A NoSQL Data Model For Scalable Big Data Workflow Execution

Aravind Mohan, Mahdi Ebrahimi, Shiyong Lu, Alexander Kotov
Wayne State University
Detroit, MI, USA
{amohan, mebrahimi, shiyong, kotov}@wayne.edu

Abstract—**While big data workflows haven been proposed recently as the next-generation data-centric workflow paradigm to process and analyze data of ever increasing in scale, complexity, and rate of acquisition, a scalable distributed data model is still missing that abstracts and automates data distribution, parallelism, and scalable processing. In the meanwhile, although NoSQL has emerged as a new category of data models, they are optimized for storing and querying of large datasets, not for ad-hoc data analysis where data placement and data movement are necessary for optimized workflow execution. In this paper, we propose a NoSQL data model that: 1) supports high-performance MapReduce-style workflows that automate data partitioning and data-parallelism execution. In contrast to the traditional MapReduce framework, our MapReduce-style workflows are fully composable with other workflows enabling dataflow applications with a richer structure; 2) automates virtual machine provisioning and deprovisioning on demand according to the sizes of input datasets; 3) enables a flexible framework for workflow executors that take advantage of the proposed NoSQL data model to improve the performance of workflow execution. Our case studies and experiments show the competitive advantages of our proposed data model. The proposed NoSQL data model is implemented in a new release of DATAVIEW, one of the most usable big data workflow systems in the community.**

Keywords- Big Data Workflows; NoSQL; Clouds;

I. INTRODUCTION

Big data workflows have recently emerged as the next generation of data-centric workflow technologies to address the five "V" challenges of big data [12]: volume, variety, velocity, veracity, and value [20, 21, 23, 25]. While its precedent, scientific workflows, focus on dataflow and automation management [17], big data workflows focus on large-scale data processing and analytics with a "scale-out" architecture and a "moving-computation-to-data" processing paradigm. More formally, a big data workflow is the computerized modeling and automation of a process consisting of a set of computational tasks and their data interdependencies to process and analyze data of ever increasing in scale, complexity, and rate of acquisition. The coining of the term "big data workflows" is timely and important to recognize the continuing relevance and importance of workflow technologies in data processing and management, as well as the challenges and opportunities of big data research in the workflow community[12]. While more and more big data tools have been developed in recent years to address the big data deluge in both science [20] and business [21], the gap between the capability of data collecting and the power of data processing and analysis continues to increase. One category of such tools are NoSQL databases [22], which deliver high read and write performance by automating the data distribution and retrieval over a cluster of tens of thousands of machines [24]. In contrast to their SQL counterpart, NoSQL databases often relax the traditional ACID properties of transactions and introduce restrictions on its query language, such as no-support-for-join, in favor of high performance of read and write. The wide adoption of NoSQL techniques in big data applications is attributed to, among other things, their large scalability, high fault-tolerance, flexible data models, and high performance query capability [24].

However, the power of big data not only lies in storing and querying large datasets, but also in performing efficient ad hoc sophisticated analysis over such datasets to shorten the cycle of "from data to insight and to value". One major research question is: Is it possible to leverage the power of NoSQL techniques in big data workflow systems to improve the performance of workflow execution? If yes, how? One approach is to integrate an existing NoSQL database system into a big data workflow system. This approach, however, will not unleash the full power of neither as data movement between the NoSQL database and the workflow engine will become the bottleneck. Moreover, many of the workflow optimization opportunities will not become available under the constraints of a NoSQL database, which decides the placement of data according to partitioning strategies that are optimized for querying, not for ad hoc analysis, in which data placement, replication, and data movement need to be decided on the fly according to the structure and data access patterns of a workflow [25, 26]. Therefore, we take another approach in this research, in which we develop our own NoSQL collectional data model, which leverages some of the capabilities of existing NoSQL data models, while enriching in capabilities, such as flexible MapReduce workflows, workflow executors, and optimization of workflow execution.

In this paper, we propose a NoSQL data model that: 1) supports high-performance MapReduce-style workflows that automate data partitioning and data-parallelism in workflow execution; in contrast to the traditional MapReduce framework, our MapReduce-style workflows are fully composable with other workflows, and thus enable dataflow applications with a richer structure; 2) automates virtual machine provisioning and deprovisioning on demand according to the sizes of input datasets; 3) enables a flexible framework for workflow executors that take advantage of the proposed NoSQL data model to improve the performance of workflow execution. Our case studies and experiments show the competitive advantages of our proposed data model. The proposed NoSQL data model is implemented in a new release of DATAVIEW, one of the most usable big data workflow systems in the community.

II. An Overview Of DATAVIEW

In Fig. 1, we present the overall architecture of DATAVIEW, which enriches the original architecture described in [12] with a refined design for the Workflow Engine. DATAVIEW consists of seven subsystems: the Webench is a Web-based interface that supports user interaction, workflow visualization, data presentation, and system configuration. The Data Product Manager features the proposed NoSQL collectional data model and a rich set of other data types, including relational, files, and scalar types. The Task Manager supports a single-component based task model that separates registration from configuration and eases the process of registering external functional components (such as Web services) into primitive workflows [13]. The Task Manager is also responsible for the run-time execution of primitive workflows. The Cloud Resource Manager (CRM) provides the provisioning and deprovisioning capabilities of cloud resources, including both virtual machines and storage resources. The Provenance Manager manages the data lineage and derivation history of data products for the reproducibility and validation of workflow execution results [18]. The Workflow Monitor supports the monitoring of workflow execution status and progress and exception handling [19]. Finally, the Workflow Engine is at the heart of the DATAVIEW system, responsible for overall workflow orchestration, scheduling, and the coordination and collaboration of all subsystems.

This paper enriches the Workflow Engine with the notion of "executors", which abstracts the execution platform that a workflow will be executed at runtime. Two categories of workflow executors are supported: *on-premises workflow executor*, which supports the execution of a workflow on a single DATAVIEW server, and *cloud workflow executor*, which supports the execution of a workflow in the cloud (e.g., Amazon EC2). Moreover, two types of cloud workflow executors have been implemented: *type-A cloud workflow executor* supports a clustering algorithm that partitions a workflow into a number of workflow clusters, with each workflow cluster executed in one virtual machine; *type-B cloud workflow executor* supports MapReduce-style workflows, which automates data partitioning, virtual machine provisioning and deprovisioning, and scalable execution of workflows. Both type-A and type-B workflow executors exploit the proposed NoSQL collectional model for improved performance of workflow execution.

III. NoSQL Collectional Data Model

A big data workflow represents a multiple-step data analysis pipeline by chaining several data analysis modules together via data links that connect the output of one analysis module to the input of another analysis module. A big data focuses on large-scale data processing and analytics with a

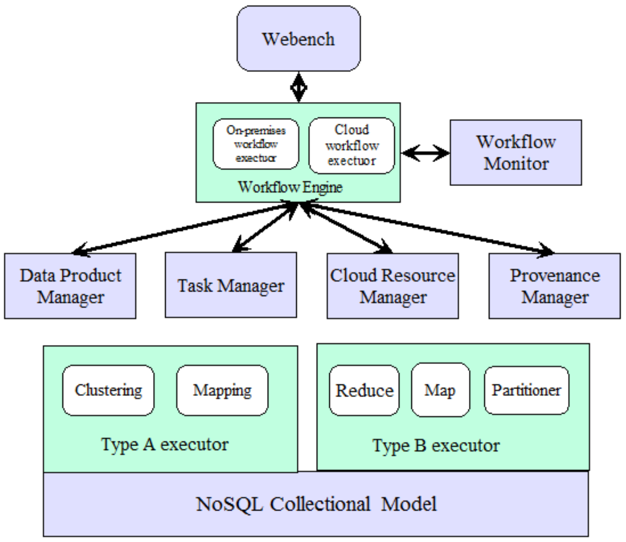


Fig. 1. Architecture of DATAVIEW.

"scale-out" architecture and a "moving-computation-to-data" processing paradigm. Big data imposes challenges in the workflow development at both the primitive and composite workflow level. A primitive workflow is the workflow that contains no sub-workflows in it. On the other hand, a composite workflow has one or more sub-workflows inside it. Our original collectional data model [1] is a counterpart of relational data model that supports creation of hierarchically organized data in a nested manner. In our new NoSQL collectional data model, we extend our original collectional data model to improve the performance of big data workflow execution.

Big data workflow is executed in the Cloud. A cloud consists of a set of virtual machines that are used to store the partitioned input data, execute the workflow and store the output data generated by the workflow. A big data workflow is defined as follows:

Definition 1. A **big data workflow w** is a 8-tuple (IP, OP, D, T, S, Consumer, Producer, DataType), where

- IP is the set of input ports for workflow w. Each individual input port is denoted by ip_m , $IP = \{ip_1, ip_2, \dots, ip_M\}$. IP^ is the set of intermediate input ports for workflow w and IP^ ⊆ IP.
- OP is the set of output ports for workflow w. Each individual output port is denoted by op_q , $OP = \{op_1, op_2, \dots, op_Q\}$.
- D is the set of workflow datasets. Each individual dataset is denoted by d_j, $D = \{d_1, d_2, d_3, \dots, d_J\}$. dj can be either connected to $ip_m \in IP$ or $op_q \in OP$.
- T is the set of workflow tasks. Each individual task is denoted by t_i , $T = \{t_1, t_2, t_3, \dots, t_I\}$. Each task can have one or more input and output ports. $t_i.ip_m$ shows the m^{th} input port of task t_i. subsequently $t_i.op_q$shows the q^{th} output port of task t_i.
- $S: D \to R^+$ is the dataset size function. $S(d_j)$, $d_j \in D$ returns the size of the dataset d_j. The size of a dataset

is defined in some pre-determined unit such as MegaBytes, GigaBytes, TeraBytes, etc. R^+ is the set of positive real number.

- $Consumer: T \rightarrow 2^\wedge T$ is the task-task function. $Consumer(t_i),\ t_i \in T$ returns the set of tasks that t_i is directly connected to and require the output of t_i for their inputs.
- $Producer: T \rightarrow 2^\wedge T$ is the task-task function. $Producer(t_i),\ t_i \in T$ returns the set of tasks that are connected to t_i directly and t_i require their output as its inputs.
- *DataType:D → {"Scalar", "File", "Relational", "Collectional"}* is the data-type function. DataType(d_j) returns the type of data, d_j □

Our NoSQL collectional data model supports registering hierarchically organized and collection oriented datasets. We formally define a NoSQL Collectional Data Model as follows:

Definition 2. A NoSQL Collectional Data Model A NCDM of level K can be formalized by $\mathbb{C}$ = ([C_1, C_2, …, C_K], V, R, I, K), where:

- I and K are two positive integers and $1 \leq I \leq K$.
- [C_1, C_2, …, C_K] is a list called **the primary key** of the collection; therefore functional dependency (C_1, C_2, …, C_K) → V holds.
- A prefix [C_1, C_2, …, C_I] of [C_1, C_2, …, C_K] is called a partition key of G, all tuples that correspond to the same value of [C_1, C_2, …, C_I] will be mapped to the same virtual machine in physical organization.
- V is an attribute for the value, it can take the type of a scalar value (INTEGER, STRING, FLOAT, DOUBLE, RELATIONNAME) or a relational name of schema specified by R. R is a relational schema. □

In Fig. 2 we show an example of OpenXC collectional data set from the automotive domain. It consists of three key attributes, *drivers*, *vehicles* and *traces*, and *value* is a relational name with three attributes <Name, Timestamp, Value>. In Fig. 3 we show the OpenXC collectional data set that is partitioned and stored in three virtual machines, vm_1-vm_3.

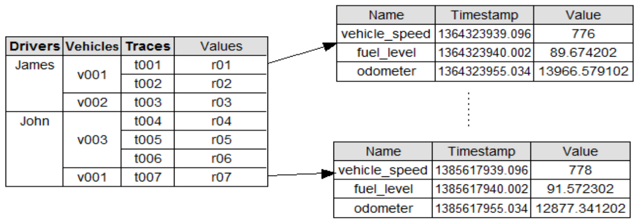
Definition 3. A Data Partitioner γ is used to partition a collectional instance c of schema $\mathbb{C}$ = ([C_1, C_2, …, C_K], V, R, I, K) into c_1, c_2, …, c_M such that $c_1 \cup c_2 \cup \ldots \cup c_M \equiv c$, where c_m is the partition for virtual machine m. Let α: $C_1 \times C_2 \times \ldots \times C_I \rightarrow [-2^{63}, 2^{63}-1]$ be a function that computes the token for a given collectional tuple t of c using only the value of the partition key (possibly composite). Let β: $[-2^{63}, 2^{63}-1] \rightarrow [1, M]$, then we have $\gamma(t) = \beta(\alpha(\pi(t, I)))$ where $\pi(t, I)$ is the projection of t over the first I attributes. □

Our proposed Map construct extends our previous notion of the Map workflow construct in [3] in two directions: 1) our Map constructs abstracts a data partitioner γ implicitly where the primary key of c is used as the partition key; 2) our Map constructs supports fully the NoSQL collectional data model, and thus fully exploits the data parallelism and the dynamic resource provisioning capability of our cloud resource manager.

Definition 4. The Map construct abstracts the partitioning and distribution of a large collectional data product over a set of M virtual machines for high-performance parallel processing. Given a workflow $w([i_1, i_2, \ldots, i_n], o)$ with n input ports and one output port, where i_1 takes a collectional data product as its input, we have $Map(w)(c, i_2, \ldots, i_n) = \bigcup_{m=1}^{M} w(c_m, i_2, \ldots, i_n)$. That is, the output of *Map(w)* on collectional data product *c* is equal to the union of the application of w on each partition c_m. In order to apply Map on *w*, *w* must satisfy the following constraints: 1) i_1 takes a collectional data product of schema $\mathbb{C}$ = *([C_1, C_2, …, C_K], V, R, I, K)* as input; 2) w is a primitive workflow or a composite workflow with no nesting Map/Reduce constructs; 3) $w(c, i_2, \ldots, i_n) = \bigcup_{t \in c} w(t, i_2, \ldots, i_n)$. That is, the output of w on input collection c is equal to the union of the application of w on each tuple in c. □

Our proposed Reduce construct extends our previous notion of the Reduce workflow construct in [3] in two directions: 1) automatic shuffling and redistribution of the input collectional dataset into multiple virtual machines; 2) executing the workflow that performs aggregation on the input collectional dataset based on the user provided key attribute in multiple virtual machines in parallel.

Definition 5. The Reduce construct abstracts the automatic shuffling, redistribution, aggregation of a large collectional data product based on a given key attribute C_I over a set of *M* virtual machines for high-performance parallel processing. Given a workflow $w([i_1, i_2, \ldots, i_n], o)$ with *n* input ports and one output port, where i_1 takes a collectional data product as its input, we have $Reduce(w, C_I)(c, i_2, \ldots, i_n)$

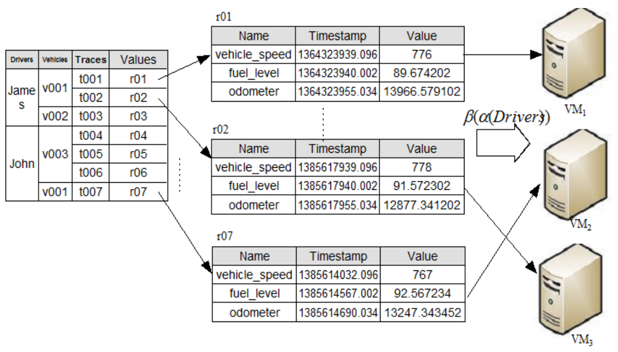


Drivers	Vehicles	Traces	Values
James	v001	t001	r01
		t002	r02
	v002	t003	r03
John	v003	t004	r04
		t005	r05
		t006	r06
	v001	t007	r07

Name	Timestamp	Value
vehicle_speed	1364323939.096	776
fuel_level	1364323940.002	89.674202
odometer	1364323955.034	13966.579102

Name	Timestamp	Value
vehicle_speed	1385617939.096	778
fuel_level	1385617940.002	91.572302
odometer	1385617955.034	12877.341202

Fig. 2. OpenXC collectional data product.



Drivers	Vehicles	Traces	Values
James	v001	t001	r01
		t002	r02
	v002	t003	r03
John	v003	t004	r04
		t005	r05
		t006	r06
	v001	t007	r07

r01

Name	Timestamp	Value
vehicle_speed	1364323939.096	776
fuel_level	1364323940.002	89.674202
odometer	1364323955.034	13966.579102

r02

Name	Timestamp	Value
vehicle_speed	1385617939.096	778
fuel_level	1385617940.002	91.572302
odometer	1385617955.034	12877.341202

r07

Name	Timestamp	Value
vehicle_speed	1385614032.096	767
fuel_level	1385614567.002	92.567234
odometer	1385614690.034	13247.343452

Fig. 3. Example of OpenXC data partitioner

$= \bigcup_{m=1}^{M} w(c_m, i_2, \ldots, i_n)$. That is, the output of *Reduce(w, C$_l$)* on collectional data product c is equal to the union of the application of w on each group c_m. In each group c_m, for t_1, $t_2 \in c_m$ we have $\pi(t_1, C_l) = \pi(t_2, C_l)$. That is, all tuples in c_m have the same value for C_l. □

Note that in the Map construct, a Map workflow run is applied to each tuple in c, while in the Reduce construct, a Reduce workflow run is applied to a group of tuples in c, with each group shares the same value for the given attribute C_l. Therefore, the Reduce construct is to be applied to a workflow that implements an aggregation function that is applicable to each group of the given input collectional data product as reshuffled according to the given key attribute.

A workflow executor takes a big data workflow, provisions virtual machines in the cloud, partitions the input collectional data product, executes the tasks of the workflow on different virtual machines, and finally deprovisions the assigned virtual machines and presents the output of the workflow to the user. While type-A workflow executor is used to execute a graph-based workflow, individual Map/Reduce workflow tasks are executed by the type-B workflow executor. We present the algorithms for type-A and type-B workflow execution in the next section.

IV. Algorithms for Workflow Executors

In this section, we propose three new algorithms that are implemented in our cloud workflow executor. We automatically provision and deprovision virtual machines based on both the size of input datasets connected to the workflow and the structure of the workflow.

We provide two types of parallelism. First, we provide a workflow level parallelism, type-A, by leveraging the structure of the workflow, we cluster the given workflow into multiple workflow clusters. Each workflow cluster consists of a set of tasks that are executed in the same virtual machine. Different workflow clusters are executed in different virtual machines in parallel.

Second, we provide a task level parallelism, type-B, by leveraging our newly proposed NoSQL collectional data model. We automatically partition the input datasets into multiple virtual machines and the task is mapped to those machines and executed in parallel. We demonstrate Algorithm 1, 2 and 3 by using the example shown in Fig. 4.

A. Task Clustering

The goal of the Task Clustering Algorithm (T-Cluster) is to generate an initial workflow schedule that is based on the structure of the workflow. The cluster map consists of a list of pairs <t_i, *VMID*>, such that t_i is the name of the task and *VMID* is the identifier of a virtual machine. Each task in a workflow consists of a set of producers and a set of consumers that are connected to it, except for the entry and exit tasks. The entry tasks in the workflow do not contain any producer and instead a set of input datasets are connected to it. In the same manner, the exit tasks do not contain any consumer and instead a set of output stubs are connected to it, in order to visualize the final results of the workflow.

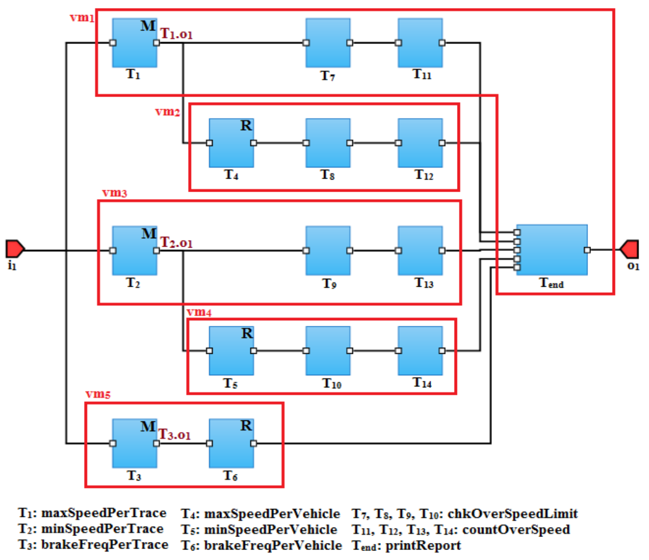


Fig. 4. An example big data workflow

In Fig. 4, we show an example big data workflow w from the automotive domain that is used to query the OpenXC dataset and compute the speeding and braking behavior of the driver. In Algorithm1, in line 4, we get all the tasks in w in some topological order and assign it to list *ts*. We validate the correctness of our algorithm for different topological order of *ts*. Let *ts* = <T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9, T_{10}, T_{11}, T_{12}, T_{13}, T_{14}, T_{end}>. In line 5, we iterate through each task t_i in *ts* and add t_i to the cluster map d. All the entry tasks in w are added to list *en* = <T_1, T_2, T_3>. Between lines 8 and 32, we iterate through each task t_i in t_s and add the consumer of t_i to list *cs*. For example consumers of T_1 are <T_4, T_7>. If task t_i is an element of list *en*, then we assign *VMID* to task t_i and increment *VMID*.

For example, <T_1,*1*> is added to the map since the default value of VMID = 1. If task t_i is not an element of list *en*, then we iterate through each consumer c in *cs*. For the first consumer, we assign the same VMID as that of the task and for the other consumers, we assign a different *VMID*. For example, for the consumers of T_1, the following pairs are added, <T_4,*1*>, <T_7,*2*>. *VMID* is incremented every time a new pair is added to the map. The final map d = <(T_1, *1*), (T_2,*3*), (T_3,*5*), (T_4,*2*), (T_5,*4*), (T_6,*5*), (T_7,*1*), (T_8,*2*), (T_9,*3*), (T_{10},*4*), (T_{11},*1*), (T_{12},*2*), (T_{13},*3*), (T_{14},*4*), (T_{end},*1*)> is returned as an output for workflow w. The output map d of w consists of five clusters assigned to five virtual machines.

B. Type-A Cloud Workflow Executor

The goal of the type-A workflow executor is to reduce the workflow makespan for a given workflow by running each cluster of the workflow in parallel in multiple virtual machines in the cloud. In Algorithm 2, in line 4, we get the cluster map d from Algorithm1(T-Cluster) for the workflow w. In line 5, we get all the tasks in workflow w in some topological order and store it in list *ts*. Between lines 6 and 24, we iterate through all task t_i in *ts* in parallel. We use

Algorithm1: Task Clustering

```
1:  function T-Cluster
2:  input: workflow specification w
3:  output: d, a map storing task-VM assignments.
4:  ts ← all tasks in w sorted in a topological order
5:  for each t ∈ ts do d[t] ← 0 end for
6:  en ← all entry tasks of w
7:  VMID = 1, fc = false
8:  for each t ∈ ts
9:     cs ← all consumers of t
10:   if (t ∈ en)
11:      d[t] ← VMID
12:      for each c ∈ cs
13:        if (d[c] = 0)
14:          d[c] ← VMID
15:          VMID ← VMID + 1
16:        end if
17:      end for
18:   else
19:      fc = true
20:      for each c ∈ cs
21:        if (d[c] = 0)
22:          if (fc = true)
23:            d[c] ← d[t]
24:            fc = false
25:          else
26:            d[c] ← VMID
27:            VMID = VMID + 1
28:          end if
29:        end if
30:      end for
31:   end if
32:  end for
33:  return d
34:  end function
```

inputready to determine whether all the datasets are available in order to execute a task. In line 7, we set *inputready[t_i]* to the total number of non-intermediate input ports of t_i. In Fig. 4, for example *inputready[T_1]* = 1, *inputready[T_4]* = 0.

At each iteration, every task except the entry tasks will wait until all the input datasets for the task becomes ready. For example, T_{end} will continue to wait until the tasks {T_{11}, T_{12}, T_{13}, T_{14}, T_6} are executed and the data movement from {T_{11}, T_{12}, T_{13}, T_{14}, T_6} to T_{end} is completed. Between lines 13 and 23, we iterate through each consumer c_i of task t_i in parallel. If the consumer c_i and task t_i are assigned to different virtual machines, then we move the data from *d[t_i]* to *d[c_i]*. We increment *inputready[c_i]* through a locking mechanism so that at a particular time only one consumer of one task can increment it. For each consumer c_i, the number of input ports is calculated. If *inputready[c_i]* is equal to the total number of input ports of c_i, then we send a signal to the consumer c_i as a wake up call. At that point *inputready[c_i]* is validated to check whether all the datasets needed to execute c_i is ready. If the datasets are ready, then c_i is executed and

Algorithm2: Type-A workflow executor

```
1:  function Type-A
2:  input: workflow specification w
3:  output: exit code
4:  d ← T-Cluster(w)
5:  ts ← all tasks in w sorted in a topological order
6:  forall task $t_i$ ∈ ts in parallel
7:    inputready[$t_i$] = |$t_i$.IP|- |$t_i$.IP^|
8:    while (inputready[$t_i$] < | $t_i$.IP|)
9:      wait(sig[$t_i$]);
10:   end while
11:   execute $t_i$ on d[$t_i$]
12:   cs ← all consumers of $t_i$
13:   forall consumer $c_i$ ∈ cs in parallel
14:      if (d[$t_i$] ≠ d[$c_i$])
15:        Move OutputOf ($t_i$, $c_i$) from d[$t_i$] to d[$c_i$]
16:      end if
17:      lock → acquire()
18:        inputready[$c_i$] = inputready[$c_i$] + 1
19:      lock → release()
20:      if (inputready[$c_i$] = |$c_i$.IP|)
21:        signal(sig[$c_i$]);
22:      end if
23:   end in parallel
24: end in parallel
25: return SUCCESS
26: end function
```

the consumers of c_i are processed. For example only after execution of the tasks *{T_{11}, T_{12}, T_{13}, T_{14}, T_6}*, the signal to wake up T_{end} is sent from T_{14}.

C. Type-B Cloud Workflow Executor

The goal of the type-B workflow executor is to reduce the task makespan for any task that has the Map or the Reduce construct applied on it. A construct is a high order function that is used to transform any given function into another sophisticated function. In our case, a construct is applied on a task to transform the task into a Map or a Reduce task. The Map construct is applied to a task that performs extracting, filtering and transformation. The Reduce construct is applied to a task that performs aggregation, summarization, filtering and transformation. Besides, the Map and Reduce constructs also differ in the way the input dataset is partitioned. Our data partitioner, which is inspired from Cassandra uses our custom hash function, to distribute any given NoSQL collectional dataset into multiple virtual machines. Our cloud infrastructure manages a ring with a range for each virtual machine. We assign multiple tokens for each range. The advantage foreseen in our approach is that we support the dynamic addition and deletion of virtual machines and still manage to keep our key to token mapping information intact. We plan to discuss our partitioner as a separate research article.

Algorithm3: Type-B workflow executor

```
1:  function type-B
2:  input: task name $t_i$, type of construct $toc$,
    partition size $ps$, number of task runs per
    virtual machine $RunsPerVM$, partition key
    for reduce $r_k$
3:  output: final output of task $t$
4:  $out \leftarrow []$, $vms \leftarrow []$, $z = 0$, $n = 0$
5:  $in \leftarrow t_i.inputdatasets$;
6:  for each dataset $d \in in$
7:    if (Type $(d)$ = collectional)
8:      $z$ = size of $d$
9:      $n = z / ps / RunsPerVM$
10:     $vms \leftarrow$ CRM.provision($n$)
11:     if ($toc$ = map)
12:        $ParKeys \leftarrow$ get all keys of $d$
13:     else if ($toc$ = reduce)
14:        $ParKeys \leftarrow r_k$
15:     end if
16:    Partition$(d, ParKeys)$ to $vms$
17:   else
18:    Move($d$) to $vms$
19:   end if
20: end for
21: forall $vm \in vms$ in parallel
22:   $result \leftarrow$ Execute task $t_i$ in $vm$
23:   $out \leftarrow out \cup result$
24: end in parallel
25: return $out$
26: end function
```

We apply the Map construct on the tasks $\{T_1, T_2, T_3\}$ and the Reduce construct on tasks $\{T_4, T_5, T_6\}$ on the big data workflow w shown in Fig. 4. In Algorithm 3, in line 4, we initialize out, z, n and vms. In line 5, we get all the input datasets connected to task t_i and store them in in. Between lines 6 and 20, we iterate through all the datasets d in in. In line 7, we validate whether the type of the dataset d is collectional. On validation success, we compute the total number of virtual machines dynamically based on the size of the dataset and the user configured value for partition size (ps) and total number of task runs per virtual machine ($RunsPerVM$). In line 10, we provision the virtual machines. In line 11, we validate the type of construct applied to task t_i. If the type of the construct is a Map, then we partition the data with all the key attributes in dataset d. If the type of the construct is Reduce, then we partition the data by a user provided key attribute r_k. For all other datasets connected to task t_i that are not of type collectional, we move the original dataset to all the virtual machines. Between lines 21 and 24, we run task runs of t_i in parallel in all the virtual machines vms. The results from all task runs are combined together and returned as the final output out.

V. Case Study and Experiments

In our DATAVIEW system, we implemented a big data workflow called the *Autoanalytics* workflow to show the strength of our proposed NoSQL collectional data model and the scalability features. The *Autoanalytics* workflow is used to analyze the data collected from vehicles and provide insights on the risk level based on the drivers driving behavior. OpenXC is an open source platform that is used as a source to generate a wealth of data from the vehicle through a hardware device that is installed in the car. As the average adult driver in the US may generate up to 75 Gb of such driving data annually, the total amount of data generated in the US may exceed 14 Eb (10^{18} bytes) per year [12, 16]. We collected data from X drivers for one hour with X= 5, 10, 15, 20, 25 resulting datasets of size in the range of 1GB-5GB. Because of the dynamic nature of the growth of size of the dataset and the 5V big data challenges incurred in the domain, we consider our *Autoanalytics* workflow as a big data workflow.

In Fig. 5, we show the *Autoanalytics* big data workflow. The first step is *ExtractDriverDetails* that accepts the collectional OpenXC dataset as input and filter by their *VehicleId*. The second step is *ComputeSpeedDistribution* that is used to find the topK vehicle speed and the distance driven without pressing the brake. The third step is *AddressFinder* that is used to find the geographic location of the vehicle during the time at which the signal was captured. We use Google Places API to determine the address from the latitude and longitude signal values. The fourth step is *ComputeRoadType* that is used to find the type of the road (highway or local) for each geographic location computed in the third step. The fifth step is *SpeedLimitFinder* that is used to find the speed limit posted on those geographic locations based on the road type. The sixth step is *ValidateVehicleSpeed* that is used to compare the vehicle speed generated in step-2 with the actual speed limit in the closest latitude longitude. The output of the task indicates if the driver was below or over the speed limit. The 7th step is *ComputeSimilarityScore* that is used to compare several drivers with different traces to identify the similarity between them. The final output of the *ComputeSimilarityScore* generates a report to show the list of good drivers and bad drivers along with their driving score.

We apply the type-B Map construct on step-1, step-2 and executed them in the range of 1-25 virtual machines simultaneously. We apply the type-B Reduce construct on step-7 by (*key=DriverName*) to partition/group all the signal values associated with each driver into the same machine. We apply the type-A cloud workflow executor for 25 drivers and executed the workflow in the range of 1-25 virtual machines in the EC2 cloud. We performed our experiments in the OpenXC dataset with the range of 1-5GB on 1 to 25 machines. Our experimental results show that our type-A executor performed well by reducing the workflow makespan. And we applied the Map and the

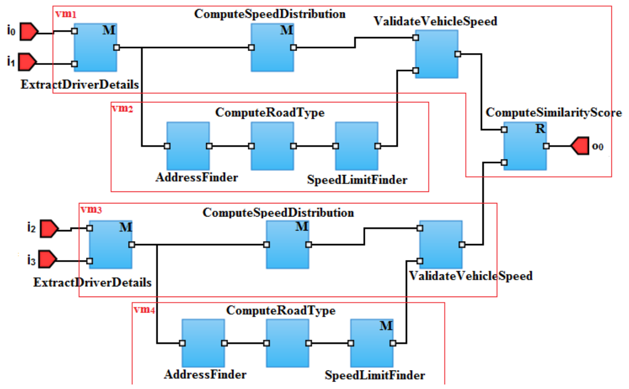


Fig.5. An automative OpenXC big data workflow

Reduce construct on the individual tasks in the workflow. Our type-B executor performed well by reducing the individual task makespan and as a whole also reducing the workflow makespan even more. In Fig.6, we show the experimental results that were performed using the OpenXC dataset in the Amazon EC2 cloud environment.

VI. Related Work

Existing workflow management systems such as Kepler [4], VisTrails [6] and Taverna [5] do not support a scalable data model that is suitable for processing big data in the cloud. Kepler proposes a collection-oriented model in which the data is nested in different levels as collections and sub collections with arbitrary data type. The data model is represented using XML and is semi structured in nature, whereas our collectional data model is well structured and is much simple to process big data workflows. VisTrails provide a good visualization framework and support a semi-structured representation of the data, it strongly lacks in storing hierarchical information such as collections and the workflow engine does not support scalable workflow execution in the cloud. Taverna supports singleton values such as strings, byte arrays and list of singletons. Lists are defined to a specific level and are not capable to handle nested data to arbitrary levels. Fei et al. [1, 3] propose a collectional data model to process scientific workflows in one machine and a set of well-defined operators and constructs. The proposed model and the operators are not scalable and do not consider challenges of data partitioning and workflow execution in the cloud. Wang et al. [2] propose an approach to improve the programmability and scaling flexibility of big data application through different parallelization techniques. They propose a list of DDP patterns that are used to process key value pairs and parallelize the execution of the user-defined functions.

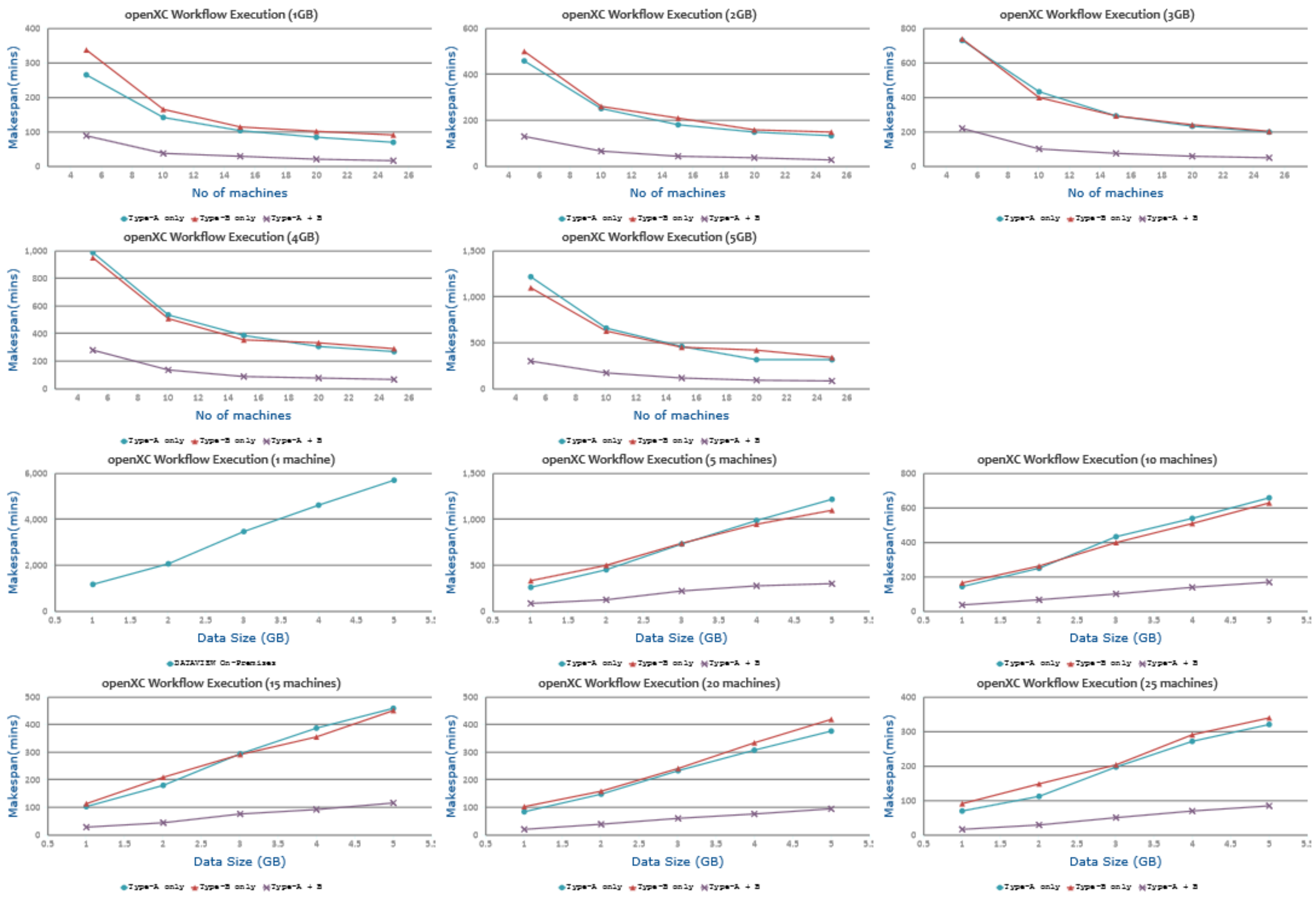


Fig. 6. Workflow makespans by varying the number of virtual machines and datasets.

Although their approach is similar to the workflow constructs proposed by us, our Map and Reduce workflow constructs can be applied to any given workflow by leveraging our proposed NoSQL collectional data model.

Existing big data NoSQL databases are classified into the following four category of databases: 1) key-value databases [7], such as Memcached and Redis, 2) document-oriented databases [8], such as RavenDB, MongoDB and CouchDB, 3) column-family databases [9, 11], such as Apache Cassandra and HBase, 4) graph databases [10], for example: Neo4J, FlockDB and GraphDB. The existing NoSQL databases are not suitable for big data workflows because they do not support ad hoc sophisticated analysis and is not extendible to workflow optimization.

None of the above techniques provides a scalable data model for data centric big data workflows. In this paper, we propose a NoSQL collectional data model that is both scalable and at the same time is well structured for performing ad hoc analysis on large datasets. Moreover, we also propose two new cloud workflow executors that take advantage of the proposed NoSQL data model to improve the performance of workflow execution.

VII. Conclusions and future work

In this paper, we propose a NoSQL data model that: 1) supports high-performance MapReduce-style workflows that automate data partitioning and data-parallelism execution. In contrast to the traditional MapReduce framework, our MapReduce-style workflows are fully composable with other workflows enabling dataflow applications with a richer structure; 2) automates virtual machine provisioning and deprovisioning on demand according to the sizes of input datasets; 3) enables a flexible framework for workflow executors that take advantage of the proposed NoSQL data model to improve the performance of workflow execution. We presented a case study and experiments that show the competitive advantages of our proposed NoSQL collectional data model and the cloud workflow executors. Ongoing work includes implementing a new set of workflow constructs that can be used for efficient parallel processing.

Acknowledgment

This work is supported by U.S. National Science Foundation under ACI-1443069 and is based upon work supported in part by the National Science Foundation under Grant No. 0910812.

References

[1] X. Fei, et al., "A Collectional Data Model for Scientific Workflow Composition", in Proc. of the 2010 IEEE International Conference on Web Services (ICWS'10), pp. 567-574.

[2] J. Wang, et al., "Big data applications using workflows for data parallel computing", Computing in Science & Engineering, vol.16, no. 4, pp. 11-21, 2014.

[3] X. Fei, et al., "A MapReduce-Enabled Scientific Workflow Composition Framework", in Proc. of the 2009 IEEE International Conference on Web Services (ICWS'09), pp. 663-670.

[4] T. McPhillips, et al., "Collection-oriented scientific workflows for integrating and analyzing biological data", in Proc. of DILS, vol. 4075, 2006, pp. 248–263.

[5] D. Turi, et al., "Taverna workflows: Syntax and semantics", in Proc. of eScience, 2007, pp. 441–448.

[6] S. Callahan, et al., "VisTrails: visualization meets data management", in Proc. of SIGMOD, 2006, pp. 745–747.

[7] R. Nishtala, et al., "Scaling Memcache at Facebook", in Proc. of 10th USENIX conference on Networked Systems Design and Implementation (nsdi'13), pp. 385-398.

[8] F. Chang, et al., "Bigtable: A distributed storage system for structured data", in Proc. of ACM Transactions on Computer Systems (TOCS'08), vol. 26, no. 2 (2008), 4.

[9] M.N. Vora, et al., "Hadoop-HBase for large-scale data", in proc of 2011 International Conference on Computer Science and Network Technology (ICCSNT'2011), pp. 601-605.

[10] R. Angels, et al., "Survey of graph database models", in proc of ACM Computing Surveys (CSUR'08), vol. 40, no. 1 (2008), 1.

[11] J. Dean, et al., "MapReduce: Simplified data processing on large clusters", in Proc. of OSDI, 2004, pp. 137-150.

[12] A. Kashlev, et al., "A System Architecture for Running Big Data Workflows in the Cloud", in Proc. of the 2014 IEEE International Conference on Services Computing (SCC'14), pp. 51-58.

[13] A. Mohan, et al., "Addressing the Shimming Problem in Big Data Scientific Workflows", in Proc. of the 2014 IEEE International Conference on Services Computing (SCC'14), pp. 347-354.

[14] C. Olston, et al., "Pig Latin: a not-so-foreign language for data processing", in Proc. of SIGMOD, 2008, pp. 1099-1110.

[15] Y. Yu, et al., "DryadLINQ: A system for generalpurpose distributed data-parallel computing using a high-level language", in Proc. of OSDI, 2008, pp. 1-14.

[16] D. Williams, "The Arbitron national in-car study", Arbitron Inc., 2009.

[17] C. Lin, et al., "A Reference Architecture for Scientific Workflow Management Systems and the VIEW SOA Solution", IEEE Transactions on Services Computing, vol.2, no. 1, pp. 77-92, 2009.

[18] Chunhyeok Lim, et al., "OPQL: Querying Scientific Workflow Provenance at the Graph Level", Data & Knowledge Engineering (DKE), 88(2013), pp.37-59, 2014.

[19] D. Ruan, et al., "A User-Defined Exception Handling Framework in the VIEW Scientific Workflow Management System", in Proc. of the IEEE International Conference on Services Computing (SCC), pp.274-281, Honolulu, Hawaii, 2012.

[20] G. Bell, et al., "Beyond the data deluge", *Science* 323.5919 (2009), pp.1297-1298.

[21] H. Chen, et al., "Business Intelligence and Analytics: From Big Data to Big Impact," *MIS quarterly* 36.4 (2012), pp.1165-1188.

[22] J. Han, et al., "Survey on NoSQL database", Pervasive computing and applications (ICPCA), 2011 6th international conference on. IEEE, 2011, pp. 363-366.

[23] J. Pokorny, "NoSQL databases: a step to database scalability in web environment", International Journal of Web Information Systems 9.1 (2013), pp.69-82.

[24] A. Chebotko, et al., "A Big Data Modeling Methodology for Apache Cassandra." *Big Data (BigData Congress),* 2015 IEEE International Congress on. IEEE, 2015, pp. 238-245

[25] M. Ebrahimi, et al., "TPS: A task placement strategy for big data workflows." Big Data (Big Data), 2015 IEEE International Conference on. IEEE, 2015, pp. 523-530.

[26] M. Ebrahimi, et al., "BDAP: A Big Data Placement Strategy for Cloud-Based Scientific Workflows." Big Data Computing Service and Applications (BigDataService), 2015 IEEE First International Conference on. IEEE, 2015, pp. 105-114.