A Comparative Study of Data Processing Approaches for Text Processing Workflows

Ting Chen
The University of Tokyo
Email: chenting@eidos.ic.i.u-tokyo.ac.jp

Kenjiro Taura
The University of Tokyo
Email: tau@eidos.ic.i.u-tokyo.ac.jp

***Abstract*—Workflows are widely used in data-intensive applications since it facilities the composition of individual executables or scripts, providing an easy-to-use parallelization to domain experts. With considerable popularity of MapReduce framework, some researchers start to develop MapReduce-enabled workflows instead of general file-based ones. Meanwhile, being commercially available for nearly two decades for large-scale data processing, parallel database systems have also gotten wide attention in the support of workflows. This paper studies three real-world text processing workflows and develops them on top of several different large data processing approaches including an open source MapReduce implementation - Hadoop, a workflow-oriented parallel database system - ParaLite, and a hybrid of MapReduce and parallel DBMS - Hive. We discuss their strength-/weaknesses both in terms of programmability and performance for each workflow. Our experiences and experimental results reveal some interesting trade-offs: (1) High-level query languages (SQL of ParaLite and HiveQL of Hive) are helpful for expressing data selection, aggregation and calculation by typical executables; (2) To reuse existing NLP tools, it is often important to be able to track the association between a document and its annotation attached by the tool, for which the expressiveness of SQL is particularly useful; (3) Each system has similar performance in the execution of overall workflows because essentially performing executables takes most of the time, but some small differences could reveal some potential trade-offs that each system entails for workflows.**

I. Introduction

Workflows [1] are widely used in data-intensive text processing applications since it facilities the composition of individually developed executables or scripts, making it easier for domain experts to focus on their research rather than the management of computation. Many systems are proposed to execute workflows, including GXP Make [2], Swift [3], Pegasus [4] and Taverna [5]. A workflow is generally a DAG with many data processing tasks and their dependencies. Each task is a typical existing binary or executable. For example, workflows in natural language processing (NLP) typically consist of sentence splitters, part-of-speech taggers, named entity recognizers, parsers, data indexers, and so on. Many of them (e.g., parsers [6], [7]) are third-party components that received a considerable amount of development efforts in the community. Others are ad-hoc scripts. Either way, they almost always work on text data which are usually stored in and transferred through files. Tasks in the workflow are communicated with each other through files [8]. Each task is fed with input files and produces at least one output file which becomes the input of a following job.

Recently, MapReduce [9] has attracted wide interests from both industry and academia due to its simple programming model and good scalability across hundreds of nodes. After the emergence of MapReduce and its open-source incarnation Hadoop [10] in particular, lots of scientific researchers start to focus on constructing map-reduce enabled workflow systems in which a heavy task can be expressed as Map and Reduce jobs [11] or a whole workflow composition is created as MapReduce style [12].

With consideration of making workflows simple and efficient, a natural idea is to build workflows on top of the parallel database systems[13] which have been commercially available for nearly two decades, including Teradata, Vertica, DB2, and Oracle. They are robust, high performance computing platforms to provide a high-level programming environment and parallelize data processing easily. In addition, with expressive SQL, database systems can simplify the description of workflows. For instance, SQL with a proper support of user-defined functions and reductions can express many data processing tasks much more elegantly and easily than MapReduce [14] [15]. Furthermore, databases are efficient for processing relational data in ways expressible in SQL due to data indexing and sophisticated query optimization [14] [15].

However, to support workflows better, both MapReduce and parallel database systems have their own disadvantages. MapReduce in general requires users to develop Map and Reduce functions; Hadoop requires them to be written in Java conforming to the class library framework, at least by default. In addition, a workflow typically consists of many third-party binaries and ad-hoc scripts written in a variety of languages, but integrating them in Hadoop is not straightforward. Fortunately, a utility coming with Hadoop – Hadoop Streaming [16] allows a simple reuse of executables by creating and running Map/Reduce jobs with any executable or script as the mapper and/or the reducer. Furthermore, to alleviate the burden of writing low-level language to express a user's requirement, some systems are proposed to provide high-level language on top of the current interface, such as Pig [17], Hive [18] and HadoopDB [19].

Database systems generally have a limited support for integrating external executable into data processing pipeline. As mentioned above, such integration is very important in workflows. Although some databases support User-defined

Functions (UDF), they either do not support parallel execution of them or lack flexibility in input/output formats, development language and reusability of code. In addition, general database systems do not optimize data transfers between data nodes and parallel clients that process large query results. Generally, big query results are always returned to a single client and then distributed to external programs for parallel execution. So the single client can easily become a bottleneck. Moreover, it prohibits us to take advantage of co-allocating computing clients with data. ParaLite, a workflow-oriented parallel database system, extends SQL syntax to integrate User-Defined Executables (UDX) into a query and proposes a concept of collective query to parallelize them.

In our previous study, we developed three real-world text processing workflows on top of ParaLite [20]. In the present paper we extend the previous study by comparing four approaches, files, Hadoop, Hive, and ParaLite, and discuss their strength/weaknesses both in terms of programmability and performance for these workflows. We target to find the trade-offs that all these systems entail for workflows. In general, high-level query languages (SQL of ParaLite and HiveQL of Hive) are helpful for expressing data selection, aggregation and join. Moreover, they provide elegant expression for pipeline-style workflows by chaining several executables in a single query while Hadoop lacks good support of such workflows since it cannot pipe multiple executables within a single command. In typical NLP workflows, to reuse existing NLP tools, it is often important to be able to track the association between a document and its annotation attached by the tool, for which the expressiveness of SQL in ParaLite is particularly useful while Hive and Hadoop need extra efforts. In terms of performance, a workflow build on top of each system takes similar execution time because most of computation comes from the execution of executables. However, compared with other systems, ParaLite has small superiority ($10\% \sim 20\%$) due to (1) data is well partitioned which allows the system to take the advantage of its underlying database system SQLite to perform local aggregation and join; (2) the powerful expressive ability mentioned above that reduces some overheads existing in other systems such as writing intermediate data into files and parsing records for each executable.

II. Review of Several Approaches

A. Hadoop

Hadoop [10] is an open-source incarnation of MapReduce model to process large-scale data across large clusters of compute nodes. It provides users easy programing model by which only two user-customized `Map` and `Reduce` functions are required to be written. Hadoop consists two layers:

(1) Hadoop Distributed File System (HDFS) layer for data storage. HDFS is a block-structured file system which splits individual files into blocks with a fixed size and distributes them across multiple DataNodes in the cluster. HDFS is controlled by a central NameNode which keeps the directory structure of all files in the file system, and tracks the location of blocks and their replicas.

(2) MapReduce layer for data processing. The MapReduce Framework follows the master/worker pattern. A single master called JobTracker receives MapReduce jobs from user applications and schedules tasks to some specific nodes in the cluster determined by the information on NameNode. The policy for job scheduling takes both data locality and load balancing into consideration. Each worker node called TaskTracker accepts tasks including Map, Reduce and Shuffle operations, and spawns separate processes to do the actual work.

Hadoop Streaming (HS) is a utility that comes with the Hadoop distribution. The utility allows you to create and run map/reduce jobs with any executable or script as the mapper and/or the reducer. For instance, to perform the word count task which is to calculate the occurrences of words from a big text, hadoop streaming uses the following statements:

```
Hadoop jar hadoop-streaming.jar
  -input myInputDir
  -output myOutputDir
  -mapper wc_mapper.py
  -reducer wc_reducer.py
```

In the above example, both the mapper and the reducer are python executables. The mapper reads the input from stdin (line by line), splits the input into words and emits the output of $\langle$`word, 1`$\rangle$ to stdout. The reducer reads the output of the mapper from stdin and calculates the total number of occurrences for each word. HS creates a map/reduce job, submits the job to a cluster, and monitors the progress of the job until it completes.

B. Hive

Hive is a data warehouse system built on top of Hadoop. It is considered as a hybrid of MapReduce model and database system since it projects structured data files to relational database tables and supports queries on the data. These queries are expressed in a SQL-like declarative language called HiveQL and compiled into MapReduce jobs executed on Hadoop. Meanwhile, Hive also allows users' own mappers and reducers which are executables written in any language to be plugged in the query when it is inconvenient or inefficient to express the logic in HiveQL.

With Hive, we can express the word count task by the following query:

```
insert overwrite table word_count
select mapout.word, count(*) from (
  map text using 'wc_mapper.py' as word
  from data) mapout group by mapout.word
```

As Hadoop does, it firstly splits text into words using a nested query in which the executable `wc_mapper.py` is specified as a mapper and outputs an intermediate table `mapout` of words. Then the outer query aggregates the occurrence of each word using `group by` operation. However, although HiveQL is similar with general SQL and targets to achieve SQL compatibility, it still introduces significant new syntax to normal SQL to integrate MapReduce scripts; for instance, in addition to the usual SELECT, it adds MAP, REDUCE and TRANSFORM.

C. ParaLite

ParaLite[21] is a workflow-oriented parallel database system. The basic idea of ParaLite is to provide a coordination layer to glue many single-node database systems together, specifically SQLite [22], and parallelize SQL query across them. To provide a better support of workflows, ParaLite extends SQL syntax to easily embed user's arbitrary command line called User-Defined Executable (UDX) into a single SQL query. A ParaLite UDX is an executable written in any language and defined within a query using `WITH` clause. It can work on and produce arbitrary columns while UDF in traditional database system can only support one column input and output. Let's also take the word count task as an example. It is straightforward to perform this task by integrating text splitter into general aggregate SQL query. The nested query splits the text into words using `awk` script as the UDX and outputs words in the format of one word per record. The occurrences for each word is simply counted by grouping words from the output of the nested query.

```
select word, count(*) from
 select F(text) from data
 with F= "awk 'for(i=1;i<=NF;i++) print $i'"
group by word
```

To parallelize the execution of UDX, ParaLite proposes a concept of `collective query`, a single query issued by multiple computing clients who collectively receive data from data nodes. Data are processed by clients in parallel using UDX. Collective queries provide co-allocation of parallel compute clients and data sources. Data are transferred from data nodes to computing clients based on the principles of both data locality and load balance of all clients. Moreover, collective queries allow the separation of data nodes and computing nodes on which UDXes-related software is installed.

III. Text Processing Workflows

Next, we introduce three real-world text-processing workflows with different structures in natural language processing:

- Japanese Word Count
- Event Recognition
- Sentence Chunking Problem

Since all these three systems do not provide any language to describe the dependencies of components/jobs, we generally perform each single job using them and leave the creation of the whole workflows to a known workflow engine called GXP Make [2]. GXP Make uses `make` to describe the workflow and provides the parallelization of tasks across clusters. So in the following sections, we ignore the descriptions of dependencies among jobs and only focus on the expressiveness of each job based on different systems.

A. Japanese Word Count

`Japanese Word Count` calculates the occurrence of Japanese words from crawled Japanese web pages. Word count task is widely used to extract key words or phrases from web data which is very useful in the web analysis of various fields, such as, revealing hot topics in Twitter, popular products in on-line stores and attracting customs in different countries.

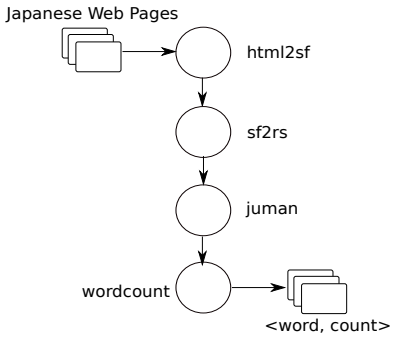

Fig. 1. Workflow of Japanese Word Count

As Fig 1 shows, this workflow is a simple pipeline with four jobs: (1) `html2sf`: convert the source data crawled from Japanese web pages to an XML-based canonical format developed by Kurohashi group at Kyoto University. The format defines XML tags to identify plain text and accommodates annotations such as named entities. (2) `sf2rs`: extract plain text, identified with tag ⟨raw_string⟩ from the canonical format data, (3) `juman`: tokenize text into Japanese words, (4) `wordcount`: calculate the frequencies of Japanese words.

Hadoop: The first two jobs are expressed by two Hadoop Streaming commands each of which only contains a mapper `html2sf` or `sf2rs`. To reduce redundant IO operations, we perform the last two jobs together using one command with a mapper `juman` and a reducer `word_count_reducer`. The mapper parses Japanese sentences into words and outputs each word with an associated count of occurrences 1 and then the reducer aggregates all counts emitted for a particular word.

The first executable `html2sf` is a file-based program which reads input data from file, so a wrapper which receives data from standard input and stores them into a file is necessary. Moreover, since the input of each executable has multiple lines per record, we have to either customize our own `InputFormat` and `InputReader` classes and pack them along with the streaming jar or write a small wrapper to convert the complex record into a single-line one. In this paper, we take the latter method. Therefore, 3 wrappers for executables are necessary.

Hive: Hive performs the workflow by only one query

```
insert overwrite table wordcount
select tokens.word, count(*) as count from (
 map rst.rs using 'juman' as word from (
  map sft.sf using 'sf2rs' as rs from (
   map html.con using 'html2sf_wrap' as sf from html)
sft) rst) tokens group by tokens.word
```

in which the first three executables are nested and the output Japanese words from `juman` are aggregated. To deal with file-based executable `html2sf`, we still need a wrapper to produce the input file with data from standard input. Since data is piped between executables and each one can handle the output data from the previous one, we don't need to write

any wrapper to deal with the complicated format of data.

ParaLite: The first three jobs are expressed by a single query where the executables are specified as three UDXes and data are piped from one to another.

```
create table tokens as
  select T(S(H(con))) as word from html
   with H="html2sf html_file" input 'html_file'
        S="sf2rs" T="juman"
partition by word
```

Then another simple SQL query with `group by` operation aggregates the occurrences of Japanese words. To take advantage of SQLite, the results from the first query are partitioned by words. As a result, ParaLite directly assigns the aggregate query to SQLite engine on each node. ParaLite can support file-based UDX, so no wrapper is required for this workflow and we only need to specify the input option for the executable.

File: In file-based workflows, the first three steps are expressed simply by a command line. However, to parallelize the command line, it is necessary to split the big input file into many small files. As a result, many intermediate files are produced. Specially, since there is no straightforward method to express the last aggregation job `wordcount`, we perform it by Hadoop Streaming.

Discussion: Japanese Word Count is a simple pipeline workflow which is expressed by Hive and ParaLite elegantly where more than one executables can be expressed within a single query. However, Hive needs more efforts to deal with file-based executable. Since Hadoop Streaming cannot support multiple mappers or reducers in a single HS job, the executables have to be expressed by several separate HS jobs, leading to a), more steps in the workflow, b), more efforts to deal with the complicated format of input data, c), longer execution time due to storing the output of each executable in files. General jobs with executables are easily expressed with file system, but the aggregate job cannot be presented straightforwardly. Users have to either write their own processing logic or rely on Hadoop.

B. Event Recognition

The goal of `Event-Recognition` [23] workflow is to recognize complex bio-molecular relations (bio-events) among biomedical entities (i.e. proteins and genes) that appear in biomedical literature. Recognition of such events including an expression of a certain gene, a phosphorylation of a protein, and a regulation of certain reactions are important to understand biomedical phenomena.

The workflow of Event Recognition is shown in Fig 2. The input of the workflow is the MEDLINE database [24] which contains over 19 million references to journal articles in life sciences with a concentration on biomedicine. The event recognition application consists of 4 steps with 6 jobs: (1) extract abstract of each article from the source xml files; (2) split the abstract into sentences with their unique identification; (3) to each sentence, apply three tools:

- Enju Parser: a HPSG parser which can effectively analyze syntactic/semantic structures of English sentences.

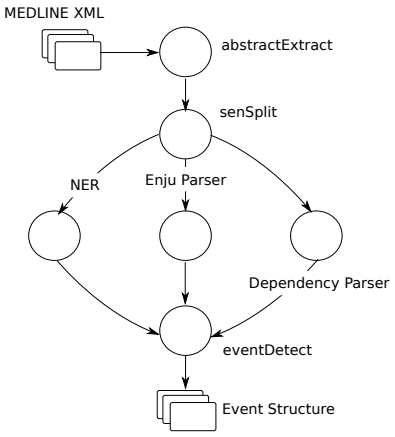

Fig. 2. Workflow of Event-Recognition Application

- Named Entity Recognizer: recognition for bio-medical entities such as gene and protein.
- Dependency Parser: a dependency parser for biomedical text.

(4) combine the results from the three tools and extract biomedical events. It is a typical real NLP workflow, which applies several existing tools to each document/sentence and combines results from them to perform a higher-level reasoning. A recurring problem in such workflows is that each tool reads texts as a single stream and does not have a notion of document boundaries. The output from such a tool is similarly a single stream that does not leave anything between document boundaries. Thus, it is the responsibility of workflow developers to track the association between a document and a result from each tool and correctly combines them.

Hadoop: Each job in the workflow could be expressed by one or several Map-Reduce jobs. Generally, tools used in the steps (3) and (4) consist of several executables and some of them work on the joined data from other two previous executables. Hadoop performs join operation separately before the executable is executed and uses several scripts for these executables. For example, the final `eventDetector` job which joins data from the previous three tools on the sentence ID to detect complex relations between entities is expressed by two MR jobs. The first one only performs the join operation using both Map and Reduce functions and outputs records each of which consists of a sentence ID followed by the sentence and the three result of this sentence. Then the next MR job specifies the executable as a mapper which reads output from the previous job and emits the final results.

Hive: Hive expresses each job with one or several equivalent queries. Different from Hadoop, Hive is able to chain several executables or express them together with a join operation within a single query. For example, the `eventDetector` is expressed by the following query:

```
insert overwrite table event_so
select out.SID, out.event
from (map abst.SID, abst.sentence, enju_so.enju,
          ksdep_so.ksdep, gene_so.gene
```

```
      using 'event-detector' as (SID, event)
      from abst
      join enju_so on (abst.SID = enju_so.SID)
      join ksdep_so on (abst.SID = ksdep_so.SID)
      join gene_so on (abst.SID = gene_so.SID)) out
```

ParaLite: ParaLite expresses each job by a similar query as Hive. Still taking `eventDetector` as an example, it is expressed by the following query:

```
create table event_so as
select F(abst.SID, abst.sentence, enju_so.enju,
         ksdep_so.ksdep, gene_so.gene) as (SID, event)
from abst, enju_so, ksdep_so, gene_so
where abst.SID = enju_so.SID
      and abst.SID = ksdep_so.SID
      and abst.SID = gene_so.SID
with F="event-detector" output_row_delimiter EMPTY_LINE
```

File: Similar with JAWC application, the input file is firstly splitted into thousands of small files and several executables are applied to each single file. Specially, for all the merge jobs, we take a in-order processing method, that is, all data is stored in the same order on the sentence ID. To fulfill this requirement, we define the name of each result file before the execution of the workflow.

Discussion: The workflow of Event Recognition generates both data access patterns of pipeline and reduce. Hadoop Streaming and file-based method are not sufficient to present join job. Hive and ParaLite are able to use queries to express the workflow elegantly. However, some extra efforts are necessary when the workflow is performed by Hive and Hadoop because they cannot track the association of the input sentence and the output from the NLP tools as we mentioned in the beginning of this section.

For example, let's say we have an executable X that reads sentences and outputs annotated sentences. In the workflow using such a tool as a component, we like to find (document id, annotated sentence) from (document id, the original sentence). In Hive and Hadoop, it is necessary to write an extra program which extracts sentences fed to the tool, receives the results and maps the annotated sentence to the original one. This is because that MapReduce programming model leaves all the computation inside of the mapper and reducer and it cannot handle complex logical processing outside. Specifically, The model reads data from HDFS, feeds them to a mapper, shuffles and sorts the output of the mapper and finally gives to a reducer. So it doesn't have any mechanism to do some complex processing to the output of mapper of reducer. Hence we need to write ten such wrappers in total. On the other hand, ParaLite, or SQL for that matter, naturally supports such an association through a simple query of the form "select sentence_id,X(sentence) from ...", as long as the output of the last executable in the chain has a fixed string , such as an empty line in most cases, between records boundaries.

C. Sentence Chunking Problem

Splitting sentences into meaningful chunks or phrases (N-grams) is very important in natural language processing since it is the first step of extracting concepts and relations within statements across a large text. Significant chunks would typically correspond to semantic units such as named entities (proteins, genes, diseases) or relations [25]. A recent paper [26] focused on the improvement of the performance of this application based on MapReduce through a proposed distributed looking-up system.

The problem for Sentence Chunking is to find the best way to chunk a sentence with the most meaningful phrases. We use a statistical model to solve it. Every sentence is generated by randomly sampling and the number of ways to chunk a sentence into phrases is finite. The model calculates the likelihood of each sentence by:

$$L(S) = \sum_{\sigma \in \Phi} \prod_{i \in \sigma} f_i$$

Φ is the set of the chunked sentence by all chunking methods; σ represents all phrases in the sentence under a specific chunking method and f_i is the probability of phrase i occurs in a corpus calculated based on its frequency.

The likelihood of whole corpus is simply calculated by the multiplication of the likelihood of each sentence:

$$L(C) = \prod_{S_i \in C} L(S_i)$$

We can then maximize the likelihood function of the whole corpus to get the best parameter f.

$$f = \underset{f}{\operatorname{argmax}}\ L(C)$$

Then, based on the best value of f, we can calculate the likelihood of each sentence in different chunking methods and the best one leads to the largest likelihood of the sentence.

The workflow of Sentence Chunking is shown in Fig 3. The input of the workflow is sentences of articles from the MEDLINE database. The workflow consists of five steps. The first three are required for initialization and only run once per input corpus: (1) `senSplit`: Extract text from input xml files and splitting into sentences, (2) `freqGen`: Generate phrases from sentence and get their frequencies, (3) `filter`: Store phrases with frequencies greater than one into a SQLite database to reduce the size of the database because most phrases occur only once according to the paper [26]. So receiving NULL for a frequency lookup query means that the frequency for the phrase is one. The following two steps are iterated: (4) `probGen`: newly estimated model parameter f is updated at the end of each iteration if a better distribution is found, (5) `likelihoodCal`: calculate the likelihood of the whole corpus based on the new parameter.

Hadoop: The `freqGen` job is expressed by a single streaming job in Hadoop in which the mapper executable generates phrases from sentences and emits pairs of $\langle$phrase, 1$\rangle$. Then an aggregate reducer reads the output of the mapper and sums the frequency for each phrase. The next `filter` job only requires a mapper which reads phrases from stdin and emits those with frequencies greater than one. Phrases

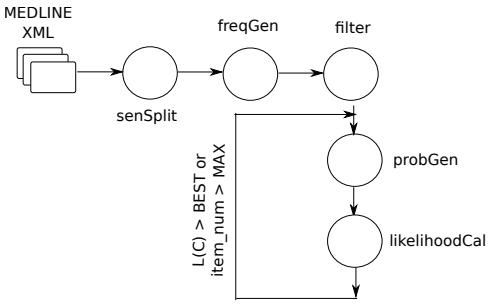

Fig. 3. Workflow of Sentence Chunking Problem

that passed the filter are stored in SQLite database which is queried against those phrase frequencies at each iteration. Next, we calculate the probability of phrase based on their frequencies, so we firstly have another aggregate job to obtain the number of phrases with the same frequency and then a script is invoked to get the related probability. Finally, the last job specifies an executable as the mapper which outputs the likelihood of each sentence to the reducer to get the likelihood of the whole corpus by multiplying likelihood of all sentences.

Hive: Hive uses equivalent streaming operations in a query to express each job, such as `freqGen` and `likelihoodCal` job. Specially, it performs `filter` job by a simple selective query without a customized mapper.

ParaLite: Similar with Hive, ParaLite expresses each job by an equivalent query in which UDXes are used instead of mappers. Specially, for the last job, ParaLite supports a user-defined aggregation `mul` to get the product of all records.

File: In this workflow, aggregate jobs such as `freqGen`, `probGen` and `likelihoodCal` appear alternately. These jobs cannot be elegantly expressed only based on file systems and as general we use Hadoop instead. Since most of jobs are performed in Hadoop style, we finally did not develop the whole workflow based on files.

Discussion: Each iteration of this workflow is a simple pipeline which is easily expressed by Hadoop, Hive and ParaLite. Although Hadoop provides an overall elegant expression, it still requires more efforts (an extra mapper or reducer) to perform data selection and aggregation. In addition, file-based method is not appropriate for such workflow in which most jobs perform aggregations to all data.

IV. Evaluation

We conducted several experiments to compare the performance of the three workflows built on top of Hadoop Streaming, Hive, ParaLite and a shared file system in a cluster of 32 nodes. Each node uses 2.40 GHz Intel Xeon processor with 8 cores running 64-bit Debian 6.0 with 24GB RAM.

A. System Configuration

Hadoop: In our experiments, we use Hadoop version 1.0.3 running on Java 1.6.0. We deploy the system on the cluster with the default configuration settings except for (1) we configure the system to run six Map instances and six Reduce instance concurrently on each node. The reason we set them to be six is that Hive often uses a single query with 2∼3 executables to perform a job, that is, 2∼3 processes are running for each Map task and we observed that Hive cat get the best performance with this configuration. (2) we allow JVM to be reused by all tasks instead of starting a new process for each Map/Reduce task. The number of mappers is decided by the system for most jobs while set manually for some time-consuming jobs to make sure that the execution time of each job is no more than 10 or 30 minutes. Since some jobs have large start-up cost, we do not limit the execution time within 1 minute as Hadoop suggested. To make the comparison fair, we store all input and output data in HDFS with the settings of one replica per block and without compression.

Hadoop	Hadoop(parallel)	Hive	Hive(parallel)	ParaLite	File
1280	126	1310	131	432	980

TABLE I
Data Preparation Time for JAWC(sec)

Hive: We use Hive version 0.8.1 with default configuration based on the Hadoop system configured as mentioned above. We set the same number of mappers and reducers for each Hive job and Hadoop job.

ParaLite: ParaLite is a serverlessness and zero-configuration system, so we do not need to configure anything before it is executed. Each table is stored in the same 32-node cluster and we start at most 6 clients on each node for each job. We set the size of blocks which is sent from data node to the computing clients within each collective query to make sure the total number of blocks equals to the number of mappers for the equivalent Hadoop job.

File: In file-based workflows, we use a shared file system NFS3 to store and transfer data. The parallelism of each job depends on the number of input files N which is determined by the parallel granularity of the most time-consuming job in other systems. For example, N should be equal to the number of mapper tasks for the job in Hadoop.

B. Japanese Word Count

We perform the experiments for Japanese Word Count workflow with a collection of Japanese web pages of size 104 GB. These web pages produce 62 GB useful text which are then loaded into different systems. The data loading time for each system is shown in Table I.

Hadoop: Since we do not need to alter the input data, we load the input file into HDFS as plain text using the Hadoop command-line utility. The input data is a single 62 GB file. If we directly invoke the command line to store it into HDFS from a node, a copy of the file is loaded to one HDFS data node. Another choice is that we split the big file into 32 small ones and store each one in the local disk of each data node respectively, then we load all local files in parallel into HDFS by issuing the command on each node. We measure these two methods and the results are presented in Table I where label `Hadoop` indicates the first method while `Hadoop`

`(parallel)` means the latter. Obviously, loading data in parallel reduce a lot time but it brings file split overhead.

Hive: Hive can load data to table from both local disk and HDFS by Hive Data Definition Language (DDL). Hive firstly copies it into HDFS and then creates metadata for the table. Since the metadata creation cost is negligible, so it takes almost the same time with Hadoop for the two cases.

ParaLite: ParaLite provides the same API with SQLite and loads data to the database by the `.import` command line. Unlike Hadoop, ParaLite distributes the big file automatically across all data nodes and really loads data to database on each data node in parallel. The process takes about 7 minutes.

File: The input file is stored in NFS, and we do not need to do anything but split it into 1000 sub-files which takes about 16 minutes. 1000 is chosen because Hadoop automatically splits each job into about 1000 tasks to be executed in parallel.

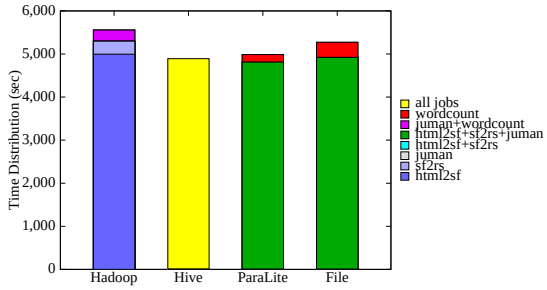

Fig. 4. The Execution Time of JAWC Workflow

As Fig 4 shows, ParaLite and Hive outperforms other systems about 15%. Since Hadoop Streaming cannot pipe multiple executables, three separate Hadoop jobs are launched which brings some extra overheads. For example, the `sf2rs` job is very lightweight and takes less than 2 minutes, but Hadoop Streaming spends 281 seconds on it. One extra overhead comes from the start-up cost of Hadoop. From our observations, it takes 15-25 seconds before all allowed Map tasks have been started. Besides, storing intermediate data (such as 25GB result of `html2sf`) into HDFS which is then read by next process also takes much more time than directly piping data between processes. For the last aggregate job, since the output results from `juman` is partitioned by words, ParaLite executes the SQL query by sending it to the SQLite database on each node and performs local aggregation. Therefore, it outperforms Hadoop since it needs to reduce data.

C. Event Recognition

Event Recognition workflow reads 30 GB data from MEDLINE database and extracts 1 GB abstracts of articles from the source xml files. We load the data into all systems using the methods mentioned in Section 4.2. For Hadoop and Hive, we directly use the single input file without splitting it and loading from all data nodes in parallel because the data is small. As a result, Hadoop and Hive take 8 seconds while ParaLite takes 11 seconds. For file-based workflow, we split the input into 10000 small ones according to the most time-consuming job `Enju Parser` and the split takes about 8 seconds.

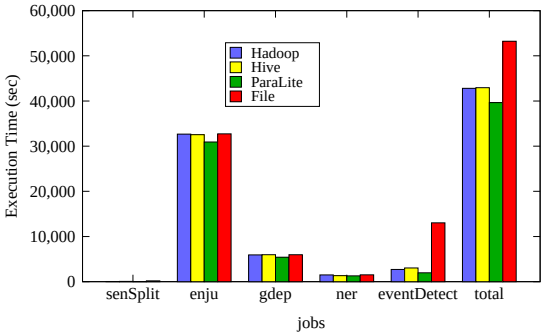

Fig. 5. The Execution Time of Event-Recog Workflow

Fig 5 shows that ParaLite outperforms other systems from 8% to 30%. Since ParaLite is able to track the association of the input and output records, most executables work on the input data directly without parsing while each executable in Hadoop- and Hive-based workflows requires to parse the input data to map the input to the according output. Another reason is that ParaLite has better performance in join operation, especially for the `eventDetector` job. The input four tables of `eventDetector` are 1 GB sentences, 55 GB `enju` results, 11 GB `gdep` results and 150 MB `ner` results. ParaLite partitions all these data on the key `SID`, so when the join operation is performed, it pushes the original join SQL query directly to the SQLite database on each data node and it only takes about 25 seconds. Hadoop performs this join operation using about 8 minutes and Hive takes about 4 minutes.

To get the best performance, we tune some parameters for each job to adjust the degree of jobs parallelization according to their compute density. Job `enju` is very computationally intensive and `eventDetector` is not as heavy as `enju` and it has high start-up overhead, so we set more parallel tasks for `enju` and less for `eventDetector`. It is easy to do the parameter tuning in ParaLite which allows you to specify the size of block for each query and Hadoop which allows you to set the number of mappers and reducers in the script for each job. However, it is not easy with Hive to tune this parameter. We have to modify the parameter of number of mappers in the configuration file and restart the Hadoop cluster every time when we want to change it. What is the worse is that this kind of parameter tuning is impossible in file-based workflow. This is the reason that the execution time of `event-detector` job in the workflow with files is much larger than that in the workflow with other systems. As mentioned in the beginning of this section, we split the input file into 10000 small ones based on the execution of `enju` job. Hence we have 10000 small sub-jobs to be processed in parallel for each step. The number of sub-jobs is much larger than that in other systems and each has high start-up overhead (about 20 seconds) , as a result, the total execution time is increased. Once the input files is splitted, users have to parallelize each job according to the number of sub-files unless internal parallelization and merge is performed independently.

D. Sentence Chunking

Sentence Chunking workflow reads 60 GB data from MEDLINE database and gets 2 GB abstract. We load the data into HDFS using the Hadoop command-line utility and it takes 14 seconds. Hive takes several seconds more to create the metadata and ParaLite takes 21 seconds. From Fig 6 we can

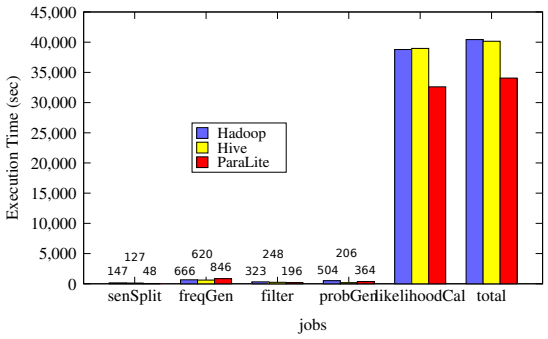

Fig. 6. The Execution Time of Sentence Chunking Workflow

see that the execution time of the whole workflow by Hive and Hadoop are almost the same and ParaLite is about 18% faster than them mainly because of ParaLite has better performance on the most time-consuming job `likelihoodCal`. For the `freqGen` job, Hadoop and Hive are about 200 seconds faster than ParaLite. That is because the output of `freqGen` is about 145 GB and storing them into database takes ParaLite another 420 seconds. For the `probGen` job, Hadoop is much more slower than others. The data to be aggregated in this job is very unbalanced and more than 90% phrases (about 137 GB) occur only once. So all these data are transferred to a single reduce task to be aggregated in Hadoop leading to longer execution time than Hive and ParaLite that firstly aggregate data locally.

V. Conclusion

We studied three real-world text-processing workflows, specifically in the discipline of Natural Language Processing (NLP), and built them on top of several large-scale data processing approaches, including Hadoop Streaming, Hive and ParaLite. We compared the programmability and performance of these workflows based on the three systems and general file-based workflows. Our development experience revealed that high-level query languages such as SQL of ParaLite and HiveQL of Hive are helpful for expressing both typical SQL jobs (data selection, aggregation and join) and jobs with executables. In NLP workflows, the expressiveness of SQL in ParaLite is particularly useful since it provides natural supports of file-based NLP executables and reusing existing NLP tools by tracking the association between a document and its annotation attached by the tool. On the other hand, workflows expressed in low-level language lacks good support of all features mentioned above, requiring some extra efforts. The evaluation experimental results show that essentially each system has similar performance in the execution of the whole workflows because performing executables takes most of time. However, ParaLite and Hive, especially ParaLite, still has small superiority in the SQL jobs due to their optimized execution plans such as allowing local aggregation and join.

References

[1] E. Deelman, D. Gannon, M. S. Shields, and I. Taylor, "Workflows and e-science: An overview of workflow system features and capabilities," *Future Generation Comp. Syst.*, vol. 25, no. 5, pp. 528–540, 2009.

[2] K. Taura, T. Matsuzaki, M. Miwa *et al.*, "Design and implementation of gxp make – a workflow system based on make," in *eScience2010*, 2010, pp. 214–221.

[3] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu, T. Stef-Praun, and M. Wilde, "Swift: Fast, reliable, loosely coupled parallel computation," in *IEEE International Workshop on Service Computing*, 2007, pp. 199–206.

[4] D. Ewa, S. Gurmeet, S. Mei-Hui, Blythe, and others., "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming Journal*, vol. 13, no. 3, pp. 219–237, Jul. 2005.

[5] T. M. Oinn, M. Addis, J. Ferris, D. Marvin *et al.*, "Taverna: a tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, vol. 20, no. 17, pp. 3405–3054, 2004.

[6] "Enju." http://www-tsujii.is.s.u-tokyo.ac.jp/enju.

[7] "Cabocha." http://code.google.com/p/cabocha.

[8] I. Raicu, I. Foster, and Y. Zhao, "Many-task computing for grids and supercomputers," in *The 1st Workshop on Many-Task Computing on Grids and Supercomputers*, 2008.

[9] J. Dean and S. Ghemawat, "Mapreduce:simplified data processing on large clusters," in *OSDI'04*, 2004, pp. 137–150.

[10] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2009.

[11] P. Nguyen and M. Halem, "A mapreduce workflow system for architecting scientific data intensive applications," in *SECLOUD '11*, 2011, pp. 57–63.

[12] X. Fei, S. Lu, and C. Lin, "A mapreduce-enabled scientific workflow composition framework," in *ICWS '09*, 2009, pp. 663–670.

[13] D. DeWitt and J. Gray, "Parallel database systems: the future of high-performance database systems," *Commun*, vol. 35, no. 6, pp. 85–98, 1992.

[14] A. Pavlo, E. Paulson, and A. Rasin, "A comparison of approaches to large-scale data analysis," in *SIGMOD 09: Proceedings of the 2009 ACM SIGMOD International Conference*, 2009, pp. 165–178.

[15] M. Stonebraker, D. Abadi *et al.*, "Mapreduce and parallel dbmss: friends or foes?" *Commun*, vol. 53, no. 1, pp. 64–71, 2010.

[16] "Hadoop streaming." http://hadoop.apache.org/common/docs/r0.15.2/streaming.html.

[17] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: A not-so-foreign language for data processing," in *International Conference on Management of Data - SIGMOD*, 2008, pp. 1099–1110.

[18] A. Thusoo, J. S. Sarma *et al.*, "Hive - a warehousing solution over a map-reduce framework," in *Proceedings of VLDB Endow*, 2009, pp. 1626–1629.

[19] A. Abouzeid, K. Bajda-Pawlikowski *et al.*, "Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads," *Proceedings of VLDB*, vol. 2, no. 1, pp. 922–933, 2009.

[20] T. Chen and K. Taura, "Data-intensive text processing workflows with a parallel database system," in *Proceedings of Summer United Workshops on Parallel, Distributed, and Cooperative Processing (SWoPP12)*, 2012.

[21] ——, "Paralite: Supporting collective queries in database system to parallelize user-defined executable," in *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, 2012, pp. 474–481.

[22] "Sqlite." http://www.sqlite.org/.

[23] M. Miwa, R. Satre, J.-D. Kim, and J. Tsujii, "Event extraction with complex event classification using rich features," in *JBCB*, 2010, pp. 131–146.

[24] D. FA, "Searching medline via pubmed," in *Clin Lab Sci*, 2008.

[25] S. Goldwater, T. L. Griffiths, and M. Johnson, "Contextual dependencies in unsupervised word segmentation," in *Meeting of the Association for Computational Linguistics - ACL*, 2006.

[26] A. S. Balkir, I. Foster, and A. Rzhetsky, "A distributed look-up architecture for text mining applications using mapreduce," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC11)*, 2011.