_1 T'_1$ is less than that of running $block_1$ and $block_2$ one by one.

5. EFFECTIVENESS OF THE DOT MODEL

In this section, we demonstrate the effectiveness of the DOT model with four types of case studies: (1) **Certification**: certifying the scalability and fault-tolerance of processing paradigms of software frameworks; (2) **Evaluation and Comparison**: evaluating and comparing software frameworks; (3) **Optimization**: optimizing job performance on various software frameworks; and (4) **Simulation**: providing a simulation-based framework for cost-effective software design.

5.1 Certification

The DOT model can be used to certify the scalability and fault-tolerance of the processing paradigm of a software framework for big data analytics.

LEMMA 5.1. *For a software framework F for big data analytics, if any job of F can be represented in the DOT model by either a single DOT block or a DOT expression, the processing paradigm of this framework F is scalable and fault-tolerant.*

PROOF. For a given software framework F for big data analytics, if any job of it can be represented in the DOT model, the processing paradigm is regulated by the processing paradigm of the DOT model. Based on the Lemma 3.1 and Lemma 3.2, the processing paradigm of this software framework F is scalable and fault-tolerant. □

With Lemma 5.1, we can study the scalability and fault-tolerance of the processing paradigms of different big data analytics frameworks. In this sub-section, we use two widely-used software frameworks, MapReduce [11] and Dryad [17], as case studies.

5.1.1 MapReduce

A MapReduce job has three *user-defined functions*, *map* function, *partitioning* function and *reduce* function. The worker running the *map/reduce* function is called *map worker/ reduce worker*. Every map function processes each input key/value pair provided from an underlying distributed file system and produces a set of intermediate key/value pairs. The partitioning function will decide which reduce worker an intermediate key/value pair will be sent to. Then, the MapReduce framework will *shuffle* intermediate results through the network and merge all intermediate key/value pairs based on the key. Finally, for every intermediate key and its values, the reduce function will generate the final results.

A computation model of the MapReduce framework is proposed by [18], in which a big data analytics job is composed of a sequence of MapReduce jobs. This model requires that the map function is stateless and the key/value pairs generated by a reduce function will have the same key with the intermediate key fed to this reduce function. In this paper, we take this model to represent the MapReduce. However, considering its practical usage, we slightly extend the power of map workers and reduce workers. Map workers can have stateful map functions, but a map worker only remembers key/value pairs it has processed. Moreover, based on the intermediate key and its values, a reduce function of a reduce worker can generate a sequence of new key/value pairs, of which new keys differ from the intermediate key.

To certify that the processing paradigm of the MapReduce framework is scalable and fault-tolerant, using Lemma 5.1 as a basis, we need to show that any MapReduce job can be represented by the DOT model.

LEMMA 5.2. *Any MapReduce job can be represented by a single DOT block.*

PROOF. A MapReduce job can be represented by an elementary or composite DOT block as follows:

$$\vec{DOT} = \begin{bmatrix} D_1 & \cdots & D_n \end{bmatrix} \begin{bmatrix} p_1(o_{map}) & \cdots & p_m(o_{map}) \\ \vdots & \ddots & \vdots \\ p_1(o_{map}) & \cdots & p_m(o_{map}) \end{bmatrix} \begin{bmatrix} t_{reduce} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & t_{reduce} \end{bmatrix},$$

where matrix O represents the map phase, T represents the reduce phase and the multiplication between $\vec{DO}$ and matrix T represents the shuffle phase. The data vector $\vec{D} = \begin{bmatrix} D_1 & D_2 & \cdots & D_n \end{bmatrix}$ represents the partitioned input data. Operator o_{map} means to apply the map function on each record of the input data. Operator p_i represents applying a partitioning function on each intermediate key/value pair generated by o_{map} and then extracting all intermediates key/value pairs which will be sent to the ith reduce worker represented by row i in the matrix T. Finally, operator t_{reduce} will first merge intermediate key/value pairs into groups based on the key and then apply the reduce function on each group. Thus, a MapReduce job can be represented by an elementary or composite DOT block. $\square$

Based on Lemma 5.2, we can then claim that the processing paradigm of the MapReduce framework is scalable and fault-tolerant.

LEMMA 5.3. *The processing paradigm of the MapReduce framework is scalable and fault-tolerant.*

PROOF. It is the corollary of Lemmas 5.1 and 5.2. $\square$

5.1.2 Dryad

In Dryad, the dataflow of a big data analytics job is represented by a directed acyclic graph (DAG)

$$G =< V_G, E_G, I_G, O_G >,$$

where V_G contains all vertices, E_G contains all edges, $I_G \subseteq V_G$ contains all inputs, and $O_G \subseteq V_G$ contains all outputs. In a DAG, there are two kinds of vertices representing data on an underlying distributed file system and operations of data processing, respectively. The edge represents the communication among workers.

To certify that the processing paradigm of the Dryad framework is scalable and fault-tolerant, we need to show that any Dryad job can be represented by the DOT model.

LEMMA 5.4. *A Dryad job represented by a DAG can be represented by a DOT expression.*

PROOF. We give the method used to write a DOT expression for a given DAG in Dryad. Given a DAG

$$G =< V_G, E_G, I_G, O_G >$$

and its graph of job stages

$$G_s =< V_Gs, E_Gs, I_Gs, O_Gs >,$$

the procedure of representing this DAG by a DOT expression is shown as follows.

1. Initialize a set S to $\emptyset$. This set contains the enumerated vertices in G_s. Initialize a set U to V_G. The set of U contains the vertices that have not been enumerated in G_s;
2. Find a vertex v in U, which does not have incoming edge from vertices in U;
   (a) If v represents input data, find all vertices in G corresponding to v and use a data vector $\vec{D}$ to represent this input data. Then, add v into S and remove v from U;
   (b) If v represents an operation of data processing, create a matrix T_{new} by using all corresponding operation vertices of v in the set V_G as operators of diagonal matrix T_{new}. Based on E_Gs, find all corresponding vertices in S pointing to v and denote the set of these vertices as P. Then, apply $\bigoplus$ to the DOT blocks or data vectors representing source vertices in P to create a new data vector $\vec{D}_{new}$. Based on the composition of edges from vertices of P to v (pointwise composition or complete bipartite graph), create a matrix O_{new}, each element of which is either operator 1 or 0. This matrix O_{new} is used to represent the edges from vertices in P to v. Then, create a new elementary/composite DOT block,$\vec{D}_{new}O_{new}T_{new}$. Finally, add v into S and remove v from U;
   (c) If v represents output data, go to the next step;
3. If all vertices in U are those vertices representing output data, go to step 4. Otherwise, go to step 2;
4. Use Property 2.1 to simplify the DOT expression;
5. Till this step, a DOT expression is created for the DAG G. The result of this DOT expression is the output of the DAG;

Thus, a Dryad job can be represented by a DOT expression. $\square$

Based on Lemma 5.4, we have shown that the processing paradigm of the Dryad framework is scalable and fault-tolerant.

LEMMA 5.5. *The processing paradigm of the Dryad frameworks is scalable and fault-tolerant*

PROOF. It is the corollary of Lemma 5.1 and 5.4. $\square$

5.2 Evaluation and Comparison

In Section 5.1, we have shown that any MapReduce/Dryad job can be represented by the DOT model. If both MapReduce and Dryad can execute big data analytics jobs represented by the DOT model, the DOT model can server as the bridge for application migration between MapReduce and Dryad. In this sub-section, we evaluate if MapReduce and Dryad can execute big data analytics jobs represented by the DOT model, i.e if MapReduce and Dryad can execute elementary/composite DOT blocks and DOT expressions. Also, since the execution of independent elementary DOT blocks in a composite DOT block is flexible, we will compare if MapReduce and Dryad support sharing data scan among operators in the O-layer of a composite DOT block. If both of MapReduce and Dryad can execute big data analytics jobs represented by the DOT model and can share data scan among operators in the O-layer of a composite DOT block, these two software frameworks will not display fundamental differences on executing a big data analytics job.

5.2.1 Executing a DOT Block on MapReduce

We first show a general method to execute a DOT block on MapReduce.

LEMMA 5.6. *A DOT block with n concurrent workers in matrix O and m concurrent workers in matrix T can be executed by at most $n \times m + m$ MapReduce jobs, where $n \times m$ concurrent map-only MapReduce jobs represent matrix O in the DOT block and m concurrent map-only MapReduce jobs represent matrix T.*

PROOF. In a DOT block, peer workers in matrix O or T do not have data dependency. For every chunk in data vector $\vec{D} = [D_1 \quad \cdots \quad D_n]$, there are m operators to be applied. In the worst case, m map-only MapReduce jobs are needed to process a chunk and then generate m intermediate results for m workers in matrix T. Thus, there are $n \times m$ independent map-only MapReduce jobs needed to perform data processing represented by matrix O. Similarly, m independent map-only MapReduce jobs are needed to perform data processing represented by matrix T. Because independent MapReduce jobs can be executed concurrently, a DOT block with n concurrent workers in matrix O and m concurrent workers in matrix T can be executed by at most $n \times m + m$ MapReduce jobs, where $n \times m$ concurrent map-only MapReduce jobs represent matrix O in the DOT block and m concurrent Map-only MapReduce jobs represent matrix T. $\square$

Although, in the worst case, $n \times m + m$ map-only MapReduce jobs are needed to execute a DOT block, if some operators can share the data scan in a worker, the number of needed MapReduce jobs for executing this DOT block can be reduced. Consider a DOT block

$$\vec{DOT} = \begin{bmatrix} D_1 & \cdots & D_n \end{bmatrix} \begin{bmatrix} o_{1,1} & \cdots & o_{1,m} \\ \vdots & \ddots & \vdots \\ o_{n,1} & \cdots & o_{n,m} \end{bmatrix} \begin{bmatrix} t_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & t_m \end{bmatrix},$$

and consider two sets $R = \{r_p | r_p \in \{1, 2, \cdots, n\}\} \subseteq \{1, 2, \cdots, n\}$ and $C = \{c_q | c_q \in \{1, 2, \cdots, m\}\} \subseteq \{1, 2, \cdots, m\}$. If, for a given $i \in R$, all $o_{i,j}$ $(j \in C)$ can share the data scan, and for a given $j \in C$, all $o_{i,j}$ for every $i \in R$ are the same, a single MapReduce job can be used to execute operators $o_{i,j}$ and t_j, where $i \in R$ and $j \in C$. In this MapReduce job, the map phase will execute operators $o_{i,j}$, where $i \in R$ and reduce phase will execute operators t_j, where $j \in C$. The partitioning function will distribute intermediate results from an operator $o_{i,j}$ to worker performing t_j. Thus, the total number of MapReduce jobs for executing a DOT block can be further reduced.

Since, an elementary/composite DOT block can be executed by MapReduce jobs and there is an underlying distributed file system associated with the MapReduce framework, a DOT expression also can be executed by MapReduce jobs.

5.2.2 Executing a DOT Block on Dryad

We first present a general method to execute a DOT block on Dryad.

Lemma 5.7. *A DOT block can be executed by a Dryad job expressed by a DAG.*

Proof. Given an arbitrary DOT block

$$\vec{DOT} = \begin{bmatrix} D_1 & \cdots & D_n \end{bmatrix} \begin{bmatrix} o_{1,1} & o_{1,2} & \cdots & o_{1,m} \\ o_{2,1} & o_{2,2} & \cdots & o_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ o_{n,1} & o_{n,2} & \cdots & o_{n,m} \end{bmatrix} \begin{bmatrix} t_1 & 0 & \cdots & 0 \\ 0 & t_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & t_m \end{bmatrix},$$

the corresponding DAG of the Dryad job is shown in Figure 6. In this DAG, vertices D_1 to D_n represent elements of the data vector. Operator $o_{i,j}$ is the operator at the ith row and j column of matrix O. Operator in matrix T are represented by t_1 to t_m in the DAG. Thus, a DOT block can be executed by a Dryad job. $\square$

If in matrix O, some operators can share the data scan, the DAG can be changed to represent the data scan by merging corresponding vertices, adding new vertices for partitioning intermediate results and adjusting edges accordingly. For example, for every worker, if all operators of a worker in matrix O can share the data scan, i.e. for all rows, all operators in a row can share the data scan, the DAG can be changed to Figure 7. In this DAG, O_i represents all operators of the ith workers in the matrix O and thus it is composed by all

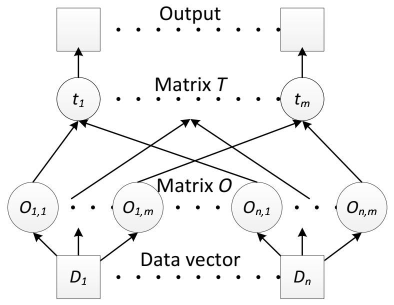

Figure 6: The DAG of a DOT block

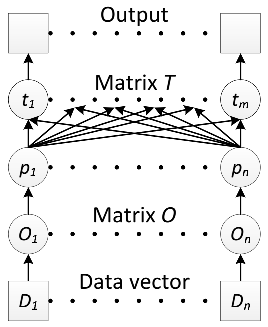

Figure 7: An example DAG for sharing data scan

operators in the ith row, i.e. $O_i = (o_{i,1}, \ldots, o_{i,m})$. Vertices p_i $(1 \leq i \leq n)$ are data distribution vertices, each of which will distribute the intermediate results generated by O_i to corresponding workers in the matrix T represented by vertices t_1 to t_m in the DAG, i.e. p_i is responsible for all $o_{i,j}$ $(1 \leq j \leq m)$ and for a given j, p_i will distribute results of $o_{i,j}$ to t_j. Finally, t_1 to t_m will generate the output.

Since, an elementary/composite DOT block can be executed by Dryad jobs and there is an underlying distributed file system associated with the Dryad framework, a DOT expression also can be executed by a Dryad job.

5.2.3 Discussion

Through 5.2.1 and 5.2.2, we have shown that a DOT block can be executed by MapReduce and Dryad. Also, if operators of a worker in matrix O can share data scan, both of these two frameworks can execute these operators with a single pass of input data. Thus, for a given DOT block, job execution of MapReduce and Dryad are equivalently represented by the DOT model. The core computing flows of MapReduce and Dryad are the same, although implementation details can be different. For a given big data analytics job represented by the DOT model, the processing paradigm of this job will be the same under the MapReduce and Dryad frameworks.

5.3 Optimization

With optimization rules provided in Section 4, the DOT model can be used to optimize big data analytics jobs on various frameworks. Moreover, the optimization made by the DOT model is framework and implementation independent, so the optimization can be automatically ported among frameworks which can execute DOT blocks. In this sub-section, we use a structured data analysis based on SQL conducted on the MapReduce framework as a case study to show the effectiveness of the DOT model in optimizing job execution. In this case, we choose the TPC-H benchmark [4] as the workload, which is a widely used data warehousing benchmark. Due to the page limit, we only present the performance evaluation of TPC-H Q17 as a representative case.

```
1 SELECT sum(l_extendedprice) / 7.0 AS avgyearly
2 FROM (SELECT l_partkey, 0.2* avg(l_quantity) AS tmp
3       FROM Lineitem
4       GROUP BY l_partkey) AS t1,
5      (SELECT l_partkey,l_quantity,l_extendedprice
6       FROM Lineitem, Part
7       WHERE p_partkey = l_partkey AND
8             p_brand = 'X' AND
9             p_container = 'Y') AS t2
10 WHERE t2.l_partkey = t1.l_partkey AND
11       t2.l_quantity < t1.tmp;
```

Figure 8: A flattened variation of TPC-H Q17

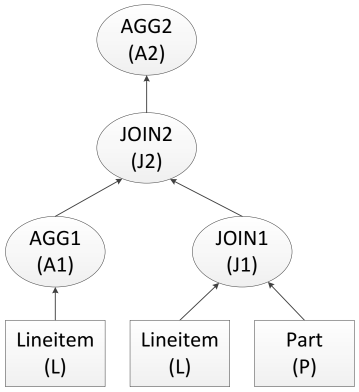

Figure 9: Query plan tree of TPC-H Q17

Because the MapReduce framework does not support nested parallelism [16], TPC-H Q17 should be "flattened" so that this query can be implemented in the MapReduce framework. The query shown in Figure 8 with the query plan tree shown in Figure 9 is flattened based on first-aggregation-then-join algorithm [19].

In Figure 8, lines 2-4 are for the aggregation operation *AGG1* (*A1*) in the query plan tree, which generates the temporary table *t1*. The *join* operation *JOIN1* (*J1*) between tables *Lineitem* (*L*) and *Part* (*P*) is shown in lines 5-9. The result of *J1* is the temporary table *t2*. Then, *JOIN2* (*J2*) will join tables *t1* and *t2*. Finally, *AGG2* (*A2*) will perform an aggregation on the result of *J2* and output the ending results. In the DOT model, each operation of *A1*, *J1*, *J2* and *A2* can be represented by a composite DOT block. Then, this query can be represented by a DOT expression, which is

$$Q17 = ((\vec{D}_L O_{A1} T_{A1} \oplus (\vec{D}_L \oplus \vec{D}_P) O_{J1} T_{J1}) O_{J2} T_{J2}) O_{A2} T_{A2}.$$

This DOT expression can be implemented by four MapReduce jobs. However, considering the optimization opportunities analyzed in Section 4, a DOT expression with only two composite DOT blocks can be used for this query. To show the optimization procedure, without loss of generality, we set the dimension of each matrix in the DOT expression based on Table 1. The original DOT expression for TPC-H Q17 is shown in Figure 10, where each operator in matrix O is combined by an operator o that simply selects desired records from input data vector and a partitioning operator p that selects desired output records of operator o for a worker in matrix T based on hash values of those records. For example, if the hash value of a output record of operator o is 1, this output record will be collected by the first worker in matrix T and thus, only partitioning operator p at first column of matrix O will select this record; Moreover, in Figure 10, an operator in matrix T will perform *aggregation* or *join* operations.

The optimization procedure has three steps:

1. Because *A1* and *J1* **share common chunks**, we can

Table 1: The dimension of each matrix in the DOT expression of TPC-H Q17

Matrix	Dimension	Matrix	Dimension
$\vec{D}_L$	1×2	$\vec{D}_P$	1×2
O_{A1}	2×2	O_{J2}	4×2
T_{A1}	2×2	T_{J2}	2×2
O_{J1}	4×2	O_{A2}	2×2
T_{J1}	2×2	T_{A2}	2×2

substitute the operator $\bigoplus$ between composite DOT blocks representing *A1* and *J1* with operator $\biguplus$. Thus, in the original DOT expression, $\vec{D}_L O_{A1} T_{A1} \oplus (\vec{D}_L \oplus \vec{D}_P) O_{J1} T_{J1}$ is replaced by

$$\begin{aligned}
&\vec{D}_L O_{A1} T_{A1} \uplus (\vec{D}_L \oplus \vec{D}_P) O_{J1} T_{J1} \\
&= \begin{bmatrix} D_{L,1} & D_{L,2} & D_{P,1} & D_{P,2} \end{bmatrix} \\
&\begin{bmatrix} p_{A1,1}(o_{A1,1,1}) & p_{A1,2}(o_{A1,1,2}) & p_{J1,1}(o_{J1,1,1}) & p_{J1,2}(o_{J1,1,2}) \\ p_{A1,1}(o_{A1,2,1}) & p_{A1,2}(o_{A1,2,2}) & p_{J1,1}(o_{J1,2,1}) & p_{J1,2}(o_{J1,2,2}) \\ 0 & 0 & p_{J1,1}(o_{J1,3,1}) & p_{J1,2}(o_{J1,3,2}) \\ 0 & 0 & p_{J1,1}(o_{J1,4,1}) & p_{J1,2}(o_{J1,4,2}) \end{bmatrix} \\
&\begin{bmatrix} t_{A1,1} & 0 & 0 & 0 \\ 0 & t_{A1,2} & 0 & 0 \\ 0 & 0 & t_{J1,1} & 0 \\ 0 & 0 & 0 & t_{J1,2} \end{bmatrix};
\end{aligned}$$

2. Because *A1* will group records based on `l_partkey` and the join condition of *J1* is `p_partkey=l_partkey`, corresponding partitioning operators of *A1* and *J1* are the same, i.e. $p_{A1,1} = p_{J1,1}$ and $p_{A1,2} = p_{J1,2}$. Because *A1* and *J1* **share common operations in matrix** O, we will have

$$\begin{aligned}
&\vec{D}_L O_{A1} T_{A1} \uplus (\vec{D}_L \oplus \vec{D}_P) O_{J1} T_{J1} \\
=&\vec{D}_{P,L} O_{A1,\,J1} T_{A1,\,J1} \\
=&\begin{bmatrix} D_{L,1} & D_{L,2} & D_{P,1} & D_{P,2} \end{bmatrix} \\
&\begin{bmatrix} p_1(o_{A1,1,1}, o_{J1,1,1}) & p_2(o_{A1,1,2}, o_{J1,1,2}) \\ p_1(o_{A1,2,1}, o_{J1,2,1}) & p_2(o_{A1,2,2}, o_{J1,2,2}) \\ p_1(o_{J1,3,1}) & p_2(o_{J1,3,2}) \\ p_1(o_{J1,4,1}) & p_2(o_{J1,4,2}) \end{bmatrix} \\
&\begin{bmatrix} (t_{A1,1}, t_{J1,1}) & 0 \\ 0 & (t_{A1,2}, t_{J1,2}) \end{bmatrix},
\end{aligned}$$

where $p_1 = p_{A1,1} = p_{J1,1}$ and $p_2 = p_{A1,2} = p_{J1,2}$. Moreover, because `t2.l_partkey=t1.l_partkey` is the join condition of *J2*, and notice that `t2.l_partkey= p_partkey=l_partkey` and `t1.l_partkey=l_partkey`, the input data vector of *J2* has already been partitioned, i.e. all records with the same `l_partkey` has been placed in the same element of the input data vector of *J2*. Thus, in *J2*, partitioning operators $p_{J2,1}$ and $p_{J2,2}$ are not needed, and O_{J2} will become to O'_{J2}, which is a a diagonal matrix

$$\begin{bmatrix} (o_{J2,1,1}, o_{J2,3,1}) & 0 \\ 0 & (o_{J2,2,2}, o_{J2,4,2}) \end{bmatrix}.$$

3. Because O'_{J2} is a diagonal matrix, considering the rule of **Exploiting the Potential of Parallelism** in section 4.4, O'_{J2} and T_{J2} can be merged into $T_{A1,\,J1}$. Thus, *A1*, *J1* and *J2* can be represented by a single composite DOT block. An equivalent DOT expression for TPC-H Q17 will only need two composite DOT blocks. This final optimized DOT expression is shown in Figure 11.

$$\begin{aligned}Q17 =&((\vec{D}_L O_{A1} T_{A1} \oplus (\vec{D}_L \oplus \vec{D}_P) O_{J1} T_{J1}) O_{J2} T_{J2}) O_{A2} T_{A2}\\
=&(([D_{L,1} \quad D_{L,2}] \begin{bmatrix} p_{A1,1}(o_{A1,1,1}) & p_{A1,2}(o_{A1,1,2}) \\ p_{A1,1}(o_{A1,2,1}) & p_{A1,2}(o_{A1,2,2}) \end{bmatrix} \begin{bmatrix} t_{A1,1} & 0 \\ 0 & t_{A1,2} \end{bmatrix} \oplus\\
&([D_{L,1} \quad D_{L,2}] \oplus [D_{P,1} \quad D_{P,2}]) \begin{bmatrix} p_{J1,1}(o_{J1,1,1}) & p_{J1,2}(o_{J1,1,2}) \\ p_{J1,1}(o_{J1,2,1}) & p_{J1,2}(o_{J1,2,2}) \\ p_{J1,1}(o_{J1,3,1}) & p_{J1,2}(o_{J1,3,2}) \\ p_{J1,1}(o_{J1,4,1}) & p_{J1,2}(o_{J1,4,2}) \end{bmatrix} \begin{bmatrix} t_{J1,1} & 0 \\ 0 & t_{J1,2} \end{bmatrix})\\
&\begin{bmatrix} p_{J2,1}(o_{J2,1,1}) & 0 \\ 0 & p_{J2,2}(o_{J2,2,2}) \\ p_{J2,1}(o_{J2,3,1}) & 0 \\ 0 & p_{J2,2}(o_{J2,4,2}) \end{bmatrix} \begin{bmatrix} t_{J2,1} & 0 \\ 0 & t_{J2,2} \end{bmatrix}) \begin{bmatrix} p_{A2,1}(o_{A2,1,1}) & p_{A2,2}(o_{A2,1,2}) \\ p_{A2,1}(o_{A2,2,1}) & p_{A2,2}(o_{A2,2,2}) \end{bmatrix} \begin{bmatrix} t_{A2,1} & 0 \\ 0 & t_{A2,2} \end{bmatrix}\end{aligned}$$

Figure 10: The original DOT expression for TPC-H Q17

$$\begin{aligned}Q17 =&(\vec{D}_{P,L} O_{A1,\,J1,\,J2} T_{A1,\,J1,\,J2}) O_{A2} T_{A2}\\
=&([D_{L,1} \quad D_{L,2} \quad D_{P,1} \quad D_{P,2}] \begin{bmatrix} p_1(o_{A1,1,1}, o_{J1,1,1}) & p_2(o_{A1,1,2}, o_{J1,1,2}) \\ p_1(o_{A1,2,1}, o_{J1,2,1}) & p_2(o_{A1,2,2}, o_{J1,2,2}) \\ p_1(o_{J1,3,1}) & p_2(o_{J1,3,2}) \\ p_1(o_{J1,4,1}) & p_2(o_{J1,4,2}) \end{bmatrix}\\
&\begin{bmatrix} t_{J2,1}((o_{J2,1,1}, o_{J2,3,1})(t_{A1,1}, t_{J1,1})) & \\ 0 & t_{J2,2}((o_{J2,2,2}, o_{J2,4,2})(t_{A1,2}, t_{J2,2})) \end{bmatrix}) \begin{bmatrix} p_{A2,1}(o_{A2,1,1}) & p_{A2,2}(o_{A2,1,2}) \\ p_{A2,1}(o_{A2,2,1}) & p_{A2,2}(o_{A2,2,2}) \end{bmatrix} \begin{bmatrix} t_{A2,1} & 0 \\ 0 & t_{A2,2} \end{bmatrix}\end{aligned}$$

Figure 11: The final optimized DOT expression for TPC-H Q17

To evaluate the benefits provided by above optimization, we compare our prototype SQL-to-MapReduce translator (YSmart) shown in [20] with Hive [26] (a widely used open source data warehousing system on top of MapReduce) on Amazon EC2 [5] and a Facebook production cluster. The optimized code generated by our prototype has two MapReduce jobs. While, the unoptimized code generated by Hive has four MapReduce jobs. On both environments, the speedup achieved by optimized code is more than 2 times. The detailed evaluation can be found in [20].

5.4 Simulation

Several software frameworks on distributed systems have been recently developed for big data analytics, e.g. Google MapReduce [11], Hadoop [1], Dryad [17], Ciel [23] and Pregel [22]. Although these frameworks may have different design and implementation features, they share the same goals and even follow the basic computing/communication patterns to ensure scalability and fault-tolerance as we have discussed in the paper. The development of these software systems can be very expensive. Since the DOT model presents an abstraction of a job execution layer for big data analytics in a scalable and fault tolerant way, it would lead to a cost-effective software development. Figure 12 presents a simulation structure for software design based on the DOT model, where the job execution layer is abstracted by the DOT model bridging between jobs and the underlying system. The storage layer simulates the data storage services, such as a distributed file system, and the network layer simulates the data transfer on a specific network configuration. Both storage and network layers are pluggable in the simulator. With this DOT-based simulation, a software design for big data analytics can be carefully evaluated in a cost-effective way before it is implemented.

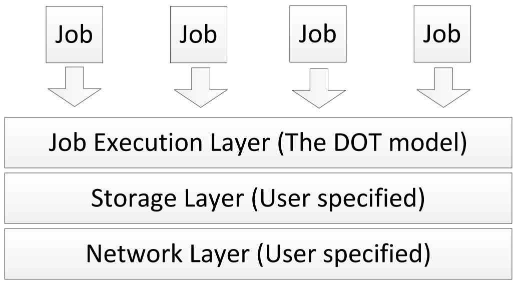
Job
Job
Job
Job
Job Execution Layer (The DOT model)
Storage Layer (User specified)
Network Layer (User specified)

Figure 12: The architecture of the big data analytics framework simulator based on the DOT model

6. OTHER RELATED WORK

In the traditional parallel computing field, there are several "scale-up" models for high performance computing. The parallel random access machine (PRAM) [6] model provides an abstract parallel machine for theoretical studies. The bulk-synchronous parallel (BSP) model [27] defines a bridging model for general-purpose parallel computation. The latency model [29] and LogP model [10] are experimental metrics for communication performance of parallel computers. Since these four models do not treat data as a first-class object, they are not suitable for modeling big data analytics.

Recent modeling efforts for big data analytics have been made for specific frameworks. The study in [13] introduces an algorithmic model for massive, unordered and distributed computation, which is an important class of computation executed on the MapReduce framework. The study in [18] proposes a model of computation for the MapReduce framework. Comparing to these framework specific models, our DOT model aims to bridge big data analytics applications and various software frameworks, and to provide a set of framework and implementation independent optimization rules for applications.

There are also a few experiment-based studies aiming to compare different software frameworks under different contexts. The study in [28] analyzes and compares different approaches to distributed aggregation in parallel database systems, Hadoop and Dryad. The study in [25] compares performance and scalability between Hadoop and parallel database systems, and the study in [7] further compares fault-tolerance among these two frameworks and a hybrid solution.

7. CONCLUSION AND FUTURE WORK

We have presented the DOT model for analyzing, optimizing, and deploying software for big data analytics in distributed systems. As a bridging model between various applications and different software frameworks to execute big data analytics jobs, the DOT model abstracts basic computing and communication behavior by three entities: (1) distributed data sets, (2) concurrent workers, and (3) simple mechanisms with optimized efficiency to regulate interactions between the workers and the data sets. We have shown that the processing paradigm of the DOT model is scalable and fault-tolerant. Based on the DOT model, we further present a set of critical optimization rules that are framework and implementation independent. We have demonstrated the effectiveness of the DOT model by certifying and comparing the scalability and fault-tolerance of processing paradigms of MapReduce and Dryad. We also have shown the effectiveness of optimization rules provided by the DOT model for complex analytical queries.

Our future work has two directions: (1) we will extend and enhance the DOT model to explore other factors that will impact the scalability and fault-tolerance of big data analytics jobs, e.g. characteristics of storage and network infrastructures; and (2) under the guidance of the DOT model, we will design and implement a simulator for big data analytics software frameworks and an automatic application migration tool among different software frameworks.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. We appreciate many discussions with big data researchers and engineers in Facebook and Intel. We thank Bill Bynum for reading the paper and for his suggestions. This research was partially supported by the National Science Foundation under grants CCF-0620152, CCF-0702380, CCF-0913150 and IIS-0916440.

9. REFERENCES

[1] http://hadoop.apache.org/.

[2] http://en.wikipedia.org/wiki/Recurrence_relation.

[3] http://en.wikipedia.org/wiki/K-means_clustering.

[4] http://www.tpc.org/tpch/.

[5] http://aws.amazon.com/ec2/.

[6] http://en.wikipedia.org/wiki/Parallel_Random_Access_Machine.

[7] A. Abouzied, K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. In *VLDB*, Lyon, France, 2009.

[8] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *SIGMOD*, 2000.

[9] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.

[10] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: A Practical Model of Parallel Computation. *Commun. ACM*, 39(11):78–85, 1996.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[12] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *PVLDB*, 3(1):518–529, 2010.

[13] J. Feldman, S. Muthukrishnan, A. Sidiropoulos, C. Stein, and Z. Svitkina. On Distributing Symmetric Streaming Computations. In *SODA*, 2008.

[14] J. Gray. Distributed Computing Economics. *ACM Queue*, 6(3):63–68, 2008.

[15] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems. In *ICDE*, 2011.

[16] B. Hindman, A. Konwinski, M. Zaharia, and I. Stoica. A Common Substrate for Cluster Computing. In *HotCloud'09*, 2009.

[17] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys*, 2007.

[18] H. J. Karloff, S. Suri, and S. Vassilvitskii. A Model of Computation for MapReduce. In *SODA*, 2010.

[19] W. Kim. On Optimizing an SQL-like Nested Query. *ACM Trans. Database Syst.*, 7(3):443–469, 1982.

[20] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang. YSmart: Yet another SQL-to-MapReduce Translator. In *ICDCS*, 2011.

[21] R. Lee, M. Zhou, and H. Liao. Request Window: an Approach to Improve Throughput of RDBMS-based Data Integration System by Utilizing Data Sharing Across Concurrent Distributed Queries. In *VLDB*, 2007.

[22] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *SIGMOD*, 2010.

[23] D. G. Murray and S. Hand. CIEL: a universal execution engine for distributed data-flow computing. In *NSDI '11*, 2011.

[24] L. Page, S. Brin, R. Motwani, and T. Winograd. The Pagerank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab.

[25] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *SIGMOD Conference*, 2009.

[26] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[27] L. G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8):103–111, 1990.

[28] Y. Yu, P. K. Gunda, and M. Isard. Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations. In *SOSP*, 2009.

[29] X. Zhang, Y. Yan, and K. He. Latency Metric: An Experimental Method for Measuring and Evaluating Parallel Program and Architecture Scalability. *J. Parallel Distrib. Comput.*, 22(3):392–410, 1994.