Provenance for MapReduce-based Data-Intensive Workflows

Daniel Crawl, Jianwu Wang, Ilkay Altintas*
San Diego Supercomputer Center
University of California, San Diego
9500 Gilman Drive
La Jolla, CA 92093-0505
{crawl, jianwu, altintas}@sdsc.edu

ABSTRACT

MapReduce has been widely adopted by many business and scientific applications for data-intensive processing of large datasets. There are increasing efforts for workflows and systems to work with the MapReduce programming model and the Hadoop environment including our work on a higher-level programming model for MapReduce within the Kepler Scientific Workflow System. However, to date, provenance of MapReduce-based workflows and its effects on workflow execution performance have not been studied in depth. In this paper, we present an extension to our earlier work on MapReduce in Kepler to record the provenance of MapReduce workflows created using the *Kepler+Hadoop* framework. In particular, we present: *(i)* a data model that is able to capture provenance inside a MapReduce job as well as the provenance for the workflow that submitted it; *(ii)* an extension to the *Kepler+Hadoop* architecture to record provenance using this data model on MySQL Cluster; *(iii)* a programming interface to query the collected information; and *(iv)* an evaluation of the scalability of collecting and querying this provenance information using two scenarios with different characteristics.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Applications*; H.2.4 [**Database Management**]: Systems—*Distributed Databases*

General Terms

Design, Experimentation, Performance

Keywords

MapReduce, Provenance, Scientific Workflows

*Corresponding author.

WORKS'11, November 14, 2011, Seattle, Washington, USA.

1. INTRODUCTION

A very large amount of scientific data is generated daily with applications in genomics, metagenomics, biological and biomedical sciences, environmental sciences, geosciences, ecology, and many other fields, posing new challenges in data, computation, network, and complexity management areas. Efficient and comprehensive analysis of the generated data requires distributed and parallel processing capabilities. New computational techniques and efficient execution mechanisms for these data-intensive workloads are needed.

Recently, many concrete distributed execution (mostly data parallel) patterns, e.g., MapReduce [5], All-Pairs [17], MasterSlave [18] (also called MasterWorker [15]), have been identified and supported by corresponding frameworks, e.g., Twister [6], Sector/Sphere [11], DryadLINQ [13], and Hadoop[1]. The advantages of these patterns include: *(i)* a higher-level programming model to parallelize user programs automatically; *(ii)* support for parallel data processing; *(iii)* good scalability and performance acceleration when executing on distributed resources; *(iv)* support for run-time features such as fault tolerance and security; and *(v)* simplification of the difficulty for parallel programming in comparison to traditional parallel programming interfaces such MPI [10] and OpenMP [3].

MapReduce [5] has been widely adopted by many business and scientific applications for data-intensive processing of large datasets. There are increasing efforts for workflows, e.g., [21, 7], including our previous work [19, 20], and other systems, e.g., Oozie[2], Azkaban[3] and Cascading[4], to work with the MapReduce programming model and the Hadoop environment. However, none of these early studies discuss provenance support for MapReduce-based workflows.

There have been a few recent studies to explore storing and querying MapReduce related provenance in distributed environments. Franke *et al.* [8] compare multiple queries against the provenance stored in HBase[5] and MySQL Cluster.[6] However, in these experiments, provenance information is loaded from existing data sources instead of live workflow executions. Ikeda *et al.* [12] discuss how to capture provenance during the execution of MapReduce workflows and supports backward and forward traversal of data

[1] http://hadoop.apache.org/core/, 2011.
[2] http://yahoo.github.com/oozie/index.html, 2011.
[3] http://sna-projects.com/azkaban, 2011.
[4] http://www.cascading.org, 2011.
[5] http://hbase.apache.org/, 2011.
[6] http://www.mysql.com/products/cluster/, 2011.

dependencies. However, this framework does not provenance inside MapReduce tasks.

Contributions. This paper extends our earlier on MapReduce in Kepler [14] with the ability to re provenance of MapReduce workflows created using *pler+Hadoop* framework. In particular, we prese data model that captures provenance inside a Ma job as well as the provenance for the workflow that ted it; *(ii)* an extension to the *Kepler+Hadoop* arc to record provenance using this data model on a Cluster; *(iii)* a programming interface to query the information; and *(iv)* an evaluation of the scalabili lecting and querying this provenance information u scenarios with different characteristics.

Outline. The rest of this paper is organized as In Section 2, we provide an extended architecture t execute, and record provenance for MapReduce wor Kepler. Section 3 describes the provenance data m querying interface used by this architecture. In S we provide two example usecases and discuss thei teristics for why they are chosen as application scer Section 5, we explain our experimental framework and show the results of our experiments for recording and querying provenance using MySQL Cluster. Section 6 discusses our conclusions and future work.

2. ARCHITECTURE

In [19], we presented the *Kepler+Hadoop* framework for executing MapReduce-based Kepler workflows on Hadoop. This architecture consists of three layers: *(i)* the Kepler GUI, *(ii)* the Hadoop Master, and *(iii)* the Haoop Slaves for Map and Reduce tasks. During the initialization of the MapReduce actor, the input data stored on the local file system will be first transferred to the Hadoop File System (HDFS) which automatically partitions and distributes the data to Map tasks. After the data stage-in phase, the Kepler execution engine and the Map and Reduce sub-workflows in the MapReduce actor will be distributed to the slave nodes for the MapReduce tasks. The Map and Reduce sub-workflows will be executed with the data blocks on the slaves. Throughout execution, Hadoop provides fault-tolerance through slave monitoring and data replication mechanisms. After execution completes on the Hadoop slaves, the MapReduce actor automatically transfers any output data from HDFS to the local file system for processing by downstream actors of the workflow.

In this section, we present an extension of the architecture in [19] to include distributed provenance recording using MySQL Cluster and a separate client interface for querying and visualizing the collected provenance information. These extensions are illustrated in Figure 1, consisting of the following components that execute either on a client or in a cluster or cloud environment:

- **Kepler Scientific Workflow System**

  The Kepler scientific workflow system [14], is developed by a cross-project collaboration and inherited from Ptolemy II[7]. Kepler provides a graphical user interface (GUI) for designing workflows composed of a linked set of components, called *Actors*, that may execute under different *Models of Computations* (MoCs)

[7] http://ptolemy.berkeley.edu/ptolemyII, 2011.

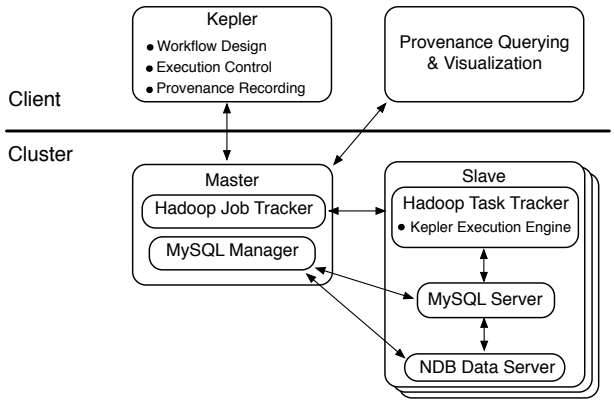

Figure 1: MapReduce provenance architecture.

[9] implemented as *Directors*. Actors are the implementations of specific functions that need to be performed and communication between actors takes place via tokens that contain both data and messages. Directors specify what flows as tokens between the actors; how the communication between the actors is achieved; when actors execute (a.k.a. fire); and when the overall workflow can stop execution. The designed workflows can then be executed through the same user interface or in batch mode from the command-line or other applications. In addition, Kepler provides a provenance framework [1, 4] that keeps a record of chain of custody for data and process products within a workflow design and execution. This facilitates tracking the origin of scientific end-products, and validating and repeating experimental processes that were used to derive these scientific products.

In this architecture, the Kepler GUI is used for designing MapReduce workflows, starting and stopping the workflow execution, and recording the provenance for the part of the workflow running on the client machine. The sub-workflows in the MapReduce actor execute on the Cluster in the Hadoop Task Trackers as Map or Reduce tasks. Provenance for these sub-workflows is stored on the MySQL Cluster running on the Cluster.

Extensions to the Kepler Provenance Framework. We made several updates to the provenance framework [1] to capture provenance in MapReduce jobs. The existing framework records both data and dependencies between actors executing on the same machine. However, in a MapReduce job, data can be transferred between nodes, e.g., the output of a Map task may be read by a Reduce task running on a different node. To capture these inter-node data dependencies, or *transfer events*, we extended the provenance recording API. When an actor creates data that will be transferred to another node, the actor first registers the data to get a unique identifier. Next, this identifier is transferred along with the data to other nodes. Finally, each actor that reads the data on a receiving node uses the identifier to notify the provenance system that the actor is reading the transfer event. The MapReduce actor uses transfer events to capture three

sets of dependencies: the inputs to each Map task, the outputs of each Map task that are read by Reduce tasks, and the output of each Reduce task. Unlike the implicit recording of data dependencies during workflow execution, actors must explicitly invoke the provenance API to manage transfer events. Section 4.1 gives examples of the MapReduce actor's transfer events.

Several updates to the provenance framework were implemented to reduce execution overhead. All writes to provenance storage are performed **asynchronously**, which allows the workflow to execute uninterrupted while provenance is captured. Further, writes are made in **batches**. While this adds a minor delay between the time provenance is captured and made available for querying, it significantly reduces the amount of network communication and processing done by the provenance storage servers.

- **Provenance Querying and Visualization**

  This component is used for querying the collected provenance information using the Kepler Query API (see Section 3.3) and visualizing the results of these queries. It is currently implemented as a stand-alone desktop application.

- **Master**

  The Master node runs management servers and consists of: *(i)* the Hadoop Job Tracker that manages and monitors the Hadoop Task Trackers on slave nodes, and distributes the Map and Reduce sub-workflows to slave nodes; and *(ii)* the MySQL Manager that monitors the MySQL Server and NDB Data Server on slave nodes.

- **Slave**

  The architecture uses slave nodes in a cluster or cloud environment, each consisting of: *(i)* the Hadoop Task Tracker that runs Map and Reduce tasks, which in turn execute the Map and Reduce sub-workflows in the Kepler execution engine; *(ii)* the MySQL Server that handles SQL commands from the provenance recorder of the Kepler execution engine, and reads and writes data to NDB Data Servers; and *(iii)* the NDB Data Server that provides a distributed storage engine for MySQL with load-balancing and data replication for fault-tolerance. (All of our experiments used a replication factor of two.)

3. PROVENANCE DATA MODEL

The provenance data model records the values of all data passed between actors during workflow execution, as well as the dependencies between data and actor executions. We use terminology from the Open Provenance Model [16] to describe the captured entities and relationships: an *artifact* is an immutable piece of data, and a *process* is a series of actions performed on artifacts possibly resulting in new artifacts. In our architecture, tokens and transfer events are artifacts, and actor executions (a.k.a. invocations) are processes.

3.1 Entity Identifiers

Artifacts and processes are represented by identifiers in the data model. The format of the identifier must meet three requirements: **unique**, **small**, and **fast to generate**. Each artifact and process is assigned a unique identifier to distinguish it from other entities generated during the same workflow execution and entities generated during other executions. Since MapReduce workflows processing large amounts of data may have millions of artifacts and tens of thousands of processes, the identifier should be as small as possible to minimize the size of provenance storage. Further, identifiers should be fast to generate to reduce workflow execution overhead.

Several choices for the identifier format were considered. One option is *counters* provided by the RDBMS (either Sequences or autoincrement columns). Counters are unique and small, but locking them to atomically increment and read their values creates a significant overhead when many MapReduce tasks running in parallel try to access it. Another identifier format is the *Universally Unique Identifier (UUID)*. While UUIDs are unique, they are slow to generate and 128-bit in size.

In order to meet all three requirements, we created a new identifier format. For artifacts, the format is $A_{id} = R\_S\_N$, where R is the workflow execution (a.k.a. run) number, S the identifier of the MapReduce task within the MapReduce job (a.k.a. sub-run) and N the artifact number within S. An identifier in this format is unique since R is different for each workflow execution, S is generated by Hadoop to uniquely identify each MapReduce task, and N is unique within the MapReduce task S. The format is also small, requiring only a few characters, and fast to generate: R is generated using a counter from the RDBMS, but only needs to be done once per workflow execution; S is automatically created by Hadoop; and N is an atomic counter only incremented and read within a single MapReduce task.

The identifier format for processes is $P_{id} = R\_S\_T\_F$ where R is the run number, S the sub-run, T the actor number, and F the number of times actor T has executed during S. The actor number T uniquely identifies the actor in the workflow, and since the workflow definition does not change during execution, T can be generated once before execution starts. Similar to the artifact number N in A_{id}, the execution number F can be generated by using an atomic counter within each MapReduce task.

3.2 Model Schema

Provenance data is stored in a relational schema defined as:

- $artifact(A_{id}, V, C)$ denotes that artifact A_{id} has a value V and the checksum of the value is C.
- $compress(C, M)$ denotes that a checksum C has a compressed value M.
- $actor(E, T, R)$ denotes that an actor named E has a number T for workflow run R.
- $dependency(A_{id}, P_{id}, D)$ denotes that artifact A_{id} was read or written, specified by D, by process P_{id}.

During workflow execution, artifact values are written to the $artifact$ table. However, many artifacts may have the same value and removing duplicates saves space. After the workflow execution has completed, a post-processing step removes duplicated values from the $artifact$ table and adds one copy to the $compress$ table. The value for an artifact

can be found in the *compress* table by using the value's checksum, which is stored in both tables.

Artifact values are also compressed. For some types of MapReduce workflows, this can reduce storage size dramatically. For example, in Word Count (see Section 4.1), many artifacts are sequences of numbers representing the occurrences of words in a file. These sequences can have high compression ratios since they often contain many repeating numbers.

The *dependency* table stores the relationships between artifacts and processes, i.e., which artifacts were read and written by each process. As defined above, the artifact and process identifier formats are $A_{id} = R\_S\_N$ and $P_{id} = R\_S\_T\_F$, respectively. Several fields are the same in both identifiers, and storage space can be reduced by not duplicating these fields when they have the same values. We therefore define a dependency table without duplication:

- *dependency_nodup*(R, S, N, T, F, D) denotes that in a workflow run R and sub-run S, artifact number N was read or written, specified by D, by execution F of actor number T.

By storing the artifact and process identifier fields separately in the table, only one copy of R and S are required for each dependency when R and S are the same for the artifact and process.

3.3 Querying

The Kepler Query API provides an interface to retrieve provenance information that is independent of the provenance data model used to store data. Applications using this API do not have to be changed to query provenance from different data models. The following list describes a few example queries:

1. `getArtifactsForProcess(p, false)` returns the artifact(s) written by process p.
2. `getProcessesForArtifact(a, true)` returns the process(es) that read artifact a.
3. `getDataValue(a)` returns the data value of artifact a.

We created an implementation of the Query API for the relational data model described in the previous section. This implementation is realized using SQL queries. The following are the SQL queries for the above examples:

1. `SELECT artifact FROM dependency WHERE process = p AND isread = false`
2. `SELECT process FROM dependency WHERE artifact = a AND isread = true`
3. `SELECT value from artifact WHERE id = a`[8]

Visualization. We created an application to display provenance graphs of workflow executions. This application uses the Query API to traverse the dependencies in the graph and retrieve artifact values and process names. A provenance graph provides insights into the workflow execution and facilitates debugging workflow components. We describe an example graph in the next section.

[8]The artifact's value may be stored in the *compress* table; in this case a second query is performed and not shown here.

4. APPLICATIONS

In this section we present two applications that were built using the MapReduce actor, and several example types of provenance information we can query by executing these applications in our architecture.

4.1 Word Count

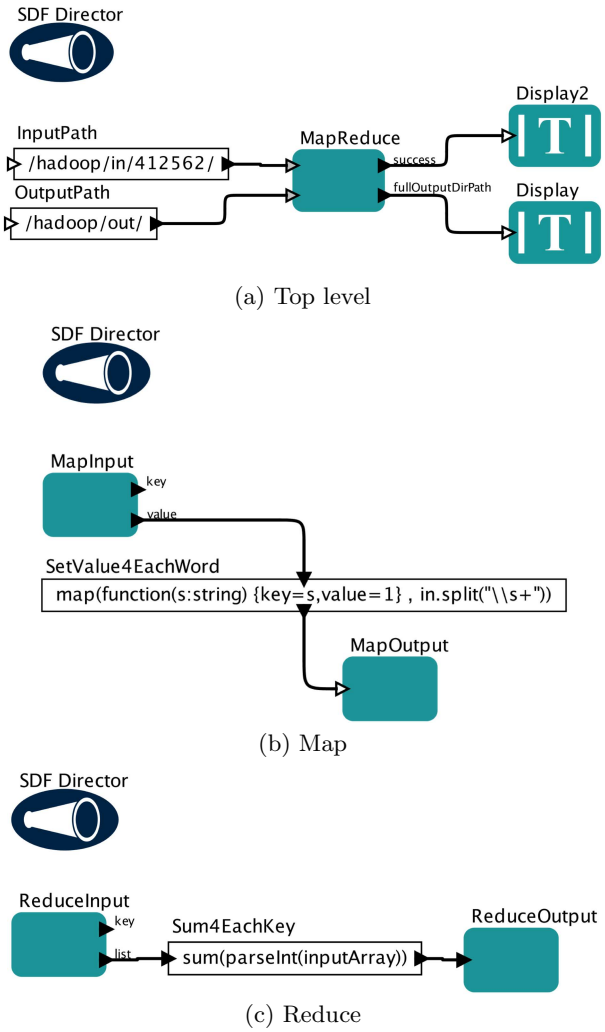

Figure 2: Word Count workflow.

Word Count is the canonical example for the MapReduce programming model [5]. This application counts the number of occurrences of each word in a large collection of text documents. Figure 2 shows the implementation of Word Count as a Kepler workflow, which uses the MapReduce actor to execute Map and Reduce sub-workflows in Hadoop. The input to the MapReduce actor is the directory name containing the input text documents. The Map sub-workflow shown in Figure 2(b) counts the occurrences for a single file, and produces a list of key-value pairs of each word and count. The Reduce sub-workflow shown in Figure 2(c) adds the occurrences for each word read in Map, and outputs the total sum. Once the Hadoop job completes, the MapReduce actor writes the output directory and a boolean value to denote if the job was successful.

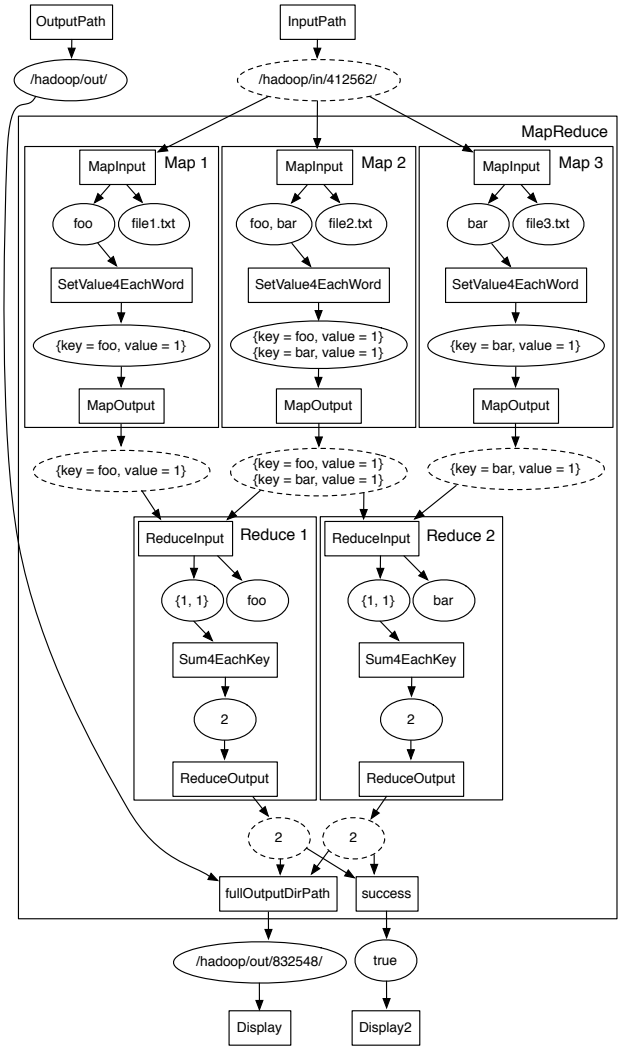

Figure 3: Provenance graph of Word Count workflow execution.

Figure 3 displays the provenance graph of Word Count reading three text files. In this graph, circles represent artifacts, rectangles are processes, and arrows denote read/write dependencies between artifacts and processes. In this example, the input directory contains three files, `file1.txt`, `file2.txt`, and `file3.txt`, and each is read by a separate Map task: *Map 1* reads `file1.txt`, which contains a single occurrence of `foo`; *Map 2* reads `file2.txt`, which contains a single occurrence of `foo` and `bar`; and *Map 3* reads `file3.txt`, which contains a single occurence of `bar`. Since there are two unique words in the input files, there are two Reduce tasks: *Reduce 1* adds the occurrences for `foo`; and *Reduce 2* adds the occurrences for `bar`.

Figure 3 contains several transfer events that are represented as circles with dashed lines. As described in Section 2, a transfer event is an inter-node data dependency, and the MapReduce actor captures three sets of these dependencies: Map tasks read the directory containing the input dataset, `/hadoop/in/412562/`; Reduce tasks read the outputs of the Map tasks; and the MapReduce actor reads the outputs of the Reduce task.

4.2 BLAST

BLAST [2] discovers the similarities between two biological sequences, and is one of the most widely used algorithms in bioinformatics. Executing BLAST can be a data-intensive process since the query or reference data can have thousands to millions of sequences.

We have created a BLAST workflow using the MapReduce actor to achieve parallel processing by splitting the input query data. The MapReduce actor in Figure 4(a) includes sub-workflows for Map and Reduce, as shown in Figures 4(b) and 4(c), respectively. Each Map sub-workflow runs BLAST to process a subset of the query sequences against the reference database. In the Reduce sub-workflow, the outputs for each subset are merged into one output.

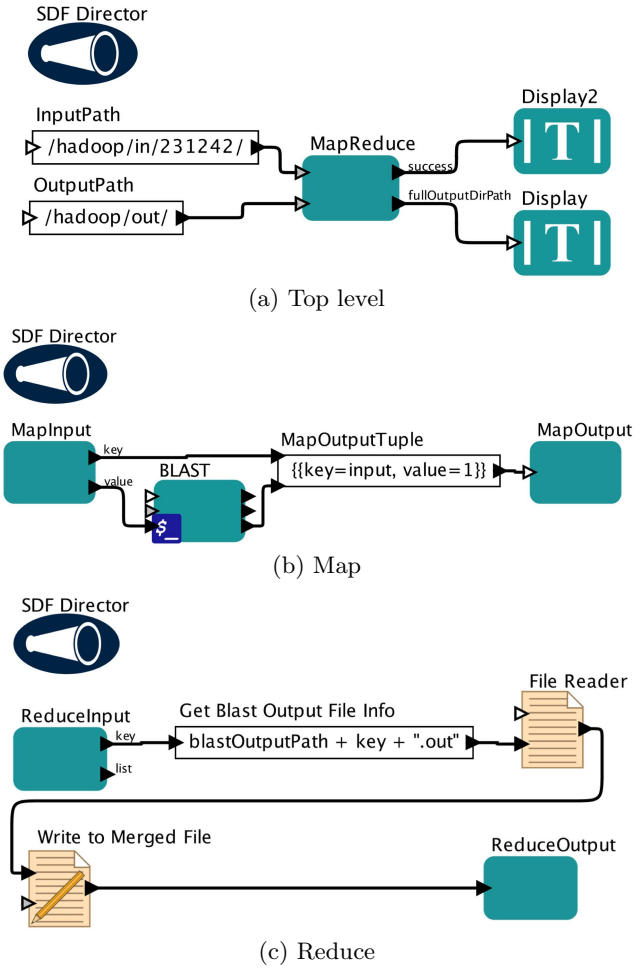

Figure 4: BLAST workflow.

4.3 Queries for the Applications

There are several provenance queries that can be answered for these applications:

- *Q*1: What is the dependency graph that led to each output artifact?

- $Q2$: What is the dependency graph that was derived from each input artifact?
- $Q3$: What are the input arguments to the BLAST program?
- $Q4$: How many times does a specific word occur in each input document in Word Count?

The data lineage captured in provenance can identify the input artifacts that led to a specific output artifact and vice versa. $Q1$ follows the dependencies for all outputs back to the initial inputs, and $Q2$ follows the derivations for each input forward to the final outputs. $Q1$ and $Q2$ can be used to measure the time traversing backwards and forwards over all the dependencies stored in provenance.

In addition to capturing the dependencies between artifacts and processes, a provenance framework can record the configuration used by a process. In our architecture, process configuration is recorded as input artifacts. $Q3$ can be answered by finding the inputs to all the processes that ran the BLAST program, and this requires capturing the provenance within the Map sub-workflow. Since the BLAST program has over 40 different command-line options[9], it is important to capture and be able to query this information so that users can know exactly how BLAST was run in previous executions of the workflow.

The output of Word Count tells how many times each word occurs in all the input documents. However, for a given word, we might want to know how many times it occurs in each document. $Q4$ can be answered by first finding the Reduce task that handled the word. (The input to each Reduce task are a key and list of values. For Word Count, the key is a word, and the list of values are a list of occurrences.) Once we find the Reduce key containing the word, we can follow the dependencies backwards to the Map task(s) that counted the word. The artifacts captured in each Map task provide the document name, as well as the number of occurrences in that document.

5. EXPERIMENTAL RESULTS

In this section we describe the experiments performed to measure the overall performance of our architecture and provenance data model. We measured the execution overhead when capturing provenance, the amount of space used to stored provenance information, and the time to execute the queries described in Section 4.3.

Environment. The experiments were performed on a cluster where each node has two quad-core Intel Xeon E5530 Nehalem 2.4 GHz processors and 24 GB of memory. We used Hadoop 0.20.2, NDB Data Server 7.1.13, and MySQL Server 5.1.56. Hadoop was configured so that each Map and Reduce task used a maximum of 4 GB of memory. NDB Data Server was configured to use two execution threads, and a maximum of 9 GB of memory for data and 1 GB for indexes. Additionally, data was replicated twice for fault-tolerance. The provenance framework was configured to use a maximum batch size of 500 writes requests, and execute them in the same number of threads as the number of slave nodes.

[9]http://www.ncbi.nlm.nih.gov/staff/tao/URLAPI/blastall.html#3, 2011.

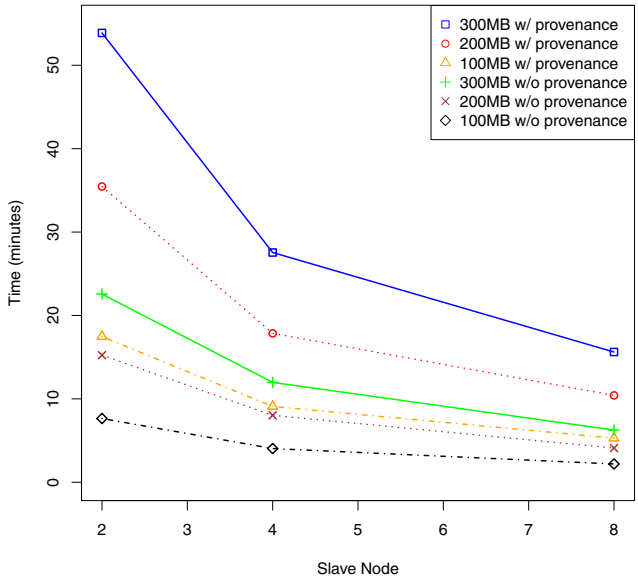

Figure 5: Word Count workflow execution times.

Application Configuration. Word Count and BLAST were chosen as application scenarios due to the different characteristics of these workflows. The number of Reduce tasks for Word Count was the same as the number of slave nodes to enable parallel Reduce execution. However, BLAST only had one Reduce task since the Reduce sub-workflow merges the results into a single output file on a shared NFS file system.

Word Count was executed with 100 MB, 200 MB, and 300 MB datasets containing 392, 782, and 1175 files, respectively. All input data were staged into HDFS before execution.

For BLAST, we split the query sequences into 160 files, each containing 125 sequences, to enable independent and parallel BLAST executions. The reference database was *AntarcticaAquatic: All Metagenomic 454 Reads*[10], which was 20 GB. The query sequences were staged into HDFS and captured in provenance. The reference database was stored on a shared NFS file system, and not staged into HDFS. The path of the reference database was captured in provenance, but not its contents.

5.1 Provenance Capture

We executed each application configuration with and without provenance support. The execution times for Word Count and BLAST are shown in Figures 5 and 6, respectively. As the number of slave nodes increases, the execution times of all workflows decrease. Word Count takes about 2.5 times longer to execute when capturing provenance, and BLAST take 1.03 times longer. These ratios do not appear to decrease as the number of slave nodes increase. The overhead for Word Count is higher than BLAST since the former's input datasets are much larger.

[10]http://camera.calit2.net/camdata.shtm, 2011.

	Word Count			BLAST
Input Data (MB)	100	200	300	2.6
Provenance Data (MB)	509	1,001	1,494	3.2
Artifacts	66,572	85,068	95,740	2,248
Processes	49,931	63,803	71,807	1,608
Dependencies	5,518,978	10,907,935	16,317,464	4,493

Table 1: Provenance information for each input data configuration.

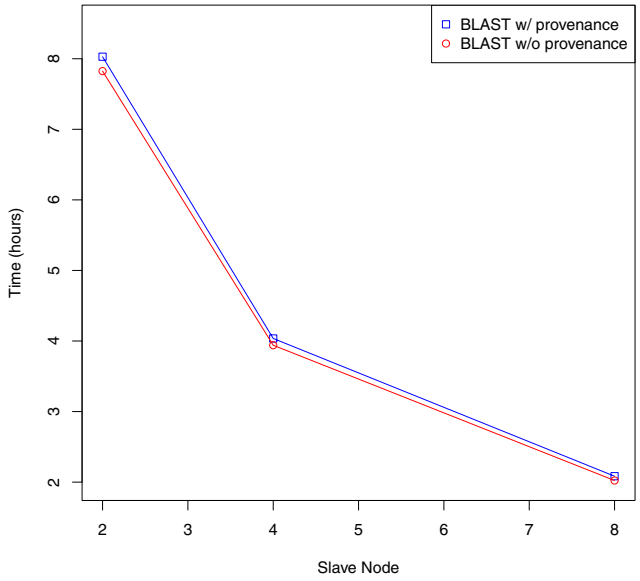

Figure 6: BLAST workflow execution times.

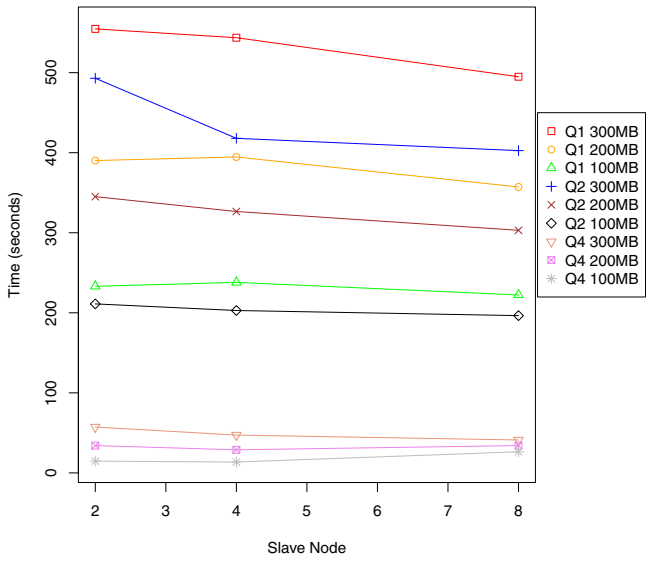

Figure 7: Word Count workflow provenance query times.

Table 1 shows the amount of provenance information captured for the experiments.[11] The Input Data row shows only the amount of data read as tokens in the Map tasks. These values do not include all the data read by workflow processes, e.g., the BLAST program reads the 20 GB reference database. The amount of captured provenance information is roughly five times larger than the input for Word Count and 1.5 times for BLAST. The next three rows show the number of artifacts, processes, and dependencies for each configuration.

5.2 Provenance Queries

We ran queries $Q1 - Q4$ described in Section 4.3 on the collected provenance data. The execution times for each query are shown in Figure 7 for Word Count and Figure 8 for BLAST.

Queries $Q1$ and $Q2$ traverse all dependencies in the provenance graph. Additionally, the data value for each artifact is downloaded to the client. For the Word Count workflow, the query times appear to slightly decrease as the number of slave nodes increases. However, the query times for the BLAST workflow are the smallest for two nodes and greatest for four nodes. Since the amount of data retrieved during the queries of the BLAST workflow is very small, network communication times between NDB Data Servers takes the majority of the time. In the experiments with two slave nodes, MySQL Server can answer any query without contacting other nodes since each node has the entire set of data. However, in the experiments with four and eight slave nodes, MySQL Server must retrieve the data from different nodes.

Query $Q3$ finds the command-line arguments used to run the BLAST program. This query takes less time than $Q1$ and $Q2$ since it can be answered using fewer SQL queries: find all the processes that ran the BLAST program, find the input artifacts for these processes, and finally get the values for these artifacts.

Query $Q4$ finds the occurrences of a specific word in each input document. We ran this query for a word that occurred a total of 1,489,296, 2,969,136, and 4,458,348 times in the 100 MB, 200 MB, and 300 MB input datasets, respectively. Similar to $Q3$, $Q4$ is faster than $Q1$ and $Q2$ since $Q4$ can be answered using fewer SQL queries. These SQL queries are described in Section 4.3.

[11] The sizes reported are the amount of data stored in the relational tables. We were unable to get an accurate measure of the indexes' sizes.

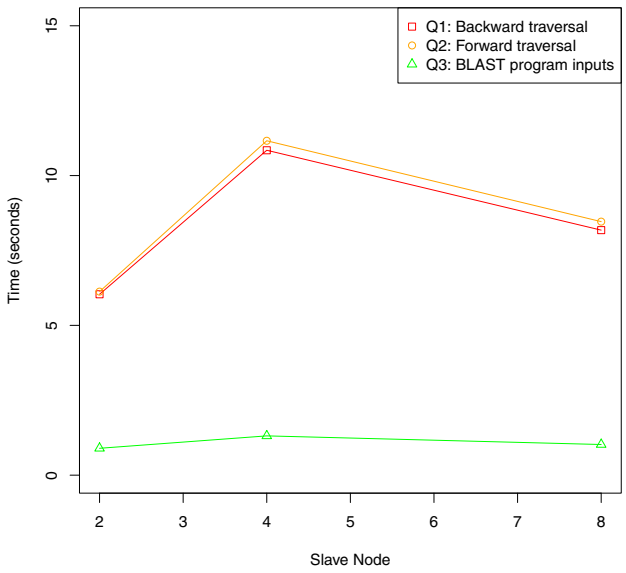

Figure 8: BLAST workflow provenance query times.

6. CONCLUSIONS

This paper presents an architecture and data model for capturing and querying provenance in data-intensive workflows. Provenance is captured for Map and Reduce tasks executing Kepler sub-workflows. We have described two data-intensive applications, Word Count and BLAST, implemented in Kepler using the MapReduce actor, and several related provenance queries. By capturing provenance inside Map and Reduce tasks for these applications, we are able to answer important queries for users, e.g., find the command-line arguments used for the BLAST program.

We evaluated the feasibility of our approach by executing Word Count and BLAST with several different input datasets and hardware configurations using MySQL Cluster to store provenance information. To the best of our knowledge, this is the first time such a study has been conducted. Based on the experimental results, we observe:

1. Provenance capturing scales well as the number of nodes increase (see Figure 5 and Figure 6).

2. Execution overhead and provenance storage size are workflow specific and increase with the input data size (see Table 1).

3. The query performance does not significantly decrease as the number of compute nodes increases (see Figure 7 and Figure 8).

As future steps, we plan to further optimize our data model to reduce execution overhead and storage space. Additionally, we are investigating the performance of storing provenance in other "shared-nothing" storage systems such as HBase and comparing the results of different storage architectures.

7. ACKNOWLEDGEMENTS

The authors would like to thank the rest of the Kepler team for their collaboration. This work was supported by NSF SDCI Award OCI-0722079 for Kepler/CORE and ABI Award DBI-1062565 for bioKepler, DOE SciDAC Award DE-FC02-07ER25811 for SDM Center, the UCGRID Project, and an SDSC Triton Research Opportunities grant.

8. REFERENCES

[1] I. Altintas, O. Barney, and E. Jaeger-Frank. Provenance Collection Support in the Kepler Scientific Workflow System. In L. Moreau and I. Foster, editors, *Provenance and Annotation of Data (IPAW 2006, Revised Selected Papers)*, volume 4145 of *Lecture Notes in Computer Science*, pages 118–132. Springer Berlin / Heidelberg, 2006.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215(3):403 – 410, 1990.

[3] B. Chapman, G. Jost, R. van der Pas, and D. Kuck. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, Cambridge, MA, USA, 2007.

[4] D. Crawl and I. Altintas. A Provenance-Based Fault Tolerance Mechanism for Scientific Workflows. In J. Freire, D. Koop, and L. Moreau, editors, *Provenance and Annotation of Data and Processes (IPAW 2008, Revised Selected Papers)*, volume 5272 of *Lecture Notes in Computer Science*, pages 152–159. Springer, 2008.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51:107–113, January 2008.

[6] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: A Runtime for Iterative MapReduce. In *The First International Workshop on MapReduce and its Applications (MAPREDUCE'10) - HPDC2010*. 2010, 2010.

[7] X. Fei, S. Lu, and C. Lin. A MapReduce-Enabled Scientific Workflow Composition Framework. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 663–670, July 2009.

[8] C. Franke, S. Morin, A. Chebotko, J. Abraham, and P. Brazier. Distributed Semantic Web Data Management in HBase and MySQL Cluster. In *Proc. of the 4th IEEE International Conference on Cloud Computing (CLOUD'11)*, pages 105–112, Washington DC, USA, July 2011.

[9] A. Goderis, C. Brooks, I. Altintas, E. A. Lee, and C. A. Goble. Heterogeneous Composition of Models of Computation. *Future Generation Computer Systems*, 25(5):552–560, 2009.

[10] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. Scientific And Engineering Computation Series. MIT Press, Cambridge, MA, USA, 2nd edition edition, 1999.

[11] Y. Gu and R. Grossman. Sector and Sphere: The Design and Implementation of a High Performance Data Cloud. *Philosophical Transactions of the Royal Society A*, 367(1897):2429–2445, June 2009.

[12] R. Ikeda, H. Park, and J. Widom. Provenance for Generalized Map and Reduce Workflows. In *Proceedings of CIDR'2011*, pages 273–283, 2011.

[13] M. Isard and Y. Yu. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 987–994, New York, NY, USA, 2009. ACM.

[14] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, J. Tao, and Y. Zhao. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice and Experience, Special Issue on Scientific Workflows*, 18(10):1039–1065, 2006.

[15] T. G. Mattson, B. A. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2005.

[16] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, and J. Van den Bussche. The Open Provenance Model core specification (v1.1). *Future Generation Computer Systems (FGCS)*, 27(doi:10.1016/j.future.2010.07.005):734–756, 2011.

[17] C. Moretti, H. Bui, K. Hollingsworth, B. Rich, P. Flynn, and D. Thain. All-Pairs: An Abstraction for Data-Intensive Computing on Campus Grids. *IEEE Transactions on Parallel and Distributed Systems*, 21:33–46, 2010.

[18] J. Wang, I. Altintas, C. Berkley, L. Gilbert, and M. B. Jones. A high-level distributed execution framework for scientific workflows. In *IEEE International Conference on eScience*, pages 634–639, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[19] J. Wang, D. Crawl, and I. Altintas. Kepler + Hadoop: A General Architecture Facilitating Data-Intensive Applications in Scientific Workflow Systems. In *WORKS '09: Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, pages 1–8, Portland, Oregon, 2009. ACM New York, NY, USA.

[20] J. Wang, P. Korambath, and I. Altintas. A Physical and Virtual Compute Cluster Resource Load Balancing Approach to Data-Parallel Scientific Workflow Scheduling. In *IEEE 2011 Fifth International Workshop on Scientific Workflows (SWF 2011), at 2011 Congress on Services (Services 2011)*, Washington, DC, USA, July 2011.

[21] C. Zhang and H. De Sterck. CloudWF: A Computational Workflow System for Clouds Based on Hadoop. In M. Jaatun, G. Zhao, and C. Rong, editors, *Cloud Computing*, volume 5931 of *Lecture Notes in Computer Science*, pages 393–404. Springer Berlin / Heidelberg, 2009.