PAPER

ParaLite: A Parallel Database System for Data-Intensive Workflows

Ting CHEN[†a)], ***Member and*** **Kenjiro TAURA**[†], ***Nonmember***

SUMMARY To better support data-intensive workflows which are typically built out of various independently developed executables, this paper proposes extensions to parallel database systems called *User-Defined eXecutables (UDX)* and *collective queries*. UDX facilitates the description of workflows by enabling seamless integrations of external executables into SQL statements without any efforts to write programs confirming to strict specifications of databases. A collective query is an SQL query whose results are distributed to multiple clients and then processed by them in parallel, using arbitrary UDX. It provides efficient parallelization of executables through the data transfer optimization algorithms that distribute query results to multiple clients, taking both communication cost and computational loads into account. We implement this concept in a system called ParaLite, a parallel database system based on a popular lightweight database SQLite. Our experiments show that ParaLite has several times higher performance over Hive for typical SQL tasks and has 10x speedup compared to a commercial DBMS for executables. In addition, this paper studies a real-world text processing workflow and builds it on top of ParaLite, Hadoop, Hive and general files. Our experiences indicate that ParaLite outperforms other systems in both productivity and performance for the workflow.

key words: *data-intensive workflow, parallel database system, user-defined executable, collective query*

1. Introduction

Workflows [1] have become one of the most important tools for data-intensive applications since they facilitate the composition of individual executable, making it easier for domain experts to focus on their researches rather than computation management. A workflow is generally a DAG with a set of independently developed jobs and their dependencies. Each job is a typical existing binary or executable and communicates with another job in the workflow. For example, workflows for natural language processing (NLP) application typically consist of data scrapers, sentence splitters, part-of-speech taggers, named entity recognizers, parsers, data indexers, and so on. Many of them (e.g., parsers [2], [3]) are third-party components that received a large amount of development efforts from the domain community and are usually developed in a variety of languages. Since a job is typically an existing executable, data transfers between jobs are generally handled by the workflow system. Usually, data are stored in files and implicitly transferred through a shared file system or explicitly moved by a staging subsystem. Such file-based workflows are often very complex with many jobs due to the low-level description. To schedule a job to computing resources for parallel execution, the input of the job is generally split into multiple small files, thus, leading to a large number of intermediate files. In addition, as data are stored in files, it is tedious to select a subset of data due to the difficulties in creating index on data. Nowadays, many systems are proposed to execute workflows, including GXP Make [4], Swift [5] and Taverna [6].

With a common goal of making large scale data processing simple and easy, MapReduce model [7] 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 [8] in particular, more and more efforts are made to enable or utilize them in scientific workflows. Researchers either create workflows with MapReduce features or integrate Hadoop into workflows to get better performance for the execution of jobs. However, MapReduce in general requires users to develop two functions map and reduce; Hadoop requires them to be written in Java conforming the class library framework, at least by default. This low-level description increases difficulties for users to develop their applications. Hence, some MapReduce systems are extended to support high-level language (e.g. SQL-like queries) [9], [10]. In addition, as data are stored in the distributed file system (HDFS in Hadoop), it is difficult to index data too.

While there is a critical need for workflow systems to manage scientific applications and data, and database technologies are well-suited to deal with some specific aspects of workflow management, a nature idea is to build workflows on top of parallel database systems [11]. Workflow systems each of which utilizes database technology to some extent, such as GridDB [12], Zoo [13] and Kepler [14], have been proposed to provide functionality such as simplifying the description of a workflow with SQL queries, improving the performance of the execution and facilitating the management of data. Firstly, with expressive SQL, database systems can simplify the description of workflows. For instance, SQL queries with a proper support of user-defined functions and reductions can express many data processing tasks much more elegantly and easily than MapReduce. Secondly, database systems facilitate the management of data naturally. Finally, databases are efficient for processing relational data in ways expressible in SQL due to data indexing and sophisticated query optimization [15], [16].

While parallel database systems facilitate the description of workflows in terms of expressing jobs with SQL

Manuscript received September 9, 2013.
Manuscript revised December 19, 2013.

[†]The authors are with Information and Communication Engineering, The University of Tokyo, Tokyo, 113–0033 Japan.
a) E-mail: chenting@eidos.ic.i.u-tokyo.ac.jp

DOI: 10.1587/transinf.E97.D.1211

queries and providing efficient management of data, they generally have some limitations. As a workflow is typically built out of various individually developed executables, integrating such executables into SQL statements is very crucial. Most databases execute external modules in the form of user-defined functions or stored procedures. Thus, programmers who want to invoke such executables as part of SQL statements have to write and compile them with respect to the strict specifications of databases, and are usually constrained by the languages they can use (e.g. C/C++/Java). It is obviously unreasonable for scientists to rewrite their applications with a large number of such executables to allow them to be run by a database. Another general limitation of parallel database systems is that they do not optimize data transfers between data nodes and parallel clients that process large query results. A significant work exists in minimizing IO costs and data transfers inside the execution of an SQL query [11], but query results are all returned to a single client who issued the query. When big results are returned to a single client and then distributed to external programs for parallel execution, the single client can easily become a bottleneck.

With consideration of the advantages of databases, our goal is to develop workflows on top of a database system *ParaLite*, with which jobs are expressed in SQL queries and all intermediate data are stored as relational tables. ParaLite is a shared-nothing parallel database system which provides an coordination layer to connect multiple single-node databases and parallelizes queries across them. To support workflows better, ParaLite provides seamless integrations of external executables (UDX, short for User-Defined Executable) into SQL statements and proposes a concept of collective query for the efficient parallel execution of UDX through co-allocation of parallel compute clients and data sources the two driving design principles. In our previous study, we introduced the detail of ParaLite and its workflow-oriented features [17]. In the present paper we extend the previous study by comparing ParaLite with a commercial database systems for both typical SQL tasks and executables. We also develop a real-world text processing workflow on top of ParaLite, Hive, Hadoop and general files and discuss their strength/weaknesses both in terms of programmability and performance for the workflow. In summary, the major contributions of our work are as follows:

- We provide *User-defined Executable (UDX)* to make it straightforward to integrate arbitrary executables within a SQL query. UDX considerably lowers users' efforts to describe jobs in workflows as it allows the user to define the executable with the input/output format directly in a query without writing any program.
- We propose a concept of *collective query* for the efficient parallel execution of UDXes, a single SQL query issued by multiple clients who collectively receive the results of the query and process them in parallel using UDXes. Collective queries enable the co-allocation of computing clients and data sources (data nodes in databases) with consideration of data locality and load balance across all clients.
- Our experiments show that ParaLite has several times higher performance over Hive [9] for typical SQL tasks and has 10x speedup comparing to a conventional DBMS for executables. In addition, ParaLite can achieve close-to-ideal speedup with the increase of computing clients.
- We study a real-world text processing workflow and develop it on top of ParaLite, Hadoop, Hive and general files. 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 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 small differences could reveal some potential trade-offs that each system entails for workflows.

The rest of this paper is organized as follows. Section 2 presents related works. In Sect. 3, we present our major work ParaLite, a shared-nothing parallel database system and the details of the integration of executables into SQL statements and the parallelization of them. Section 4 gives the evaluation of ParaLite to verify the scalability and performance of the system. In Sect. 5, we show our efforts on a real-world text-processing workflow. Finally, we give our concluding remarks in Sect. 6.

2. Related Work

2.1 Integrating Databases and Workflows

Many researches have been done in the field of managing workflows with databases. Some solutions for workflows focus on the control flow among processes such as active databases [18], relational transducers [19] and enhanced datalog [20]. As scientific jobs are typical long-running and resource-intensive, it is essential to efficiently manage the data-flow. Furthermore, it is necessary to have sophisticated tools to query and visualize the data. Zoo [13], a desktop experiment management environment built on top of Horse OODBMS, is developed for this purpose. GridDB [12] models the inputs and outputs of programs as relational tables. It allows users to define programs and the dependencies between their inputs and outputs with a functional data modeling language (FDM). The execution of programs in the workflow is triggered by the change of the input tables such as insertion of tuples. [21] is another work related to the execution of scientific programs. The work demonstrated the advantages of modern DBMSs such as data indexing, query parallelization and efficient joins by the

cluster-finding example from the Sloan Digital Sky Survey. It obtained several times better performance with a database (Microsoft's SQL Server [22]) than previous approach. The work involves invoking program modules from SQL statements which is usually in the form of User-defined Functions (UDF) or stored procedures. Therefore, the developers have to conform to the database specifications while coding and compiling them and are usually constrained by languages (C, C++ and Java). Our work solves this problem by introducing the idea of User-defined Executable which provides straightforward integration of programs without conforming to database specifications.

2.2 Integration of External Executable

To allow integration of data processing methods into query execution plans, relational database systems support user-defined functions (UDF) [23], [24] which are longstanding database features for database extensibility. There are significant research focusing on efficiently using UDFs within database queries in terms of both optimization and execution, such as [25], [26]. However, most of their work in the context of single-node database systems rather than shared-nothing parallel databases.

There are some works related to the parallelization of user-defined aggregates, table operators [27] and scalar functions [28]. By specifying local and global finalize functions, conventional user-defined aggregates can be executed in parallel [28]. Furthermore, [27] proposes user-defined table operators which use relations as both input and output. The idea of a user-defined table function is supported in most commercial databases. To enable parallelism and tell the system the usable of the operator, user-defined table operators require the user to specify a partitioning method, which is inconvenient for the user. In addition, they lack flexibility in input/output formats, development language and reusability of code [28], [29].

On the other hand, MapReduce framework with its most popular implementation Hadoop [8] provides the user procedural API (Map and Reduce functions) to customize their own processing logic. However, it requires programming in relatively low-level languages (most commonly C++ or Java) even for very straightforward tasks that would be trivial in SQL. Integrating third-party binaries and ad-hoc scripts need similar efforts just to wrap them. In addition, Hadoop programs are often slower than equivalent SQL queries because the former lack data indexing and require multiple MapReduce jobs each accessing files for intermediate results.

Therefore, many hybrids of relational databases and MapReduce have been proposed. On one hand, SQL/MapReduce [30], a part of Aster nCluster Database, proposes an approach to polymorphic user-defined functions providing users with a procedural API (row and partition methods like map and reduce methods in Hadoop) through which they can implement a UDF in the language of their choice. These user-defined methods are parallelized by MapReduce. However, the user still needs to write programs conforming to database APIs and thus cannot directly use existing file-based applications. On the other hand, SQL-like declarative languages are supported by MapReduce frameworks with optimizations to some extent, such as Hive [9], Scope [31], HadoopDB [32] and Tenzing [33]. While these languages significantly improve the productivity of MapReduce programs, they generally have some limitations for the support of workflows. Firstly, they are not productive for most of NLP workflows. For example, Hive does not provide the functionality of tracking the association between a document and a result from NLP tools and cannot support executables which take the input data from files. Secondly, as almost all these systems run the executable using Hadoop, they cannot take advantage of database technologies, such as good data partitioning and query optimization. Finally, all data are stored in distributed file system, bringing difficulties in creating indexes on them.

3. ParaLite

3.1 Architecture of ParaLite

ParaLite is a shared-nothing parallel database system based on a popular single-node database SQLite [34]. The basic idea of ParaLite is to provide a coordination layer to glue many SQLite instances together, and parallelize an SQL query across them. The architecture of our system is shown in Fig. 1. It uses classic master/worker pattern to organize resources. ParaLite is designed to be a serverless and zero-configuration system, so no process is running before a query is executed. ParaLite has multiple clients which present an SQL interface to users and allows a group of queries to be submitted at the same time. The master is responsible for transforming received queries into the logical plan (a DAG of operators) which is the key structure to connect each logical component, creating processes for operators on data nodes and scheduling and dispatching jobs to corresponding processes. Data are transferred among data nodes and computing clients, thus the master works only for the controlling decisions and is not a bottleneck for any data transfer. Each process on a data node receives data from another, handles them using its own processing logic (e.g. join,

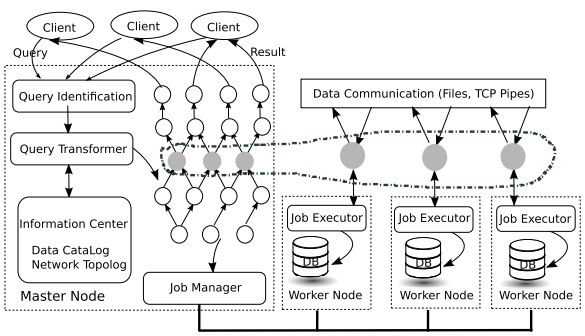


Fig. 1 Architecture of ParaLite.

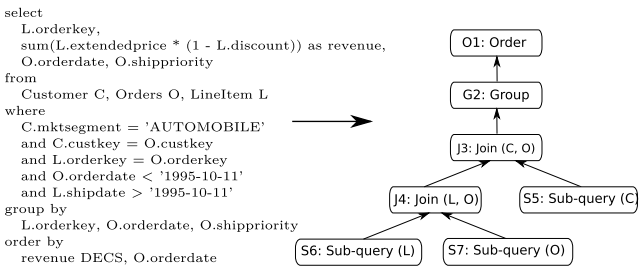


Fig. 2 A logical plan for TPC-H Query 3.

sort and aggregate) and sends the output data to the next process. The root operator in the logical plan returns results to the clients.

3.2 Query Model

A query is expressed by a DAG (query plan) of relational operators each of which forwards data tuples to the next. For example, the query plan of Query 3 of TPC-H benchmark [35] is shown in Fig. 2. As ParaLite stores data in SQLite database, it uses SQL query to access data. To take advantage of database technologies of SQLite, e.g. indexing and query optimization, we push as much operations as possible into the query to the underlying SQLite. So the leaf nodes of the plan are sub-query operators which read relations using corresponding predicates and produce a row-and-column subset of the relational table. Operators *J3* and *J4* join two relations. The output from *J3* is then aggregated and sorted. Specially, some operators can be integrated into a single query if no repartitioning operations are necessary for them. For example, if relations *Customer* and *Orders* are both hash-partitioned across data nodes by the join attribute, *J4*, *S6* and *S7* can be integrated into a single query.

Each operator is either a *pipeline* operator, which can process each tuple independently without the knowledge of all tuples, or a *blocking* operator which must receive all tuples before emitting the result, e.g, an aggregation operator and a sorting operator. Each operator is split into multiple logical tasks and assigned to a set of processors. The number of tasks is determined by the number of partitions for the input tuples of the operator and is usually much larger than the number of assigned processors. If an operator's successor is a pipeline operator, it forwards the output tuples of each task to the its successor as soon as the task is finished. The target processor is chosen based on its processing capacity in terms of estimated runtime. For an operator whose successor is a blocking operator, it holds all output data on memory until it reaches a threshold, at which point it writes them into an intermediate file. Tasks of a blocking operator is scheduled to processors using a greedy algorithm to balance the load across all processors. Once a processor becomes idle, a task is allocated to the processor.

3.3 Integration of Executable

As the intended applications for ParaLite are workflows typically built out of various independently developed executables and scripts, ParaLite extends SQL to support arbitrary executable called User-Defined Executable (UDX).

3.3.1 Syntax of UDX

A ParaLite UDX is an executable file which can be written in any language. This is very flexible because a user does not need to develop a program respecting to rigid formatting rules such as <key, value> input/output format or write code according to pre-defined procedural methods. User can use arbitrary format of input and output and any programming language to implement their functions. The only condition is just to make sure the program is executable. This has been very useful for data-intensive science computations such as a linear algebra package for solving linear equations and a natural language processing library. In these applications, most functions are developed by domain experts and then reused by others on diverse workflows. ParaLite programs allow such functions to be reused without changing any code.

The syntax of UDX shown below is similar to that of traditional User-Defined Function. Firstly, the name of an UDX could be a random string e.g *X* in the example. Secondly, an UDX can work on and produce arbitrary columns. It extends *AS* syntax a little bit so that multiple columns are allowed to be the output of a UDX by using "*AS (col1, col2,...)*". Finally, the definition of an UDX provides flexible input/output format with a set of options such as, *input*, *input_row_delimiter*, *output* and *out_row_delimiter*. The options related to input allow the system to correctly extract and organize data for the executable while output-related options tell the system how to parse the output of the executable and store them in a relational table. Specially, *input* and *output* options can specify that the input/output for the executable comes/goes from/to the standard input/output or files. This ability is especially useful for file-based programs commonly existed in NLP applications. In addition, to avoid create and compile an UDX before the query is executed, ParaLite allows users to define it within the query using *WITH* clause. It starts from a command line followed by data format options mentioned above.

```
select col1, X(col2) as new_col2
from T
where <predicate1, predicate2, ...>
with X = "command_line"
```

3.3.2 Examples

In this section, we take some examples to elaborate the usage of UDX. First of all, let's define the schema for a table *DOCUMENT*:

```
paper_id | title | author | year | text
```

- Grep Task
  Grep task is considered as a typical MapReduce task

which scans through a large set of records looking for a three-character pattern. This task can be easily performed by a query with shell scrip command "grep" as a UDX:

```
select F(text) from DOCUMENT
with F = "grep XYZ"
```

All data of column `text` retrieved from table `DOCUMENT` is processed by the UDX `grep XYZ` and the filtered data are returned.

- Word Count Task
  Word count task is also a typical MapReduce task [7] to count the number of occurrences of words in a large collection of documents. This task processes a document table in which a single row is a single document with its descriptions and generates a word table in which a single row is a word with its occurrences. While this task could be easily expressed by MapReduce researcher with a Map and a Reduce job, there is no easy way to perform it in database community unless the big text could be split into words. With UDX, it is straightforward to integrate text splitter into general group by SQL task to calculate the count for each word.

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

  Column *text* is a long article split into independent words by an *awk* command in the nested SQL. The occurrences for each word is simply counted by grouping words from the output of the nested query. The command *awk* reads data from standard input and writes results to standard output.
- Sentence Split Task
  The sentence split task is to split a big text into sentences and it is normally the first step of almost all text-processing applications. It involves a third-party binary *geniass* [36] developed by domain researchers which reads a text from file, splits it into sentences by inserting line breaks between sentences within a paragraph and empty lines before the sentence from another paragraph, and finally outputs the sentences to a file.

```
select paper_id, F(text) as sentence
from DOCUMENT
with F = "geniass"
         input 'src_file' output 'dest_file'
         output_row_delimiter EMPTY_LINE
```

  In the query above, the user needs to specify the input and output for the executable to be a file. This is different from the word count task in which the executable reads/writes data from/to the standard input/output. Furthermore, if the option *output_record_delimiter* is empty line, it means that all results between two empty lines belong to a single record. Each result record has two columns of *paper_id* and *sentence*. The second column contains multiple sentences separated by line breaks.
- Web Log Analysis Task
  This task first generates a collection of HTML documents which are crawled from web pages. There are random links to other pages in each document. Then this task computes the inlink count for each document which is often used as a component of PageRank calculations. The schema of the HTML document is: `Pages (url | content)`.
  The SQL query for this task is similar with that for the word count task. In the nested query, the executable *url_finder* reads the contents of each page, searches for all the URLs that appear in the contents and outputs the result pairs of URL and its value. As a result, the nested query produces a temporary table with two columns (*to_url* and *value*). Note that we extend the syntax of SQL to support arbitrary output columns. The user needs to tell the system the separator between different columns by specifying the option *output_col_delimiter* to be TAB (it could be any strings). Then the outer query just simply aggregates all *to_url* and calculates their values respectively.

```
select to_url, sum(value) from (
  select F(content) as (to_url, value) from
  Pages with F = "url_finder"
  output_col_delimiter TAB
)
group by to_url
```

3.4 Parallel Execution of UDX

3.4.1 Concept of Collective Query

A parallel database system provides the same functionality as a centralized DBMS with the ability of transparently distributing data across nodes and parallelizing queries. It typically consists of a single master node and multiple data nodes. A master is responsible for receiving a query from a client, converting the query to a parallel execution plan, scheduling the plan to worker nodes, and assembling the individual "pieces" of the final results up into a single result set to the client. As such, the master node hides the distributed nature of the system and presents users a single system image. Data nodes provide the data storage and the query processing backbone of the appliance.

Let's assume that a user wants to apply a parallel processing on the data accessed from a parallel database such as loading the data into another system (e.g. MapReduce system) or performing a further analysis which cannot be expressed within a SQL query. As the left part of Fig. 3 shows, a traditional approach is that a user issues a query, gets all result data from the database system and then distributes them to multiple clients who perform the further processing on the result data in parallel. Obviously, the single query issuer can easily become a bottleneck. Moreover,

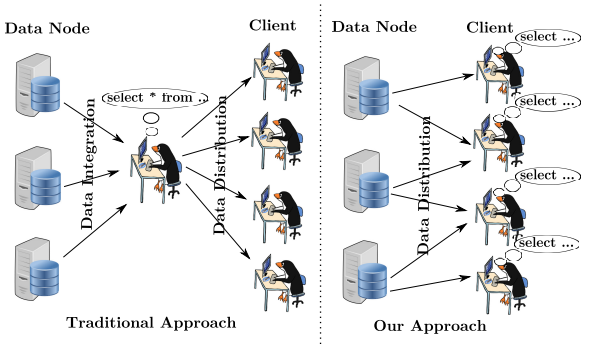


Fig. 3 Data transfer in conventional parallel DBMS.

as the approach prohibits us to take the advantage of co-allocating computing clients with data in the database, data transfer probably brings much unnecessary overhead to the overall execution. For example, if the computing clients are located on the same physical data nodes, it is straightforward and efficient that each client gets data locally from the corresponding data node, instead of integrating data from data nodes first and then distributing them to the same nodes again.

Therefore, to solve this problem, we propose the concept of *collective query* to take advantage of co-allocation of parallel compute clients and data sources. A collective query is a single query issued by many clients who collectively receive the results of the query. As the right part of Fig. 3 shows, a set of clients issue the same query to the database system. They just notify the system to get a part of data and don't care about what exact data they will obtain. The system performs the query received from the clients and distributes the result to them. As the system knows the destinations of the result in advance, it allows the data transfer directly from the data nodes to the clients without passing through the master. In this case, data are not required to be integrated first and distributed later. The best situation is that when a computing client is running on each data node and data are already balanced among them, no data are transferred between nodes at all and each client gets data locally from the data node the client is running on. Note that Fig. 3 does not show the architectures of both systems but only indicates how data are transferred from data nodes to computing clients. In both approaches, there is a master node who is responsible for coordinating the behavior of all data nodes and computing clients.

ParaLite uses the concept of collective query for the parallelization of executables. The query issued by multiple clients contains one or more UDX(es). The results of the query are distributed to the clients and then processed by them in parallel, using the executables defined in the query.

3.4.2 Data Distribution

The key issue for collective query is to efficiently distribute data to clients from data nodes. The result data are always stored in multiple nodes, thus, a more general problem setting should be that given data on N data nodes and m clients, to determine which data should go to which client. In data intensive applications, a client may perform a significant amount of computation, therefore, ParaLite should consider not only communication cost but also computation cost.

ParaLite implements two-phased DLLB (Data Locality with taking Load Balance into consideration) algorithm to solve the problem. The first phase in the algorithm is to generate tasks on each data node by splitting data into small blocks. Each block size is denoted by *bsize*. A block is the smallest unit that is transferred to a computing client at a time and a process of the executable is spawned for a block of data. Invoking a process on a block of data rather than a single tuple reduces the start-up overhead a lot for most NLP programs. From our experiments which are not shown in this paper due to space limitation, the block size heavily affects the overall performance. Large block size brings difficulty in rigorously balancing the load of all clients while small block size increases the start-up overhead. ParaLite does not optimize it automatically but leaves the decision to users. Users can specify the block size based on features of the executable and their experience.

Once splitting is complete and execution has begun, we enter the load-balancing phase. When a client i on the node A becomes idle, a task (block) must be transferred to it from some data node. Our goal is to balance load across all clients but with the smallest data transfer cost. Thus, the target data node T should be:

(1) node A, if A is a data node, or,

(2) node B, who has the maximum expected completion time:

ECT is an expected completion time as measure of the load of a node. *ECT* of a data node J relies on the total data and clients it holds and can be calculated as (assuming c clients are running on it):

$$ECT_J = lsize_J / \sum_{i=1}^{c} s_i + max_{i=1}^{c}(bsize/s_i - stime_i)$$

$lsize_J$: the size of left (unassigned) data on data node J

s_i: the speed of client i

$stime_i$: the time for client i starting to process a task

The speed of each client is initiated as a random number. The *ECT* should be infinite if a data node has no client on it. Once a client completes a task, it sends an ACK message to the master to notify about its IDLE status and report its processing speed which is used for the next scheduling. Since it is difficult to provide an exact measure of real-time speed for a client, ParaLite always estimates it by the last speed of a client. On the other hand, after the master receives a ACK message from a client, it updates its speed and data information of related data node whose data are processed by this client. Once a client becomes idle, the master firstly calculates the *ECT* of all data nodes and decides a target data node based on the formula above. The computational complexity for deciding a target data node is $O(n)$ where n is the number of client. It costs 10^{-6} seconds

if there is only one client based on some experiments we conducted.

Once the target T is chosen for an idle client i, it must decide whether data (one block) is needed to be transferred to the idle client or not. DLLB employs the following equation to calculate the number of blocks:

$$n = \begin{cases} 0 & \text{if } T = B \text{ and } ECT_T(cur) < Com(1) + ECT_i(1) \\ 1 & \text{otherwise} \end{cases}$$

Data are transferred to the idle client only when the transfer provides a gain in the completion time, that is, the sum of the task transfer time ($Com(1)$) and the estimated completion time of the task on the idle client ($ECT_i(1)$) should be smaller than the estimated completion time of all tasks on the target node ($ECT_T(cur)$).

The DLLB algorithm is flexible for data scheduling. It first takes data locality as a main consideration and always tries to perform calculation on local data. But if the calculation is CPU-intensive and data transfer can get a gain to the calculation, data are transferred to a remote client to be calculated. At most one block is transferred at a time, which lets the master node have better control on data. Fine-grained data scheduling adds extra overhead of making decision to the master node, but the cost of making decisions is negligible even the data unit to be transferred is 1 compared with the cost of data transferring and calculation.

3.5 UDX Compared to UDF

Compared with ordinary User-Defined Function (UDF) in conventional parallel database systems, our implementation of User-Defined Executable (UDX) has the following advantages:

- UDX doesn't require the user to write any program and register to the database before it is executed. To define UDF in conventional databases, the user has to write a specific program conforming to strict database specifications. This is very troublesome in workflows as typically only executables are provided. In this case, for each executable, a program that encapsulates the invoking of the executable is required. Furthermore, programs are usually constrained with the language they can use as most database systems only support UDF written in C/C++ or Java.
- The UDX implementation does not invoke the executable on every single tuple while the implementation of UDF does. As typical NLP programs have high start-up overhead, invoking such programs on every single tuple would heavily degrade the overall performance.
- The execution of an UDX is not bound to database nodes and it can be distributed to arbitrary clients for larger scale execution and computational load balancing. This loose coupling with database nodes is also very useful in the case of that data are stored in a set of nodes while the related executable is installed in other sets of nodes for some reasons, e.g. the licenses. On the other hand, an UDF can only be parallelized across data nodes pre-configured before the database server starts.
- UDX parallelization is efficient as it optimizes data transfer between data nodes and computing clients. Most commercial database systems take a naive strategy to parallelize UDF which assigns a whole partition of data to a local processor without consideration of its load. Moreover, the implementation of the UDX allows flexible control on the parallelism degree by increasing or reducing the number of computing clients.

4. Evaluation

We conducted all experiments in a 32-node cluster. Each node uses 2.40 GHz Intel Xeon processor with 8 cores running 64-bit Debian 6.0 with 24GB RAM and 500G SATA hard disk. According to hdparm, the hard disks deliver 86MB/sec for buffered reads.

4.1 Compared Systems

We compare ParaLite with a commercial parallel database system DBMS-X from a major relational database company and a popular MapReduce system Hive.

DBMS-X: We installed the newest release of DBMS-X, a parallel row-oriented SQL DBMS. The official TPC-H benchmark conducted by the DBMS-X vendor used a slightly older version of the system. We specified our parameters for our installation the same with that in the official TPC-H benchmark. Specially, we did not enable the replication features in DBMS-X because all queries in the benchmark are read-only and enabling replication features makes it more complex for the installation process. We installed this version on each node.

Hive and Hadoop: For experiments in this paper, we used Hive version 0.8.1 and Hadoop version 1.0.3, running on Java 1.6.0. We configured both systems according to the suggestions offered by members of Hive's development team in their report on running TPC-H on Hive [37]. To reflect our hardware capacity, we configured the system to run eight Map instances and eight Reduce instances concurrently on each node. We also allowed JVM to be reused by all tasks instead of starting a new process for each Map/Reduce task. To make the comparison fair, we stored all input and output data in HDFS with the settings of one replica per block and without compression.

4.2 Data Preparing and Loading

The TPC-H benchmark data were generated in parallel on every node using the dbgen program provided by TPC. We used the appropriate parameters to produce a consistent dataset across the cluster.

DBMS-X: We followed the suggestions from DBMS-X vendor to create the tables and indices, and to distribute

data across the cluster. All tables were hash-partitioned across the nodes by their primary keys while *PartSupp* and *LineItem* relations were hash-partitioned on only the first column of their primary keys. In addition to creating index on the primary key for each table, the *Supplier* and *Customer* relations were indexed on their nation keys respectively, and the *Nation* table was indexed on its region column. Finally, the *LineItem* and *Orders* relations were organized by the month of the date columns for a partial ordering by date on each node of the cluster. The loading process worked as follows: data were first partitioned across the cluster and then the partitioned data were loaded on each node in bulk. The time for loading TPC-H data (with scaling factor of 100) to 16 nodes is about 3 hours and 21 minutes.

Hive and Hadoop: We first loaded the source data into HDFS using the Hadoop command-line utility. The utility was run in parallel on all nodes and copied unaltered data files into HDFS under a separate directory for each table. Each file was automatically broken into 128MB blocks and stored on a local DataNode. Then we executed Hive DDL scripts provided by the Hive development team special for TPC-H benchmark to put relational mapping on the files. Since the metadata creation cost is negligible, the entire data preparing time is considered as loading data into HDFS and it took only 7 minutes for the TPC-H data.

ParaLite: In ParaLite, all tables are hash-partitioned across the cluster and indexed on the same key with that in DBMS-X respectively. But ParaLite cannot organize the *LineItem* and *Orders* relations by the month of their date columns. The main process of loading data is almost the same with DBMS-X. It first parses each record and sends it to the correct partition. Then each node loads received records to the SQLite database locally in parallel. The whole process took about 2 hours and 23 minutes for the same TPC-H data.

4.3 Weak Scalability

We perform the TPC-H Query 3 as shown in Fig. 2. As the two relations *Orders* and *LineItem* are partitioned on the join key, the join operation on them is pushed into a SQL query to be executed by SQLite directly in ParaLite. On the other hand, Hive needs another single MapReduce job to join these two relations. As a result, Hive requires 4 MapReduce jobs to express this query in total.

In the experiments, we increased the nodes from 10 to 30 with about 10GB on each node. The results are shown in Fig. 4 as we expected: (1) Both systems scale well with the increase of data nodes; (2) ParaLite is about 4 times faster than Hive. As we explain before, firstly, join and aggregation are faster in ParaLite due to data partitioning. Then writing all intermediate data to durable storage in Hive degrades the overall performance. In addition, the start-up overhead of Hadoop cannot be ignored since there are 4 MapReduce jobs in total and each one takes about 15 seconds until the first Map task begins.

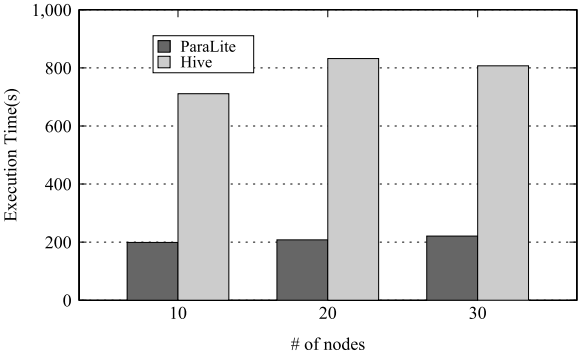


Fig. 4 The performance of TPC-H Query 3.

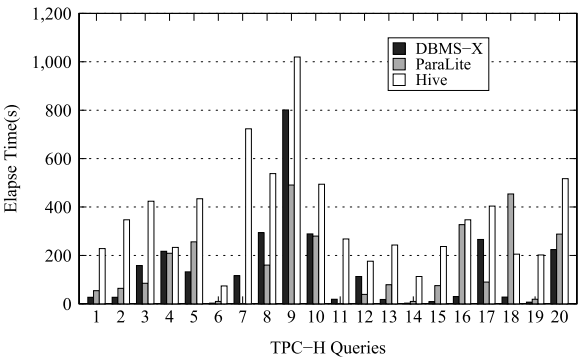


Fig. 5 TPC-H performance of several approaches.

4.4 Completion Time for TPC-H Query

We run TPC-H queries from Query 1 to 20 with scaling factor 100 in a cluster of 16 nodes. For DBMS-X and Hive we executed the queries as suggested in the official TPC-H reports by the vendors. Since the syntax of HiveQL is just a subset of SQL, for may queries, the original TPC-H queries were rewritten into a series of simple queries. For ParaLite, several queries are also rewritten into a series of simple queries as Hive does and Query 7, 11 cannot run successfully because currently ParaLite does not support operations like *left join* and nested query in *where* clause.

Figure 5 shows the benchmarking results for all three systems. First, it is not surprising that DBMS-X and ParaLite significantly outperform Hive for almost all queries. The main reason for the superior performance of DBMS-X's and ParaLite is the ability to take advantages of partitioning and indexing. Without this ability, Hive performs a full data scan for every selection and most of the joins in Hive require to repartition and shuffle all records across the cluster. However, ParaLite is slower than Hive for Query 18. ParaLite and Hive rewrite this query into several simpler ones. ParaLite stores the output of each query to the database, which takes much more time than storing them to HDFS in Hive.

Second, compared to DBMS-X, the performance of ParaLite is competitive. ParaLite loses for some queries

and slightly wins for other queries. The main reason for the inferior performance is the lack of organization of data. DBMS-X organizes the *LineItem* and *Orders* relations by the month of their date columns for a partial ordering by date. DBMS-X gains much from the organization of data for many queries such as Query 2, 15. Another reason is that for some rewritten queries, ParaLite is required to store intermediate data from each step to database as mentioned above while DBMS-X does not need to materialize any of them.

4.5 Performance for Executable Execution

Providing the straightforward and efficient integration of external executables into query plans is a major feature of ParaLite for workflows. In this section, we run several queries with the integration of both heavy (Enju) and lightweight (simple_tokenizer.pl) executables.

Enju [2] is a fast, accurate, and deep parser for English text and widely used in natural language processing (NLP) applications. It is a cpu-intensive program with very high start-up overhead. It firstly needs to load dictionary before processing sentences which takes about 8 seconds.

ParaLite expresses the task simply by a query with a UDX definition as follows.

```
select sid, F(sentence) as enju_result
from   Abstract
with   F = "Enju" output_row_delimiter EMPTY_LINE
```

DBMS-X performs the task with an UDF in the query below. In the experiments, we implement the function *F* in a java program which receives each sentence, starts the Enju process and returns the result tuple.

```
select sid, F(sentence) as enju_result
from   Abstract
```

Hive provides the syntax to integrate any executable into the HiveQL straightforwardly. It runs the executable as a MapReduce job. However, it cannot map the sentence ID to the enju result for the sentence. So we encapsulate the executable in another program to fulfill the mapping. The program reads records with two columns, feds only sentences to the enju program and maps the output of enju of a sentence to its ID.

```
from (
  from Abstract map sid, sentence
       using 'enju_wrap' as SID, enju
) map_output
select map_output.sid, map_output.enju
```

The executable *simple_tokenizer.pl* is also a NLP tool to tokenize the words in sentences. It reads sentences from standard input and returns the sentences with tokenzied words. Similar with *Enju*, this executable is expressed by ParaLite, DBMS-X and Hive in the same way. For example, ParaLite performs it using the following query:

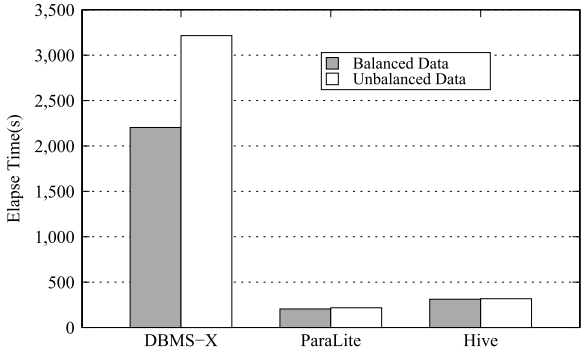


Fig. 6 Completion time of the heavy executable.

```
SELECT sid, S(sentence) as tk_sentence
FROM   Abstract
with   S = "perl simple_tokenizer.pl"
```

We tested the completion time of the two executables with both balanced and unbalanced data across a 16-node cluster.

For the execution of *Enju*, 1.6MB data are distributed across the cluster. Each node has 0.1MB data when the data are evenly distributed while each of 8 nodes holds 0.15MB and each of the other nodes has only 0.05MB data when data are not evenly distributed. The results are shown in Fig. 6. Firstly, ParaLite is slightly faster than Hive but about 10 times faster than DBMS-X in both situations. The main reason for the inferior performance of DBMS-X is the high start-up overhead of enju. DBMS-X takes about 8 seconds to initiate a enju process for each sentence while ParaLite takes the 8 seconds for a bulk of data (50KB in our experiments). Secondly, ParaLite has similar performance no matter whether data are evenly distributed or not, so does Hive. However, there exists big differences between the completion time of both cases for DBMS-X because it uses static data scheduling policy for the parallelization of executables. Once data are partitioned, DBMS-X assigns a processor for a partition of data. When the assigned data is finished, the processor does not get the data from another partition. ParaLite and Hive take dynamic data scheduling policies in which once a processor becomes idle, data from other partitions are dispatched to it.

For the lightweight executable, 32GB data are distributed across the cluster. Each node has 2GB data when the data are evenly distributed while each of 8 nodes holds 3GB and each of the other nodes has only 1GB data when data are not evenly distributed. Figure 7 shows the completion time with the two types of data distribution. The results are similar with that for the heavy executable. ParaLite and Hive are 25 times faster than DBMS-X. Even if *simple_tokenizer.pl* does not have high start-up overhead, the time for a large number of processes creation is considerable. Similar with the heavy executable execution, DBMS-X has worse performance when data are not evenly distributed due to the lack of efficient data scheduling policy.

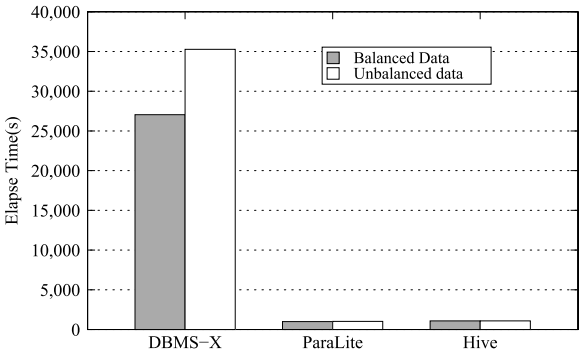
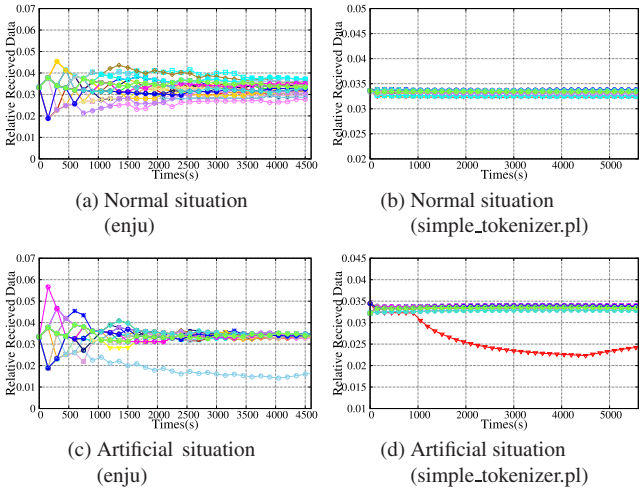


Fig. 7 Completion time of the lightweight executable.



(a) Normal situation (enju)

(b) Normal situation (simple_tokenizer.pl)

(c) Artificial situation (enju)

(d) Artificial situation (simple_tokenizer.pl)

Fig. 8 loads of 30 clients for both executables.

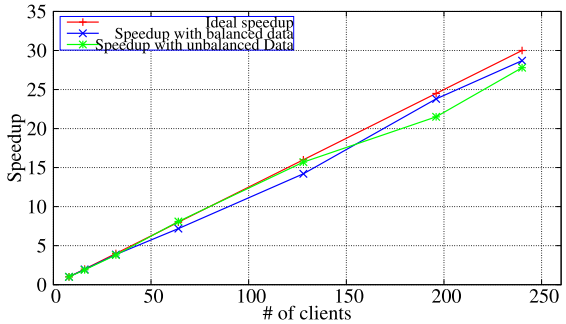


Fig. 9 Speedup for the heavy executable.

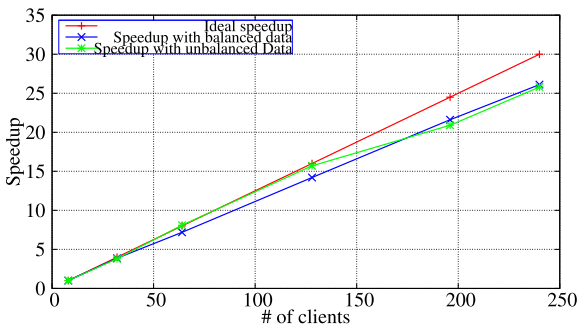


Fig. 10 Speedup for the lightweight executable.

4.6 Evaluation of Data-Distribution Algorithm

In this section, we evaluate our DLLB (Data Locality with taking Load Balance into consideration) algorithm for the data distribution in both normal and abnormal situations. We run 30 computing clients with one client on each node.

In the normal situation that no machine is artificially loaded, the load balance for the cluster over time for both executables are shown in (a) of Fig. 8 and (b) of Fig. 8. The Y-axis is the fraction of whole data that each client receives including its local data and data received from other data nodes. If the system is perfectly load balanced, all clients should receive equal size of data. The system works exactly as we expected while performing the lightweight executable and the 30 clients hover around 0.033 of the whole load. A small deviation from our expectation exists for the execution of the heavy executable. From our observation, the abilities of all clients to perform the heavy executable are not perfectly equal. Some clients are several seconds faster than others for a block of data, leading faster clients getting more data. However, the lines tending to be straight indicate that they get data at the same speed. Most of data are executed locally and only 463KB and 500MB data are transferred through network for heavy and lightweight executables respectively.

In the abnormal situation, we adopted a system stressing utility called CPU Burn-in [38] to artificially load a given machine. CPU Burn-in spawns a number of processes to consume system resources. At $t = 800$, we start CPU Burn-in on *client18*. The results for the heavy and lightweight executables are shown in (c) and (d) of Fig. 8 respectively. Before we push the stress on the client, the load is balanced as in the normal situation. After the client is artificially loaded, its received data decreases and other clients compensate for the over-loaded client's loss quickly. The data transferred through network is 1025KB for the heavy executable and 1.6GB for the lightweight one.

4.7 Scalability of Executable

We test the scalability of ParaLite when the executable is performed with the increase of computing clients. The data distribution for all executable related experiments in the rest of this section is set as follows. For the heavy executable, 30MB data are distributed across a 30-node cluster. Each node has 1MB data when the data are evenly distributed while each of 15 nodes holds 2MB data when data are not evenly distributed. For the lightweight executable, 120GB data are distributed across the same cluster. Each node has 4GB data when the data are evenly distributed while each of 15 nodes holds 8GB data when data are not evenly distributed. The number of clients varied from 8 to 240 for both executables.

As Fig. 9 and Fig. 10 show, the speedup is close to the

ideal (linear) one. The reasons for the small deviation are: (1) the time for accessing data from the database is not decreased with the increase of clients; (2) when data are unevenly distributed, data transfer is increased when the clients are located in the nodes who don't have data.

5. Real-World Text Processing Workflow

Next, we introduce a real-world text-processing workflow in the field of natural language processing – *Event Recognition* (ER) [39], [40]. We build ER on top of ParaLite, Hadoop (specifically Hadoop Streaming [41]), Hive and general files and discuss their strengths/weaknesses in terms of both productivity and performance for the workflow. Since all these systems do not provide any language to describe the complex dependencies of jobs, we generally perform each single job using them and leave the creation of the whole workflow 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.

5.1 Description

The goal of *Event Recognition* 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 process to extract the events from an English sentence is show in Fig. 11.

The workflow of ER is shown in Fig. 12. The input of the workflow is the MEDLINE database [42] 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 using a NLP tool *geniass*; (3) to each sentence, apply three tools:

- Enju Parser: a HPSG parser which can effectively analyze syntactic/semantic structures of English sentences.
- 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. Specifically, the executable *geniass* used in step (2) 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. 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. Moreover, since the input of each executable has multiple lines per record, we have to either customize our own InputFormat and InputReader

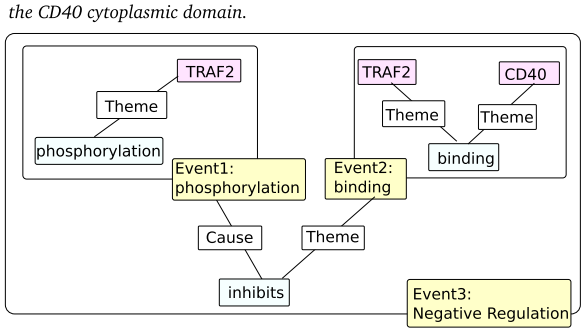


Fig. 11 Extracted events from an English sentence.

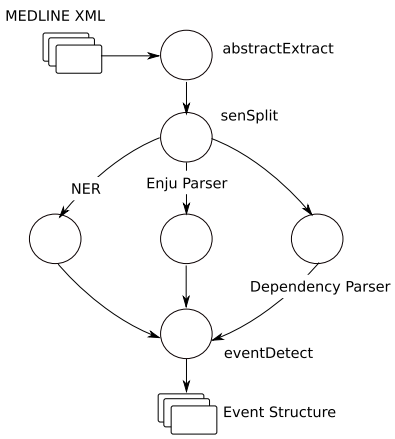


Fig. 12 Workflow of event-recognition application.

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. In summary, 12 extra programmers (wrappers) are required.

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
```

However, to deal with file-based executable *geniass*, 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. In this case, 10 wrappers are required in total.

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
```

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. As a result, ParaLite only needs 5 wrappers to deal with considerable complex of input and output data.

File: To enable the parallel execution of executables, the input file is firstly split 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 are 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. 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. 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.

Table 1 Comparison of productivity of systems for ER.

	Hadoop	Hive	ParaLite	File
# of intermediate files	No	No	No	50000
# of wrapper	12	10	5	10

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. The summary for the efforts made by each system is shown in Table 1.

5.2 Execution

The experiments are conducted in a 30-node cluster which is introduced in Sect. 4. ER workflow reads 30 GB data from MEDLINE database and extracts 1 GB abstracts of articles from the source xml files. For file-based workflow, we split the input into 10000 small ones according to the most time-consuming job `Enju Parser`.

Figure 13 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

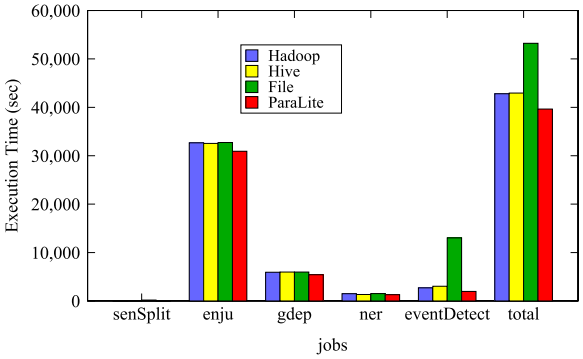


Fig. 13 The execution time of event-recog workflow.

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 split, users have to parallelize each job according to the number of sub-files unless internal parallelization and merge is performed independently.

6. Conclusion

This paper proposes ParaLite – a shared-nothing parallel database system for data-intensive workflows. With ParaLite, jobs in a workflow are expressed with SQL queries and all intermediate data are stored as relational tables. As workflows are typically built out of various executables, ParaLite provides seamless integrations of external executables (UDX, short for User-Defined Executable) into SQL statements and proposes a concept of *collective query* for the efficient parallel execution of UDX. With UDX, users do not need to write any programs that conform to the strict specifications of databases. The implementation of collective query makes ParaLite 10x speed up comparing to UDF implementation in the conventional database.

In addition, we studied a real-world text-processing workflow in the field of natural language processing, and built it on top of ParaLite, Hadoop, Hive and general files. Our development experience revealed that high-level query languages such as SQL of ParaLite and HiveQL of Hive are helpful for expressing data selection, aggregation and calculation by typical executables. 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. The experimental results show that essentially each system has similar performance in the execution of the whole workflows because performing executables takes most time. However, the small differences still revealed some potential superiority of ParaLite due to data partitioning and query optimization.

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] "Enju." http://www-tsujii.is.s.u-tokyo.ac.jp/enju, 2011.
[3] "Cabocha yet another japanese dependency structure." http://code.google.com/p/cabocha.
[4] K. Taura, T. Matsuzaki, M. Miwa, Y. Kamoshida, D. Yokoyama, N. Dun, T. Shibata, C.S. Jun, and J. Tsujii, "Design and implementation of gxp make – a workflow system based on make," eScience, 2010.
[5] 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," IEEE International Workshop on Service Computing, pp.199–206, 2007.
[6] T.M. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M.R. Pocock, A. Wipat, and P. Li, "Taverna: A tool for the composition and enactment of bioinformatics workflows," Bioinformatics, vol.20, no.17, pp.3405–3054, 2004.
[7] J. Dean and S. Ghemawat, "Mapreduce:simplified data processing on large clusters," OSDI'04, pp.137–150, 2004.
[8] "Hadoop." http://hadoop.apache.org/.
[9] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive - A warehousing solution over a map-reduce framework," VLDB Endow, pp.1626–1629, 2009.
[10] A.F. Gates, O. Natkovich, S. Chopra, P. Kamath, S.M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a high-level dataflow system on top of map-reduce: The pig experience," VLDB, pp.1414–1425, 2009.
[11] D. DeWitt and J. Gray, "Parallel database systems: The future of high-performance database systems," Commun., vol.35, no.6, pp.85–98, 1992.
[12] D.T. Liu and M.J. Franklin, "The design of griddb: A data-centric overlay for the scientific grid," VLDB, pp.600–611, 2004.
[13] Y.E. Ioannidis, M. Livny, S. Gupta, and N. Ponnekanti, "Zoo : A desktop experiment management environment," VLDB, pp.274–285, 1996.
[14] D. Barseghian, I. Altintas, M.B. Jones, D. Crawl, N. Potter, J. Gallagher, P. Cornillon, M. Schildhauer, E.T. Borer, E.W. Seabloom, and P.R. Hosseini, "Workflows and extensions to the kepler scientific workflow system to support environmental sensor data access

and analysis," Ecological Informatics, pp.42–50, 2010.
[15] A. Pavlo, E. Paulson, A. Rasin, D.J. Abadi, D.J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," SIGMOD, pp.165–178, 2009.
[16] M. Stonebraker, D. Abadi, D.J. DeWitt, and S. Madden, "Mapreduce and parallel dbmss: Friends or foes?," Commun., vol.53, no.1, pp.64–71, 2010.
[17] T. Chen and K. Taura, "Paralite: Supporting collective queries in database system to parallelize user-defined executable," CCGRID, pp.474–481, 2012.
[18] U. Dayal, E.N. Hanson, and J. Widom, "Active database systems," Modern Database Systems, pp.434–456, 1995.
[19] S. Abiteboul, V. Vianu, B.S. Fordham, and Y. Yesha, "Relational transducers for electronic commerce," J. Comput. Syst. Sci., vol.61, no.2, pp.236–269, 2000.
[20] A.J. Bonner, "Workflow, transactions, and datalog," Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp.294–305, 1999.
[21] M.A. Nieto-Santisteban, J. Gray, A.S. Szalay, J. Annis, A.R. Thakar, and W. O'Mullane, "When database systems meet the grid," CIDR, pp.154–161, 2005.
[22] http://www.microsoft.com/en-us/sqlserver/default.aspx.
[23] M. Stonebraker, J. Anton, and E.N. Hanson, "Extending a database system with procedures," ACM Trans. Database Syst., vol.12, no.3, pp.350–376, 1987.
[24] M. Stonebraker and G. Kemnitz, "The postgres next generation database management system," Commun. ACM, vol.34, no.10, pp.78–92, 1991.
[25] S. Chaudhuri and K. Shim, "Optimization of queries with user-defined predicates," ACM Trans. Database Syst., vol.24, pp.177–228, 1999.
[26] J.M. Hellerstein and M. Stonebraker, "Predicate migration: Optimizing queries with expensive predicates," Proc. SIGMOD Conf., pp.267–276, 1993.
[27] M. Corporation, "Table-valued user-defined functions," tech. rep., Microsoft, 2009.
[28] M. Jaedicke and B. Mitschang, "On parallel processing of aggregate and scalar functions in object-relational dbms," Proc. SIGMOD Conf., pp.379–389, 1998.
[29] M. Jaedicke and B. Mitschang, "User-defined table operators: Enhancing extensibility for ordbms," VLDB, pp.494–505, 1999.
[30] E. Friedman, P. Pawlowski, and J. Cieslewicz, "Sql/mapreduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions," VLDB Endow., pp.1402–1413, 2009.
[31] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "Scope: Easy and efficient parallel processing of massive data sets," PVLDB, vol.1, no.2, pp.1265–1276, 2008.
[32] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, "Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads," VLDB, vol.2, no.1, pp.922–933, 2009.
[33] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong, "Tenzing a sql implementation on the mapreduce framework," PVLDB, vol.4, no.12, pp.1318–1327, 2011.
[34] "SQLite." http://www.sqlite.org/.
[35] "TPC-H." http://www.tpc.org/tpch.
[36] "Genia sentence splitter." https://github.com/TsujiiLaboratory/geniass.
[37] "Running tpc-h queries on hive. web page." https://issues.apache.org/jira/browse/HIVE-600, 2009.
[38] "CPU Burn-in." http://users.bigpond.net.au/CPUburn/.
[39] M. Miwa, R. Satre, J.D. Kim, and J. Tsujii, "Event extraction with complex event classification using rich features," JBCB, pp.131–146, 2010.
[40] Bjorne, F. Ginter, S. Pyysalo, J. Tsujii, and T. Salakoski, "Complex event extraction at pubmed scale," Bioinformatics, vol.26, no.12, pp.382–390, 2010.
[41] "Hadoop streaming." http://hadoop.apache.org/common/docs/r0.15.2/streaming.html.
[42] D. FA, "Searching medline via pubmed," Clin Lab Sci, 2008.

Ting Chen born in 1986. She received the B.S. degree in Computer Science from Beijing Language and Culture University in 2007 and received the M.S. degree in Computer Science from Beihang University in 2010. Since 2010, she has been a Ph.D. candidate in Information Science and Technology from the University of Tokyo, Japan. Her current research interests reside in data-intensive computing and parallel database system. E-mail: chenting@eidos.ic.i.u-tokyo.ac.jp

Kenjiro Taura born in 1969. He is a associate professor at Department of Information and Communication Engineering, University of Tokyo. He received his B.S., M.S., and DSc degrees from University of Tokyo in 1992, 1994, and 1997. His major research interests include parallel/distributed computing and programming languages. He is a member of ACM and IEEE. E-mail: tau@eidos.ic.i.u-tokyo.ac.jp