Request Window: an Approach to Improve Throughput of RDBMS-based Data Integration System by Utilizing Data Sharing Across Concurrent Distributed Queries

Rubao Lee, Minghong Zhou, Huaming Liao
Research Centre for Grid and Service Computing Research,
Institute of Computing Technology, Chinese Academy of Sciences
P.O. Box 2704, Beijing, China
{lirubao,zmh,lhm}@software.ict.ac.cn

ABSTRACT

This paper focuses on the problem of improving distributed query throughput of the RDBMS-based data integration system that has to inherit the query execution model of the underlying RDBMS: execute each query independently and utilize a global buffer pool mechanism to provide disk page sharing across concurrent query execution processes. However, this model is not suitable for processing concurrent distributed queries because the foundation, the memory-disk hierarchy, does not exist for data provided by remote sources. Therefore, the query engine cannot exploit any data sharing so that each process will have to interact with data sources independently: issue data requests and fetch data over the network.

This paper presents Request Window, a novel DQP mechanism that can detect and employ data sharing opportunities across concurrent distributed queries. By combining multiple similar data requests issued to the same data source to a common data request, Request Window allows concurrent query executing processes to share the common result data. With the benefits of reduced source burdens and data transfers, the throughput of query engine can be significantly improved. This paper also introduces the IGNITE system, an extended PostgreSQL with DQP support. Our experimental results show that Request Window makes IGNITE achieve a 1.7x speedup over a commercial data integration system when running a workload of distributed TPC-H queries.

1. INTRODUCTION

With the rapidly increasing application requirements of integrating remote data objects from various distributed, heterogeneous, and autonomous data sources, traditional RDBMS are extended to support distributed query processing [16], such as the extension to DB2 [15] and to the SQL Server [1]. Such DQP extensions deliver RDBMS-based data integration systems that can reuse the existing RDBMS components, including access interfaces, query optimizer and execution engine, with necessary modifications and extensions.

VLDB '07, September 23-28, 2007, Vienna, Austria.

Typically, the RDBMS-based data integration system employs a wrapper architecture that allows various data sources to be wrapped and to be plugged into the system. Moreover, the query engine utilizes a new operator that is response for interacting with wrappers to issue data requests and fetch results, such as the SHIP operator in extended DB2. For a distributed query, the corresponding query plan tree will be constructed with such operators as leaf nodes, so that the query engine can execute a distributed query as a common query. Meanwhile, the distributed query optimizer is added on the basis of the existing optimizer to improve the query execution performance.

In addition, facing the evolution from disk-oriented RDBMS to network-oriented data integration system, academic researchers have developed many new algorithms for relational operations, especially for the join operation, such as XJoin [30], MJoin [31], and Hash-Merge-Join [22]. These algorithms aim at improving join performance including initial delay and total response time on slow/bursty network transfers. To our knowledge, however, these algorithms are not implemented practically in current RDBMS-based data integration systems. Inside the systems, traditional disk-oriented join algorithms are still being used even for processing distributed queries.

For RDBMS-based data integration system, no prior work considers the throughput problem: "how to execute concurrent multiple distributed queries more efficiently to improve the overall throughput?" To address this problem, it is not enough to only consider new techniques in query optimizer [10] and new operation algorithms, which aim at making a single query be executed faster and more efficiently. The improvement of overall throughput greatly relies on data sharing across concurrent queries, which is more necessary in data integration system than in traditional RDBMS, since real data are actually stored in remote data sources and have to be transferred to the query execution engine over the network.

Unfortunately, the query execution model of RDBMS-based data integration system, which is inherited from the underlying RDBMS and can be expressed as "executing each query independently" as pointed out in the paper [9], makes it difficult to provide data sharing across concurrent distributed queries. Existing data sharing mechanisms in RDBMS are not suitable for executions of distributed queries. The query execution model in RDBMS is to execute concurrent queries independently and to employ a global buffer pool manager to share disk pages across queries. To improve the utilization of main memory and the performance of query processing, researchers have developed

various cache replacement algorithms [13][21][23][26]. Despite their concrete differences in the replacement policy, all these algorithms follow the same idea: "to provide run-time data sharing on the basis of the memory-disk hierarchy." Unfortunately, when processing distributed queries, the global buffer pool mechanism in RDBMS cannot be directly used because its foundation, the memory-disk hierarchy, is bankrupted. In typical data integration scenarios, a data source may only support a data access interface that takes an SQL statement as the input and returns the result tuples. In this way, random access to the data source like access to a disk is not supported.

Being different from page caching and tuple caching, the semantic caching technique, which manages client caches as a collection of semantic regions [5][14], is used in various distributed data applications, such as web querying, application servers, etc. However, to be a general method for providing data sharing across queries in data integration systems, semantic caching is facing the following difficulties:

(1) In some data integration scenarios, a data source may not allow clients to cache its data for some reasons related to copyright and policy. Under such limitations, the only way to obtain data from the data source, whenever a client needs the data, is to send data requests to the data source. Moreover, the client must discard all data after consuming them. [1]

(2) Another difficulty for caching data of sources lies in the fact that data sources are likely autonomous and they can change their data without any notice. More importantly, a data source may provide no support for data synchronization between its own data storage and copies on the client sides. This is very common in service-oriented applications. In this situation, the maintenance work for data cache consistence is a very big challenge.

In this paper, we solve the DQP throughput problem using a novel approach: Request Window, which can detect and exploit data sharing opportunities across multiple concurrent distributed queries. The main idea behind Request Window is to remove unnecessary data requests by combining multiple similar data requests sent to the data source to a single common request and dispatching the common result data returned from the data source. Although Request Window is a batch processing technique, which processes a group of data requests as a batch, it does not require that multiple queries must enter the system simultaneously. The key is the DIOP (**Delay Indicated by OPtimizer**) technique we present in this paper. By using the DIOP technique, each data request generated when executing a query has a tolerable delay time, which makes it possible to construct a group of such data requests. Moreover, Request Window utilizes the DAW (**Dynamically Adjusting Window**) technique to adjust the window size dynamically, i.e. to determine when to process a group of data requests as a batch. Request Window can improve throughput of executing concurrent queries without sacrificing the response time of individual query by utilizing the DIOP technique and the DAW technique.

This paper also introduces the IGNITE system, an extended PostgreSQL [24] with DQP support. Our experimental result shows that Request Window makes IGNITE achieve a 1.7x speedup over a commercial data integration system when running a workload of distributed TPC-H queries.

In summary, this paper presents the first study of how to utilize data sharing opportunities across concurrent queries to improve the DQP throughput. We make the following contributions:

- We formalize the problem of data sharing across concurrent queries in RDBMS-based data integration system.
- We present a classification of related data sharing techniques employed in traditional RDBMS and in data integration systems.
- We introduce a set of query execution techniques to provide data sharing across queries including the Start-Fetch wrapper architecture, the Request Window mechanism, the DIOP technique, and the DAW technique.
- We describe an implementation of the IGNITE system, a relational data integration system based on the PostgreSQL RDBMS.

The remainder of this paper is organized as follows. Section 2 discusses various related work and presents a classification of data sharing techniques. Section 3 introduces the IGNITE system and the Start-Fetch wrapper architecture that provides the foundation of Request Window. In Section 4, we describe Request Window in detail and illustrate the DIOP technique and the DAW technique. Section 5 carries out various experimental results. In section 6, we give further discussions about Request Window. We conclude this paper and introduce our future work in Section 7.

2. RELATED WORK & CLASSIFICATION

This section first briefly introduces related work in various contexts and then presents a classification of four data sharing techniques.

2.1 Related Work

As we discussed in the Introduction section, several commercial database products are extended to provide functionalities of data integration, for example IBM DB2 [15] and Microsoft SQL Server [1]. To our knowledge, these systems currently provide no data sharing across distributed queries.

More closely related to this paper is the research work on the data and work sharing in traditional RDBMS. A recent paper is about the QPipe query engine in [9], which introduces a new "one-operator, many-queries" query execution model to replace the traditional "one-query, many-operators" model. The QPipe engine can provide not only sharing of table scans but also sharing of common computations across concurrent queries. In addition, several RDBMS products provide similar sharing of table scans, for example SQL SERVER [3], RedBrick Data Warehouse [6], and Teradata [32].

Multi-query optimization techniques in RDBMS [4][25][27][33] are related to this paper considering the common goal of data sharing across queries. The main idea of multi-query optimization technique is to process a group of queries that contain common subexpressions and to produce a globally optimal plan. To our knowledge, no prior work considers multi-query optimization in data integration context.

Ideas on obtaining maximized throughput by delaying processing of data requests exist in OS research fields, which are similar to Request Window. In [18], an anticipatory scheduling framework is proposed to solve the problem of deceptive idleness in disk scheduling to improve throughput of the disk subsystems for concurrent disk-intensive applications.

[1] This is a real requirement we met when we applied our system to a National Railway Information Grid project in China.

2.2 Classification of Data Sharing Techniques

In this subsection, we compare Request Window with other three data sharing techniques in traditional RDBMS (details are following) and present a classification as the comparison result. Although these techniques have different problem contexts, their goal is same: providing data sharing across queries. In our classification, we consider two correlative factors. The first one is the restriction on the arrival times of different queries across which data sharing can be possible, and the second one is the maximized amount of data that can be shared among multiple queries when data sharing is possible.

We take an example to illustrate the total four categories of data sharing techniques. Consider two identical queries Q1 and Q2: "*select * from a_table*". In the query statement, "*a_table*" may be a local relation or a remote relation, which determines the query to be a traditional query in RDBMS or a distributed query in a data integration environment. We assume that Q1 and Q2 may arrive at the query processing engine simultaneously, or that Q2 may arrive later than Q1. In the example, the first factor means the difference between the arrival times of Q1 and of Q2, while the second factor means the amount of data shared between Q2 and Q1.

The first category of data sharing techniques is the multi-query optimization techniques utilized in traditional RDBMS [4][25][27][33]. Multi-query optimization is a technique working at query compilation phase. The major problem that multi-query optimization solves is how to find common subexpressions and to produce a global-optimal query plan for a group of queries. The multi-query optimization technique has the most restrictive requirement on the arrival times of different queries due to the limitation that multiple queries must be optimized as a batch. Such the limitation means that multiple queries must enter the query engine simultaneously in order to be optimized together, otherwise queries arrived earlier have to be sacrificed to wait for other queries. However, the multi-query optimization technique can provide maximized capabilities of data sharing across queries once multiple queries are optimized as a batch. More importantly, multi-query optimization can provide not only data sharing but also common computation sharing.

The second category of data sharing technique is Request Window presented in this paper. What should be noted is that, although Request Window is mainly proposed for DQP, it can also be employed in traditional RDBMS to provide data sharing. Request Window is a technique working at the query execution phase. Its basic idea is to combine similar requests for the same data source to a single common request and then to dispatch the results received from the data source to multiple independent query execution processes. In this way, each query engine process can share the common results. In the example, data requests generated by processing Q1 and Q2 will be combined to a common data request, and the query engine processes for Q1 and Q2 can share all the results of the common data request received from the data source. The restriction on arrival times of queries in Request Window is looser than that in multi-query optimization as illustrated later in this paper. This is because that some data requests sent to the data source when executing complex queries may have a delay opportunity so that it is tolerable to wait for other similar data requests. For this category, the amount of data shared depends on the difference of arrival times of queries.

The third category of data sharing techniques is table scan sharing techniques [3][6][32]. The main idea of scan sharing is that new scan request for the same table can "piggyback" on an existing scan process. In the example, when Q2 arrives, it can directly obtain tuples from the output of existing Q1' scan process. However, because Q2 misses some tuples that have already been consumed before Q2's arrival, a new partial scan will have to be restarted for Q2 to fetch the missed tuples. Scan sharing techniques have a loose restriction on arrival times of queries as long as a previous scan process is still ongoing. For example, Q2 can share Q1's scan output as long as Q1 is not finished. Like Request Window, the amount of data shared depends on the difference of arrival times of queries. Although it works well in RDBMS, using table scan sharing in DQP faces a big challenge. It may be hard to fetch missed tuples for later queries from the data source exactly because the data source may not promise an identical response for even two same data requests.

The fourth category of data sharing techniques is the widely used cache mechanism in RDBMS [13][21][23][26]. This is a kind of low-level page-based data sharing techniques. Such data sharing has no restriction on the arrival times of queries. In the example, even Q1 has already finished, Q2 can still benefit from pages in main memory placed by the execution of Q1 so that unnecessary I/O operations can be removed. However, the amount of data shared is unsure and largely dependent on the concrete cache replacement policy and run-time situations. Because of the lack of memory-disk hierarchy, the cache mechanism is not suitable for providing data sharing when processing data integration queries.

Finally, we summarize our classification in Table 1.

3. IGNITE & START-FETCH WRAPPER

In this section, we first present an overview of IGNITE, our extended PostgreSQL with DQP support. Next, we introduce the Start-Fetch wrapper architecture in IGNITE, which is the base of Request Window.

Table 1: Classification of Data Sharing Techniques

Category	Multi-Query Optimization	Request Window	Table Scan sharing	Cache Mechanisms
Application scope	RDBMS or Data Integration	RDBMS or Data Integration	RDBMS	RDBMS
Restriction on interarrival times	Most restrictive Arrivals must be simultaneous.	Less restrictive No requirement for simultaneousness	Ongoing scans must exist	No restriction
Amount of shared data	Maximized (including working sharing)	dependent on interarrival times	dependent on interarrival times	Unsure and dependent on concrete policy

3.1 Overview of IGNITE

The IGNITE system is a relational data integration system that provides a collection of virtual views for integrating relational data objects from various distributed, heterogeneous, and autonomous data sources. The implementation of IGNITE is on top of PostgreSQL, which is similar to the distributed extensions to IBM DB2 [15] and Microsoft SQL Server [1]. Because PostgreSQL is a traditional RDBMS that has no built-in distributed query processor, to extend it to a data integration system, we have to solve several problems about the system architecture, user interface and query execution performance. We present an overview of IGNITE as follows.

The cornerstone of IGNITE is the function mechanism provided by PostgreSQL [28]. We build a wrapper framework to enable various data sources to plug into IGNITE. A wrapper in IGNITE consists of the implementations of several pre-defined function interfaces, which are responsible for providing metadata of the data source, executing a data request in the data source, obtaining statistics information from the data source, etc. Via a wrapper, a real relation in the data source can be registered into IGNITE to be a virtualized view that can then be queried with no difference from common tables.

Although the function mechanism and the wrapper framework enable data integration functionalities, the performance of distributed query processing could not be improved if we do not modify the underlying query execution engine of PostgreSQL. Our modifications involve several aspects, including (a) the re-implementation of a pipelined FunctionScan operator, (b) the implementation of query shipping [7] (we currently implement functionalities for pushing down three operations: projection, selection, and sorting), and (c) the implementation of an improved optimizer which can utilize various statistics information of data sources with the help of wrappers.

Several limitations in the architecture and the implementation of PostgreSQL, including the process-per-connection model and single-threaded query engine implementation that is not thread-safe [2], make it difficult to provide data sharing and improve throughput when processing concurrent distributed queries in IGNITE. Solving this problem is the major motivation of this paper. In the IGNITE system, we overcome these limitations by utilizing the Start-Fetch wrapper architecture and implementing the Request Window mechanism. We only make necessary modifications to the core of the PostgreSQL and implement additional features in the wrappers.

For the experiments of this paper, we develop an IGNITE wrapper for the data source which is a PostgreSQL database using the libpq library. This wrapper is an independent multi-threaded application implemented using the pthread library on a POSIX platform, which conforms to the design of the Start-Fetch wrapper and the Request Window mechanism. We utilize the shared-memory and UNIX domain socket as the IPC mechanisms between this wrapper and the query engine process. The details of the architecture and performance evaluation of IGNITE can be found in [17].

3.2 Start-Fetch Wrapper

From the point of view of the underlying PostgreSQL, each wrapper in IGNITE is only a collection of specific functions. IGNITE uses a special R_Scan operator as the bridge between the query engine and the wrappers. The R_Scan operator follows the iterator model [8] and implements the Open, Next and Close iterator functions. Because the backend of PostgreSQL is single-threaded, the wrapper code will be executed within the same process of the query engine for a query. Therefore, no prefetching can be available in wrappers, and executions of multiple wrappers will have to be synchronized.

To decouple wrappers from the query engine, we present the Start-Fetch architecture for wrappers. The Start-Fetch wrapper employs a "multi-process" model: to implement the wrapper functionalities in a separate process and to employ some kind of inter-process communication mechanisms to connect the query engine process and the wrapper process at execution time. The multi-process model enables parallelized execution between a query engine process and a wrapper process, and there exists a consumer-producer relationship between them. When the Open function of the R_Scan operator is invoked, the query engine process sends a data request to the wrapper process, and the wrapper process must return a "ticket" to the query engine process as the response immediately. Then the wrapper process needs to send the data request to the underlying data source and receive results, which occurs independently in its own process. This is the "Start" step of Start-Fetch. When the Next function of the R_Scan operator is invoked, the query engine process asks for next tuple from the wrapper process using the ticket obtained at the "Start" step. This is the "Fetch" step of Start-Fetch.

One benefit of Start-Fetch is that the wrapper process can prefetch more tuples from the data source while the query engine process is consuming some tuples. The parallelized execution can accelerate query processing. However, the overhead of communications between the wrapper process and the query engine must be reduced to as low as possible.

Besides the ability of parallelized execution, another big advantage of Start-Fetch is that the multi-process model makes data sharing across queries to be possible. To enable data sharing across multiple independent query engine processes, all data requests must be submitted to a common place. In traditional RDBMS, the buffer pool manager can provide functionalities of such a common place. In a data integration system, the decoupled wrapper process is actually such a common place that is responsible for receiving data requests from multiple independent query engine processes and dispatching results received from the data source.

Note that, the goal of data sharing across queries could not be achieved only by employing the Start-Fetch architecture. A Start-Fetch wrapper only provides the foundation for implementing the Request Window mechanism that is the key enabling technique.

4. REQUEST WINDOW

4.1 Overview

In the Start-Fetch wrapper architecture, multiple query engine processes will independently issue data requests for the same data source to a common wrapper process. The default action of the wrapper process is to deal with each data request independently, i.e. sending it to the data source, receiving result tuples from the data source, and finally returning all result tuples to the corresponding query engine process. This execution model does not exploit any data sharing opportunity across multiple data requests.

To provide data sharing across multiple query engine processes, we present a new batch processing technique working in a Start-Fetch wrapper: the Request Window mechanism. The main idea

behind the mechanism is to process multiple relative data requests as a batch in the wrapper instead of processing each of them independently. To implement the batch processing, the wrapper combines a group of similar data requests to a common data request and only sends the common data request to the data source. Then, the wrapper will dispatch corresponding result tuples for the common data request returned by the data source to each participating query engine process. By doing this, multiple query engine processes can share common result data so that the burdens of the data source for processing data requests can be reduced and the amount of result data transferred over network can be reduced.

When a query engine process submits a data request to the wrapper process, instead of instantly sending the request to the data source, the wrapper will add the new request to a corresponding waiting queue that stores each request and the corresponding query engine process. If no corresponding waiting queue exists, then the wrapper will first create an empty queue. We call such a waiting queue a "request window." At a time, the wrapper will process the waiting queue. We call this action "window issue" and call the processing time the "window issue time." The window issue action involves several steps: (a) combining all data requests in the waiting queue to a single SQL statement, (b) sending the SQL statement to the underlying data source, (c) receiving results from the data source, and (d) dispatching result tuples to each query engine process. Of course, the last two steps can be executed in a pipelined way. After all query engine processes have received all results, the window will be destroyed. Figure 1 shows an illustration of the request window technique.

Currently in our implementation, each data request is a SQL statement which has the form of "*select (columns) from a_table where (predicate)*". The common data request generated by combining multiple such statements has a synthesized "*where*" clause. The result tuples for the common data request contains all tuples needed by each participating query engines. The result dispatcher will only dispatch corresponding tuples to each query engine process. It means that the query engine process will never receive unnecessary tuples that cannot pass the filter for the request.

We take an example to illustrate the request window mechanism. At a time the wrapper receives the data request Q1 (*"select * from a_remote_table where key > 10"*) from a query engine process e1; and later, the wrapper receives Q2 (*"select * from a_remote_table where key > 20"*) from another query engine process e2. For the two data requests, the wrapper performs the following actions.

- It creates an empty request window and adds Q1 in the window when it receives Q1.
- It adds Q2 into the window when it receives Q2.
- It combines Q1 and Q2 to a common SQL statement Q3 (*"select * from a_remote_table where key > 10"*) when the issue time of the window arrives. Note that, Q3 is as same as Q1 because the query result of Q1 contains the result of Q2.
- It sends Q3 to the corresponding data source and receiving result tuples.
- It dispatches corresponding result tuples to e1 and e2.

A natural question that will be asked is when the wrapper should issue a request window. We define the "window size" concept for a request window as the interval between the time for creating the window and the time for issuing the window. The key point of the request window mechanism is to determine the window size.

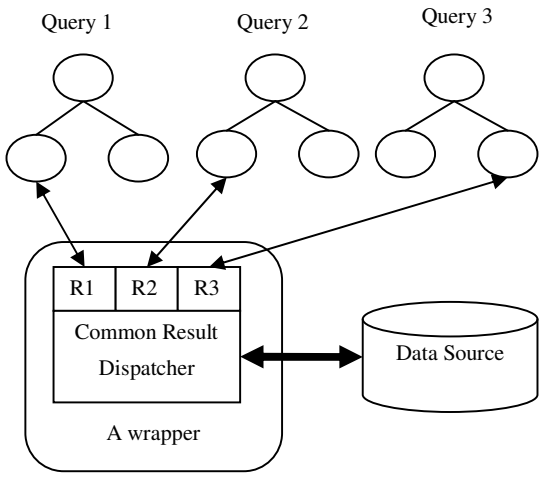

Figure 1: An architecture overview of the Request Window. In a Start-Fetch wrapper, multiple similar data requests will be combined to a common data request so that the underlying data source only needs to process the common request. The common results will be dispatched to each participating query engine process.

Intuitively, the larger the window size is, the more requests the window may contain, and the more data can be shared. However, for a large window size, early requests have to wait for a long time so that their response times will be increased. To solve this problem, we present a solution that consists of two techniques. The first one is the DIOP technique, which is utilized to determine the maximized delay time of each data request. The second one is the DAW technique, which is utilized to adjust the window size dynamically according to the maximized delay time of each new data request.

4.2 DIOP: Delay Indicated By OPtimizer

The key to determine the size of a request window is to determine how long each data request can be delayed by the wrapper process without increasing the total response time of the query execution. According to the Start-Fetch model, the wrapper will receive a data request when the Open function of the corresponding R_Scan operator is invoked (the Start step) and will provide tuples when the Next function is first invoked (the Fetch step). The interval between the Start step and the Fetch step provides the opportunity for the wrapper to delay the data request. Intuitively, the wrapper must at least have one result tuple available when the Fetch step begins, otherwise the query engine will have to be blocked by waiting for results. Therefore, to determine the maximized delay time of a data request, two factors must be determined: (a) the interval between the Start step and the Fetch step, and (b) the time for the wrapper to obtain the first tuple from the data source.

We present the DIOP (Delay Indicated by OPtimizer) technique. By using DIOP, the delay time of a data request is not determined by the wrapper process blindly but by the query optimizer. For each data request, the query optimizer will make an estimation of its tolerable delay time. When the query engine process submits the data request to the wrapper, the engine will additionally tell the wrapper the estimated delay time for the data request. Therefore, each data request received by the wrapper has an annotation of its maximized delay time. In addition, the wrapper will dynamically adjust the window size according to the

annotation value of each new data request, which is the job of the DAW technique discussed in section 4.3. Next, we explain why the delay time of a data request can exist and introduce how the query optimizer makes the estimation for the delay time.

4.2.1 Opportunities for Delaying a Data Request

The foundation on which a data request can be delayed is that there is an available interval from the time when the data request is created to the time when the results of the request begin to be consumed. According to the iterator model for executing a query plan tree, such intervals exist and can be estimated by the query optimizer when it creates the query plan tree.

When the query engine begins to execute the query, it will first invoke the Open function of each node in the query plan tree recursively. When the Open function of a leaf node is invoked, a corresponding data request is created. However, the Next function of a node will not be invoked until its parent node begins to consume the output tuples of the node. This is the nature of on-demand data consuming of the iterator model. According to the implementation algorithm of the parent node, it is possible that data consuming of one child node will not happen until some event occurs, such as the one that another child node has outputted all tuples. Such event dependency in the query plan tree may constitute a critical path diagram. In the diagram, a data request can be delayed as late as possible until its dependent events occur.

We consider an example of the hash join operator that consists of two phases: (1) building the hash table using all tuples from the left child node and (2) probing the hash table using each tuple from the right child node. For the right child node, its Next function will not be invoked until the building phase is completed. Therefore, the underlying wrapper for the right child node can delay the data request received when the Open function is invoked for a while.

4.2.2 Estimating the Maximized Delay Time

Now, we formalize the problem as:

For a given data request R generated by a leaf node N in the query plan tree, how to determine the "Maximized Delay Time" of the data request: MDT_R^N , i.e. the interval between the time when the corresponding wrapper receives the data request to the time when the wrapper sends the data request to the data source?

We define several concepts as follows to solve this problem.

Definition 1*: "Begin Time" of a query plan tree:* BT_Q

The "Begin Time" of a query plan tree is the time when the query plan tree is executed. Considering that execution of the Open function of each node in the tree is very fast[2], we can think that the time when the Open function of each node is invoked is equal to the Begin Time of the query plan tree. This time is also the time when each wrapper process receives corresponding data request[3], i.e. the time of the Start step of each wrapper.

[2] In PostgreSQL, the Open function of each node in the query plan does not execute too much code. Even for the Sort operator, the real sorting code will be executed when the Next function is first invoked.

[3] Actually, we can guarantee this by first sending all involved data requests to corresponding wrappers when the query plan is executed without keeping waiting until each Open function is invoked.

Definition 2*: "First Fetch Time" of a node N:* FFT_N

The "First Fetch Time" of a node N in a query plan tree is the time when the Next function of node N is first invoked. For a leaf node, this time is the begin time of the Fetch step for the corresponding wrapper. In addition, we assume that the First Fetch Time of the root node is equal to the Begin Time of the query plan tree.

Definition 3*: "Wait Opportunity" of a node N:* WO_N

The Wait Opportunity of a node N in a query plan tree is the interval from the Begin Time of the tree to the First Fetch Time of node N. It can be expressed using the following formula:

$$WO_N = FFT_N - BT_Q$$

Definition 4*: "Initial Delay" of a data request R:* ID_R

The "Initial Delay" of a data request R is the interval from the time when the wrapper sends the data request R to the data source to the time when the wrapper receives the first tuple returned by the data source.

According to the above definitions, the Maximized Delay Time of a data request R genereated by a node N can be calculated according to the following formula:

$$MDT_R^N = WO_N - ID_R$$

If the result is a negative, then we set it to zero. Now, the question becomes the one of how to estimate the Wait Opportunity of a leaf node. We make the following definition.

Definition 5*: "Algorithm Related Delay" of a node N:* ARD_N

For the root node, the "Algorithm Related Delay" is zero. For a non-root node N, the "Algorithm Related Delay" is the interval from the First Fetch Time of the parent node P of node N to the First Fetch Time of node N. It can be expressed using the following formula:

$$ARD_N = FFT_N - FFT_P$$

By the definition, the Algorithm Related Delay of a node is the elapsed period from the time when the Next function of its parent node is first invoked to the time when the Next function of its own is first invoked. How long this interval will be is determined by the implementation algorithm of its parent node, so different operators have different Algorithm Related Delays. For a hash-join operator, the Algorithm Related Delay of the left child node (building hash table) is zero because the parent join node will instantly fetch tuples from the child once the join begins, while the Algorithm Related Delay of the right child node (probing hash table) is the elapsed time for finishing the hash table building.

Now, we can get the following formulas to recursively calculate the Wait Opportunity of each node.

$$WO_N = 0 \; if \; node \; N \; is \; root$$

$$WO_N = WO_P + ARD_N \; if \; node \; N \; is \; not \; root$$

4.2.3 Examples to Illustrate DIOP

Next, we provide two examples to illustrate the estimation of Wait Opportunity for each node.

The first example is about a query plan tree shown in Figure 2. It is a right-deep join tree for a star-join among three relations:

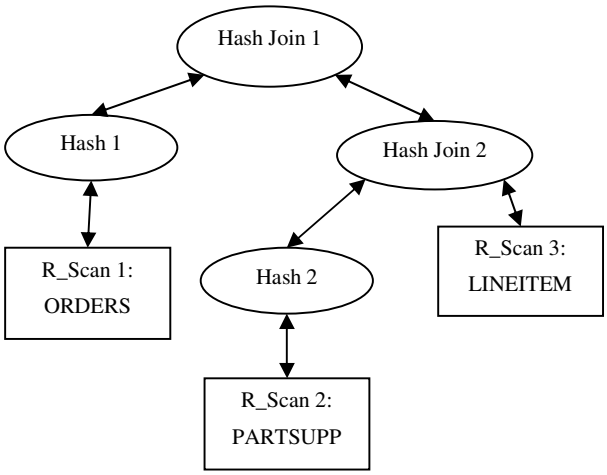

Figure 2: a right-deep hash-join tree for join of three relations

$$ORDERS \bowtie LINEITEM \bowtie PARTSUPP .$$

According to the above formulas, we can know that:

(1) The Wait Opportunity of the leaf node "R_Scan 1" is equal to 0.
(2) The Wait Opportunity of the leaf node "R_Scan 2" is equal to the time for finishing the node "Hash 1".
(3) The Wait Opportunity of the leaf node "R_Scan 3" is equal to the sum of the time for finishing the node "Hash 1" and the time for finishing the node "Hash 2".

The second example is about a query plan tree shown in Figure 3. The tree is for a union operation of three relations:

$$ORDERS1 \bigcup ORDERS2 \bigcup ORDERS3 .$$

In the tree, the root node is an Append operator (PostgreSQL's implementation for union operation) which is executed with three distinct stages in turn for feching tuples of the three child nodes respectively.

According to the above formulas, we can know that:

(1) The Wait Opportunity of the leaf node "R_Scan 1" is equal to 0.
(2) The Wait Opportunity of the leaf node "R_Scan 2" is equal to the time for finishing the node "R_Scan 1".
(3) The Wait Opportunity of the leaf node "R_Scan 3" is equal to the sum of the time for finishing the node "R_Scan 1" and the time for finishing the node "R_Scan 2".

4.2.4 Approximate Estimation for Hash-Join Tree

According to the above formulas, estimation of the Wait Opportunity of each node is actually a recursive calculation process. However, for a query plan tree, for example the hash-join tree, we can utilize a more straightforward method to make approximate estimations. Actually we don't need accurate estimations since the purpose of DIOP is just to indicate a tolerable delay opportunity for a data request.

Because in typical data integration scenarios, the time for data transfer over the network dominates the whole query execution, we can ignore the time for local computations. This is the key of making approximate estimations.

In this subsection, we introduce an approximate estimation method for a hash-join tree. In a hash-join tree, only three kinds of nodes exist: the hash-join operator, the hash operator and the

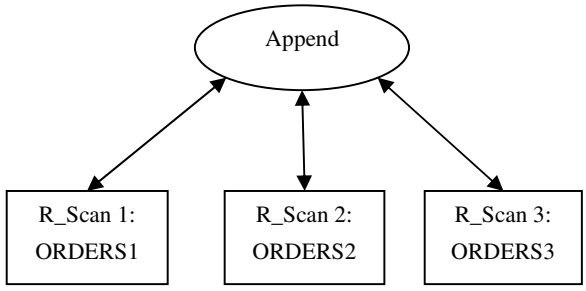

Figure 3: a query plan tree for unions of three relations

R_Scan operator at the leaf level. All wait opportunies in such a tree come from the fact that the hash operator is a blocking operator. It means that the hash operator cannot output any tuple before all tuples of its child are consumed for building the hash table. In this way, we can approximately estimate the time for finishing a hash node as the time for finishing its child node. And further, we can estimate this time using the total time for finishing all result data transfers of data requests generated by all the leaf nodes in the sub-tree under the hash node.

We first make the following definition and then present formulas for approximate estimations.

***Definition 6**: "Total Transfer Time" of a data request R: TTT_R*

The "Total Response Time" of a data request R is the interval from the time when the underlying wrapper receives R's first result tuple returned by the data source to the time when the underlying wrapper receives the last tuple returned by the data source. According to related definitions, we know that the time for finishing a data request is actually the sum of its Initial Delay and Total Transfer Time. To estimate the total transfer time, two factors are related. The first one is the data transfer speed which is affected by the communication speed and the tuple output speed of the data source. The second one is the number of result tuples which is affected by the cardinality of the remote relation and the selectivity of corresponding "where clause" in the data request statement. To obtain these values, the query optimizer needs the help of wrappers which provide values of predefined parameters and statistics information of underlying data sources.

To approximately estimate the Wait Opportunity of a leaf node in a hash-join tree, we traverse the query plan tree using an inorder traversal. After this, for each leaf node N, we obtain a set of data requests RS_N, which comprises all data requests generated by the leaf nodes that are before the node N in the traversal. We use the following formulas to approximately estimate the Wait Opportunity of a leaf node N:

$$RS_N = \{requests\ generated\ before\ N\ by\ inorder\ traversal\}$$

$$WO_N = \sum_{r \in RS_N} (ID_r + TTT_r)$$

Then, the corresponding Maximized Delay Time can be calculated using the formula:

$$MDT_R^N = \left(\sum_{r \in RS_N} (ID_r + TTT_r) \right) - ID_R$$

Further approximation is possbile if we can ignore the factor of Initial Delay when it is small compared with the Total Transfer Time. This requires that the data request must satisfy the following conditions:

(1) The target data source supports pipelined data fetch, for example, if the source is a RDBMS supporting resultset fetch via a cursor.
(2) The target data source can rapidaly return initial results. This depends on whether the target relation is a physical table.
(3) The tuple count of the resultset is large so that the Total Transfer Time is very long which is limited by the network speed.

By ignoring the initial delays, we get a formula for more approximate estimations:

$$MDT_R^N = \sum_{r \in RS_N} TTT_r$$

According to this approximate estimation method, we get the same estimation result for the query plan trees shown in Figure 2 and in Figure 4:

1) the Maximized Delay Time of the data request for ORDERS is 0.
2) the Maximized Delay Time of the data request for PARTSUPP will be the Total Transfer Time for ORDERS.
3) the Maximized Delay Time of the data request for LINEITEM will be the sum of the Total Transfer Time for ORDERS and the one for PARTSUPP.

4.2.5 Implementation Issues

There are two special considerations in implementing DIOP. We use two configurable parameters to allow IGNITE system administrators to make specific settings.

The first parameter is empolyed for those data requests which are estimated to have a very high selectivity, for example a data request which may only has one tuple in the result. Such a data request should not be processed by the request window mechanism for two reasons: (a) its total response time is very small so that any optimizing effort is unnecessary, and (b) it is unfair if the request is unfortunately placed in a window with another request which has a very small selectivity so that the corresponding query engine will have to select only a very small fraction of all tuples from the common result. Only data requests whose selectivity is smaller than the value of this parameter can be processed by the request window.

The second parameter is employed for data requests whose Maximized Delay Time is 0. The optimizer has a choice to reset the Maximized Delay Time of such a data reqeust to the value of the parameter. By doing this, even if a data request should not be delayed, the wrapper will still delay it to wait for other similar data requests. For a data request which will have a very long total transfer time (for example more than 100 seconds), it is tolerable if the request is delayed for only several seconds to wait for other similar requests. More importantly, many queries containing aggregations, for example the typical TPC-H queries in DSS workloads, are not concerned about the initial delay of query results. For systems busied by running concurrent such workloads, it is tolerable to set the value of this parameter to be higher in order to improve the overall throughput.

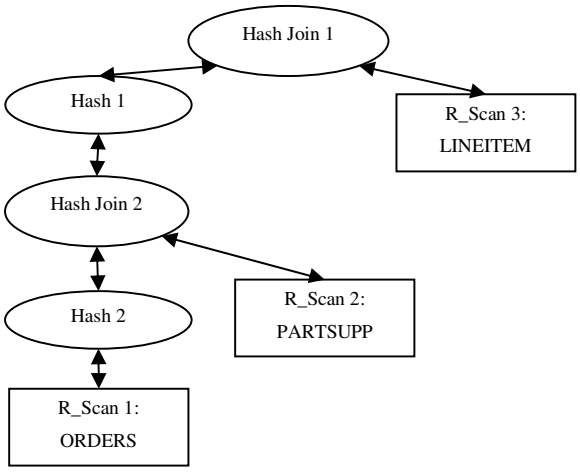

Figure 4: a hash-join tree for a star-join of three relations (compared with Figure 2)

4.3 DAW: Dynamically Adjusting Window

Remember that, our goal is to determine when to issue the window, i.e. to determine the window size. The DIOP technique provides hints to make the decision by annotating each data request with its maximized delay time. But, this is only the first step to achieve the goal. Because a request window, i.e. the waiting queue, contains multiple data requests which may have different delay times, a coordinator should be used to calculate the window size on the basis of delay times of all participating data requests.

We present the DAW (Dynamically Adjusting Window) technique to solve the problem of determining window size. The basic idea of DAW is to dynamically adjust the window size using some policy when a new data request with its annotation of maximized delay time is added into the window. The DAW technique consists of two components. The first one is the adjusting policy which specifies how to get a new window size after a new request is added into the window, and the second one is the adjusting executor which is triggered by the arrival of new data request. The adjusting executor will enforce the adjusting policy.

We first describe how the adjusting executor works. The window size is stored as an integer value which means the left time in seconds to issue the window. When the window receives a new data request, the wrapper will adjust this value by enforcing the policy which is implemented as a function hook. In addition, an independent daemon thread in the adjusting executor will wake up at the interval of one second. When the thread wakes up, it will check the current window size value. If it is zero, then the thread will start a new working thread to issue the window. Otherwise, it will shorten the current window size by one second.

The core of DAW is the adjusting policy. Whatever policy is employed, its goal is to determine a new window size according two basic inputs: the current window size and the Maximized Delay Time of the newcome data request. Intutively, we should use a policy which always keeps the smallest delay time to be the window size. In this way, the new window size should be the smaller one between the two input values. This policy can guarantee that no data request will be delayed beyond its tolerable maximized time. This is an emergency-oriented policy. However,

we can define a policy which can provide more data sharing by calculating a larger window size.

Now, we describe our adjusting policy. Related symbols are:

WS: current window size (in second)

RC: current number of data requests waiting in the window

MDT: Maximized Delay Time of the newcome data request

The policy distinguishs between two conditions.

1) If the Maximized Delay Time of the newcome data request is larger than or equal to current window size, then it doesn't change the current window size.

$$WS = WS \quad if\ MDT \geq WS$$

2) Otherwise, it use the following formula to calculate the new window size:

$$WS = \frac{WS \times RC + MDT}{RC + 1} \quad if\ MDT < WS$$

The principle of the policy consists of two points: (a) the window size will never be increased, and (b) the window size is the average of all the delay times of data requests in the window. This policy can provide more data sharing opportunities than the emergency-oriented policy by enlarging the window size using the average of all delay times as the window size instead of the smallest one. Moreover, according to this policy, the arrival of a later data request with long delay time will not cause that existing data requests in the window are delayed further.

Our proposed policy here is suitable for running DSS workloads, such as the TPC-H queries, which are not concerned about the initial delays of query results.

5. PERFORMANCE EVALUATION

5.1 Experimental Setup

In this section, we present the experimentation with IGNITE. Considering no common data integration query benchmark [12], we use a TPC-H [29] (scale factor 0.1, 100MB) database as the dataset of our experiments. The goal of our experiments is to examine how the request window can improve the performance and throughput when executing distributed TPC-H queries in IGNITE by providing data sharing across queries.

A TPC-H database has eight relations (REGION, NATION, CUSTOMER, SUPPLIER, PART, PARTSUPP, ORDERS, and LINEITEM). Therefore, we use eight data sources, and each of them is responsible for providing one relation respectively. Each data source is actually a PostgreSQL 8.1 RDBMS running on a 2.8GHz Intel P4 machine with 768MB of RAM, running the FreeBSD 5.4 stable operating system. Our IGNITE system is running on a SMP machine with four Intel P4 Xeon 2.4GHz CPUs, 2GB of RAM, running Linux 2.4.18 SMP. All the machines are connected on a 100Mbit/sec Ethernet.

In our experiments, each relation from a data source is registered into the IGNITE system to be a view via several extended SQL statements. For example, the real table LINEITEM in a data source is actually mapped to the view V_LINEITEM in IGNITE. Each query issued to IGNITE in our experiments is actually executed over these virtual views. For our experiments, we force the query optimizer to choose hash-join for join parts in the query plans.

To avoid introducing additional client-server communication overhead, we discard all result tuples of executing queries. In addition, all experiments are run a minimum of four times. In all of the graphs, the "Baseline" represents the IGNITE system without enabling the Start-Fetch wrapper and the Request Window mechanism.

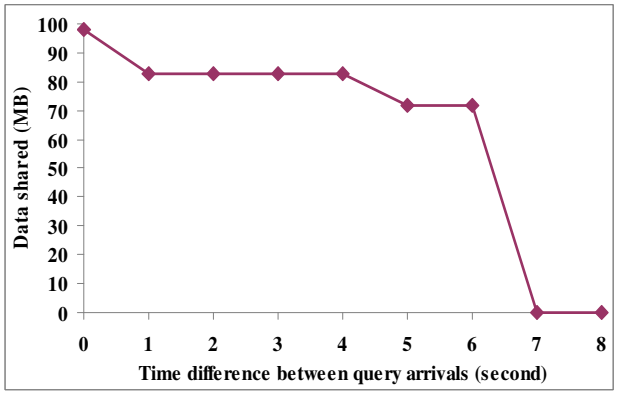

Figure 5: The amount of data shared between two queries when increasing the interarrival times up to 8 seconds.

5.2 Sharing Transferred Data

In this experiment, we examine how the request window mechanism can provide data sharing when processing two identical queries (say Q1 and Q2 respectively) concurrently in IGNITE. We vary the arrival time of the second query later than the first one from 0 second to 8 seconds and check how much data can be shared between these two queries. The query is:

*select * from V_ORDERS, V_PARTSUPP, V_LINEITEM where O_ORDERKEY = L_ORDERKEY and PS_SUPPKEY = L_SUPPKEY and PS_PARTKEY = L_PARTKEY.*

This query contains a star-join among three relations LINEITEM, ORDERS, and PARTSUPP. The query plan tree is actually shown in Figure 2.

The test results are shown in Figure 5. The vertical axis is the total data shared between Q2 and Q1, and the horizontal axis is the interarrival time. We can see that when the time difference between query arrivals increases, the amount of shared data decreases from the maximized about 100MB to zero.

We explain the results in detail. First, the maximized sharing opportunity between Q1 and Q2 is an interval of 6 seconds (0-6), which is actually the Wait Opportunity of the node R_Scan 3 for LINEITEM. Second, if these two queries arrive simultaneously, then they can share all data of tuples of ORDERS, PARTSUPP, and LINEITEM. Third, there is an opportunity of 4 seconds (0-4) for Q2 to share tuples of PARTSUPP and LINEITEM, which is time for finishing R_Scan 1 for ORDERS. Fourth, there is a last opportunity of 2 seconds (4-6) for Q2 to share tuples of LINEITEM, which is time for finishing R_Scan 2 for PARTSUPP.

5.3 Running Concurrent Same Queries

In this group of experiments, we examine how well the request window in IGNITE performs when running multiple concurrent same queries. We vary the number of concurrent clients from 1 to 12. We set the query interarrival times to zero, i.e. all concurrent queries will arrive simultaneously. The goal of these experiments is to evaluate how the total response times can be reduced by combining multiple data requests to only one in the wrapper.

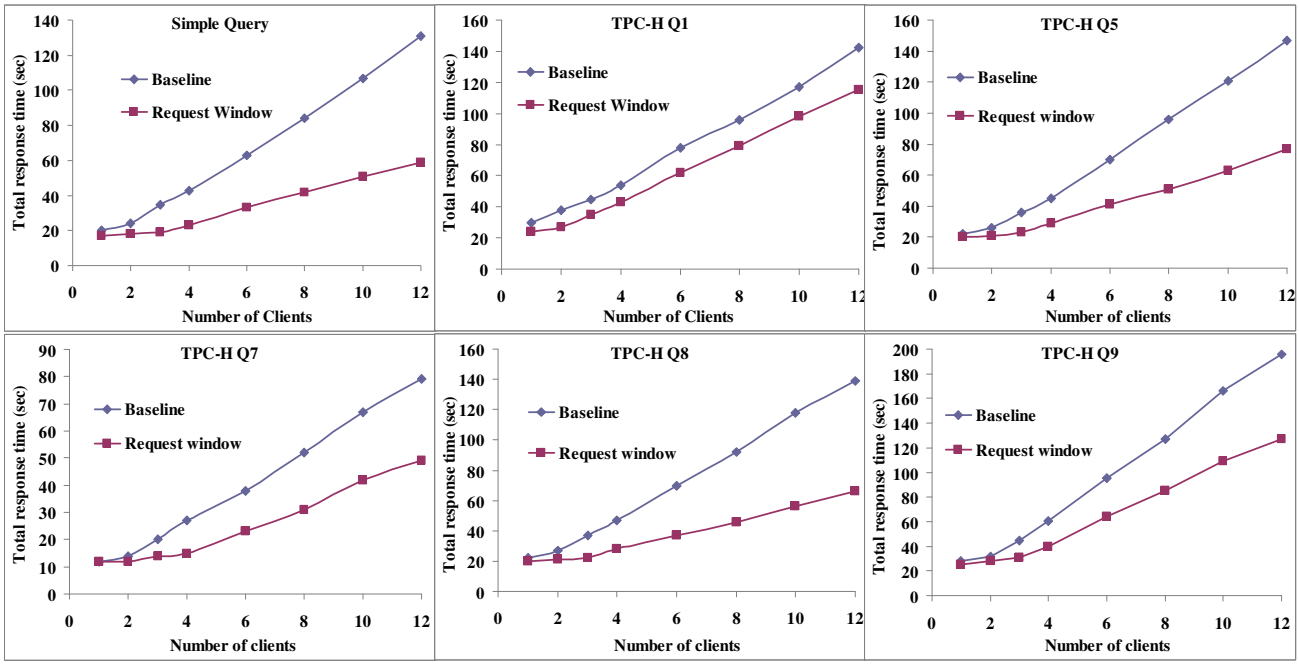

Figure 6: Total response times for executing concurrent same queries that simultaneously arrive. Without enabling Request Window, data transfers dominate the query executions. By minimizing data transfers, total times are reduced significantly.

We choose six queries in this group of experiments. The first one (we call it "Simple Query") is a simple SQL statement: "*select * from v_lineitem*". The execution of this query has very few local computations and is dominated by transferring all tuples of the LINEITEM relation from the data source to the query engine. The other five queries are the standard TPC-H queries 1#, 5#, 7#, 8#, and 9#.

In Figure 6, six graphs show the test results for each query respectively. We present several explanations for the results of this group of experiments.

First, for all queries except the TPC-H query 1#, the request window can significantly reduce the total response times by removing unnecessary data transfers when executing concurrent queries.

Second, when running concurrent workloads, since request window can reduce the data transfer times to the minimum, query executions are no longer limited by the network speed but the CPU performance. From the graphs, we can see that when the number of concurrent clients is less than four, the total response times are almost unchanged with enabling request window.

Third, how much request window can reduce total response times depends on the amount of local computations involved in query executions. For the TPC-H query 1#, because local computations (the HashAggregate operation) dominate query executions when running multiple concurrent workloads, request window cannot provide a significant speedup over the baseline as it does for other queries. The maximum speedup is for the "Simple Query" which has the fewest computations among all six queries.

5.4 Running Full Workloads

In the next experiment, we compare the overall performance of IGNITE with enabling Start-Fetch and Request Window against the baseline system using a set of clients executing a mix of

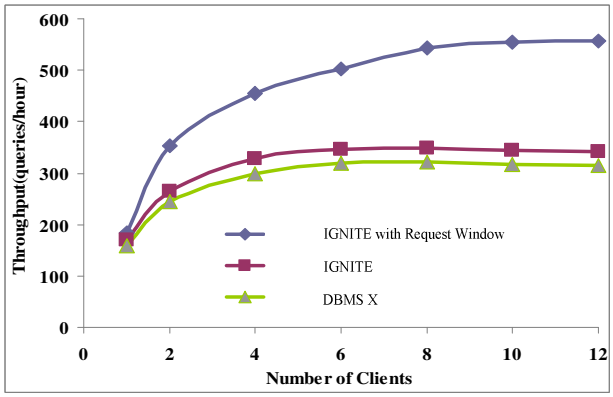

Figure 7: TPC-H throughput for three systems with varying the number of clients from 1 to 12.

queries from the TPC-H benchmark. And in this experiment, we also test a major commercial data integration system (we call it DBMS X for licensing restrictions) as a comparison. We choose eight standard TPC-H queries: #1, #3, #4, #5, #7, #8, #9, and #10. The query execution sequence of each client is listed in Table 2, which is generated according to the TPC-H throughput test specification by removing unused queries. We vary the number of concurrent clients from one to twelve and measure the throughput.

Unlike the last group of experiments in which concurrent queries are arranged to arrive simultaneously, this experiment cannot pre-determine the arrival time of each query from different execution streams. Therefore, data sharing opportunities are determined dynamically by the DIOP technique and the DAW technique as discussed in this paper. The experimental results in Figure 7 show that the request window can significantly improve the throughput of IGNITE by providing data sharing across concurrent queries.

Table 2: TPC-H query sequences of 12 concurrent clients for the throughput experiment. Eight queries are selected. We generate this table according to the TPC-H throughput test specification. (C for Client)

C	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
1	3	5	7	10	8	9	1	4
2	10	9	8	5	7	1	4	3
3	8	5	4	7	1	9	10	3
4	5	4	9	8	10	1	7	3
5	4	7	3	1	5	8	10	9
6	10	3	8	9	7	4	1	5
7	8	4	1	9	3	5	7	10
8	1	5	8	9	7	4	3	10
9	8	3	10	4	1	7	9	5
10	1	7	10	9	3	4	8	5
11	10	1	8	4	5	3	9	7
12	1	7	8	9	4	5	10	3

When the number of concurrent clients is 12, IGNITE with Request Window can perform an outstanding throughput speedup over two opponent systems.

From the graph, we can also see that our IGNITE system has a higher throughput than the DBMS X even without enabling Request Window. This may be explained by the difference in implementing the wrapper: the IGNITE utilizes a wrapper for the PostgreSQL database built on the libpq library, while the DBMS X utilizes a wrapper built on the ODBC interface.

6. FURTHER DISCUSSIONS

6.1 Application Range

Request Window is suitable for running concurrent DSS queries, which often contain aggregations on the results of a join operation between multiple relations. For such queries, current IGNITE optimizer prefers hash-join-based query plans. Although pushing sorting down to sources to accelerate sort-merge join is an attractive strategy in data integration applications, it is only useful for multi-join based on a common attribute. Moreover, many data sources do not support sorting operation, which only accept queries with the input of a target relation and a selection predicate, although the query form does not always follow the SQL syntax. For this situation, it is impossible to push sorting down. If the IGNITE optimizer chooses a sort-merge join for a query involving such sources, the sorting operations will be executed by the engine of IGNITE. Because sorting is also a blocking operator as the hash operator, there will be wait opportunities in the query plan which can be utilized by Request Window.

If a query contains multi-join based on a common attribute and involved sources accept sorting, then Request Window is not useful. To execute a single such query, the engine should push all sorting operations down and merge ordered tuples from each source. In this case, pure Start-Fetch execution without Request Window works well because each data request has not any available wait opportunity. However, to run multiple concurrent such queries, whether and how Request Window can be used to reduce data transfers and source burdens is unsure. We are planning to improve the current query optimizer to challenge such situation.

In addition, for queries that contain union of multiple relations from different data sources, Request Window is suitable.

6.2 Estimating Delay Times

Request Window is actually a framework that allows alternative implementations for each component. For the DIOP technique, a concrete implementation can make the most conservative estimations as we do using our approximate approach, or can utilize a more accurate but more complex model to make exact estimations.

To estimate delay times of data requests in DIOP, the big challenge is the cardinality estimation of the result of each request. To do this, current IGNITE system maintains various statistics information for each registered remote relation with the help of data source wrappers.

However, as discussed in [11][12], statistics information of some sources may be not available in data integration so that making exact estimations for query cardinality is very difficult. Currently, IGNITE cannot deal with this situation and it can only trust the values returned by wrappers. Fortunately, the Black-Box approach to query cardinality estimation introduced in [19][20] shows the feasibility of accurately estimating query cardinality using machine-learning techniques without knowing data distribution. We are learning the Black-Box approach and planning to design and implement a new component to estimate delay times in DIOP.

6.3 Compatibility

Request Window can work together with the semantic caching technique if the latter is feasible in a specific application. Even if the query engine side has caches of some semantic regions, the remainder queries, which cannot be answered only by cached regions, need to be sent to data sources. Request Window still can deal with those remainder queries. Moreover, because the data request can be partially answered by cached regions, the remainder query part will have an enlarged "wait opportunity" considering the corresponding result will not be consumed before the query engine has consumed cached result tuples.

7. CONCLUSION & FUTURE WORK

It is feasible to extend traditional RDBMS to support distributed query processing. Yet the big challenge is how to provide data sharing across concurrent distributed query instances without the memory/disk hierarchy, the foundation of RDBMS's buffer pool management mechanism. We present Request Window as the solution. Its core idea is to combine multiple similar data requests to only one common data request and make concurrent query execution instances share the common result data. The benefit of exploiting such data sharing is the ability of significantly reducing the amount of result data that will have to be transferred over the network and the burdens of data sources for processing data requests. Request Window does not require that multiple queries must arrive simultaneously. It utilizes the DIOP technique to detect the delay opportunity for a data request, and utilizes the DAW technique to construct a group of data requests processed as a batch dynamically. We implemented these techniques in IGNITE, a PostgreSQL-based data integration system. The experimental results show that our solution for data sharing can significantly improve throughput when processing concurrent distributed TPC-H queries in IGNITE.

In future, we shall extend Request Window in three aspects.

Supporting Subquery: How to detect and exploit waiting opportunities when executing various forms of subqueries is a challenge, for which we need to develop flexible model for estimating delays in DIOP.

Using Black-Box Approach: To estimate query cardinality and total transfer times of data requests, the Black-Box approach based on machine-learning methods [20] seems to be a better way, which can avoid maintenance work of statistics information of sources, and is suitable for general data integration scope beyond federated DBMS.

Adding Window Notification: Currently the wrapper can only trust the one-off delay annotation given by DIOP. We are studying how to monitor query execution progress and re-calculate delay times, and then use window notification to hasten window issuing.

8. ACKNOWLEDGEMENTS

We sincerely thank Professor Xiaodong Zhang for helping us better understand capabilities and limitations of the cache mechanism in various contexts. We thank Dr. Zhiwei Xu for his excellent suggestions about the idea and the experiment. And we also thank the VLDB reviewers for their helpful comments. This work is supported in part by the National Science Foundation of China (Grant No. 90412010) and the China National 973 Program (No. 2005CB321807).

9. REFERENCES

[1] J. A. Blakeley, C. Cunningham, N. Ellis, B. Rathakrishnan, and M. C. Wu. "Distributed/Heterogeneous Query Processing in Microsoft SQL Server." In *Proc. ICDE*, 2005.

[2] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World." In *Proc. CIDR*, 2003.

[3] C. Cook. "Database Architecture: The Storage Engine." Microsoft SQL Server 2000 Technical Article, July 2001. Available at: http://msdn.microsoft.com/library.

[4] N. Dalvi, S. K. Sanghai, P. Roy, and S. Sudarshan. "Pipelining in Multi-Query Optimization." In *PODS*, 2001.

[5] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, and M. Tan. "Semantic Data Caching and Replacement." In *Proc. VLDB*, 1996.

[6] P. M. Fernandez. "Red Brick Warehouse: A Read-Mostly RDBMS for Open SMP Platforms." In *Proc. SIGMOD*, 1994.

[7] M. J. Franklin, B. T. Jonsson, and D. Kossmann. "Performance tradeoffs for Client-Server Query Processing." In *Proc. SIGMOD*, 1996.

[8] G. Graefe. "Query Evaluation Techniques for Large Databases." ACM Computing Surveys, 25(2), pp. 73–170, June 1993.

[9] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. "QPipe: A Simultaneously Pipelined Relational Query Engine." In *Proc. SIGMOD*, 2005.

[10] L. M. Hass, D. Kossmann, E. L. Wimmers, and J. Yang. "Optimizing Queries Across Diverse Data Sources." In *Proc. VLDB*, 1997.

[11] Z. G. Ives, D. Florescu, M. T. Friedman, A. Y. Levy, and D. S. Weld. "An Adaptive Query Execution System for Data Integration." In *Proc. SIGMOD*, 1999.

[12] Z. G. Ives, A. Y. Halevy, and D. S. Weld. "Adapting to Source Properties in Processing Data Integration Queries" In *Proc. SIGMOD*, 2004.

[13] T. Johnson and D. Shasha. "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm." In *Proc. VLDB*, 1994.

[14] B. Jonsson, M. Arinbjarnar, B. Jorsson, M. J. Franklin, and D. Srivastava. "Performance and Overhead of Semantic Cache Management." In ACM TOIT, 6(3), pp. 302-331, August 2006.

[15] V. Josifovski, P. Schwarz, L. M. Hass, and E. Lin. "Garlic: a New Flavor of Federated Query Processing for DB2". In *Proc. SIGMOD*, 2002.

[16] D. Kossmann. "The State of the Art in Distributed Query Processing." ACM Computing Surveys, 32(4), pp. 422-469, December 2000.

[17] R. Lee and M. Zhou. "Extending PostgreSQL to Support Distributed/Heterogeneous Query Processing." In *Proc*.DASFAA, 2007.

[18] S. Lyer and P. Druschel. "Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O." In *Proc. SOSP*, 2001.

[19] T. Malik, R. Burns, N. Chawla, and A. Szalay. "Estimating Query Result Sizes for Proxy Caching in Scientific Database Federations." In *SuperComputing*, 2006.

[20] T. Malik, R. Burns, and N. Chawla. "A Black-Box Approach to Query Cardinality Estimation." In *Proc.* CIDR, 2007.

[21] N. Megiddo and D. S. Modha. "ARC: A Self-Tuning, Low Overhead Replacement Cache." In *Proc. FAST*, 2003.

[22] M. F. Mokbel, M. Lu, and W. G. Aref. "Hash-Merge Join: A Non-blocking Algorithm for Producing Fast and Early Join Results." In *Proc.* ICDE, 2004.

[23] E. J. O'Neil, P. E. O'Neil, and G. Weikum. "The LRU-K Page Replacement Algorithm for Database Disk Buffering." In *Proc. SIGMOD*, 1993.

[24] PostgreSQL homepage, 2007. http://www.postgresql.org.

[25] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. "Efficient and Extensible Algorithms for Multi Query Optimization." In *Proc. SIGMOD*, 2000.

[26] G. M. Sacco and M. Schkolnick. "Buffer Management in Relational Database Systems." In ACM TODS, 11(4), pp. 473-498, December 1986.

[27] T. K. Sellis. "Multiple Query Optimization." In ACM TODS, 13(1), pp. 23-52, March 1988.

[28] M. Stonebraker and G. Kemnitz. "The POSTGRES Next Generation Database Management System". In Communications of ACM, 34(10), pp. 78-92, 1991.

[29] TPC Homepage. TPC-H benchmark. www.tpc.org

[30] T. Urhan and M. J. Franklin. "XJoin: A reactively-scheduled pipelined join operator." IEEE Data Engineering Bulletin, 23(2), June 2000.

[31] S. Viglas, J. F. Naughton, and J. Burger. "Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources." In *Proc. VLDB*, 2003.

[32] T. Walter. "Explaining cache — NCR CTO Todd Walter answers your trickiest questions on Teradata's caching functionality." http://www.teradata.com/t/page/116344/.

[33] Y. Zhao, P. M. Deshpande, J. F. Naughton, and A. Shukla. "Simultaneous Optimization and Evaluation of Multiple Dimensional Queries". In *Proc. SIGMOD*, 1998.