Journal of Network and Computer Applications 34 (2011) 1035–1042

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

# Optimization of sub-query processing in distributed data integration systems

Gang Chen, Yongwei Wu*, Jia Liu, Guangwen Yang, Weimin Zheng

*Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 100084, China*

A R T I C L E  I N F O

*Article history:*
Received 28 October 2009
Received in revised form
21 April 2010
Accepted 7 June 2010
Available online 15 June 2010

*Keywords:*
Cloud computing
Grid computing
Data integration
Query
Data flow

A B S T R A C T

Data integration system (DIS) is becoming paramount when Cloud/Grid applications need to integrate and analyze data from geographically distributed data sources. DIS gathers data from multiple remote sources, integrates and analyzes the data to obtain a query result. As Clouds/Grids are distributed over wide-area networks, communication cost usually dominates overall query response time. Therefore we can expect that query performance can be improved by minimizing communication cost.

In our method, DIS uses a data flow style query execution model. Each query plan is mapped to a group of μEngines, each of which is a program corresponding to a particular operator. Thus, multiple sub-queries from concurrent queries are able to share μEngines. We reconstruct these sub-queries to exploit overlapping data among them. As a result, all the sub-queries can obtain their results, and overall communication overhead can be reduced. Experimental results show that, when DIS runs a group of parameterized queries, our reconstructing algorithm can reduce the average query completion time by 32–48%; when DIS runs a group of non-parameterized queries, the average query completion time of queries can be reduced by 25–35%.

## 1. Introduction

As cloud and grid computing is becoming more and more popular, increasing number of applications needs to access and process data from multiple distributed sources. For example, a bioinformatics application needs to query autonomous databases across the world to access different types of proteins and protein–protein interaction information located at different storage clouds.

Data integration in Clouds/Grids is a promising solution for combining and analyzing data from different stores. Several projects (e.g., OGSA-DQP Lynden et al., 2009; CoDIMS-G Fontes et al., 2004; and GridDB-Lite Narayanan et al., 2003) have been developed to study data integration in distributed environments. For example, OGSA-DQP (Lynden et al., 2009) is a service-oriented, distributed query processor, which provides effective declarative support for service orchestration. It is based on an infrastructure consisting of distributed services for efficient evaluation of distributed queries over OGSA-DAI wrapped data sources and analysis resources available as services.

Queries to data integration systems are generally formulated in virtual schemas. Given a user query, a data integration system typically processes the query by translating it into a query plan and evaluating the query plan accordingly. A query plan consists of a set of sub-queries formulated over the data sources and operators specifying how to combine results of the sub-queries to answer the user query. As Clouds/Grids are generally built over wide-area networks, high communication cost is the main reason of leading to slow query response time. Therefore, query performance can be improved by minimizing communication cost. In this paper, our objective is to reduce communication overhead and therefore improve query performance, through optimizing sub-query processing.

We optimize sub-query processing by exploiting data sharing opportunities among sub-queries. IGNITE is a method proposed in Lee et al. (2007) to detect data sharing opportunities across concurrent distributed queries. By combining multiple similar data requests issued to the same data source, and further to a common data request, IGNITE can reduce communication overhead, thereby increase system throughput. However, IGNITE does not utilize parallel data transmission so that it does not always improve query performance. Our approach proposed here enhances IGNITE by addressing its drawbacks so that query performance in distributed systems can be further improved.

Our data integration system employs an operator-centric data flow execution model, also proposed in Harizopoulos et al. (2005). Each operator corresponds to a μEngine, which has local threads for data processing and data dispatching. Queries are processed by routing data through μEngines. All the μEngines work in parallel, thus they can fully utilize intra-query parallelism. Based

* Corresponding author. Tel.: +86 10 62796341.
*E-mail addresses:* c-g05@mails.tsinghua.edu.cn (G. Chen), wuyw@tsinghua.edu.cn (Y. Wu), liu-jia04@mails.tsinghua.edu.cn (J. Liu), ygw@tsinghua.edu.cn (G. Yang), zwm-dcs@tsinghua.edu.cn (W. Zheng).

1084-8045/$ - see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jnca.2010.06.007

on such an operator-centric data flow execution model, all similar query plans are allocated to the same group of μ Engines. Therefore sub-queries from different queries are grouped in a common place for processing to enable data sharing across the sub-queries.

In the μEngine for processing sub-queries, a query reconstruction mechanism with a Merge-Partition (MP) reconstruction algorithm is developed. The query reconstruction mechanism can construct a set of new queries to eliminate data redundancy among the sub-queries being processed by the μEngine. All the sub-query answers can be obtained by evaluating the new queries and therefore the required communication overhead can be reduced.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 describes the execution model of our DIS. Section 4 proposes the Merge-Partition (MP) query reconstruction algorithm used in our DIS. Section 5 discusses the experiments that we conducted to evaluate our solution. Section 6 concludes the paper.

## 2. Related work

IGNITE system proposed in Lee et al. (2007) was developed based on the PostgreSQL database, and is a work mostly related to the work presented in this paper. IGNITE decouples the source wrappers from the execution engine (adopted from the PostgreSQL database), and enables the execution engine to send sub-queries to same source, which therefore makes data sharing across sub-queries possible. Meanwhile, IGNITE employs the iterator model proposed in Graefe (1993) so that sub-queries may have delay opportunities – a sub-query can wait for other similar requests. Because of this, IGNITE develops a Start-Fetch wrapper with Request Window mechanism. The wrapper combines a group of similar sub-queries to a common sub-query and only sends the common sub-query to the data source, so that redundant answers among sub-queries can be eliminated.

There are two major differences between our method and IGNITE. First, our method reconstructs original sub-queries to alternative sub-queries, which may not eliminate all redundant answers, but never introduce unnecessary data. IGNITE combines a group of sub-queries into a single common sub-query to eliminate redundant answers; however by doing so it may introduce unnecessary data and in some cases may increase the size of query answers. IGNITE increases communication traffic in two ways: (1) it requires not only output attributes, but also predicate attributes to identify sub-query answers; (2) all tuples including common tuples must contain all required attributes for all sub-queries. The second major difference between our method and IGNITE is that if the source wrapper manages multiple work threads, our method can take advantage of parallel sub-query processing, whereas IGNITE cannot.

A significant amount of work on data integration (i.e. Ives, 2002; Halevy et al., 2006; Deshpande et al., 2007; Haas et al., 1997) has been conducted. Several projects (e.g., OGSA-DQP (Lynden et al., 2009); CoDIMS-G (Fontes et al., 2004); and GridDB-Lite (Narayanan et al., 2003)) particularly focus on data integration in Clouds/Grids. With a service-oriented architecture, OGSA-DQP supports pipeline and partition parallelism for efficient evaluation of distributed queries. Different from our method, OGSA-DQP uses iterator model and relevant research on OGSA-DQP often focuses on improving the performance of a single query. Similarly, CoDIMS-G and GridDB-Lite also focus on improving the performance of a single query.

Many efforts have been made on exploiting data sharing in data integration area as well as database area (e.g., Dalvi et al., 2001; Harizopoulos et al., 2005; Lee et al., 2007; Goldstein and Larson 2001; Sacco and Schkolnick, 1986), including: (1) Multiple-query optimization (MQO) techniques (e.g., Dalvi et al., 2001), which exploit data sharing by identifying common sub-expressions in query execution plans during optimization; (2) buffer pool management (e.g., Sacco and Schkolnick, 1986), which typically reuses disk pages in a buffer pool; (3) caching and view materialization (e.g., Kossmann, 2000; Goldstein and Larson, 2001), which typically reuse pre-stored data in cache or materialized view.

There are also techniques proposed in the distributed data processing area, aiming to improve query efficiency (e.g. parallel query processing techniques proposed in Gounaris (2005) and adaptive query processing techniques proposed in Deshpande et al. (2007) and Gounaris (2005)). The technique proposed in Kossmann (2000) is one of them, which achieves the objective by decreasing communication cost. For example, semi-joins are proposed in Kossmann (2000) to reduce data transition while processing joins between tables stored at different sites, and row blocking is used to reduce the number of communication occurrences by delivering tuples in batches.

## 3. Query engine

In this section, we discuss the execution engine of our DIS. The engine employs a data flow style execution model (Section 3.1), based on it, sub-queries can be gathered to a common place for evaluation through source wrappers (Section 3.2). We also discuss in Section 3.3, in detail, why is required to have a delay for each request in order to better utilize data sharing.

### 3.1. Data flow execution model

As previously discussed, our DIS employs a data flow style execution model, also referred to as operator-centric (one-operator, many-queries) model in Qpipe (Harizopoulos et al., 2005). In this model, each operator uses an independent μEngine. μEngines serve requests from submitted queries. Each request specifies input and output data buffers, and operator arguments. By linking a μEngine's output to another's input, producer–consumer relationships can be established among μEngines. Queries can then be evaluated by pushing data through μEngines.

Fig. 1 describes the runtime model of our data flow execution. In this model, there are four kinds of elements: Query Plans, requests, *dispatcher* and μEngines.

- Query Plans: a Query Plan consists of a set of sub-queries formulated over the data sources and operators specifying how to combine results of the sub-queries to answer the user query.
- requests: are generated according to the Query Plans. They can be considered a group of operations need to be performed by μEngines.
- *dispatcher*: is a component which is responsible for sending the requests to proper μEngines.
- μEngines: Each round box in Fig. 1 represents a μEngine and the text in the box indicates its corresponding operator. In Fig. 1, μEngines labeled with "*wrapper*" or "*WSP*" is used to process sub-queries or invoke web services, respectively. μEngines labeled with "*Sort*", "*Selection*" and "*Hashjoin*" are used to process relational operators "Sort", "Selection" and "Hashjoin", respectively.

The process of evaluating a query plan is as follows. After the arrival of the query plan, the *dispatcher* creates as many requests as the nodes in the query plan and dispatches these requests to their corresponding μEngines. Then, the μEngines work in parallel

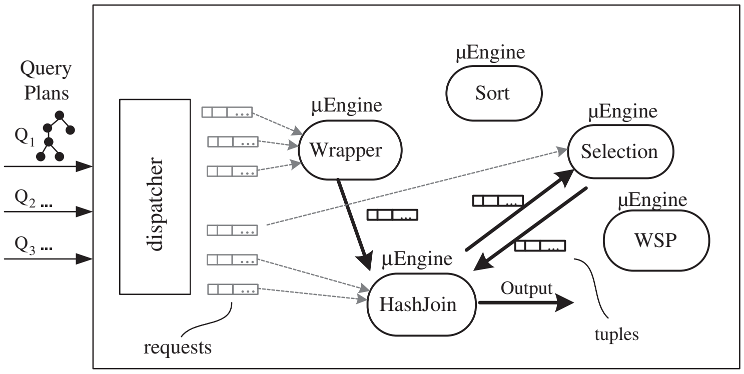


**Fig. 1.** Runtime architecture of the execution engine.

to process the requests, and the data tuples flow among the buffers of different μEngines in push mode for evaluation.

The reasons why we choose the operator-centric data flow model instead of the iterator model are as follows. First, in the data flow model, all μEngines can work in parallel; therefore intra-query parallelism can be achieved, and query processing can be accelerated. This feature is particularly important to DIS for the reason that queries of DIS (e.g., OGSA-DQP (Lynden et al., 2009) and CoDIMS-G (Fontes et al., 2004)) always contain time-consuming operators such as external web service calls and local function calls. Second, the data flow model can group requests with the same nature together, and can process each group of similar requests using a dedicated μEngine. This feature enables the execution engine to send all sub-queries to same source and therefore enables data sharing across sub-queries.

### 3.2. *Sub-query evaluation through source wrappers*

Wrappers are used to evaluate sub-queries in data integration systems, where wrappers hide the heterogeneity of accessing data sources of different types. For example, OGSA-DAI acts as a grid-enabled wrapper, providing service access interfaces for various data sources.

Our DIS has one specific μEngine, referred to as μEngine-W, to invoke wrapper procedures of evaluating sub-queries. As shown in Fig. 2, the μEngine-W consists of the following elements.

- Sub-query requests: a Query Plan consists of a set of sub-query requests.
- Query Reconstructor: the component which is responsible for reconstructing the requests by using Query Reconstruction Algorithm proposed in this paper (Section 4).
- Reconstructed sub-queries: the sub-queries generated by Query Reconstructor.
- Coordinator: the component is responsible for reorganizing the results of reconstructed sub-queries for the original sub-queries.
- Wrapper Handler: the component is responsible for performing the queries on Data Sources.

A data sharing mechanism is proposed in this paper (Section 4) and applied in μEngine-W to optimize sub-query processing.

Fig. 2 shows the overview architecture of μEngine-W with a query reconstruction mechanism. To process sub-query requests, μEngine-W first reconstructs a set of sub-queries (say *W*) contained in the sub-query requests into a substitute set of sub-queries (say *W*′) using the *Query Reconstructor*. This step identifies and eliminates redundant data among the sub-queries

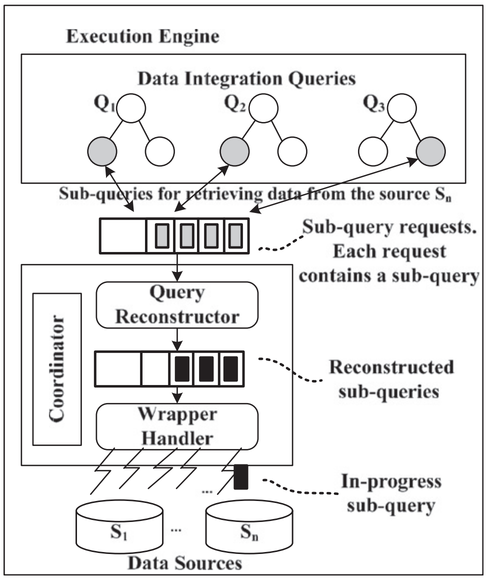


**Fig. 2.** Architecture of μEngine-W.

in *W*. Then μEngine-W evaluates the reconstructed sub-queries *W*′ to get required answers for the original queries *W* using the *Wrapper Handler*.

### 3.3. *Sub-query delay to better facilitate data sharing*

This subsection discusses when is appropriate for μEngine-W to trigger a reconstruction.

The target of the reconstruction algorithm is the sub-query requests waiting in μEngine-W. Assume that μEngine-W schedules an available work thread to remove and process a request immediately after the request arrives. The only case in which concurrent waiting requests occur is that the requests arrive at the exact same time or μEngine-W has no idle work thread. Therefore, the opportunity for reducing query cost through exploiting data sharing in this case is very low and thus the expected cost saving is quite limited.

In order to increase the cost saving, μEngine-W makes use of the allowed delays of sub-query requests. As described in IGNITE (Lee et al., 2007), when complex queries are executed, some sub-queries sent to the data sources may have a delay opportunity so that it is tolerable to wait for other similar sub-queries. Let's take the following query for example: *select *from t*1, *t*2 *where t*1.*id*=*t*2.*id*. Suppose we use a hash join to calculate the tuples satisfying $t1.id = t2.id$. The left child node of the hash join is a sub-query to fetch tuples from $t1$, and the right child node is a sub-query to fetch tuples from $t2$. The processing of the hash join consists of two phases: (1) building the hash table using tuples from $t1$, and (2) probing the hash table using tuples from $t2$. We can see that there is an interval between the arrival time of the sub-query to $t2$ and the time that the hash join takes its result. Though IGNITE discusses the delays in the context of the iterator model, we can see that the same mechanism can be similarly applied in our context: the operator-centric data flow model. By setting a delay to each sub-query request, the opportunity of the presence of concurrent requests can be increased and the cost saving of the query reconstructing mechanism can be increased accordingly.

After adding a delay to each sub-query request, μEngine-W removes the requests from its queue, classifies them into groups according to their target data sources, and then triggers a reconstruction in the following two situations: (1) the waiting time of a request has reached its tolerable delay and (2) the result of a request is about to be consumed and μEngine-W is notified by the consumer μEngine of the request to trigger a reconstruction to its corresponding group.

## 4. Query reconstruction algorithm

In this section, we introduce the Merge-Partition (MP) reconstruction algorithm applied in our DIS. First, we model the problem of query reconstruction in Section 4.1. Then, in Section 4.2, we present the algorithm to see how it reconstructs a set of queries and computes the answers of the queries.

### 4.1. Problem description

We assume every sub-query is a Select–Project–Join (SPJ) query with duplicate-preserving semantics. Every query is in the form of $\pi_L(\sigma_P(R_1 \times R_2 \times \cdots \times R_a))$, where L is the list of output attributes, P is the selection predicate, and $R_1, R_2, \ldots, R_a$ are queried relations.

For a given query $q_i$:

Let $L^o(q_i)$be the set of output attributes.
Let $P(q_i)$ be the selection predicate of $q_i$.
Let $L^c(q_i)$ be the set of attributes that appear in $P(q_i)$.
Let $S(q_i) = \{R_1, \ldots, R_a\}$be the set of source relations.
Let $R(q_i)$ be the answer of $q_i$.

For two queries $q_i$ and $q_j$, let $R(q_i \cap q_j)$be their common answer.

The problem of query reconstruction can be described as follows. Given a set of SPJ queries $Q = \{q_1, \ldots, q_n\}$, we compute another set of queries $Q^* = \{q_1^*, \ldots, q_m^*\}$ such that: (1) we can produce the answers to the original queries $Q = \{q_1, \ldots, q_n\}$ from the answers to $Q^*$; (2) $\mathrm{Size}(R(Q^*)) \leq \mathrm{Size}(R(Q))$, which means the network overhead incurred by delivering $R(Q^*)$is smaller than that of $R(Q)$.

The query reconstruction mechanism consists of two activities: (1) reconstructing the set of original queries before dispatching them to the data sources, and (2) computing the answers to the original queries based on the answers to the reconstructed queries.

### 4.2. MP query reconstruction algorithm

In this section, we describe our Merge-Partition query reconstruction algorithm, which reconstructs queries by two steps: query merging and query partitioning, for both parameterized queries (Section 4.2.1) and non-parameterized queries (Section 4.2.2).

#### 4.2.1. Parameterized query

Parameterized queries are queries that have one or more embedded parameters in a SQL statement. The main advantages of a parameterized query are: (1) it makes a SQL statement less prone to errors and (2) it saves query preparation time since you can prepare the query one time, and execute the query as many times as you wish.

There are two steps to reconstruct a group of parameterized queries: merging all the queries using the method proposed in IGNITE (Lee et al., 2007) and using a range-partition method to partition the merged query into query fragments.

Let us take the following example to illustrate our method:

Q: select* from a_table where key > ?
Q1: select* from a_table where key > 10
Q2: select* from a_table where key > 20
Q3: select* from a_table where key > 50

Suppose Q1–Q3 are three sub-queries, which are generated by embedding values "10", "20" and "50" in the parameterized query Q. To reconstruct the queries Q1–Q3, the first step is to merge these three queries, which results the following merged query:

Q4: select* from a_table where key > 10.

Then, the MP query reconstruction algorithm partitions the merged query Q4. Assume that the domain of attribute "key" is 0–2000, and the number of the fragments is 4. An example of query partitioning is Q5–Q8:

Q5: select* from a_table where key > 10 and (key ≤ 500);
Q6: select* from a_table where key > 10 and (key > 500 and key ≤ 1000);
Q7: select* from a_table where key > 10 and (key > 1000 and key ≤ 1500);
Q8: select* from a_table where key > 10 and (key > 1500 and key ≤ 2000);

To computer the answer of Q1, we should apply its predicate (key > 10) to the answer of the merged query Q4. It is obvious that:

$R(Q4) = R(Q5) \cup R(Q6) \cup R(Q7) \cup R(Q8)$

The answers to Q2 and Q3 can be computed in a similar way.

#### 4.2.2. Non-parameterized query

*4.2.2.1. Merge queries.* For any two queries $q_i$ and $q_j$ in $Q = \{q_1, \ldots, q_n\}$, they can be merged if all of the following three conditions can all be satisfied:

(1) $L^c(q_i) \subseteq L^o(q_i)$;
(2) $L^c(q_j) \subseteq L^o(q_j)$;
(3) $L^o(q_i) = L^o(q_j)$.

Let $q_m$ be the merged query of $q_i$ and $q_j$. The answers to $q_i$ and $q_j$ can be computed from the answer to $q_m$ because of the following reasons:

(1) the answer to $q_m$ get all the tuples required by $q_i$ and $q_j$;
(2) the answer to $q_m$ get all the columns required by $q_i$ and $q_j$;
(3) the answers to $q_i$ and $q_j$ can be computed since the attributes required by their predicates are provided in the answer to $q_m$.

Besides, from the third condition $L^o(q_i) = L^o(q_j)$ (see above), we can see that no unnecessary data is introduced by evaluating the merged query $q_m$ to get answers for $q_i$ and $q_j$.

We use the set $Q^1=\{q^1_1,\ldots,q^1_s\}$ to represent the result after the step of merging queries. In this step, we record all the pairs of queries that can be merged. Let $MP=\{\langle q_{11},q_{12}\rangle,\ldots,\langle q_{p1},q_{p2}\rangle\}$be these query pairs. The communication overhead can be saved by this step is$size(R(q_{11}\cap q_{12}))+\cdots+size(R(q_{p1}\cap q_{p2}))$. For each query pair, say $\langle q_i, q_j\rangle$, we also record the following information as lineage expressions:

(1) $\langle q_m, func_i, q_i\rangle$
(2) $\langle q_m, func_j, q_j\rangle$

Where $func_i$ is the function used to compute the answer to $q_i$ from the answer to $q_m$, and$func_j$ is the function used to compute the answer to $q_j$.

*4.2.2.2. Partition queries.* In this step, we further partition the queries in $Q^1=\{q^1_1,\ldots,q^1_s\}$ to eliminate overlapping data.

We use an example to explain the basic idea of the query partition first. Fig. 3 shows two queries: $q_1$ and $q_2$. We use a two-dimensional area to represent a query answer. As shown in Fig. 3(a), if you process the two queries directly, the common data represented by the yellow area is transmitted twice. Fig. 3(b) shows the way after applying query partition. Queries $q_1$ and $q_2$are partitioned into three fragments: (1) $q^*_2$: to get the overlapping tuples (i.e., a query with condition($P(q_1)\cap P(q_2)$)), (2) $q^*_1$: to get the remaining tuples of $q_1$ (i.e., a query with condition($P(q_1)-P(q_2)$)), and (3) $q^*_3$: to get the remaining tuples of $q_2$ (i.e., a query with condition($P(q_2)-P(q_1)$)). After the answers to $q^*_1$, $q^*_2$ and $q^*_3$ are retrieved, the answer to $q_1$ can be computed directly from the answers to $q^*_1$ and $q^*_2$. The answer to $q_2$can be extracted directly from the answers to $q^*_2$ and $q^*_3$.

Given two queries, if the amount of their common data is small, the method of query partition may lead to performance degradation. The reason is that partition increases the number of queries and each query brings an initial delay. Thus, we need to estimate if a query partition can bring in performance improvement before take any actions. Given two queries $q_i$ and $q_j$, without a query partition, the cost of sub-query processing is

$$C^{ij}_{before}=\frac{size(R(q_i))}{r}+\frac{size(R(q_j))}{r}+2ID$$

where $r$denotes the data rate over the link between the query engine and the target data source, $ID$denotes the initial delay of processing a sub-query. With a query partition, the cost is

$$C^{ij}_{after}=\frac{size(R(q^*_{ij1}))+size(R(q^*_{ij2}))+size(R(q^*_{ij3}))}{r}+3ID$$

$C^{ij}_{after}$ which can also be expressed as

$$C^{ij}_{after}=\frac{size(R(q_i))+size(R(q_j))-size(R(q^*_{ij2}))}{r}+3ID$$

To determine if the query partition can improve the query performance, we need to compare $C^{ij}_{before}$ with$C^{ij}_{after}$. Generally, if $C^{ij}_{before}-C^{ij}_{after}\geq\alpha$ is satisfied (the amount of improvement exceeds a specified threshold $\alpha$), a query partition can be considered. We

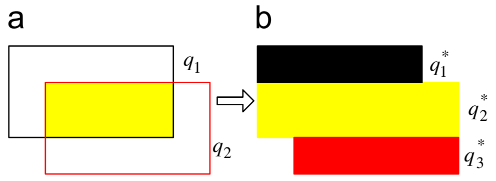


**Fig. 3.** Partition the queries $q_1$ and $q_2$ to the queries $q^*_1$, $q^*_2$ and $q^*_3$, to eliminate overlapping data (each two-dimensional area represents a query answer).

use$C^{ij}_{imp}$ to denote $C^{ij}_{before}-C^{ij}_{after}$, and the following equation can be derived:

$$C^{ij}_{imp}=\frac{size(R(q^*_{ij2}))}{r}-ID$$

To check if $C^{ij}_{imp}\geq\alpha$ is satisfied, an estimate of the amount of the common data $R(q^*_{ij2})$ is required. We can use statistic techniques in database area (e.g., Ahad et al., 1989; Getoor et al., 2001) to perform the estimation. Due to space limitations, we omit the discussion of the estimaiton from this paper.

After the basic idea of partition and the basic conditions to consider a partition are clear, we now describe the process of performing partitions over the queries in $Q^1=\{q^1_1,\ldots,q^1_s\}$.

*Step* 1 estimates the amount of common data between every two queries, and build the following matrix $M_{pri}$ to denote the priorities of query partitions:

$$M_{pri}=\begin{array}{c} \\ q^1_1 \\ q^1_2 \\ \vdots \\ q^1_s \end{array}\begin{array}{c} \begin{array}{cccc} q^1_1 & q^1_2 & \cdots & q^1_s \end{array} \\ \begin{pmatrix} p_{11} & p_{12} & \cdots & p_{1s} \\ p_{21} & p_{22} & \cdots & p_{2s} \\ \vdots & \vdots & \ddots & \vdots \\ p_{s1} & p_{s2} & \cdots & p_{ss} \end{pmatrix}\end{array}$$

where, $p_{ij}$ denotes the priority of the query partition between $q^1_i$and $q^1_j$. If $i=j$, then $p_{ij}=0$; otherwise, $p_{ij}$ can be computed as

$$p_{ij}=C^{ij}_{imp}-\alpha=\frac{size(R(q^*_{ij2}))}{r}-ID-\alpha$$

*Step* 2 selects the biggest and positive priority from the matrix $M_{pri}$. Biggest one means the one with the highest priority, and positive one means that the requirements of performing query partition are satisfied. if $p_{ij}$ is the biggest one, remove the $i_{th}$ and $j_{th}$ rows, and the $i_{th}$ and $j_{th}$ columns from the matrix $M_{pri}$, perform a partition to the queries $q^1_i$and $q^1_j$, and record the following information as lineage expressions:

(1) $\langle q^*_{ij1}$, null, $q^1_i\rangle$

(2) $\langle q^*_{ij2} func_i, q^1_i\rangle$

(3) $\langle q^*_{ij2} func_j, q^1_j\rangle$

(4) $\langle q^*_{ij3}$, null, $q^1_j\rangle$

where, $\langle q^*_{ij1}$, null, $q^1_i\rangle$ means the answer to $q^*_{ij1}$ is a subset of the answer to $q^1_i$. "null" means no computation on R($q^*_{ij1}$) is needed before returning it to $q^1_i$. $func_i$ is the function used to compute the answer to $q^1_i$ from the answer to $q^*_{ij2}$, which is actually a projection operation. Other symbols can be explained similarly.

*Step* 3 repeats *step* 2 until there is no positive number in the matrix $M_{pri}$.

We use the set $Q^*=\{q^*_1,\cdots,q^*_m\}$to represent the result after the step of partitioning queries. Then, we can begin to evaluate queries in $Q^*$.

When the data returned from the data source arrives, we can first use the lineage expressions recorded in the step of partitioning queries to compute the answers of the queries $Q^1=\{q^1_1,\ldots,q^1_s\}$, and then use the lineage expressions recorded in the step of merging queries to compute the answers to the original sub-queries $Q=\{q_1,\ldots,q_n\}$.

## 5. Evaluation

In this section, we present our evaluation method and results. The overall experimental setup is discussed in Section 5.1, followed by the detailed discussion of each experiment and its results in Section 5.2.

### 5.1. *Experimental setup*

The experimental setup consists of two parts: the server side to deploy databases and the client side to run our data integration system. The experimental environment is presented in Fig. 4, including the location and configuration of each machine, and the data distribution over the machines.

On the server side, we use a TPC-H database (http://www.tpc.org/tpch/) (scale factor 1) as the dataset of our experiments. The TPC-H database has eight relations: REGION, NATION, CUSTOMER, SUPPLIER, PART, PARTSUPP, ORDERS, and LINEITEM. To build a distributed environment, we created eight PostgreSQL (version: 8.3) databases on four different machines, as shown in Fig. 4. Each of the eight databases provides one of the above relations. The dataset in the relations was generated by DBT (http://osdldbt.sourceforge.net/). On the client side, our DIS was deployed on a separate machine, as shown in Fig. 4.

The value of sub-query delay is set according to the cardinality of tables and the structure of the whole query tree. This principle is followed by Lee et al. (2007).

### 5.2. *Experiment analysis and results*

Three experiments have been designed and conducted to evaluate (1) the effectiveness of the operator-centric data flow query model by comparing it with the iterator model, (2) the effectiveness of the Merge-Partition algorithm by comparing it with IGNITE having the merge algorithm used in μEngine-W and an original approach where μEngine-W processes sub-queries without using any query reconstruction mechanism, when processing a group of parameterized queries, and (3) the effectiveness of the Merge-Partition algorithm when processing a group of non-parameterized queries.

#### 5.2.1. *Experiment 1*

Experiment 1 evaluates the effectiveness of the operator-centric data flow query model while executing DIS queries. We evaluated a group of queries that follow the parameterized query described in Fig. 5(a). The parameterized query contains two web services $WS_1$ and $WS_2$. The query plan described in Fig. 5(a) is used to answer the queries.

In this experiment, the selectivities of $WS_1$ and $WS_2$ are 1, and the average response time of $WS_1$ and $WS_2$ are 0.06 and 0.11 s, respectively. During different runs, the parameter values of the query were set as “1992–02–01”, “1992–03–01”, “1992–04–01”, …, and “1992–11–01”, respectively.

We ran the queries in both the iterator model and our data flow model. The experiment result is shown in Fig. 5b), where the *x*-axis and *y*-axis denote the parameter value the query and the completion time of the query, respectively. From the figure, we can see that the data flow model achieves 40% less query completion time than the iterator model. This is a reasonable result because the data flow model invokes web services $WS_1$ and $WS_2$ in parallel to process data, which is the rationale why we chose the data flow model rather than the iterator model employed in IGNITE.

#### 5.2.2. *Experiment 2*

This experiment evaluates the effectiveness of the Merge-Partition algorithm by processing a group of parameterized queries. In this experiment, we compare the average query completion time and communication traffic of the following three approaches. (1) MP: the MP algorithm is used in μEngine-W; (2) IGNITE: the merge algorithm is used in μEngine-W;

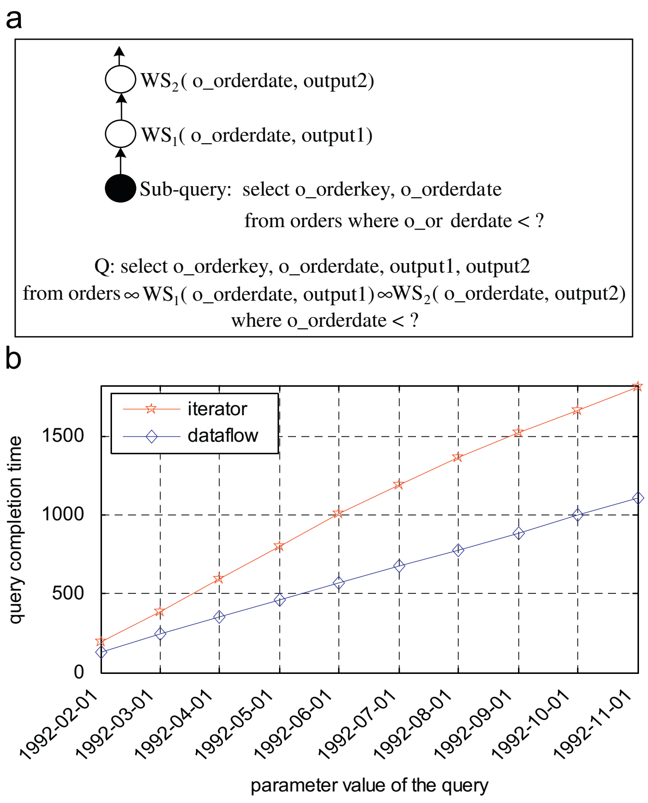
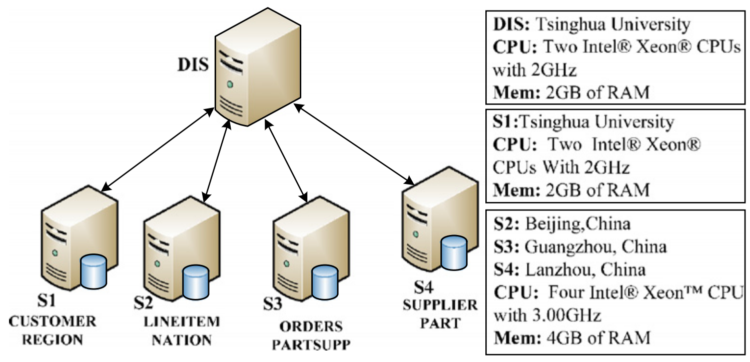


**Fig. 5.** (a) Parameterized query and query plan. (b) Query completion time.



**DIS:** Tsinghua University
**CPU:** Two Intel® Xeon® CPUs with 2GHz
**Mem:** 2GB of RAM

**S1:**Tsinghua University
**CPU:** Two Intel® Xeon® CPUs With 2GHz
**Mem:** 2GB of RAM

**S2:** Beijing,China
**S3:** Guangzhou, China
**S4:** Lanzhou, China
**CPU:** Four Intel® Xeon™ CPU with 3.00GHz
**Mem:** 4GB of RAM

**Fig. 4.** Experimental environment.

(3) Original: µEngine-W processes sub-queries without using any query reconstruction mechanism.

In this experiment, 50 queries ($q_1$–$q_{50}$) are generated complying with the query described in Fig. 6(a). 49 random numbers ($r_1$–$r_{49}$) in the range of 0–60 are generated as query intervals. Queries $q_1$–$q_{50}$ are submitted sequentially. The time interval between submitting $q_i (1 \leq i \leq 49)$ and submitting $q_{i+1}$ is $r_i$ seconds.

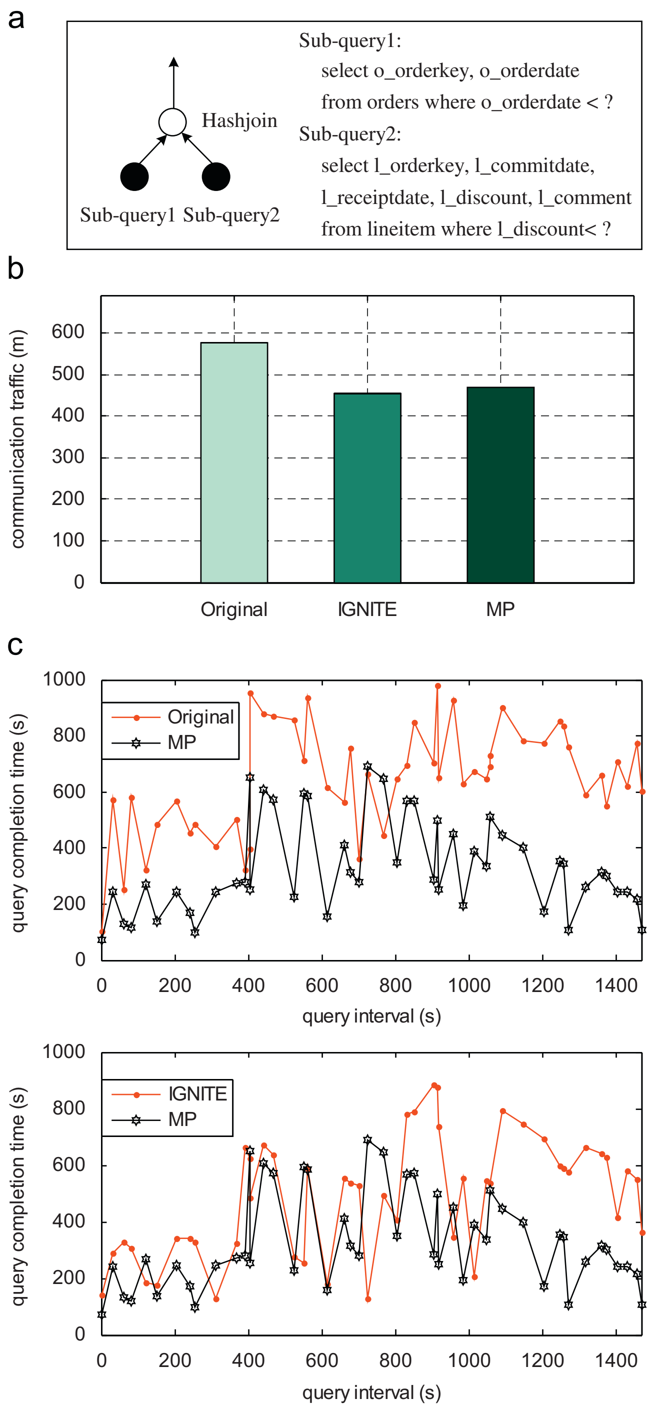
Fig. 6(b) and (c) presents the results of the experiment. Fig. 6(b) records the total communication traffic of the queries of each of the three approaches. Compared with the Original approach, IGNITE and our MP approach can reduce the communication traffic by around 20% and 15%, respectively. Fig. 6(c) shows two plots of the query completion time of the three approaches at each query interval. The *y*-axis of each plot is the completion time of $q_1$–$q_{50}$, the *x*-axis is the interval between the arrival time of a query and the arrival time of the first query $q_1$, and each point in each plot represents a query. The average query completion time of the Original, IGNITE and MP is 64 6332.914, 490 724.16 and 333 633.88 in decreasing order, respectively.

From Fig. 6(c), the following conclusions can be drawn. First, compared with Original, our MP approach can reduce the average completion time by 48%. This is obviously because MP can exploit data sharing among sub-queries. This is due to one fact. Though reconstructing small and large queries together may take shorter time than reconstructing them separately, this may lead to reconstructing small queries takes more time. Third, the average completion time of MP is smaller than that of IGNITE by 32%. It is because MP can get a higher degree of parallelism during the phase of partition than IGNITE.

### 5.2.3. Experiment 3

This experiment evaluates the effectiveness of the Merge-Partition algorithm by processing a group of non-parameterized queries. Similarly to Experiment 2, in this experiment, 50 queries ($q_1$–$q_{50}$) and 49 random numbers ($r_1$–$r_{49}$) in the range of 0–60 are generated as query intervals. The generated queries are submitted sequentially.

Fig. 7 shows the experimental results. Fig. 7(a) records the total communication traffic of the queries of each approach. Compared with Original, both IGNITE and MP can reduce the communication traffic by about 15%. Fig. 7(b) shows a plot of the average query completion time of every five queries of the three approaches. Each point in the plot represents an average completion time of each five queries. For example, the point



```
Sub-query1:
    select o_orderkey, o_orderdate
    from orders where o_orderdate < ?
Sub-query2:
    select l_orderkey, l_commitdate,
    l_receiptdate, l_discount, l_comment
    from lineitem where l_discount< ?
```

**Fig. 6.** Experimental results of parameterized queries: (a) parameterized query, (b) total communication traffic of queries and (c) query completion time of each query.

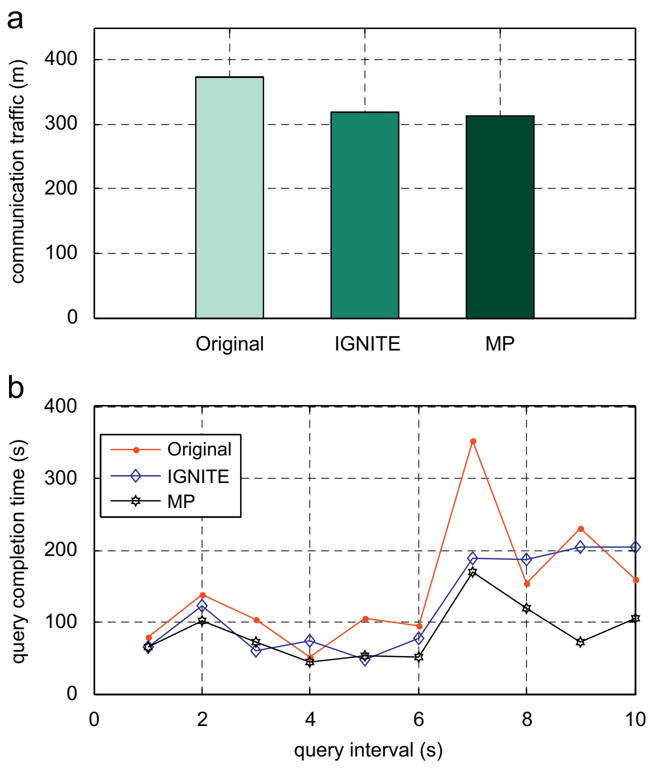


**Fig. 7.** Experimental results of non-parameterized queries: (a) total communication traffic of queries, (b) average completion time of each query group.

corresponding to the number 1 of the $x$-axis represents the average completion time of queries $q_1$–$q_5$. From Fig. 7(b), we can observe the following facts. (1) Compared with Original, MP can reduce the average completion time of queries by more than 35%. (2) For most of the queries, MP takes less time to complete the queries than IGNITE; MP can reduce the average completion time of all the queries by 25%, to compare with IGNITE.

## 6. Conclusion

Distributed data sources can be heterogeneous, and managing, analyzing, and processing data from different sources in an integrated way is becoming more and more important. Distributed data integration applications are always processed on distributed infrastructures, and communication cost becomes the main factor of determining query response time. Therefore we can expect that query performance can be improved by minimizing communication cost. The objective of this paper is to propose an approach to improve the query processing performance of data integration systems by optimizing sub-query processing.

Our data integration system adopts a data flow style execution model, which allows the system to exploit data and work sharing opportunities across queries during the process of query evaluation, at the same time, improves intra-query parallelism. An experiment was conducted to demonstrate the effectiveness of the data flow model of our choice by comparing it with the iterator model employed in IGNITE. The experiment result confirms our choice: the data flow model achieves 40% less query completion time than the iterator model.

We also developed a Merge-Partition query reconstruction algorithm in a μEngine (Harizopoulos et al., 2005) for processing sub-queries. The proposed reconstruction algorithm is able to exploit data sharing opportunities among the concurrent sub-queries, which can reduce the average communication overhead required by the sub-queries and can therefore improve the overall query performance. Two experiments have been designed and conducted to evaluate the effectiveness of our Merge-Partition algorithm by comparing it with two approaches: IGNITE having the merge algorithm used in μEngine-W and an original approach where μEngine-W processes sub-queries without using any query reconstruction mechanism, when processing a group of parameterized and non-parameterized queries, respectively. The results show that, by applying query reconstruction mechanism with our Merge-Partition algorithm, communication overhead of executing sub-queries can be reduced, and performance of DIS queries can be improved correspondingly. More detailedly, compared with the original approach, our MP approach can reduce the average completion time of queries by more than 48% and 35% for parameterized and non-parameterized queries, respectively. Compared with IGNITE, MP can reduce the average completion time of all the queries by 32% and 25% for parameterized and non-parameterized queries, respectively.

## Acknowledgement

This Work is supported by Natural Science Foundation of China (60803121, 60773145, 60911130371, 90812001, 60963005), National High-Tech R&D (863) Program of China (2009AA01A130, 2006AA01A101, 2006AA01A108, 2006AA01A111, 2006AA01A117), MOE-Intel Foundation and Tsinghua National Laboratory for Information Science and Technology (TNLIST) Cross-discipline Foundation.

## References

Ahad Rafiul, Bapa Rao KV, Mcleod Dennis. On estimating the cardinality of the projection of a database relation. ACM Transactions on Databases 1989;14(1):28–40.

Dalvi Nilesh N, Sanghai Sumit K, Roy Prasan, Sudarshan S. Pipelining in Multi-Query Optimization. In PODS 2001.

Deshpande Amol, Ives Zachary, Raman Vijayshankar. Adaptive query processing. Foundations and Trends in Databases 2007;1(1):1–140.

Fontes V, Schulze B, Dutra M, et al. CoDIMS-G: a data and program integration service for the grid. In: Proceedings of the 2nd workshop on Middleware for grid computing. Ontario, Canada, 2004. p. 29–34.

Graefe Goetz. Query evaluation techniques for large databases. ACM Comput Surv 1993;25(2):73–170.

Getoor Lise, Taskar Benjamin, koller Daphne. Selectivity estimation using probabilistic models. SIGMOD 2001.

Goldstein J, Larson P. Optimizing queries using materialized views: a practical, scalable solution. SIGMOD 2001:331–342.

Gounaris Anastasios. Resource aware query processing on the grid. PhD thesis, School of Computer Science of the University of Manchester, 2005.

Harizopoulos S, Shkapenyuk V, Ailamaki A. QPipe: a simultaneously pipelined relational query engine. In: Proceedings of SIGMOD, 2005.

Alon Halevy, Rajaraman A, Ordille J. Data integration: the teenage years. In: Umeshwar Dayal, Kyu-Young Whang, editors. Proceedings of the VLDB'06. Seoul, Korea, 2006. p. 9–16.

Haas LM, Kossmann D, Wimmers EL, Yang J. Optimizing queries across diverse data sources. In: Proceedings of VLDB, 1997. p. 276–85.

Ives Z. Efficient query processing for data integration. PhD thesis, University of Washington, 2002.

Kossmann Donald. The state of the art in distributed query processing. ACM Comput Surv 2000;32(4):422–69.

Lee Rubao, Zhou Minghong, Liao Huaming. Request window: an approach to improve throughput of rdbms-based data integration system by utilizing data sharing across concurrent distributed queries. In: Christoph Koch, Johannes Gehrke, editors. Proceedings of the VLDB'07. Vienna, Austria, 2007. p. 1219–30.

Lynden Steven, Mukherjee Arijit, Hume Alastair C, et al. The design and implementation of OGSA-DQP: A service-based distributed query processor. Future Generation Computer Systems 2009;25(3):224–36.

Narayanan S, Kurc TM, Saltz J. Database support for data-driven scientific applications in the grid. Parallel Processing Letters 2003;13(2):245–71.

OGSA-DAI, 〈http://www.ogsadai.org.uk/〉.

Sacco GM, Schkolnick. M. Buffer management in relational database systems. ACM TODS 1986;11(4):473–498.

〈http://www.tpc.org/tpch/〉.

〈http://osdldbt.sourceforge.net/〉.