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