Vol.2012-HPC-135 No.23
2012/8/2

Data-Intensive Text Processing Workflows with a Parallel Database System

Ting Chen[1,a)] Kenjiro Taura[1,b)]

Abstract: This paper studies three real-world text processing workflows and tries to accomplish them using a parallel database system called ParaLite as the backend storage instead of files. Database system could be helpful to simplify the description of workflows and reduce a large number of intermediate files. To better support workflows which are typical built out of various independently developed executable and scripts, ParaLite provides an extended syntax of SQL to embed arbitrary external programs (user-defined executable) into a single SQL query and implements a concept of collective queries. A collective queries is an SQL query whose results are distributed to multiple clients collectively and then processed by them in parallel, using user-defined executable. With ParaLite, these three workflows are easy to be performed and experimental results show that ParaLite has achieved good scalability in terms of the parallelization of UDXes with the increase of computing clients.

Keywords: Workflow, Parallel Database System, User-Defined Executable, Collective Query

1. Introduction

Workflow [1] has become one of the most important and necessary tool for data-intensive applications since it facilities the composition of individual executable or binary scripts, making it easier for domain experts to focus on their research rather than computation management. Workflow is widely used to process a large number of text, especially in the discipline of Nature Language Processing (NLP). Common tasks in NLP are to extract the features of data (aspects of the representations of the data) some of which may be superficial, such as the words and sequences of words themselves while others are more complex, such as both the grammatical and semantic relationship between words. To accomplish these tasks with workflows, easy description and parallel processing of tasks readily accessible to NLP scientists. Many system are proposed to execute workflows, including GXP Make [2], Swift [3], Pegasus [4] and Taverna [5].

A workflow is generally a DAG with a set of jobs and their dependencies. Each job is a typical existing binary or executable. For example, NLP workflows 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 [6], [7]) are third-party components that received a considerable amount of development efforts in the community. Others are adhoc scripts. Either way, they almost always work on text data which is usually stored in and transferred through files. Each job in the workflow is fed with input files and produces at least one output file which becomes the input of a follow-up job. A file-based workflow is always very complex with many jobs due to its low-level description. Besides, to process file in parallel, a big file is splitted into small files, thus, leading to a large number of intermediate files. Users always complain that it is troublesome for them to manage thousands of files, and also inconvenient to extract useful information from so many files. In addition, Using files to store data may lead to poor performance for the execution of a workflow. Since creating index for data stored in files is usually impossible, it is very tedious and inefficient to select a subset of data which requires a full scan to files.

Recently, MapReduce [8] 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 [9] 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 [10] or a whole workflow composition is created as MaprRduce style [11]. 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. Data is stored in the parallel file system HDFS which makes indexing data impossible too. Moreover, a workflow typically consists of many third-party binaries and ad-hoc scripts written in a variety of language, but integrating them in Hadoop is not straightforward. Hadoop Streaming API supports external programs but only to a limited extent; it generally does not bring to them as much flexibility as to the native Hadoop programs.

With consideration of making workflows simple and efficient, a nature idea is to build workflows on top of the parallel database system[12]. On the one hand, with expressive SQL, database systems can simplify the description of workflows. For instance, SQL with a proper support of user-defined functions and reduc-

1 The University of Tokyo, Bunkyo-ku, Tokyo 101–0062, Japan
a) chenting@eidos.ic.i.u-tokyo.ac.jp
b) tau@eidos.ic.i.u-tokyo.ac.jp

tions can express many data processing tasks much more elegantly and easily than MapReduce [13] [14]. Good expressive ability of SQL also leads to many proposals on hybrids of relational databases and MapReduce [15] [16] [17]. On the other hand, database are efficient for processing relational data in ways expressible in SQL due to data indexing and sophisticated query optimization [13] [14]. However, to support workflows better, 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. 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 on minimizing IO costs and data transfers inside the execution of an SQL query [12], 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. Moreover, it prohibits us to take advantage of co-allocating computing clients with data.

Therefore, we proposed a lightweight parallel database system called ”ParaLite” [18]. It provides an extension to database system to support the integration of external programs ” User-Defined Executable (UDX)” into a single SQL query. It implemented a concept of collective queries that facilitate description of workflows by making data parallel execution of UDX on big data easy and streamlined and also provide the workflow developers with a familiar and powerful language SQL, for flexible data filtering and stereotypical data processing tasks. In this paper, we demonstrate several real-world text-processing workflows using ParaLite. Section 2 firstly gives a brief review of ParaLite and then section 3 introduces three workflows and the development with ParaLite. Some experimental results are shown in section 4. Finally, conclusion and future works are introduced in section 5.

2. Review of ParaLite

ParaLite is a shared-nothing parallel database system based on a popular single-node database SQLite [19].

2.1 Syntax of User-Defined Executable

The syntax of UDX is shown in Fig 1.

```
select a1, F(a2) as b2, a3 from t where ...
with F = "cmd_line"
collective by id1
```

Fig. 1 The syntax of a collective query

ParaLite extends SQL to support the definition of User-Defined Executable (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. An UDX can work on and produce arbitrary columns while UDF in traditional database system can only support one column input and output. Moreover, to avoid registration to system before the query is executed, ParaLite allows users to define the UDX within the query using `WITH` clause. It starts from a command line followed by data format options, such as, `input`, `input_row_delimiter`, `output` and `out_row_delimiter`. These options not only provide more flexible input and outputs of UDXes than traditional UDFs, but also allow data to be processed in bulk. However, a single invocation of traditional UDF by a single row produces a single result record, which leads to a significant performance degradation if the UDF has much start-up overhead which commonly happens in the NLP applications.

2.2 Examples of UDX

I will take some examples to illustrate the usage of UDXes. The schema of table `data` is :

`data` : | 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 expressed by a simple SQL query:

  ```
  select * from data where text like '%XYZ%'
  ```

  It is also easily performed by a query with shell scrip ”grep” as a UDX:

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

  All data of column `text` retrieved from table `data` is processed by the UDX `grep XYZ` and the filtered data is returned.
- Word Count Task
  Word count, another canonical example of MapReduce, is to calculate the occurrences of words in a big text document. 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 splitted 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) from data
     with F= "awk 'for(i=1;i<=NF;i++) print $i'"
  group by word
  ```

  The nested query is to split the text into words using `awk` script and output words in the format of one word in one line. The occurrences for each word is simply counted by grouping words from the output of the nested query.
- Sentence Split Task
  Splitting documents into sentences is the first step for almost all text-processing applications. It involves an third-party binary developed by domain researchers.

  ```
  create table sentence as
     select F(text) as (SID, sentence) from data
     with F="geniass" output_col_delimiter '\t'
  ```

  The sentence splitter `geniass` takes `text`, outputs sentences with their identification separated by '\t'. So in the definition of `geniass`, an user should specify the `output_col_delimiter` to let database system know how to convert data into the right schema. For this query, the output of `geniass` is stored into a table "sentence" with two columns "`| SID | sentence |`".

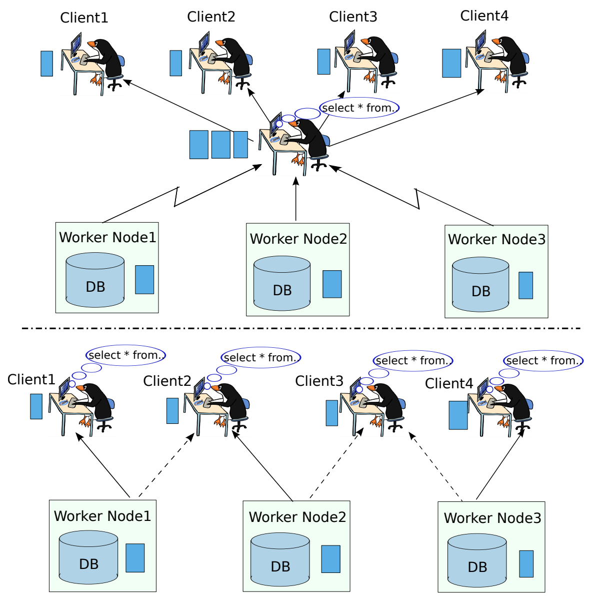

Fig. 2 The support of parallelization of UDX in a collective query

2.3 Collective Query

The main goal of collective query is to parallelize the execution of UDX. Multiple computing clients issue the same query with the same collective ID specified by the `collective by` clause, get data collectively directly from data nodes and perform UDXes on data in parallel. This feature makes it differ a lot from traditional database system which only allows a single client to issue a query whose results are all returned to the client. When the computation is too complex to use an UDF with a SQL query, then client has to distributes all result data to external programs for parallel execution as shown in the top of Fig 2. In this case, the single query issuer can easily become a bottleneck. Moreover, it prohibits us to take advantage of co-allocating computing clients with data. It is a big waste of time and bandwidth if computing clients and data are located on the same node. On the other hand, a collective query transparently parallelizes UDXes across multiple clients. Data transfer between database processes and client processes executing UDX is optimized taking data locality and load balance among all computing clients into account (see the bottom of Fig 2). In the best case, a computing client is running on each database node and data are already balanced among them, no data are transferred between nodes. In addition, collective queries allow the separation of data nodes and client nodes on which UDXes-related software is installed. The degree of parallelization is controlled by the number of clients and new clients can join in the group to get a part of data during the execution but before all data is distributed to clients.

A simple and straightforward flow to process a group of collective query is as follows: First of all, data are partitioned and stored in database managed by SQLite on each data node. A master node parses received collective queries and translates them into a execution plan. A execution plan is a job graph composed of relational operators such as `join` and `group by`, sub-query executed by SQLite and UDX. Once the execution plan is created, jobs of operators and sub-queries are scheduled into data nodes to be executed while UDXes are executed by computing clients. The query is completed successfully after all jobs are finished.

3. Text-Processing Applications

With the increase of text data, it becomes more and more necessary but difficult to extract useful information in text-processing applications. In this section, we will introduce three typical text-processing applications either in web analysis or Natural Language Processing. They are developed by workflow system called GXP Make [2] which use `make` to describe the workflow thanks to its nature feature in describing dependencies between targets and prerequisites. GXP Make supports all the features of GNU make but extends its platforms from single node systems to clusters, clouds, supercomputers, and distributed systems. In the past, files are used to store and transfer data and the backend storage system is NFS. To solve the problems with file (mentioned in the first section), we will use ParaLite, the workflow targeted parallel database system, as the backend storage system in GXP Make. A significant feature of ParaLite in a workflow is that a single SQL query can do everything including file split, data reduce and parallelization of programs.

a, First, data is naturally partitioned on many data nodes eliminating explicit big file split.

b, Then, SQL query with proper reduction can automatically spawns processes to handle data globally based on efficient execution plan.

c, Finally, with support of UDX and collective query, ParaLite provides transparently parallelization of UDX across multiple computing clients with optimized data transfer from data sources to clients.

3.1 Japanese Word Count

`Japanese Word Count` is to calculate 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 topic in Twitter, popular products in on-line stores and attracting customs in different counties.

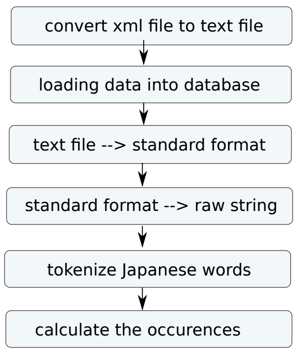
convert xml file to text file
loading data into database
text file --> standard format
standard format --> raw string
tokenize Japanese words
calculate the occurences

Fig. 3 The workflow of Japanese Word Count

This task reads raw data crawled from main Japanese websites, extracts Japanese words and calculates their occurrences. The main steps of the workflow is shown in Fig 3. With ParaLite, each task in the workflow is expressed by either a collective query with a third-party executable or a SQL query for general purpose and all intermediate data is stored in ParaLite system. The detail of the description in the Makefile is shown as follows:

```
all: $(WORD_COUNT)

$(HTML_FILE) : $(INPUT_FILES)
        ./readcrawl.py $< > $@
$(HTML_TABLE) : $(HTML_FILE)
        paralite DB "create table html(rowid, rar,url,rbt
            ,res,tim,req,sta,hdr,con) on DATA_NODE"
        paralite DB ".import $< html -column_separator
            ::::: -row_separator =====" 
$(STANDARD_FORMAT) : $(HTML_TABLE)
        paralite DB "create table standard_format as
            select S(con) as sf from html with S = "
            html2sf.py" input_row_delimiter '====='
            output_row_delimiter '=====' on DATA_NODE
            collective by 1"
$(RAW_STRING) : $(STANDARD_FORMAT)
        paralite DB "create table raw_string as select R(
            sf) as rs from standard_format with R ="
            sf2rs.py" input_row_delimiter '====='
            output_row_delimiter '=====' on DATA_NODE
            collective by 2"
$(TOKENS) : $(RAW_STRING)
        paralite DB "create table tokens as select T(rs)
            as word from raw_string with T = "juman" on
           DATA_NODE partition by word collective by 3"
$(WORD_COUNT) : $(TOKENS)
        paralite DB "create table word_count as select
            word, count(*) as frequency from tokens
            group by word on DATA_NODE"
```

Note that, while all tasks are easy to be performed using SQL query, they are general difficult or inefficient to be completed with files especially for the final word count task as shown in Fig 4. A big file should be splitted into a several small files each of which is executed by different programs, leading to three problems:

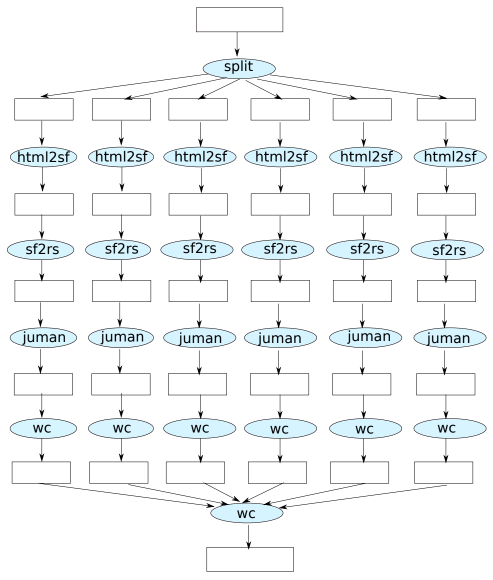

Fig. 4 Japanese Word Count with Files

(1) every program has to parse each record to get desired columns.

(2) programs may not be scheduled to the nodes who have target data by workflow systems. For example, the performance should be degraded when `sf2rs` programs are scheduled on nodes [4-6] while all output of `html2sf` are stored in nodes [1-3].

(3) when word count is performed, all data are required to be reduced into one node and be counted globally. Of course, users could use some sophisticated reduce methods such as MapReduce, but they are obviously much complex than only a SQL query.

3.2 Sentence-Chunking Problem

The first step of extracting concepts and relations within statements across a large text is to chunking sentences into its constituent phrases which represent N-gram in a sentence. Significant chunks would typically correspond to semantic units such as named entities (proteins, genes, diseases) or relations [20] [21].

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. This model assumes that every sentence is generated by randomly sampling from a dictionary which contains phrases (N-gram). Due to the finite number of methods to chunked a sentence into phrases, this model assigns a likelihood value to each phrase based on its frequency, and chooses the chunking with the maximum likelihood. This model requires multiple iterations to maximize the corpus likelihood.

The workflow of Sentence Chunking is shown in Fig 5.

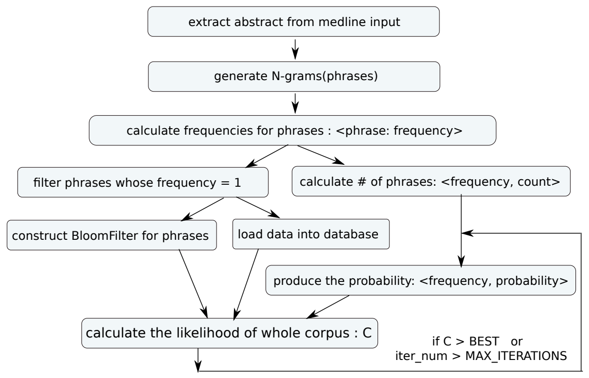
extract abstract from medline input
generate N-grams(phrases)
calculate frequencies for phrases : <phrase: frequency>
filter phrases whose frequency = 1
calculate # of phrases: <frequency, count>
construct BloomFilter for phrases
load data into database
produce the probability: <frequency, probability>
calculate the likelihood of whole corpus : C
if C > BEST or
iter_num > MAX_ITERATIONS

Fig. 5 The workflow of Sentence Chunking

This workflow is as always expressed with several SQL queries using ParaLite. Since expressing iteration is not easy in Makefile, so we split the whole Makefile into two parts and integrate them in a shell script. The first two features of ParaLite are especially important in this workflow since data split and reduce occurs alternately.

```
ALL: $(FREQUENCY_COUNT) $(BLOOMFILTER) $(SQDB)

$(ABST_TXT) : $(PUBMED_XML_GZ)
        zcat $^ | xml2text '###' > $@
        paralite DB "create table abst_txt(abstract) on
            DATA_NODE"
        paralite DB ".import $@ abst_txt"
$(ABST_SS) : $(ABST_TXT)
        paralite DB "create table abst_ss as select G(F(
            abstract)) as sentence from abst_txt with G=
            "tokenize /dev/stdin" F="geniass INPUT
            OUTPUT" input 'INPUT' output 'OUTPUT)' on
            DATA_NODE collective by 1"
$(N_GRAM) : $(ABST_SS)
        paralite DB "create table n_gram as select G(
            sentence) as phrase from abst_ss with G="
            n_gram_splitter MAX_LENGTH" on DATA_NODE
            partition by phrase collective by 1"
$(PHRASE_FREQUENCY) : $(N_GRAM)
```

```
        paralite DB "create table phrase_frequency as
             select phrase, count(*) as frequency from
             n_gram group by phrase on DATA_NODE"
$(SMALL_PHRASE_FREQUENCY) : $(PHRASE_FREQUENCY)
        paralite DB "select phrase, frequency from
             phrase_frequency where frequency > '1'"
$(SQDB) : $(SMALL_PHRASE_FREQUENCY)
        sqlite3 SQDB "create table phrase_frequency (
             phrase varchar(100), frequency int)"
        sqlite3 SQDB ".import $^ phrase_frequency"
$(BLOOMFILTER) : $(SMALL_PHRASE_FREQUENCY)
        ./bf_producer SIZE NUN_HASH $^ \| $@
$(FREQUENCY_COUNT) : $(PHRASE_FREQUENCY)
        paralite DB "select frequency, count(*) from
             phrase_frequency group by frequency"
$(FREQUENCY_PROBABILITY) : $(FREQUENCY_COUNT)
        ./prob_producer $^ $@ \|
$(SENTENCE_LIKELIHOOD) : $(FREQUENCY_PROBABILITY)
        paralite DB "create table sentence_likelihood as
             select F(sentence) as likelihood from
             abst_ss with F=\"likelihood_prod BLOOMFILTER
              SIZE NHASH CACHE_SIZE SQDB $^ | \" on
            DATA_NODE collective by 1"
$(CORPUS_LIKELIHOOD) : $(SENTENCE_LIKELIHOOD)
        paralite DB "select sum(likelihood) from
             sentence_likelihood"
```

3.3 Event-Recognition

`Event-Recognition` [22] [23] 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 6.

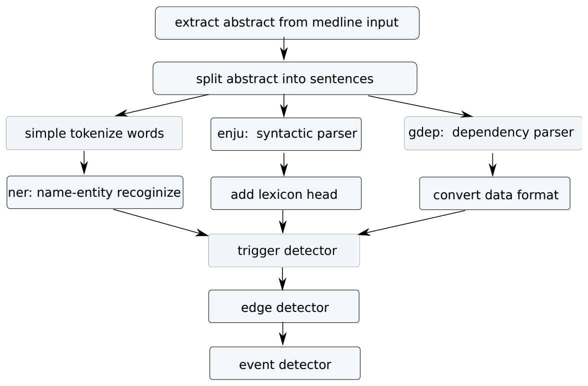

Fig. 6 The workflow of Event-Recognition Application

This workflow takes a collection of data from the MEDLINE database of journal description [24] [25]. It first extracts the abstract of bio-medical related abstract and title, then processes each sentence by different programs to find bio-medical entities and dependencies of or between words. Finally, event recognizer tries to find complex event among entities using all the result for each sentence. ParaLite accomplishes the workflow simply by several collective queries with UDXes and a join SQL to get all result for a single sentence. To perform this in a workflow with files, a common method is to define file name in advance to know the location of sentences. This is really troublesome and inefficient.

```
ALL: $(ENJU_SO)

$(ABST_TXT) : $(PUBMED_XML_GZ)
        zcat $^ | xml2text '###' > $@
        paralite DB "create table abst_txt(PMID, abstract
             ) on DATA_NODE"
        paralite DB ".import $@ abst_txt"
$(ABST_SS) : $(ABST_TXT)
        paralite DB "create table abst_ss as select F(
            PMID, abstract) as (SID, sentence) from
             abst_txt with F="geniass"
             input_col_delimiter '###'
             output_col_delimiter '###' on DATA_NODE
             partition by SID collective by 1"
$(ENJU_SO) : $(ABST_TXT)
        paralite DB "create table enju_so as select SID,
            A(E(sentence)) as enju from abst_ss with E="
             enju" A="add_lex_head" output_row_delimiter
             '=#=' on DATA_NODE partition by SID
             collective by 2"
$(GDEP_OUT) : $(ABST_SS)
        paralite DB "create table ksdep_out as select SID
             , F(sentence) as pre_ksdep from abst_ss with
              F="gdep" output_row_delimiter EMPTY_LINE on
              DATA_NODE partition by SID collective by 3"
$(GDEP_SO) : $(GDEP_OUT)
        paralite DB "create table ksdep_so as select SID,
              F(sentence, pre_ksdep) as ksdep from
             abst_ss, ksdep_out where abst_ss.SID =
             ksdep_out.SID with F="dep2so -g"
             input_row_delimiter EMPTY_LINE
             input_col_delimiter '###'
             output_row_delimiter EMPTY_LINE on DATA_NODE
              partition by SID collective by 4"
$(GENE_NE_TEMP) : $(ABST_SS)
        paralite DB "create table gene_ne_temp as select
             SID, G(T(sentence)) as pre_gene from abst_ss
              with T="gene_ner_tokenizer" G="
             gent_ner_gtag" on DATA_NODE partition by SID
              collective by 5"
$(GENE_NE_SO) : $(GENE_NE_TEMP)
        paralite DB "create table gene_ne_so as select
             SID, F(N(M(sentence, pre_gene)))as gene from
              abst_ss, gene_ne_temp where abst_ss.SID =
             gene_ne_temp.SID with M="pos_marker" N="
             dict_matcher" F="ner" on DATA_NODE partition
              by SID collective by 6"
$(EVENT_SO) : $(GENE_NE_SO) $(GDEP_SO) $(ENJU_SO) $(
     ABST_SS)
        paralite DB "create table event_so as select F(
             SID, sentence, enju, ksdep, gene) as (SID,
             event) from abst_ss, enju_so, ksdep_so,
             gene_ne_so where abst_ss.SID = enju_so.SID
             and abst_ss.SID = ksdep_so.SID and abst_ss.
             SID = gene_ne_so.SID with F="all_detectors"
             on DATA_NODE collective by 6"
```

4. Evaluations

All experiments are conducted on a 32 nodes cluster. Each node has 2.40 GHz Intel Xeon processor with 8 cores and 24GB RAM. SQLite 3.7.3 is installed on each node.

4.1 Japanese Word Count

The input for this workflow is 28GB and stored in 32 nodes. Among all tasks, resources consuming tasks are listed below:

- html2sf: Conversion from crawled data to standard format which is an XML-based format developed by Kurohashi's group at Kyoto University to represent a document in a manner that can easily extract necessary pieces such as plain text.
- sf2rs: Extraction of plaintext part, marked with <rawstring> tags in the data with standard format.
- juman: A morphological analyzer for Japanese.

These three tasks are all performed by a collective query. The execution time for them is shown in Fig 7. The number of clients in this experiment is 64 which means that each node has 2 clients.

The program of `html2sf` is very cpu-intensive and cost about 94 minutes while the other three are lightweight programs, consuming only several minutes.

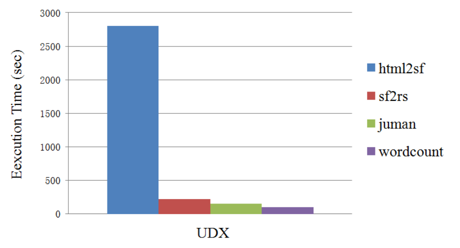

Fig. 7 The execution time of tasks in Japanese Word Count

To verify the mechanism of the parallelization of UDX, we tested the speedup for both cpu-intensive <html2sf> and lightweight UDXes <sf2rs> with the increase of clients as shown in 8. The speedup for <html2sf> is a litter smaller than linear one when the number of clients is up to 128. This is mainly because with the reduction of total execution time, overheads for some operations that cannot be parallelized, such as, data scheduling and storing final result data into database, have become bigger part in the total time than before. So the ideal speed up could not be achieved. However, The speedup of <sf2rs> exceeds the linear one a litter. One possibility is that the execution time of <sf2rs> is not the linear function of the size of data. Then when the data size is reduced to be half for each client, the execution time may reduce more than half.

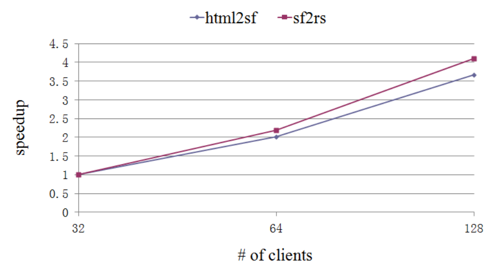

Fig. 8 The speedup of two UDXes in Japanese Word Count

4.2 Sentence-Chunking Application

The experiments are performed with 7 GB data from MEDLINE database. We extract the 1 GB journal abstract which produces about 50 GB phrases. Time consuming tasks in this workflow are shown below:

- phrase-generator: Generate N-grams (phrases) for each sentence.
- phrase-frequency: Calculate the frequencies of phrases.
- db-load: store all phrases whose frequencies are larger than 1 into a SQLite database which is located in a shared file system.
- bf-producer: construct BloomFilter (BF) [26] for phrases whose frequencies are larger than 1. The BF is created to reduce the latency of a look up to the SQLite database. For example, if we want to query the frequency for word `A`, firstly, we check if `A` is in the BF. If not, it means the frequency of this word is 1. Otherwise, another query to the SQLite database is required to get the exactly frequency. Since the latency to look up in a BF is much smaller than SQLite and phrases whose frequencies equal to 1 take a great part in the whole phrases, BF provides a significant improvement of performance.
- frequency-count: Calculate the number of phrases group by their frequencies.
- likelihood-prod: Calculate the likelihood of each sentence.

First of all, Fig 9 shows the execution time for each task with 32 data nodes and 64 clients. We can see that the calculation of sentence likelihood is the most time-consuming task even with the optimization of BloomFilter. For a sentence with n words, there are 2^{n-1} methods to chunking the sentence and all of them should be calculated.

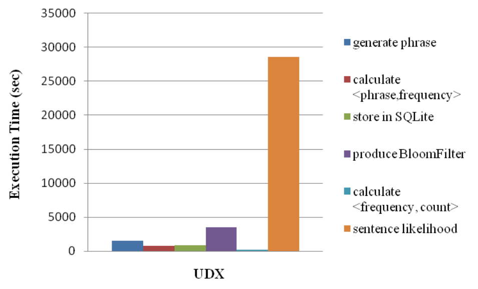

Fig. 9 The execution time of tasks in Sentence-Chunking

To verify the scalability of this task, we performed another experiment with 128 clients and the result is shown in Fig 10. When the number of clients increase up to 128, the elapse time of this task reduces only one fifth, that is, the speedup for this program is 1.2. One possibility for the poor speedup is bad performance of SQLite in support of concurrent read/write. The SQLite database is stored in NFS and 128 clients read it very frequently. The lock mechanism of SQLite may lead the read sequentially. However, this is not verified and I will do it later.

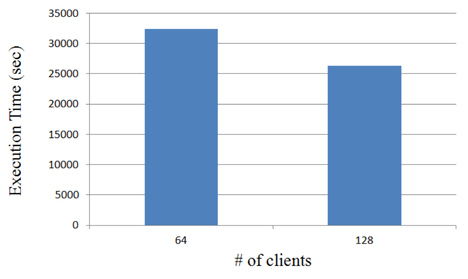

Fig. 10 The execution time of sentence-likelihood with increase of clients

4.3 Event-Recognition

We use the same MEDLINE data set with Sentence-Chunking application. Main tasks in this workflow are:

- enju: a HPSG parser which can effectively analyze syntactic/semantic structures of English sentences and provide a user with phrase structures and predicate-argument structures.
- ner: recognition for bio-medical entities such as gene and protein.

- gdep: a dependency parser for biomedical text.

Fig 11 gives the execution time for the three time-consuming tasks above. While `ner` and `gdep` cost about one hour, `enju` is much heaver than them and cost about 24 hours.

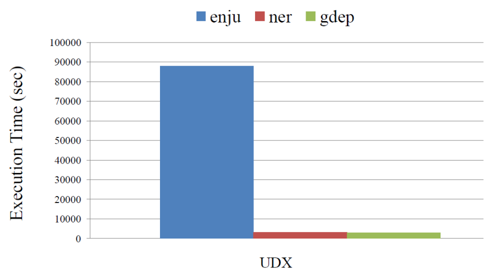

Fig. 11 The execution time of tasks in Event-Recognition

Due to the time limitation, we take `gdep` as an example to show the scalability of ParaLite. From Fig 12, we can see that the execution time decreases with the increase of clients and the speedup exceeds the linear one a little. The reason for this may be the same with that for program `sf2rs` in the workflow of Japanese Word Count.

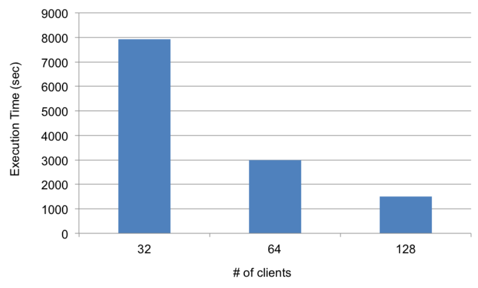

Fig. 12 The execution time of gdep with increase of clients

5. Conclusion

We proposed a concept of `collective query` which embeds arbitrary external programs (user-defined executable) into a single SQL query and allows multiple clients to perform them in parallel efficiently. This concept is implemented in `ParaLite`, a lightweight shared-nothing parallel database system. The intended applications are data intensive workflows, typically built out of various independently developed executable and scripts. We applied ParaLite into three real-world typical text-processing workflows. ParaLite are very expressive to describe the workflows with a collective query for a task. Finally, experimental results showed that ParaLite has achieved good scalability in terms of the parallelization of UDXes with the increase of the number of clients. In future, we will compare ParaLite with several popular approaches performing workflows, such as, Hive and Hadoop.

Acknowledgments This research is partially supported by the project for "R&D on cloud service infrastructure for recovering wide-area disaster (Reliable cloud service platform technology)" of the Ministry of Internal Affairs and Communications.

References

[1] Ewa Deelman, Dennis Gannon, Matthew S. Shields, and Ian Taylor. Workflows and e-science: An overview of workflow system features and capabilities. *Future Generation Comp. Syst.*, 25(5):528–540, 2009.

[2] Kenjiro Taura, Takuya Matsuzaki, Makoto Miwa, et al. Design and implementation of gxp make – a workflow system based on make. In *e-Science 2010 conference (eScience2010)*, 2010.

[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*, pages 199–206, 2007.

[4] Deelman Ewa, Singh Gurmeet, Su Mei-Hui, Blythe, James Gil, and others. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal*, 13(3):219–237, July 2005.

[5] Thomas M. Oinn, Matthew Addis, Justin Ferris, Darren Marvin, et al. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3405–3054, 2004.

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

[7] Cabocha yet another japanese dependency structure :. http://code.google.com/p/cabocha.

[8] J. Dean and S. Ghemawat. Mapreduce:simplified data processing on large clusters. In *OSDI'04*, pages 137–150, 2004.

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

[10] Phuong Nguyen and Milton Halem. A mapreduce workflow system for architecting scientific data intensive applications. In *SECLOUD '11*, pages 57–63, 2011.

[11] Xubo Fei, Shiyong Lu, and Cui Lin. A mapreduce-enabled scientific workflow composition framework. In *ICWS '09*, pages 663–670, 2009.

[12] D.J. DeWitt and J. Gray. Parallel database systems: the future of high-performance database systems. *Commun*, 35(6):85–98, 1992.

[13] Andrew Pavlo, Erik Paulson, and Alexander Rasin. A comparison of approaches to large-scale data analysis. In *SIGMOD 09: Proceedings of the 2009 ACM SIGMOD International Conference*, pages 165–178, 2009.

[14] Michael Stonebraker, Daniel Abadi, David J. DeWitt, and Sam Madden. Mapreduce and parallel dbmss: friends or foes? *Commun*, 53(1):64–71, 2010.

[15] Ashish Thusoo, Joydeep Sen Sarma, et al. Hive - a warehousing solution over a map-reduce framework. In *Proceedings of VLDB Endow*, pages 1626–1629, 2009.

[16] Azza Abouzeid, Kamil Bajda-Pawlikowski, et al. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proceedings of VLDB*, 2(1):922–933, 2009.

[17] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. In *International Conference on Management of Data - SIGMOD*, pages 1099–1110, 2008.

[18] Ting Chen and Kenjiro Taura. 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)*, pages 474–481, 2012.

[19] Sqlite :. http://www.sqlite.org.

[20] Sharon Goldwater, Thomas L. Griffiths, and Mark Johnson. Contextual dependencies in unsupervised word segmentation. In *Meeting of the Association for Computational Linguistics - ACL*, 2006.

[21] Chiara Sabatti and Kenneth Lange. Genomewide motif identification using a dictionary model. *Proceedings of The IEEE*, 90(11):1803–1810, 2002.

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

[23] Bjorne, F. Ginter, S. Pyysalo, J. Tsujii, and T. Salakoski. Complex event extraction at pubmed scale. *Bioinformatics*, 26(12):382–390, 2010.

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

[25] L. E. Notter. Medlinenewest service in the medical information network. In *Nursing Research*, 1972.

[26] B.Bloom. Space/time tradeoffs in the hash coding with allowable errors. *CACM*, 13(7):422–426, 1970.