Vol.2011-HPC-130 No.28
2011/7/28

Supporting Collective Queries in Database System for Data-intensive Workflow Applications

Ting Chen[*1] and Kenjiro Taura[*1]

Workflow system becomes the most important and necessary tool for data-intensive applications, enabling the composition and execution of complex analysis on distributed resources. Due to simple and effective query expression and high performance on query processing of parallel DBMS, we proposed a database system for workflows in which task in workflows can be expressed as a SQL query and processed by our systems. To be effectively used in the support of workflows, our database system supports collective queries in which multiple clients collectively issue a set of queries operated on a collection of data set as if a large data-scanning query is sent. Collective queries are integrated into a single SQL query which will be transformed into a logical plan composed of relational operator and sub-query and parallelized into multiple underlying resources. Finally result data is distributed into multiple clients in a coordinated way which can reduce a lot of data communication cost. Experiments show that our current database system is feasible and scalable to some extends.

1. Introduction

The world of science has experienced paradigm of experimental science, theoretical science, computational science and now come to the forth paradigm *data-intensive science* termed by Jim Gray[1]. This new paradigm introduces a wave of data-driver approaches to data captured by instruments or generated by simulations before being processed by software. Researches in many disciplines such as environment, weather, and medicine, are considered as typical data-intensive applications. A typical situation is that researchers are collecting data either from instruments or sensors, or from running simulations, then pretty soon they end up with a large amount of data (files) and there is no easy way to manage or analyze these data.

There is very wide agreement that data-intensive methods are key to data-intensive applications in many disciplines and receive more and more interest[2]. Such methods are expected to play an increasing role in providing evidence for well-informed decisions and policies. Therefore, they are of great scientific and social importance. There are many approaches to support data-intensive methods, such as workflow technologies[3][4][5], augmented familiar tools and Map-Reduce framework[6] over large-scale distributed systems.

Scientific workflow systems have become the most important and necessary tool for many applications, enabling the composition and execution of complex analysis on distributed resources. To make it easier for scientists to focus on their research rather than computation management, scientific workflow systems aim at automating a cycle of moving data to a supercomputer for analysis , simulation, launching the computations and managing the storage of the output results. A Workflow refers to the activity of defining the sequence of tasks needed to manage a business or computational science or engineering process. A Workflow Instance is a specific instantiation of a workflow for a particular problem and includes the definition of input data. To take advantage of workflow system, firstly a composition of the workflow is designed which specifically is tasks with their dependencies through a number of different means, such as text and graphic. Then, the workflow (tasks) is mapped to underlying resources. Each task may perform an independent computation to the data or a similarly complex computation that involves dividing the data set into smaller ones in order to support concurrent processing. So finally task is executed based on some semantic rules on mapped resource and the result data comes out after execution of the workflow.

Another attractive method for data-intensive applications is MapReduce Model[6]. It provides a simple model through which users can express relatively sophisticated distributed programs while hiding the messy details of parallelization, fault-tolerance, data distribution and load balancing. A user-defined map function is applied to each key/value pair of the input, which produces a set of intermediate key/value pairs, and then a user-defined reduce function is used to aggregate all the values with the same key. For data-intensive applications, lots of researchers focus on constructing map-reduce enabled workflow systems in which a heavy task can be expressed as Map and Ruduce jobs[7] or a whole

[*1] Presently with the University of Tokyo

workflow composition is created as mapreduce style[8].

Given this, a natural question is proposed that why don't take advantage of parallel DBMS. Parallel database systems[9] have been commercially available for nearly two decades, and there are now about a dozen in the marketplace, including Teradata, Microsoft SQL Server, Vertica[10], DB2 (via the Database Partitioning Feature), and Oracle (via Exadata). They are robust, high performance computing platforms. Like MapReduce, they provide a high-level programming environment and parallelize readily. Therefore, we would like to introduce them into workflow systems to support data-intensive applications better.

First of all, during the task execution phase, users have to specify the computation on the data. It is a great burden for them if they are not very familiar with computer technology and the structure of data. Using a kind of high-level language such as SQL which is a very general and powerful to express user's requirement is a good alternate. Secondly, parallel DBMSs have very good performance on query processing. Parallel DBMS and Map-Reduce framework are always considered as two main approaches to perform large-scale data processing. There are some publications[11][12] to compare the two approaches and the result shows that parallel DBMSs have much better performance on query processing due to data scheme, index and flexible query optimization mechanism.

With so many good properties of database system, we do not hesitate to use it. People may ask that, however, there must be many sophistcated parallel DBMSs, why not using them directly? A significate reason is that all the current parallel DBMSs are not constructed for workflow systems, therefore, we cannot customize special functions to support workflow systems well. Moreover, current parallel DBMSs have several common drawbacks. (1) it costs too much time to load data since data has to be adhered some schema and stored into pre-defined tables. (2) They can only support transaction-level fault recovery, which means that during the query processing, no matter a error occurs in whichever stage, the whole query will be re-processed. Finally, since parallel DBMSs are commercial, they are too expensive for some users.

Therefore, our goal is to make a database system to be widely and effectively used in the support of workflow systems. To achieve it, some key issues should be addressed such as collective query supporting, complex computation expressive, fast data loading. We focus on how to support collective query in this paper. In section 2, we introduce what is collective query and why it is necessary. System architecture is proposed in section 3 to explain how each component works. Then some experiments are shown to verify our system in section 4, only query processing performance experiments are done and others are ongoing. Finally, conclusion and future works are introduced in section 5.

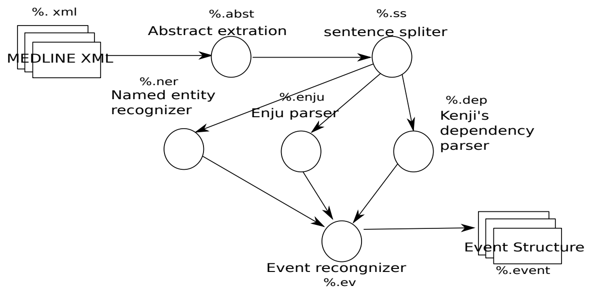


Fig. 1 Simple workflow of event recognition application

2. Collective Query

In this section, I will explain what is collective query and why it is necessary in workflow systems using a NLP (Natural Language Processing) application which is called event-recognition[13][14]. Its purpose is to extract and classify bio-molecular events mentioned in English texts. Example bio-molecular events of interest are an expression of a certain gene, a phosphorylation of a protein, a binding of several proteins, and a regulation of certain reactions. The entire data set is split into many chunks and the structure of the workflow for each chunk is shown in **Fig. 1**.

In the workflow, there are six dependent tasks. Usually, system checks dependent file for each task . Once it exists, system will start a job to perform the task and schedule it to an available node. Since the entire data set is split into many small ones, assuming N chucks, N jobs actually are started to perform the task. So the task does not care about the exact input file and takes any input file (such as %.xml, %.abst) to perform computation on.

Now we want to use database system to perform the computation. Firstly,

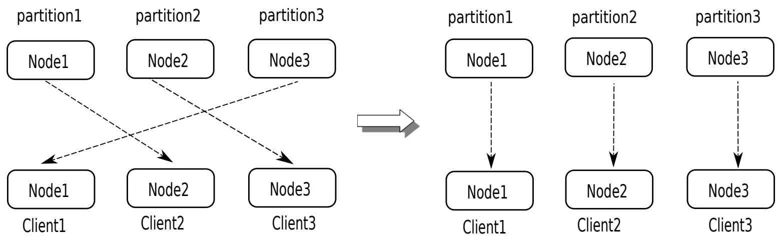


Fig. 2 The benefit of collective query

data is partitioned into N chucks and stored as a table *medline_data* respectively on multi-nodes. For simple task such as an abstract extraction task in Fig. 1, a simple query is used to express the computation shown below. (Of course, some complex computations cannot be expressed by general SQL query. How to support them is one of our future works.)

select abstract from medline_data.

Then N clients have to issue a group of queries at the same time to the database server but each query is applied to a part of data set. The queries operate not on single table, but on entire table collections. These kinds of query are called *collective query* which is collectively issued by multiple clients to perform a large data-scanning task. If collective queries are processed as many small, distinct SQL queries to database server, the performance degrades drastically due to the communication cost under an incoordination pattern. **Fig. 2** shows the problem in detail.

In the left part, since each query is processed as a distinct single query, data has to be transferred among different nodes to meet the data requirement of each client. Note that, in workflows, actually clients only care about if all data is processed and results are returned. They do not care about which exact partition of data they will get. So we can easily collect all quries from these clients and coordinate them so that data transfer can be reduced as much as possible. The ideal situation is shown in the right part of Fig. 2 where no data is transferred among nodes any all.

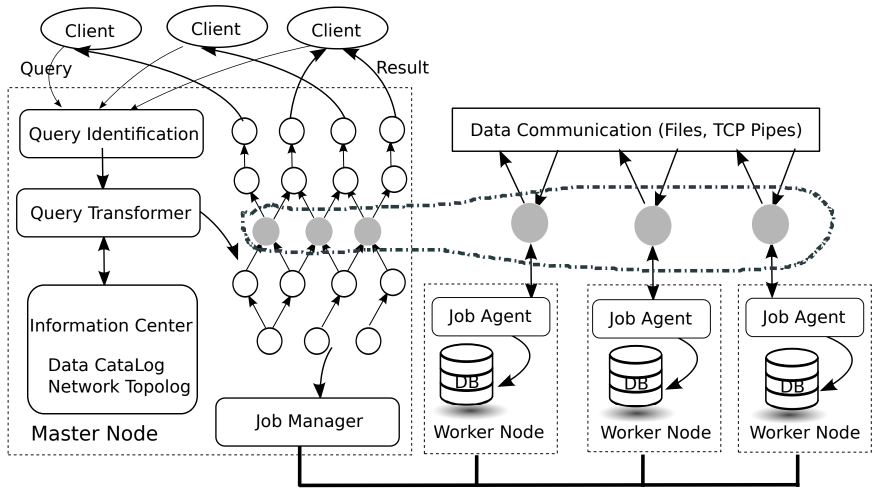


Fig. 3 System architecture

3. Architecture

The key structure to connect each logical component in our system is a logical plan which is organized as a tree (may with a virtual root node). Each node in the logical plan is a job either a relational operator or a sub-query and edges represent data channels. It is a logical computation tree that is automatically mapped onto physical resources by the runtime. At run time each channel is used to transport a finite amount of data. This channel abstraction may has several concrete implementations that use TCP pipes, or files temporarily persisted in a file system.

The architecture of our system is shown in **Fig. 3**. We use classic master/-worker pattern to organize nodes. A master constructs the job tree and schedule jobs across worker nodes coordinately. All data is sent directly between worker nodes and thus the master is only responsible for control decisions and is not a bottleneck for any data transfers.

A simple and straightforward flow to process a (set of) query(es) is as follows: First of all, we use database system (here is SQLite[15]) pre-installed on each worker node to store our data. Data is partitioned into worker nodes based on either hashing fashion, range fashion or round-robin fashion and stored as a predefined table in a database. A master node receives queries from clients and identifies them as collective queries or general ones firstly. If all queries received are collective queries, the master integrate them into a single SQL query which is then translated into a logical plan. After the logical plan is created, a set of dependent jobs are scheduled into worker nodes and will be executed by Job Manager. On each worker node, there is a job agent which is a daemon to start a process to handle jobs and monitor job execution states. A query is completed successfully after all jobs in the logical plan are finished. Finally, results stored on worker nodes are distributed to clients.

3.1 Collective Query Identification

Since collective queries have the same expression but are applied to different partitions, we express them using the modification of SQL query by adding an "partition" clause:

*SELECT * FROM table WHERE 'conditions' PARTITION 'expression'*

Here the expression could be the enumeration of the partition, e.g. '1, 2, 3...' , numeric set notation such as [[1-4]], 'ALL' or 'ANY'. So we use two methods to identify them:

- Identified with a specific string: Each collective query is appended a string 'CT' at the start of the modified SQL. Master node groups all the queries which start with 'CT' and considers them as one single query.
- Identified without a specific string: Note that the expressions of the collective queries are the same except the PARTITION clauses which target at different partitions of data. So a natural idea is to identify them by comparing the substring of queries.

The identification for collective queries should have a prerequisite that all queries are issued at the same time or within a limit period of time.

3.2 Query Transformation

Query Transformer transforms SQL query into a logical plan which is a set of jobs. The logical plan is a DAG of relational operators (such as filter(where), select (project), join, aggregation or sub-query) that operate as iterators: each operator forwards a data tuple to the next operator after processing it.

Considering the set of following collective queries:

*select * from x join y on x.a = y.b and x.a > 10*
join z on x.c = z.d and z.d < 20 PARTITION i

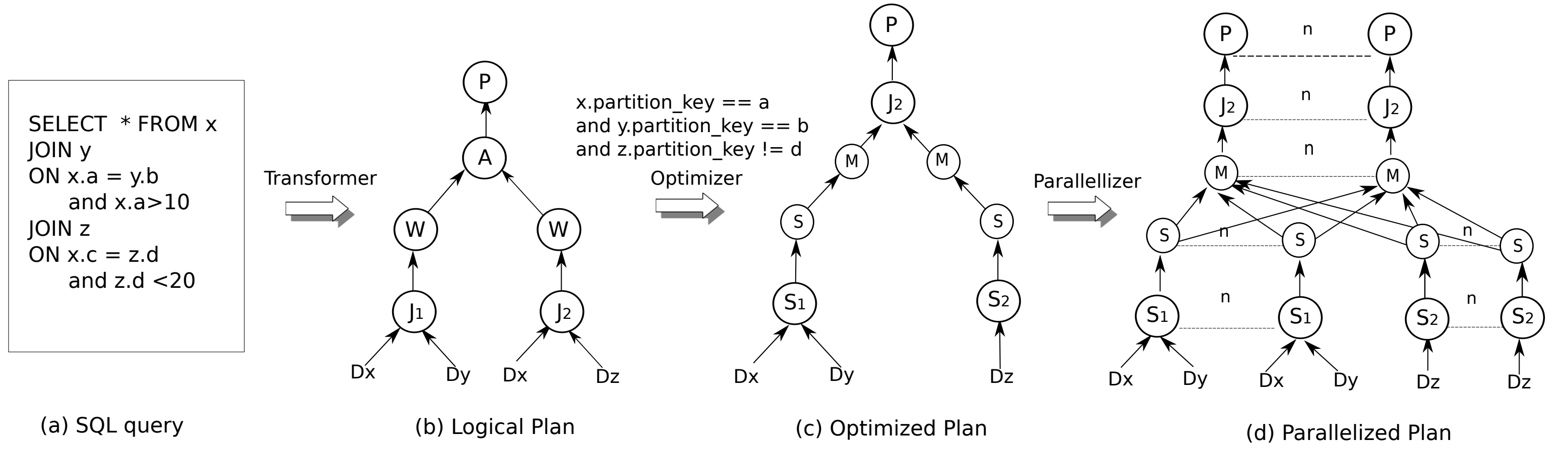
They are firstly merged into a single query without partition clause which are processed along with the following chain:

(1) Syntax Parser: It translates query into a abstract syntax tree consisted of keyword tokens based on modified SQL grammar. At the same time, it analyzes the query semantically through the interaction with information center which stores information about data partitioning, table attributes, resource usage situation and so on.

(2) Logical Planner: It transforms the syntax tree to logical plan composed of relational operators that are executable program or sub-queries. For the example query, the logical plan is shown in (b) graph of **Fig. 4**. The operators *J(oin)1* and *J(oin)2* implement join x, y and x, z and the *W(here)* operators apply the filter condition $x.a > 10$ and $z.d < 20$ to the output of J operators respectively. Then the *A(nd)* operator integrates two out streams from W operators. Actually, the exist of *A(nd)* node is to make our plan constructed easier. It does not perform real process for data and will be eliminated in the optimized phase. Finally, *P(roj)* operator output the satisfied columns of result (all columns here).

(3) Plan Optimizer: It re-constructs the query plan to be more efficient based on rules.

- WHERE-RULE: Filter(where) operator applies some arithmetic calculation to data and gets satisfied data. In most of cases, it reduces a large amount of data. Hence, we push these *where* operators as close as possible to data to reduce the data communication. In Fig. 4 , the WHERE operator in (2) is put before join operator.
- AND-RULE: it integrates *A(nd)* operator's children that perform computation on the same table. In the example, both two join operators have operations on table x, so one join can be a child of another instead of the same table.
- SPLIT-RULE: We cannot directly send the SQL query to each worker to execute due to the *join* and *group by* operators. For example, if the key to be joined is different from the data partitioned key, we surely cannot



```
SELECT  * FROM x
JOIN y
ON x.a = y.b
      and x.a>10
JOIN z
ON x.c = z.d
      and z.d <20
```

Fig. 4 The process of a single SQL query

get right results. So SPLIT-RULE is used for *join* and *group by* operators to repartition data based on appropriate keys. Two special operators are inserted into logical plan before each *join* and *group by* operator. *Splitter* is responsible for splitting the output of an operator into several stream based on the join key while *merger* merges the several input streams before the next operator executes. In the *join1* operator in Fig. 4, the join keys are a for table x and b for table y which are the same with the partition keys of these two tables, so there is no *splitter* and *merger* operators before the *join1*. Oppositely, these two special operators are required for the *join2* operator since the join keys are different from the partition keys of table x and z.

- SUB-SQL-RULE: To take full advantage of SQL server, that has more sophisticated, adaptive or cost-based optimizers on each worker node, we want to re-organize operators as many as possible into a single SUB-SQL to be executed by the SQL server. We retrieve the logical plan bottom to up and put all operators into a single SQL until a splitter operator is encountered. In Fig. 4 SQL1 is $\underline{select\ *\ from\ x\ join\ y\ on\ x.a = y.b\ and\ x.a > 10}$
SQL2 is $\underline{select\ *\ from\ z\ where\ z.d < 20}$

(4) Parallel Planner: We provide partitioned parallelism which is due to the data being partitioned. All input data files are partitioned into n parts. The operator $S1_i$ is a SQL query involved both X and Y tables to be executed by SQLite. It takes the partitioned *Xi* and *Yi* as the input data and then a S(plitter) operator partitions the output of *S1* into n parts called *U1* through *Un* by the c attribute of table x. Similarly, the $S(QL)2_i$ operator is fed by partitioned *Zi* and the output is also partitioned into *P1* to *Pn* by the d attribute of table Z. Then $J2_i$ operator implement join by taking partitioned *Ui* and *Pi* inputs from all *S1* and *S2* nodes and merging them (keyed on $x.c$ and $z.d$ respectively) to produce satisfied records.

At this point, we assume that communication cost is much larger than computation cost during the query processing. So We want to put operators as many as possible on a node which means that all operators in one branch can be executed on one node along the logical plan tree.

3.3 Job Execution

Job Manager schedules jobs to underlying resources and traces their execution state. There is a simple daemon called job agent running on each computer in the cluster that is responsible for creating processes on behalf of the job manager. When an operator is executed on a computer its binary is sent from the job manager to the daemon. The job manager can communicate with the remote job agent and monitor the state of the computation. Later, some fault tolerant mechanisms will be implemented. An operator may be executed multiple times due to failures, and more than one instance of a given vertex may be executing at any given time to solve the skew problem.

At present the job manager performs greedy scheduling which means no matter whether the node is busy or not, a job will be scheduled on mapping resources immediately when all of an operator input files become ready. Here file-based method is used to transfer data between nodes. Each operator outputs data into a single file or a set of file if the data is partitioned by a splitter operator. Operators that rely on these files read data from them and perform computations.

During execution the job manager receives periodic status updates from the Job Agent on. If any operator is re-run more than a set number of times then the entire job is failed. Temporary files are stored in directories managed by the daemon and cleaned up after the job completes.

3.4 Data Distribution to Multiple Clients

After a query is processed successfully, an important question is how to distribute result data to clients (Result Distribution Stage in Fig. 3). We always split data into N streams, so, a more general problem setting should be, given N data $(D_1, D_2, ..., D_n)$ and M clients $(C_1, C_2, ..., C_m)$, determine which data should go to which clients shown in **Fig. 5**. In data intensive applications, client may perform a significate amount of computation, therefore, we should consider not only communication cost but also computation cost.

First of all, let's think about the granularity of data to be distributed. If we just simply map each data D_i to C_j, the performance will be degraded due to the bad load balance if the size of D_i is large. To achieve load balance, we split data on each node into small pieces called data units and only one data unit can be transferred at one time.

A dynamic probing method is used for load balance of the computation on multi-nodes. It detects available nodes (CPU usage is low) and sends data unit to them according to the order:

(1) Allocate data unit on the same node to the available node so that there is no data transfer.

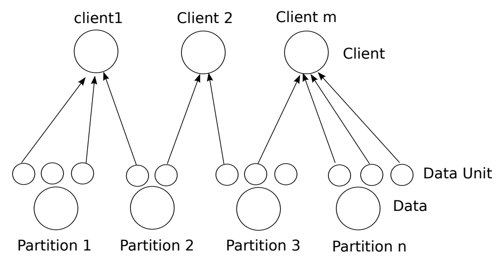


Fig. 5 Distribution of result to multiple clients

(2) Allocate data unit on the closest nodes that don't have clients on to the available node. The 'close' is defined by the hop from source node to destination node according to network topology.

(3) Allocate data unit on the closest rest nodes to the available node.

4. Experiments

We have done several experiments to verify our system in a 20-node cluster. Each node has 16 2.40 GHz Intel Xeon processor running 64- bit Debian 2.6.26 with 24GB RAM. We use NFS to share files within cluster. The source data set comes from the Andrew Pavlo's comparison paper[11]. They are two documents with random HTML, similar to that which a web crawler might find. We created two tables to store them in database system. The schema of tables is as follows:

```
CREATE TABLE Rankings (
  pageURL VARCHAR(100)
            PRIMARY KEY,
  pageRank INT,
  avgDuration INT );
```

```
CREATE TABLE UserVisits (
  sourceIP VARCHAR(16),
  destURL VARCHAR(100),
  visitDate DATE,
  adRevenue FLOAT,
  userAgent VARCHAR(64),
  countryCode VARCHAR(3),
  languageCode VARCHAR(6),
  searchWord VARCHAR(32),
  duration INT );
```

Each node is assigned 1GB document for Rankings(18,028,863 Records) and 0.5GB for UserVisits (4,114,693 Records). We designed three tasks related to HTML document processing.

- Selection Task

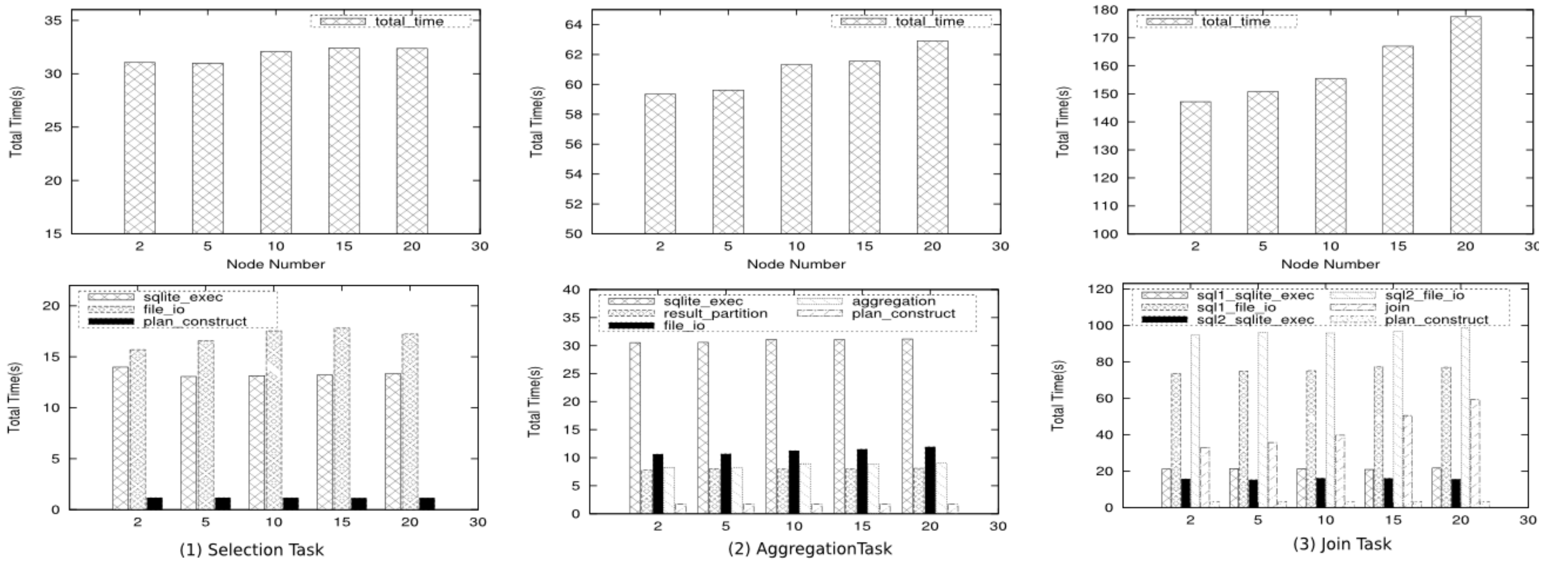


Fig. 6 Time cost for the three tasks

It is a lightweight filter to find the pageURLs in the Rankings table (1GB/node) with a pageRank above a user-defined threshold. For our experiments, we set this threshold parameter to 2, which yields approximately 3,177,285 records per data file on each node. Our system executes the selection task using the following simple SQL statement:
select pageURL, pageRank from Rankings where pageRank > X;
This query is executed by SQLite on each node. From **Fig. 6**, we can see that average processing time is approximately 32 sec mainly due to the sub-query execution and writing result data into file.

- Aggregation Task
  Aggregation task requires system to calculate the total adRevenue generated for each sourceIP in the UserVisits table, grouped by the sourceIP column and produces 1,978,880 records (54 MB). In this task, nodes need to exchange intermediate data with one another in order compute the final value. The following SQL query is used to express the task:
  select sourceIP, sum(adRevenue) from UserVisits group by sourceIP;
  The sub-query is executed on each node and the result is partitioned by *sourceIP* to all nodes and finally aggregated to get the sum of *adRevenue*. The experiments show that the total execution time increased less then 4 sec when the number of node varies from 2 to 20 and the average time is about 60 sec. It mainly includes time of sub-query execution, data partitioning, data persisting to files and data aggregation. Each part of time almost has little change with the increasement of nodes.
- Join Task
  Join task is to get the sourceIP and from its destination URLs there exists one whose rank is more than 2. It is expressed as following SQL query:
  select UserVisits.sourceIP from Rankings, UserVisits
  where Rankings.pageRank > 2 and Rankings.pageURL = UserVisits.destURL;
  A *join* operator takes the partitioned output of two sub-query on the key of *pageURL* and *destURL* respectively. The first sub-

query is <u>*select pageURL from Rankings where pageRank > 2;*</u> which produces about 167MB (3,177,258 records)intermediate data. Another is <u>*select sourceIP, destURL from UserVisits;*</u> producing approximately 271MB (4,114,693 records). The *join* operator applies the condition *Rankings.pageURL = UserVisits.destURL* to the intermediate result, gets all satisfied records and outputs *UserVisits.destURL*, approximately 5.4MB (386,008 records). The experiments show that the total time increases a little with the increasement of nodes mainly due to the join operator. Join operator firstly sorts all input files and gets two big files, one is for table Rankings and the other is the result of table UserVisits. The sort operation cost more time when the number of files to be sorted increases.

Discussion

File Writing and sub-query processing cost most of total execution time. We persist intermediate data into file and use file-based data communication method due to the consideration of node failure. If some nodes fail, the whole query does not need to be reprocessed and only jobs on the failed node are required to be re-executed since all the data files are already there. However, writing data into file really costs much time. A good alternate is to integrate the data pipeline mechanism which transfers data to a node directly maybe by TCP pipes into file-based method and find a trade-off between them.

The sub-query processing relies on the performance of SQLite. We are considering to improve it by changing some compiling parameters such as *temp_store* which indicates the position of intermediate result either in memory of temp files and *cache_size* setting the number of database file pages stored in memory at a time.

5. Future Works

We constructed a database system to support collective query for workflow systems. Experiments shows that our database system is scalable at least 20 nodes for query processing. However, at this point, we only have one client to issue query. So in the future, we will firstly test the collective queries from multiple clients and result data distribution to them. In addition, since data has to be loaded into database system before each task is executed, the extra cost will be increased a lot leading to user frustration. Hence, another important future work is to find a method for fast data loading.

References

1) Jim Gray. : *The Fourth Paradigm: Data-Intensive Scientific Discovery*, Tony Hey, Stewart Tansley, and Kristin Tolle (Editors), editors, 2009.
2) Gordon Bell, Tony Hey, and Alex Szalay.: COMPUTER SCIENCE: Beyond the Data Deluge, *Science*, Vol.323, No.5919, pp.12971298 (2009).
3) 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.*, Vol.25, No.5, pp.528540 (2009).
4) Ian Taylor, Matthew Shields, Ian Wang, and Andrew Harrison.: The triana workflow environment: Architecture and applications. *Workflows for e-Science*, pp. 320339 (2007).
5) Kenjiro Taura, Takuya Matsuzaki, Makoto Miwa.: Design and Implementation of GXP Make – a Workflow System Based on Make. *e-Science 2010 conference (eScience2010)*, Brisbane, Australia, IEEE, pp.214 - 221 (2010).
6) J. Dean and S. Ghemawat.: MapReduce:Simplified Data Processing on Large Clusters. *OSDI 04*, pp.1010 (2004).
7) Phuong Nguyen, Milton Halem.: A MapReduce workflow system for architecting scientific data intensive applications. *SECLOUD '11*, ACM, New York, doi:10.1145/1985500.1985510 (2011)
8) Xubo Fei, Shiyong Lu, Cui Lin.: A MapReduce-Enabled Scientific Workflow Composition Framework. *ICWS '09*, IEEE, ISBN: 978-0-7695-3709-2 doi¿10.1109/ICWS.2009.90 (2009)
9) DeWitt, D.J. and Gray, J.: Parallel database systems: the future of high-performance database systems. *Commun*, ACM, Vol.35, No.6, pp.8598 (1992).
10) Vertica. available from ⟨http://www.vertica.com/⟩.
11) Andrew Pavlo, Erik Paulson, Alexander Rasin.: A comparison of approaches to large-scale data analysis. *SIGMOD 09: Proceedings of the 2009 ACM SIGMOD International Conference*, ACM (2009).
12) Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden.: MapReduce and parallel DBMSs: friends or foes? *Commun*, ACM, Vol.53, No.1,pp.6471 (2010).
13) M. Miwa, R. Satre, J.-D. Kim, J. Tsujii.: Event extraction with complex event classification using rich features. *JBCB*, pp.131146 (2010).
14) Bjorne, F. Ginter, S. Pyysalo, J. Tsujii, and T. Salakoski.: Complex event extraction at PubMed scale. *Bioinformatics* Oxford, England, Vol.26, No.12, pp.382390 (2010)
15) SQLite. available from ⟨http://www.sqlite.org⟩.