Performance Analysis of Relational Operator Execution in N-Client 1-Server DBMS Architectures

Nelson Padua-Perez, nelson@cs.umd.edu
Dec 5 1996

Distributed Information Management Systems
at the University of Maryland (DIMSUM)
Department of Computer Science
University of Maryland, College Park

Abstract

Relational DBMS N-Client-1-Server architectures represent a model of data processing that allows several clients to share a pool of resources residing at the server, and one in which data is managed by the set of operators belonging to the relational algebra. One of the decisions to be taken concerning operators is where their execution takes place (at the server or at the client). This defines two well known paradigms for the execution of a relational operator in a Client-Server environment: the query and data shipping shipping paradigms. In this paper we show performance results of the execution of relational operators which have been implemented following query and data shipping strategies. For this study a Client-Server relational query engine based on the Shore storage manager have been implemented on a cluster of IBM rs/6000 machines. The results of the study have shown the performance trade-offs that each operator implementation have under each execution strategy, have pointed out the relevant parameters affecting the performance of a particular operator under each configuration, and have partially validated the expected behavior of the query and data shipping strategies. In addition the study has illustrated the potential performance gains achievable when relational operators are executed following hybrid execution strategies. The impact of a high speed network in the above mentioned strategies has also be assessed.

1 Introduction

The are two well known strategies for the execution of a database query in a client-server environment: query shipping and data shipping. In query shipping, all query processing is performed at the server with results sent to the client, while in data shipping, data is downloaded to the client which then performs any required processing. A third alternative, hybrid shipping [KOSS95] allows the execution of queries at clients, server, or any combination of the two.

Query, data and hybrid shipping have associated to them different benefits. Among the benefits of query shipping are: low communication cost for queries with high selectivity and support for low cost client machines. On the other hand, data shipping has the advantages of exploiting the resources of powerful client machines (CPU, memory, and disk) and of reducing communication cost for queries with large results [FRAN95]. If caching of downloaded data is being performed at the client in data shipping, then locality of data also reduces the communication cost.

Hybrid shipping tries to combine the advantages of both query and data shipping; it distributes the execution of a query between server and client based on a criteria to optimize (response time, server/client resource utilization, etc.).

In this study, we take a look at the execution performance of typical relational operators in a client-server environment. We study the performance of relational operators as they are executed following query, data and hybrid shipping strategies, and as the number of clients in the system increases. This study provides quantitative data, that confirms the expected behavior of the alternatives previously described.

As part of the study, we have developed a simple relational-database engine called **Tornadito**. This engine allows the execution of typical relational operators (select, project and join) in a multiuser environment, using as the model of computation the iterator [GRAE94].

The reminder of this document is organized as follows: Section 2 presents the experimental environment used for the study. Section 3 describes the set of tests used to compare the performance of the strategies. Section 4 presents the results of performed experiments. Finally, Section 5 contains the conclusions and proposes future work.

2 Experimental Environment

2.1 Software

The set of queries used for the performance comparison of query, data and hybrid shipping strategies, has been implemented using a software we developed, called Tornadito. Tornadito is a simple relational-database engine developed on top of the Wisconsin Scalable Heterogeneous Object REpository (SHORE) storage manager [CAREY94]. SHORE's storage manager provides database concurrency and recovery support, B+-Trees, and file and index scans among other services. In addition, SHORE provide threads (light-weight process) and synchronization (semaphores, condition variables) facilities. One of the features of Tornadito is that it allows remote file and index scans among different SHORE storage managers [1]. The following sections provide information related to the design and implementation of Tornadito.

2.1.1 Tornadito Base Code

As was described previously, SHORE storage manager provides us with a foundation over which we developed the rest of the relational environment we set as a target. Two main modules were added on top of the storage manager in order to complete the relational database engine: the **File and Index module** and the **Iterator module**.

The File and Index module implements basic file, index (create, destroy, open, etc.) and record (get next record, get a particular record, etc.) operations, using storage manager methods. Some functions in this module are implemented through a direct call to the equivalent storage manager method, while others build upon other manager methods.

The Iterator module, is the one responsible for implementing the relational operators and the remote data retrieval facilities of the Tornadito system. All the operations defined in this module, follow an iterator model of computation [2]. In this model there are three types of operations to be associated with any entity declared as an iterator:

- **open()** - This operation does any initialization particular to an iterator. For example, during the initialization of a join iterator, memory buffers are allocated, parameters are processed, etc.
- **next()** - Retrieval of data is achieved through the next() operation. The next() operation returns the next unit of data (in majority of the cases a character string representing a data record) if there is one available. Otherwise the operation indicates that an end of data condition has been reached.
- **close()** - Makes the iterator return any resources it has allocated.

Any query to be processed by Tornadito, will be represented as a tree of iterators. At the leaves level of this tree, the iterators to be found are those provided by SHORE (index or file scans). At the root of the tree, we will have an iterator returning the result of the query, a data unit at a time.

The file and index scans that SHORE provides can be of two kinds: local or remote scans. A local scan is a scan over an index or relation residing in the local volume [3] associated with a SHORE storage manager; a remote scan is a one over an index or relation residing in a different storage manager volume.

Many of the iterators to be presented, use a boolean predicate in order to filter the data it is returned by the iterator. The boolean predicate assumes the form of an interval of allowed values. This interval is specified by indicating a pair of boolean operators, and a pair of boundary values. For example, if we want an iterator to return only those values between 12 and 17, we specify the predicate by using the pairs (GREATER_THAN_OP, LESS_THAN_OP) and (12, 17). Tornadito support integers, float, and string types for the specification of predicates.

The iterators that Tornadito implements are the following:

- **FullFileScanIterator** - Allows a full scan of a file (relation). It is implemented using the file scan class provided by the storage manager.

[1] The remote file and index scans facility was developed by Markos Zaharioudakis at the University of Wisconsin, Madison.

[2] This concept is similar to the one used by [GRAE94].

[3] In SHORE a device is either an operating system file or operating system device and is identified by a path name. In the current SHORE implementation, there is a maximum of 1 volume per device.

- **FileScanIterator** - Performs a file scan, returning those tuples satisfying a specified predicate. This iterator is a FullFileScanIterator and select iterator in one.
- **IndexRecIdIterator** - This iterator uses the index scan iterator provided by the storage manager in order to do index retrieval. It returns the logical record identifiers of records whose index key value are within an specified interval.
- **IndexScanIterator** - Uses an IndexRecIdIterator iterator, in order to retrieve the logical records identifiers of records whose index key value are within an specified interval. Then it retrieves the record which is associated with each logical record identifier returned by the IndexRecIdIterator.
- **Join Iterators** - Tornadito implements tuple-oriented join, block-oriented join, index nested loop join and hybrid hash-join [SHAPI86] using an iterator model of computation.
- **Project Iterator** - Projects fields of the records returned by another iterator.
- **Select Iterator** - Returns records satisfying a particular predicate.
- **Remote Server Iterator** - When processing queries using a query shipping strategy, a mechanism is needed to transfer initialization commands and results between the client and the server. The Remote Server Iterator implements this mechanism. This iterator opens a communication channel between the server and a client, retrieves pages of data, and finally returns data to the client application a tuple at a time [4].

2.1.2 Tornadito Server Application

Using Tornadito's iterator facilities, and SHORE threads and synchronization facilities, we developed the Tornadito Server Application. This application has as its main responsability, the processing of submitted queries (queries following a query shipping strategy) and controlling the remote access of relations/indices by client applications.

The Tornadito Server Application is completed threaded [5]. There is a thread called the **listener thread**, responsible for detecting any client request for service.

Among the services provided by the Tornadito Server application we can mention: query processing, session management [6], coordination mechanisms for simultaneous clients execution, server buffer management (warming or invalidation), execution performance data collection, and others.

The query processing service is the most important of the services offered by the Server application. When a request for the processing of query is detected by the listener thread (this request is generated by the Remote Server Iterator), a thread called the **query processing thread (QPT)** is created, and will be in charge of handling initialization, and next page requests submitted by a client application.

The query processing thread has hard-wired the set of queries it can process. For each initialization request, it identifies the query to be processed, and performs the particular initialization associated with the query, using a set of parameters it receives from the client. For each next request submitted by the client, the query processing thread returns a page of results [7] or end of data condition.

All the processing performed by a query processing thread occurs within a transaction environment; that is, the initialization of query starts a transaction and the completion of the query generates a commit operation.

All the communication between the Tornadito server application and the clients is through the SHORE Communications System which is based on TCP/IP.

2.1.3 Tornadito Client Application

Clients in our environment are represented by a hard-coded query which is implemented using the Tornadito Base Code. This implies that each client has a storage manager, and a query processing engine with the same capabilities as the server. Queries implemented at a client, follow query, data and hybrid shipping approaches.

2.1.4 Tornadito Miscellaneous Applications

In addition to the two previous applications, we have developed applications that allow us to create devices, and to create, destroy, and load indices and relations.

[4] When processing queries following a data shipping approach, all the data transfer between server and client is taken care by the storage manager scans facilities.

[5] The SHORE storage manager runs in a threaded environment. The volcanito server application adds threads which runs concurrently with other manager threads.

[6] Experiments are performed within a session context. This is explained further in section 3.3.

[7] Prefetching of results is not done by the server application.

These applications are implemented by using the Tornadito Base Code.

2.2 Hardware/Operating System Support

The software described previously, corresponding to the client, was installed in five IBM RS/6000 running AIX Version 4.1 in a stripped down multiuser mode. The server software was also installed in a similar machine. For the experiments, the device of the server was configured as a raw disk partition. All the communication between the clients and the server took place by using an FDDI LAN.

3 Tests Description

A total of five tests where developed using different implementations of relational operators. Three tests were based on the select operator and two were based on the join operator. For each test, we varied a set of parameters which we will describe below. Detailed description of each test (and the corresponding setting of the parameters), will be provided in the *Analysis of Tests Results* section. All the tests performed are read tests (based on file or index scans) so shared (SH) file and index locks were used.

3.1 Operators Implementations

A relational operator can be implemented on different ways. In this study we took well known approaches for the execution of an operator, and implement those approaches following query, data and hybrid shipping strategies.

For the select operator the following implementations were used:

1. File Scan Based Select
2. Index Scan Based Select (using clustered and unclustered indices)

For the join operator the following implementations were used:

1. Index Nested Loop Join - Based on the joining attribute of tuples retrieved from the outer relation, tuples from the second relation are retrieved by using an index and then joined.
2. Hybrid Hash-Join - We are using the algorithm described in [SHAPI86].

3.2 Tests Parameters

The following parameters represent the main parameters used to analyze the performance of the strategies under consideration.

1. **Shore Server Buffer Size** - Two values for the Shore server buffer size were used: 1024K bytes which represented a small buffer size and 12288K which represented a large buffer size.
2. **Client Buffer Size** - Two values for the Shore client buffer size were used: 512K bytes which represented a small buffer size and 6144K which represented a large buffer size.
3. **System Load** - Load was varied in the system by changing the number of clients executing a test. The number of clients ranged from 1 up to 5 clients.
4. **Hybrid Hash-Join Parameters** - Two parameters are associated with the hybrid hash-join algorithm: the Hybrid Hash Memory Size and the size of the smaller relation involved in the join. We explored two cases: one where the hybrid hash memory size was bigger than the smaller relation, and the other where it was smaller.
5. **Relations** - Two Wisconsin benchmark relations were used. Each one had 100000 tuples. Each tuple occupies 208 bytes.
6. **Selectivity** - Fraction selected from a relation.

3.3 Performance Metrics

The execution performance of a particular experiment, was measured by using three groups of metrics:

1. **Getrusage and Gettimeofday System Call Metrics**
   By using the Unix getrusage system call we were able to obtain server CPU time, input/output blocks, and other operating system metrics. The gettimeofday() system call allowed us to compute the response time.
2. **Storage Manager Metrics**
   These metrics (volume reads, volume writes, pages fixed and unfixed, etc.) are provided by a method of the SHORE storage manager.
3. **Network Information** - We added methods to the SHORE communication system, which allow us to estimate the number of bytes and messages transmitted during the execution of a query.

In the current presentation we use a subset of the metrics specified above. The subset of metrics used is the following:

1. **Response Time** - Elapsed time from initialization up to the time the last unit of data is retrieved by a client during the execution of a query (see discussion below).
2. **Client Bytes Transmitted** - Bytes transmitted by the client across the network in order to complete a query execution.
3. **Client Messages Transmitted** - Messages transmitted by the client across the network in order to complete a query execution.
4. **Server CPU Time** - User plus system time invested by the server during a session (see session discussion below).
5. **Client CPU Time** - User plus system time invested by the client during a query execution.
6. **Server CPU Utilization** - Server CPU Time divided by session duration time (see session discussion below).
7. **Client CPU Utilization** - Client CPU Time divided by Response Time.
8. **Server Volume Read Operations** - Read operations performed on the raw disk partition where the volume resides.

Each experiment performed to evaluate the shipping strategies was executed within a session context. A session is started after all initialization at the clients is completed (this initialization includes client cache warming, server information requests, etc.). A session ends when all clients finish execution. Performance information corresponding to the duration of the session is kept by the server.

Each client involved in an experiment, measured the execution performance of a query from the point where initialization of the query starts (open method execution), until the point where the last unit of data associated with the query results was retrieved.

In the experiments presented, a query was executed once by all the clients involved in an experiment, in order to reach what we call a "steady state". This execution is responsible for warming server/client buffers, and for allocating the required memory address space for the query (at both the client and the server). After the initial run, the query was executed three times, and the performance of these three executions was measured and averaged. Two extra executions were done, in order to maintain a constant load on the server.

4 Analysis of Tests Results

4.1 Select Operator Results

In this section we present performance results of different implementations of the select operator in the N-Client 1-Server environment.

For the examples to follow, we will use Wisconsin benchmark relations consisting of 100000 tuples. A clustered index on the unique2 attribute and an unclustered index on the unique1 attribute are associated with each relation. Each client can use a unique set of relations/indices or all of them can share the same set.

The select query to be executed by the examples to be presented is the following:

```
select *
from R1
where R1.unique2 < X
(clustered index case)

or

select *
from R1
where R1.unique1 < X
(filescan and unclustered index case)
```

The X variable will assume a unique1 or unique2 attribute value which allow us to retrieve a particular percentage of the relation.

Three examples will be presented: the first one does the selection using filescans, the second one does the selection using a clustered index, and the third one does the selection using an unclustered index.

For each of the select examples to be presented next, we will discuss the impact that different parameters have on the performance of each query processing strategy, and then we will compare the strategies performance against each other.

4.1.1 FileScan Based Selects

The most simple strategy to retrieve a set of tuples of a relation satisfying a particular predicate, is to scan the full relation and filter those tuples satisfying

the predicate. In this section we explored the performance of query and data shipping strategies for the evaluation of selections executed via file scans.

The main results obtained are that, as the percentage of data retrieved is small, and the server resources are plenty, query shipping outperforms data shipping. Data shipping, on the other hand, outperforms query shipping in those cases where the relation fits in the client buffer. In this case, the server does not needs to be contacted in order to perform the selections, what avoids server cpu/disk usage and network traffic.

In general, we observed that filescan does not fit well with a data shipping strategy. Only in those cases where the whole relation can fit in the cache, is where relevant performance gains are observed. This, of course, tends no to be the general case as the ratio of (relation size/ buffer size) usually tends to be big. Any overhead that the data shipping strategy incurs, in downloading and caching data does not pay off in the long run (for the big size relation case) as data keeps being flush continuously as new data arrives. Thus, query shipping, seems to be a preferable alternative in this scenario, as it avoids (for high or medium selectivity values) network traffic and any caching effort at the client.

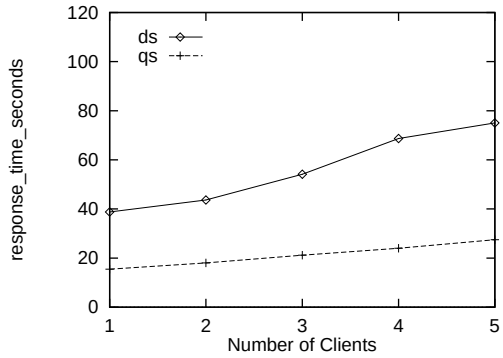

Figure 1: Client Response Times, Select Filescan 10%, Shared, Server Buffer 12288K, Client Buffer 6144K

Figure 1 is a representative example of the response times obtained when following query and data shipping approaches to perform a selection of 10% of a particular relation, in an scenario where neither the client nor the server buffer can fit completely the relation used for the selection [8]. For both query and data shipping cases, the server is constantly replacing the buffer content and doing I/O, in order to retrieve relation pages. The client in the data shipping case, in addition, keeps submitting requests to the server for relation pages.

[8] All clients share the same relation.

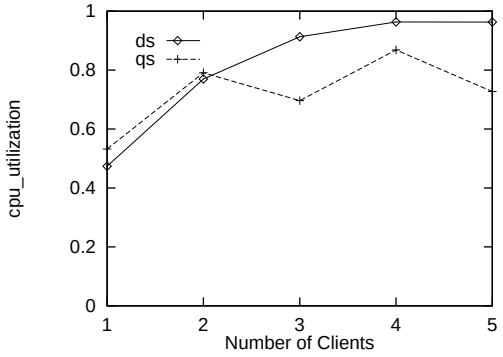

Figure 2: Server Cpu Utilization, Select Filescan 10%, Shared, Server Buffer 12288K, Client Buffer 6144K

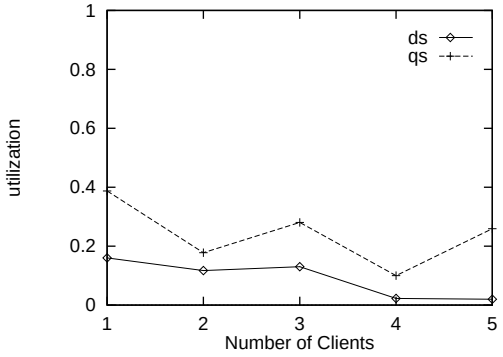

Figure 3: Server Disk Utilization, Select Filescan 10%, Shared, Server Buffer 12288K, Client Buffer 6144K

The server and device disk utilizations observed for this case are presented in figures 2 and 3 respectively. From these graphs we can see that when following a data shipping strategy the server is not I/O bounded, but more CPU bounded (server is busy sending data across the network) as the load (number of clients in the system) increases.

We wanted to see the impact an increase on I/O activity had in the strategies performance. To achieve this increase, we made each client use a different relation for the selection. Figure 4 illustrates the response times obtained for this case. Notice that as the number of clients increases, the response times generated are much higher than for the previous case (e.g., in the previous case (figure 1) when there were four clients in the system, the response time for the selection using a query shipping strategy was around 25 seconds,

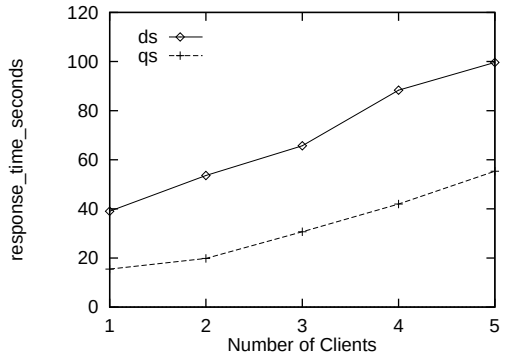

Figure 4: Client Response Times, Select Filescan 10%, Non Shared, Server Buffer 12288K, Client Buffer 6144K

while in the current case is above 40 seconds). Server device disk utilizations around 0.6 were observed for query shipping and utilizations around 0.2 were observed for data shipping, for any number of clients in this example that increased the I/O activity.

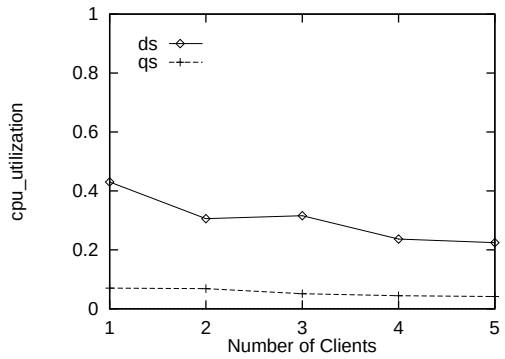

Figure 5: Client Cpu Utilization, Select Filescan 10%, Shared, Server Buffer 12288K, Client Buffer 6144K

The response times observed for the data shipping strategies are due to the amount of data that is being downloaded. The data shipping strategy downloads approximately 25 MB of data, while the corresponding query shipping strategy downloads less than 2.5 MB. An example of the effort by the client (in terms of cpu utilization) to download the data can be seen in figure 5.

Reducing the size of the results of the selection, generated performance improvements in term of response time for the query shipping case. For example, figure 6 presents the results of a query which returned the count of tuples satisfying a selection, instead of the tuples themselves. When we compare these results with

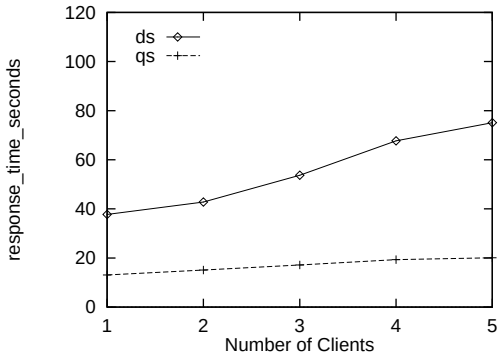

Figure 6: Client Response Times, Select Filescan Count 10%, Shared, Server Buffer 12288K, Client Buffer 512K

the ones shown in figure 1 (non count case) we can observe how as the load (number of clients) increases in the system, the fact that we just transmit the count makes a difference in the response time observed. The reader should keep in mind that this difference is not so big in this example, due to the use of the fast network (FDDI). No significant performance gains were observed for data shipping when the count of tuples was returned. This is expected, as the same network transmission occurs for the computation of the tuple count in the data shipping case.

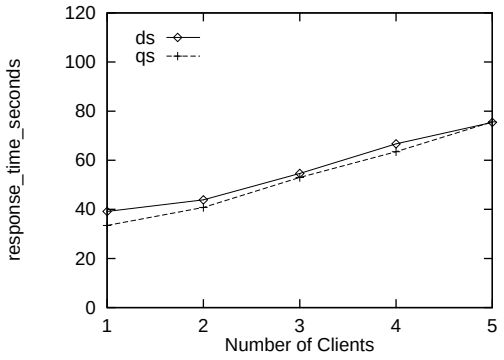

Figure 7: Client Response Times, Select Filescan 100%, Shared, Server Buffer 12288K, Client Buffer 6144K

At this point the reader may wonder how the data and query shipping strategies compare, when the full relation (instead of a fraction of it) is retrieved. This is shown in figure 7. Notice that their performance is quite similar.

4.1.2 Index Based Selects

The previous section presented filescans as the retrieval method used to obtain a set of tuples satisfying a selection criteria. Another well know retrieval mechanism are indices. In this section, we present tests which illustrate the performance trade-offs that indices have when they are used for the processing of queries under query and data shipping strategies.

In general, we observed that query shipping techniques using indices tend to performed well as the load (represented in our case by number of clients) and the demands on the server (e.g., disk utilization, and others) are low. As the number of clients increases, the average response time for query execution increases considerably with each additional client added, what makes query shipping strategies not so appealing, for loaded systems.

We observed that indices, contrary to filescans, allow data shipping techniques to provide significant performance gains during the execution of the query, by making an efficient use of the client cache, by reducing network traffic, and by promoting an overall off-loading of the server. The dependency on the client cache size, that the data shipping strategy has, was heavily noticed, and it was the factor determining in many cases, the overall performance of data shipping. This dependency pointed out that smarter buffer replacement techniques and overall buffer management at the server and client, can make a real impact on the performance of data shipping.

The performance gains observed through data shipping when compared against query shipping, were more significant for those cases where the server was experiencing high disk utilization and/or high cpu demand, as the data shipping strategy offloaded the server. The performance advantage of data shipping over query shipping held even though the network utilization it generated, in terms of messages and bytes transmitted, was higher. The reader should recall, however, that we are using an FDDI network, what makes the network factor, at least for this example, one probably negligible.

Next we will present examples that illustrate the main issues already pointed out in the previous discussion. First, we will take a look at a clustered index example and then we will look at a unclustered index one. For each example we will discuss how server/client buffer size, load, I/0 server load and size of results generated, affects a particular shipping strategy.

Clustered Index

The response times for a 10% selection of a relation using a clustered index for different server/client buffer sizes can be seen in figures 8, 9, 10, 11.

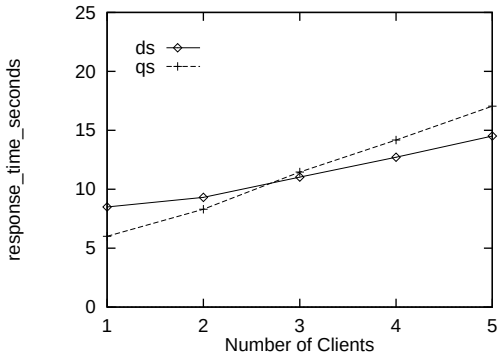

Figure 8: Client Response Times, Select Clustered Index 10%, Shared, Server Buffer 1024K, Client Buffer 512K

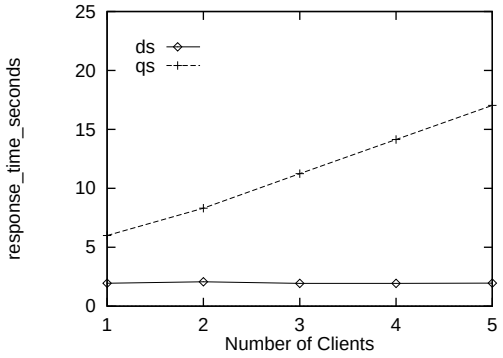

Figure 9: Client Response Times, Select Clustered Index 10%, Shared, Server Buffer 1024K, Client Buffer 6144K

Query Shipping

An increase in the server buffer size from 1024K to 12288K, in general, generate significant performance improvements in terms of response time (compare the query shipping curves of figures 8 and 10) for the selections explored when following a query shipping approach. For example, a reduction by 2 seconds can be observed in the response time generated by one client in the system, when the server buffer size is increased. Notice that as the number of clients increases, the gains achieved by an increased in buffer size are not as significant as for the case where the number of clients is low.

The reductions in response time observed by using a big server buffer are due to the reduction in I/O that this buffer increase generates. The server disk

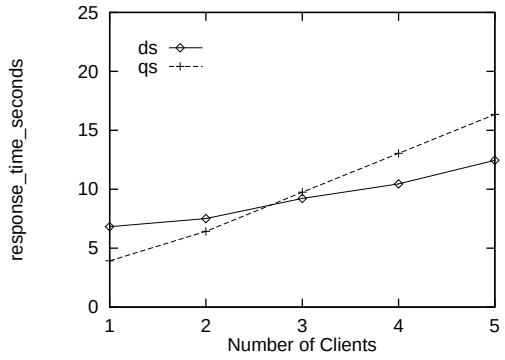

Figure 10: Client Response Times, Select Clustered Index 10%, Shared, Server Buffer 12288K, Client Buffer 512K

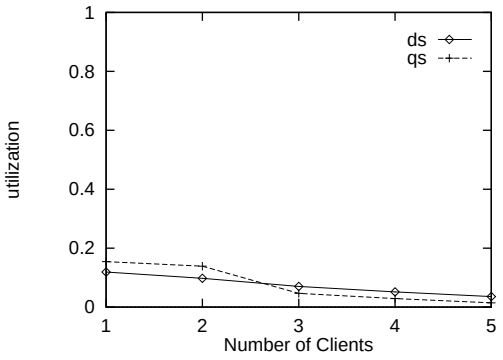

Figure 12: Server Device Disk Utilization, Select Clustered Index 10%, Shared, Server Buffer 1024K, Client Buffer 512K

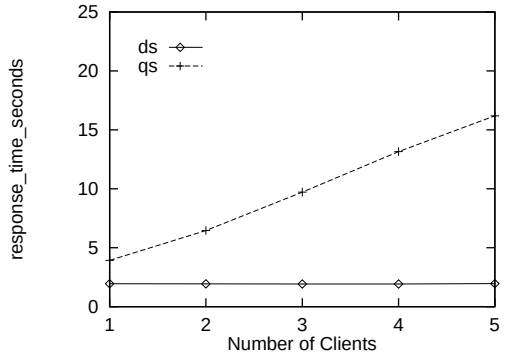

Figure 11: Client Response Times, Select Clustered Index 10%, Shared, Server Buffer 12288K, Client Buffer 6144K

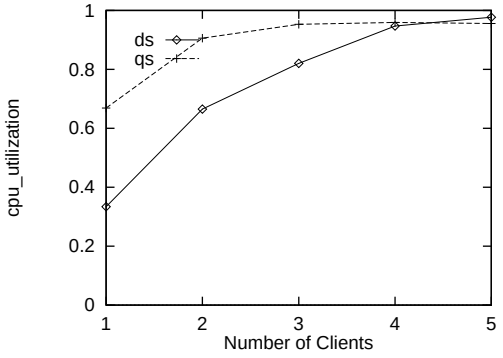

Figure 13: Server Cpu Utilization, Select Clustered Index 10%, Shared, Server Buffer 12288K, Client Buffer 512K

utilization illustrated on figure 12 goes to zero when a big server buffer size is used. The fact that as the number of clients increases, the gains of a buffer increase are less, is because for higher number of clients the disk utilization is not the predominant factor (the system is cpu bounded as figure 13 illustrates). Notice that after 3 clients, the server reaches an utilization very close to 0.9; this illustrates that query shipping strategies do not seem to scale well.

The selections performed by each client for the above results used the same relation at the server. In order to see how query shipping will respond to an increased in I/0 activity at the server, we repeat the experiment, but in an scenario where each client used a different relation. An example of the impact on response time, that this had for the query shipping strategy, can be seen in figure 14 (compare against figure 9); the disk utilization for this case is presented in figure 15 (compare against figure 12). As expected the increase in I/O has made quite an impact on the response times observed, specially as the number of clients increases.

Data Shipping

Increases in the server buffer size promoted some gains in the performance, in terms of response times, for the data shipping strategy (compare figures 8 and 10). Now, the impact in performance was not as big as an increase in client buffer generated (compare figure 8 and figure 9). Notice how in this case, increasing the client buffer from 512K to 6144K, has reduced the response time for any load by at least a factor of four.

The performance of data shipping as the client buffer is increased comes as no surprise. Having the data locally at the client avoids any interaction with the server, what avoids network traffic and any load on the server (figure 16).

It is important to notice that the impact of data

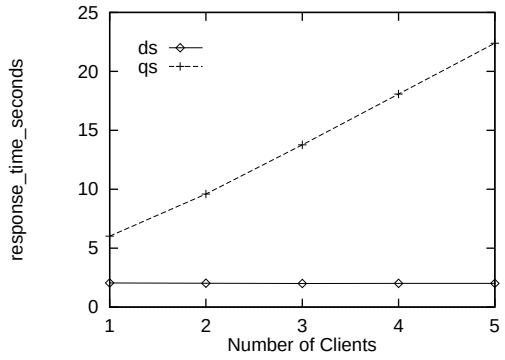

Figure 14: Client Response Times, Select Clustered Index 10%, No Shared, Server Buffer 1024K, Client Buffer 6144K

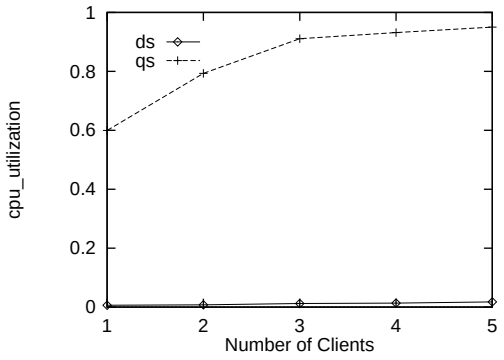

Figure 16: Server Cpu Utilization, Select Clustered Index 10%, Shared, Server Buffer 1024K, Client Buffer 6144K

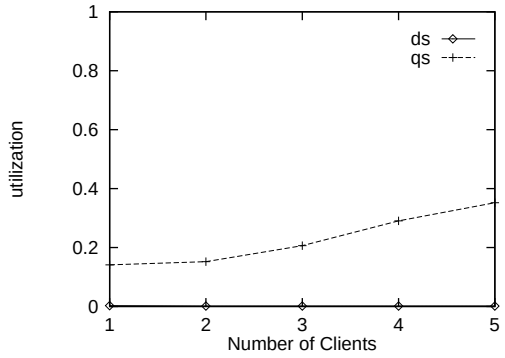

Figure 15: Server Device Disk Utilization, Select Clustered Index 10%, Non Shared, Server Buffer 1024K, Client Buffer 6144K

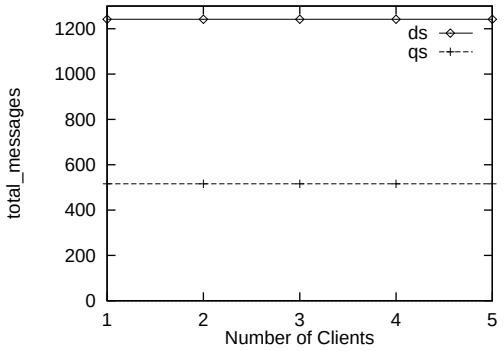

Figure 17: Client Total Messages Transmitted, Select Clustered 10%, Shared, Server Buffer 1024K, Client Buffer 512K

shipping client caching when compared against query shipping, is even more significant when each client retrieves a different relation. When we compare the data shipping curves of figures 14 and 9 we can see how data shipping is not affected at all by the fact that now each client retrieves a different relation. The same cannot be said about query shipping.

Network Usage

Figures 17 and 18 illustrate typical messages and bytes transmission occuring for the query and data shipping strategies that use a clustered index. The curves for the data shipping strategy correspond to the case where a small client buffer size was used. For the big client buffer size, bytes and messages transmited was a minimum (very close to zero).

Reducing the size of the results produced by the query, by returning the count of tuples satisfying the selection, showed performance improvements, in term of response times, for all the loads explored in the query shipping case. For example, compare figures 10 (response time for no count case) with figure 19 (response time for count case). No significant gains were achieved, however, when following a data shipping strategy. This is expected because when following a data shipping strategy, the network transmission is the same doing count or not. Notice that the example that returns the count, give us an approximation of the network contribution to the total response time observed in the query when all the resulting tuples of the selection, instead of the count, are transmitted.

The clustered index select operator implementation represents one of the best examples of how an appropriate use of the client buffer, under a data shipping strategy, can produce significant performance gains when compared against a query shipping strategy.

Unclustered Index

In the discussion to follow, we take a look at the performance of selections implemented using unclus-

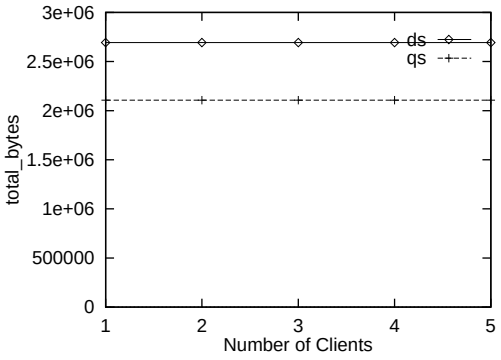

Figure 18: Client Total Bytes Transmitted, Select Clustered 10%, Shared, Server Buffer 1024K, Client Buffer 512K

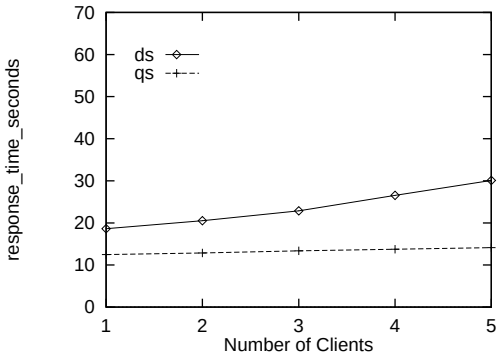

Figure 20: Client Response Times, Select Unclustered Index 0.5%, Shared, Server Buffer 1024K, Client Buffer 512K

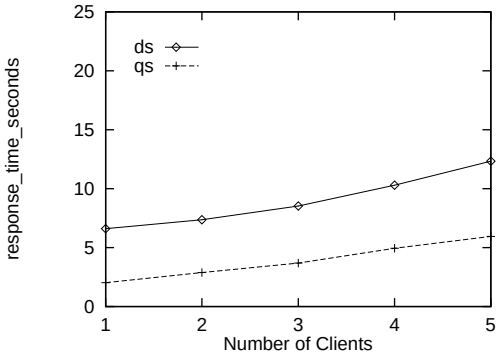

Figure 19: Client Response Times, Select Clustered Index Count 10%, Shared, Server Buffer 12288K, Client Buffer 512K

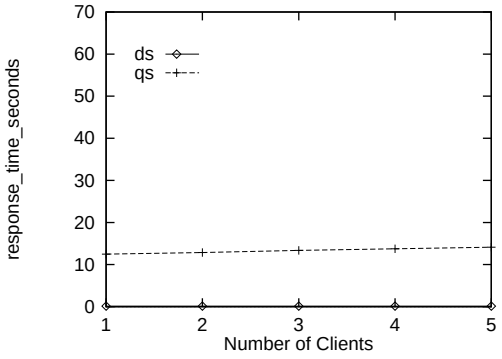

Figure 21: Client Response Times, Select Unclustered Index 0.5%, Shared, Server Buffer 1024K, Client Buffer 6144K

tered indices. The discussion below will focus on a selection example that retrieves 0.5% of a Wisconsin relation by using an unclustered index constructed on the unique1 attribute.

The response times obtained for selecting 0.5% of a relation [9] using unclustered indices, for different combinations of server/client buffer sizes, can be seen in figures 20, 21, 22, 23.

Query Shipping

Variations in the server buffer size generated significant improvements in the response times observed for the query shipping strategy. For example, increasing the server buffer from 1024K to 12288K, reduced the selection response time for 1 client by approximately 10 seconds (compare figures 20 and 22). Similar performance gains were observed as the load in the system increased.

An increase in server buffer size reduces the response time observed for query shipping, because it reduces the disk utilization at the server. If we take a look at the utilization of the disk for the small server/ small client case (figure 24) we can see that the disk utilization is at least 0.6, for any number of clients. When the buffer size is increased, the utilization goes to zero. If we take a look at the server cpu utilization for the case where the server buffer size is small, we can notice that the system in this case, is not CPU-bounded (figure 25).

Figures 27 (small server buffer case), and 28 (big server buffer case), illustrate the same selection under discussion but for the case where each client uses a different relation; these cases present an extra effort for I/O at the server. Notice in the graphs how the response time for the query shipping alternative has suffered a dramatic increase when this new environment was used. As in the previous scenario the response time increased is due to the disk utilization occurring

[9] All clients share the same relation.

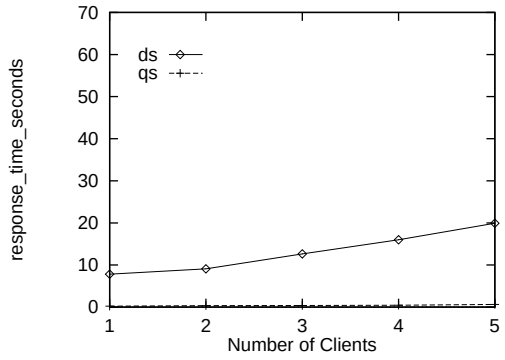

Figure 22: Client Response Times, Select Unclustered Index 0.5%, Shared, Server Buffer 12288K, Client Buffer 512K

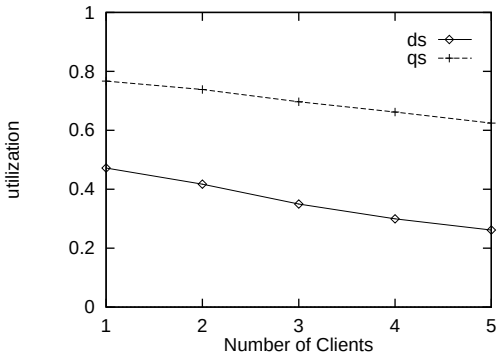

Figure 24: Server Disk Utilization, Select Unclustered Index 0.5%, Shared, Server Buffer 1024K, Client Buffer 512K

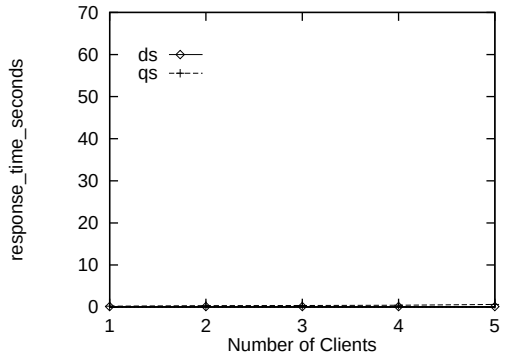

Figure 23: Client Response Times, Select Unclustered Index 0.5%, Shared, Server Buffer 12288K, Client Buffer 6144K

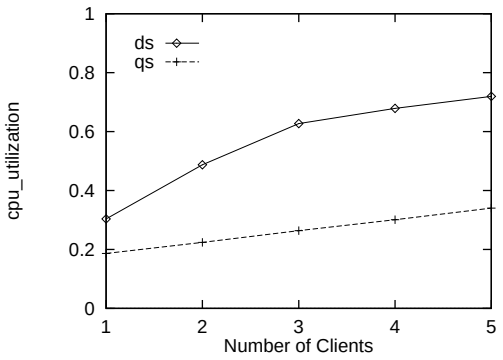

Figure 25: Server Cpu Utilization, Select Unclustered Index 0.5%, Shared, Server Buffer 1024K, Client Buffer 512K

in the system as figure 29 (small server buffer case) and figure 30 (big server buffer case) illustrate.

The two scenarios (the I/O bounded and the not I/O bounded one) we just presented, are very important, as they illustrates how query shipping responds under an environment where the server has different loads.

Data Shipping

Increasing the server buffer size reduced the response times observed for the data shipping strategy for all the number of client explored (compare the data shipping curves of figures 20 and 22). The increases in server buffer size reduced the disk utilization generated at the server to basically zero, for the data shipping case, as it did for the query shipping strategy, and it explains the performance improvements seen in terms of response times.

As seen in the clustered index case, the improvements in response times observed when the client buffer size was increased, were more significant than when the server buffer size was increased (compare data shipping curves of figures 20 and 21).

The client cpu utilizations observed under the data shipping strategy for the different buffer combinations, can be seen in figures 31, 32, 33, and 34. Notice that as more resources are added to the system, in this case in terms of buffer space, the utilization of the client is higher, what implies a higher exploitation of the client resources and a faster response time.

It is important to indicate that the data shipping strategy under sub optimal conditions (under the current discussion this translates to insufficient buffer space and high I/O demand) can be detrimental for the full client server environment, as server and client resources will be more heavily utilized under data shipping than under other strategy (e.g., query shipping) without achieving any performance gains (e.g., response time reductions in the processing

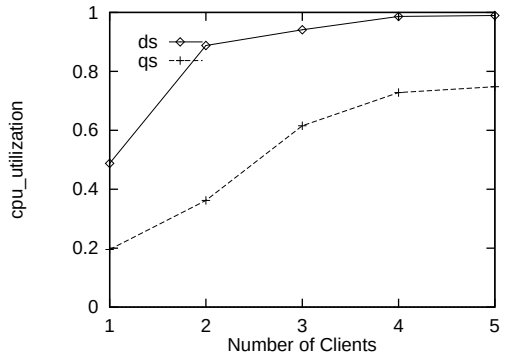

Figure 26: Server Cpu Utilization, Select Unclustered Index 0.5%, Shared, Server Buffer 12288K, Client Buffer 512K

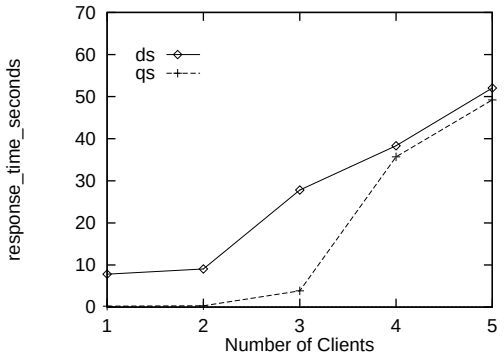

Figure 28: Client Response Times, Select Unclustered Index Non Shared 0.5%, Server Buffer 12288K, Client Buffer 512K

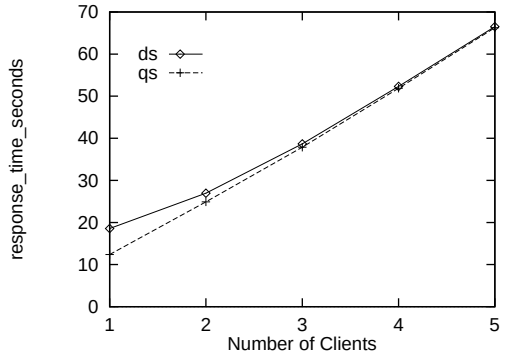

Figure 27: Client Response Times, Select Unclustered Index Non Shared 0.5%, Server Buffer 1024K, Client Buffer 512K

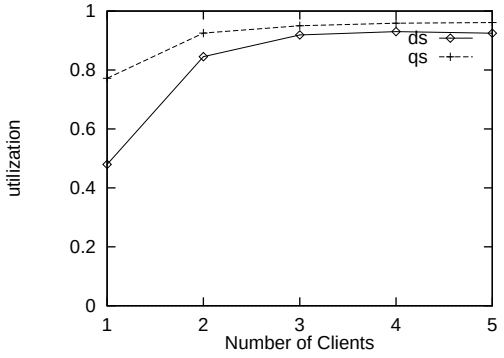

Figure 29: Server Disk Utilization, Select Unclustered Index Non Shared 0.5%, Server Buffer 1024K, Client Buffer 512K

of the query). An example of this situation can be observed by looking at the server and client cpu utilization graphs (figures 25 and 31 respectively), and to the server device disk utilization graphs (figure 24) for the case where the server buffer is 1024K and the client buffer is 512K. In this case server and client resources are being heavily utilized without achieving a reasonable response time (figure 20).

The results obtained when the I/O activity at the server was increased for the data shipping strategy, where similar to the ones seen for query shipping, for those cases where the client was not able to exploit the client cache. For those cases where the client buffer was big enough, trivially, the increased in I/O activity had no impact as everything was cached in the client buffer during the warm up runs. Notice that although there was no impact, there was a big saving achieved in this case, by the exploitation of the cache at the client. Each client in this case, is caching portions of a relation which is not being used by any other client; the impact on the server if each client had to download this portion would be big.

Network Usage

The reader can see the representative network usage that a client generates for this example, in terms of messages and bytes transmitted, by looking at figures 35 and 36. Notice the high communication overhead imposed by data shipping.

The selection through an unclustered index presented in this section, illustrated, among other things, how query and data shipping strategies can outperform each other under different run time conditions for the evaluation of the same query. Also it illustrates how as the server services become scarce (I/O demand increases) and the load increases, the data shipping strategy tends to outperform query shipping, specially if it can exploit the cache in an proper manner.

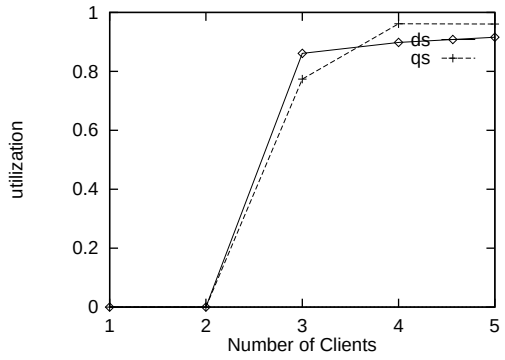

Figure 30: Server Disk Utilization, Select Unclustered Index Non Shared 0.5%, Server Buffer 12288K, Client Buffer 512K

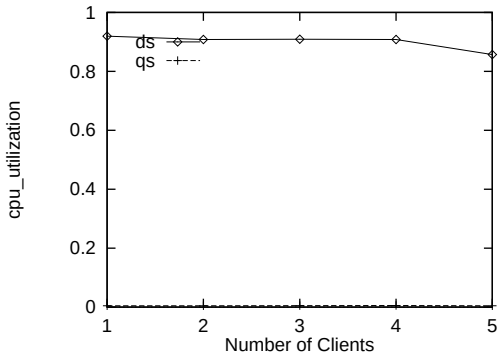

Figure 32: Client Cpu Utilization, Select Unclustered Index 0.5%, Shared, Server Buffer 1024K, Client Buffer 6144K

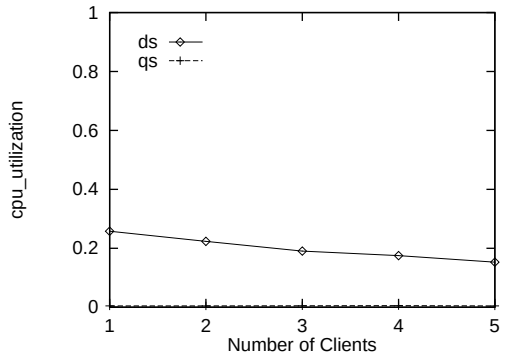

Figure 31: Client Cpu Utilization, Select Unclustered Index 0.5%, Shared, Server Buffer 1024K, Client Buffer 512K

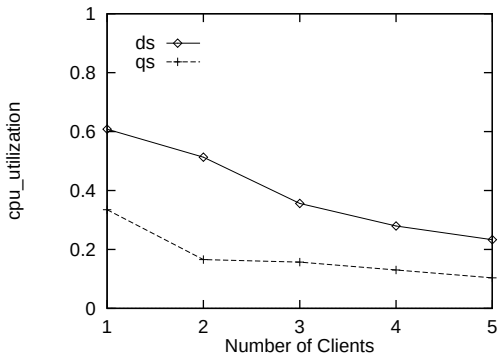

Figure 33: Client Cpu Utilization, Select Unclustered Index 0.5%, Shared, Server Buffer 12288K, Client Buffer 512K

4.2 Join Operator Results

In this section we present performance results of different implementations of the join operator in the N-Client-1-Server environment. In contrast to the select operator implementations addressed in the previous section, the join operator allow us to explore the load balancing effect that each query processing strategy has in the Client-Server environment. Issues already addressed for the select operator case (e.g., server/client buffer size, load impact, etc.) will also be addressed.

From the executions of the join tests we observed some general trends. For example, the average response times observed for each additional client added, grew faster for query shipping than for any of the other alternatives. In spite of this behavior, the query shipping strategy tends to have a good performance, when the number of clients in the system is low (reduced contention for server resources).

The data shipping strategy, as was seen in the select examples, performed at its best, when the client cache was fully exploited. Overall, the response times observed where the lowest ones, when the data shipping strategy can use the content of the client buffer. The offloading effect the strategy has and the use of indices for the retrieval of data, are the responsible for the performance observed for the data shipping strategy.

It was interesting to see that the data shipping strategy, was able to outperform the other two strategies (query and hybrid shipping ones), for the case where the client buffer was small, and as the load (number of clients in the system) increased. This performance advantage was seen in spite of the fact that the network utilization that the data shipping strategy generated was higher for the data shipping strategy than for the other alternatives.

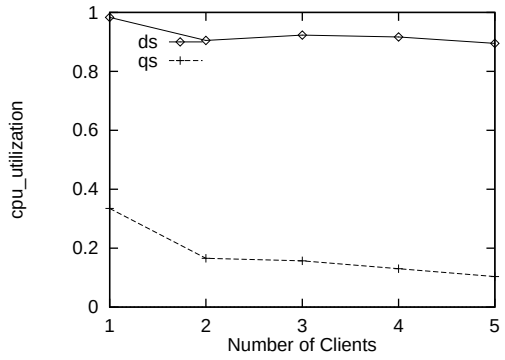

Figure 34: Client Cpu Utilization, Select Unclustered Index 0.5%, Shared, Server Buffer 12288K, Client Buffer 6144K

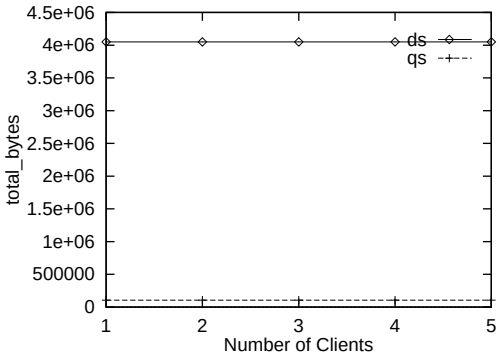

Figure 36: Client Total Bytes Transmitted, Select Unclustered 0.5%, Shared, Server Buffer 1024K, Client Buffer 512K

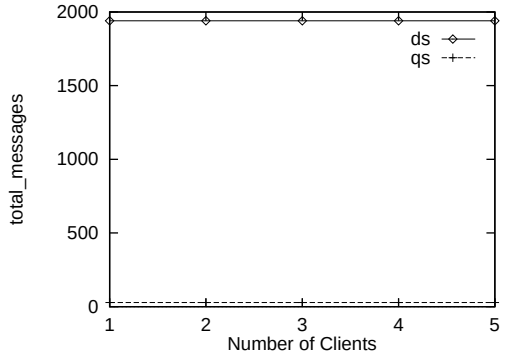

Figure 35: Client Total Messages Transmitted, Select Unclustered 0.5%, Shared, Server Buffer 1024K, Client Buffer 512K

The hybrid shipping plans showed how displacing work to the client (in this case the actual computation of the join once one or both selections involved in the join have been done at the server) can have overall performance improvements. The system by using hybrid shipping plans, exploited the client resources, and diminished the server load, what translated in overall reduction of client response time and increased server scalability. Through the hybrid shipping queries presented we saw the potential performance gains that can be achieved through hybrid approaches, gains that pure query or data shipping systems may not be able to achieve.

In the sections to follow, two 2-way join tests will be presented. One test implements the join using the hybrid hash-join algorithm and the other one using the index nested loop join algorithm. Each test implements the join using query and data shipping approaches. In addition, for each test, we will present performance results of a hybrid shipping strategy plan associated with the test.

For the tests to follow, we will use Wisconsin benchmark relations, each with 100000 tuples. In addition a clustered index on the unique2 attribute will be associated with each relation. The clients in the system will use the same set of relations/indices in order to compute the join (shared relation/indices set case) or a set of relations/indices which is unique to them (non-shared relations/indices case).

For each of the join examples to be presented next, we will discuss the impact that different parameters have on the performance of each query processing strategy, and then we will compare the strategies performance against each other.

4.2.1 Hybrid Hash Join Test

This example presents the computation of a join by using the hybrid hash-join algorithm.

The join query that this example computes is the following:

```
select *
from R1, R2
where ( (R1.unique1 == R2.unique1) AND
        (R1.unique2 < 10000)       AND
        (R2.unique2 < 10000)         )
```

The query shipping plan implemented does selections on each relation to be joined using index scans, and then joins those selections using the hybrid hash-join algorithm. A similar plan is followed by the data shipping strategy, but the index scans are done remotely from the client. The hybrid shipping plan does the selections at the server (using indices), which are then used by the client to perform the join. Notice that the

joining attribute is different from the attribute used in the selections.

Two cases for the hybrid hash-join were explored: one case where the hybrid hash memory was bigger than the smaller relation involved in the join, and another case where the hybrid hash memory was smaller. In the current presentation we concentrate on the "small memory case".

The response times for the different combinations of server/client buffer sizes for the case where each client is using the same set of relations/indices [10] is presented in figures 37, 38, 39, 40.

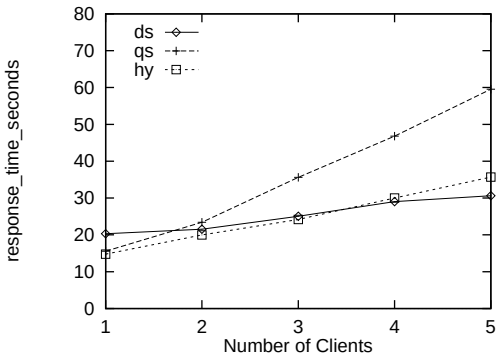

Figure 37: Client Response Times, Hybrid Hash Join, Shared, No Memory, Server Buffer 1024K, Client Buffer 512K

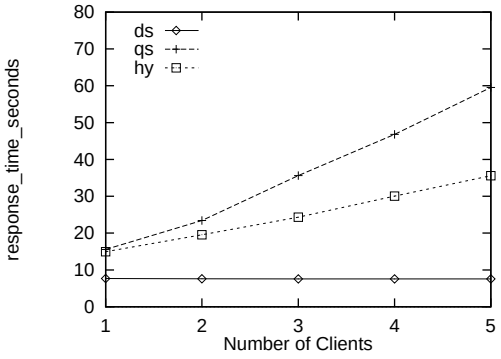

Figure 38: Client Response Times, Hybrid Hash Join, Shared, No Memory, Server Buffer 1024K, Client Buffer 6144K

Query Shipping

Increases in server buffer sizes, generate no significant improvement, in terms of response time, for the query shipping strategy. Some gains are observed for the case where the load is only 1 or 2 clients, however. For example, increasing the server size from 1024K to 12288K (figures 37 and 39), reduced the response time to compute the join by approximately three seconds for the 1 client case. The impact the query shipping strategy has on the server utilization for the small (1024K) and big (12288K) server buffer size can be seen in figures 41 and 42 respectively.

[10] In the discussion to follow the reader can assume we are using the same set of relations/indices unless otherwise stated.

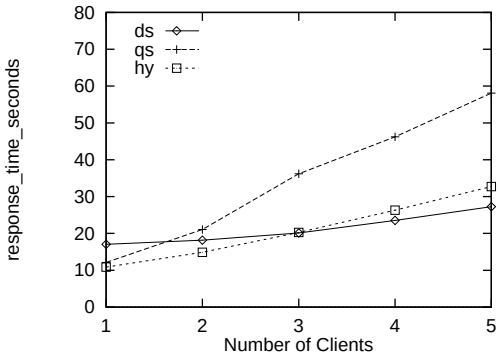

Figure 39: Client Response Times, Hybrid Hash Join, Shared, No Memory, Server Buffer 12288K, Client Buffer 512K

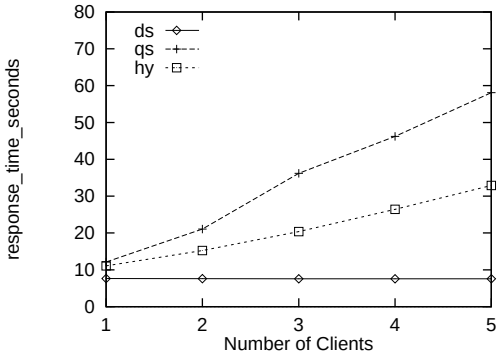

Figure 40: Client Response Times, Hybrid Hash Join, Shared, No Memory, Server Buffer 12288K, Client Buffer 6144K

The factor responsible for the response time observed in the query shipping strategy, is the I/O required to generate the partitions used in the hybrid hash join. Figure 43 illustrates an representative example of the I/0 effort done in order to process the join when following the hybrid hash join algorithm. When we reduced this effort to 0 (by allocating plenty of memory to be used by the hybrid hash algorithm) the utilization of the disk dropped to 0 (as expected), and the response times observed diminished dramatically as figure 44 illustrate (compare against figure

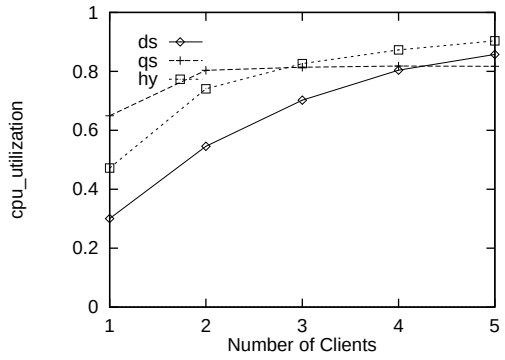

Figure 41: Server Cpu Utilization, Hybrid Hash Join, Shared, No Memory, Server Buffer 1024K, Client Buffer 512K

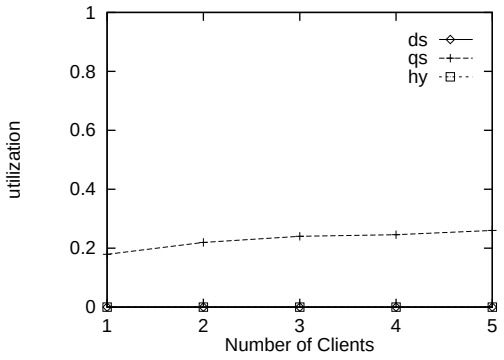

Figure 43: Server Non Device-Disk Utilization, Hybrid Hash Join, Shared, No Memory, Server Buffer 1024K, Client Buffer 512K

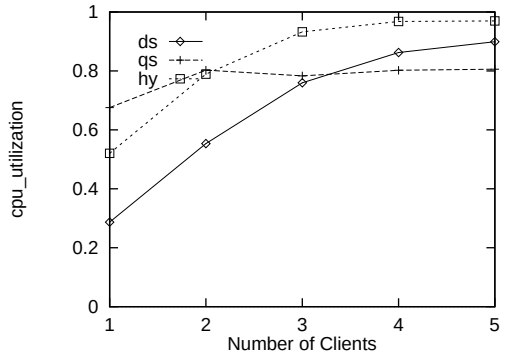

Figure 42: Server Cpu Utilization, Hybrid Hash Join, Shared, No Memory, Server Buffer 12288K, Client Buffer 512K

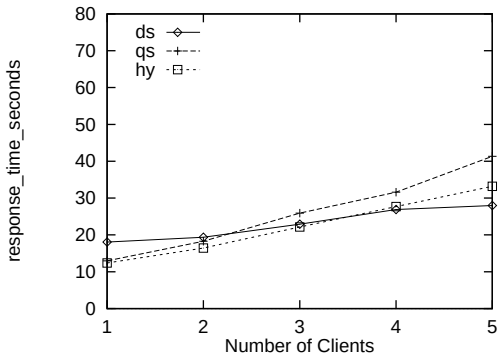

Figure 44: Client Response Times, Hybrid Hash Join, Shared, Memory, Server Buffer 1024K, Client Buffer 512K

37). The representative device disk utilizations for both cases mentioned above (the plenty memory and non-plenty memory case) is illustrated in figure 45.

We wanted to observe the impact an increase in device I/O (I/O responsible for relation/indices retrieval) had on response time on the system. In order to increase the I/0 activity we made each client use a unique set of relation/indices. The impact of I/0 increase for the small server buffer case in terms of response time can be seen in figure 49 (compare against figure 37). Notice that as the number of clients is increased, the impact of having to retrieve different set of relations shows a dramatic increase in the response time observed for each client (e.g., in the case where reduced I/O was done we observed a response time of approximately 47 seconds when four clients are executing, while for the case where the I/O was increased we observed a response time of approximately 55 seconds). The server device disk utilization for this case can be seen in figure 50.

Data Shipping

The data shipping strategy, in contrast to the query shipping strategy, allow us to move the computation of the join from the server to the client promoting load balancing in the system. The response time observed for the case where all clients shared the same set of relations and where the hybrid hash algorithm uses a small memory size can be seen in figures 37, 38, 39, 40.

Overall, the data shipping strategy was not significantly affected by increases in server buffer size. Some reductions in response times were observed, particularly for a high number of clients, when the server buffer was increased from 1024K to 12288K. The data shipping strategy, however, showed a significant performance improvement, in terms of response time, when the client buffer size was increased. A reduction in the response time by at least 10 seconds was seen in

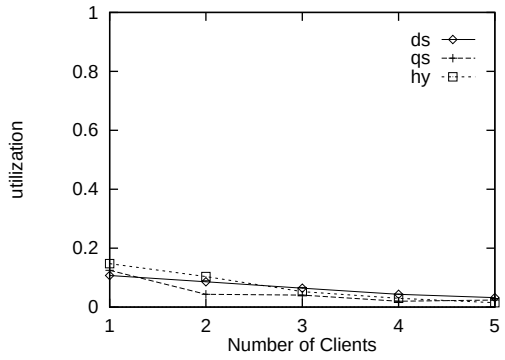

Figure 45: Server Device Disk Utilization, Hybrid Hash Join, Shared, No Memory, Server Buffer 1024K, Client Buffer 512K

the cases studied, for any number of clients (compare figures 37 and 38). This improvement in response time is caused because the selections required for the join, can now be performed completely at the client with no server intervention at all. In those cases where the client cache is not big enough to cache the data, we need to contact the server what explains the response times observed for the small client buffer case. A representative server cpu utilization for the case that the server needs to be contacted is presented in figure 42.

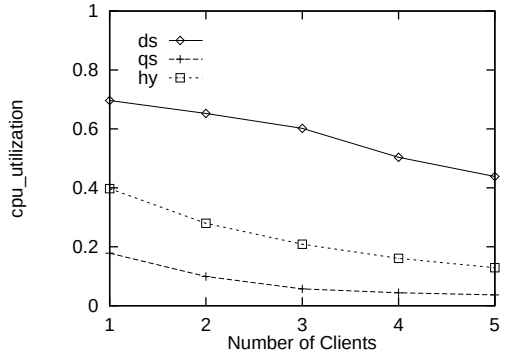

Figure 46: Client Cpu Utilization, Hybrid Hash Join, Shared, No Memory, Server Buffer 12288K, Client Buffer 512K

A high client cpu utilization was observed in the clients when the client buffer used had a big size. The cpu utilization at the client was higher (very close to 1) for those cases where the client buffer was big and when the computation of the join used plenty of memory. In those cases where the client buffer was small we saw an utilization with a maximum value of 0.7, which decreased with the number of clients in the system (figure 46).

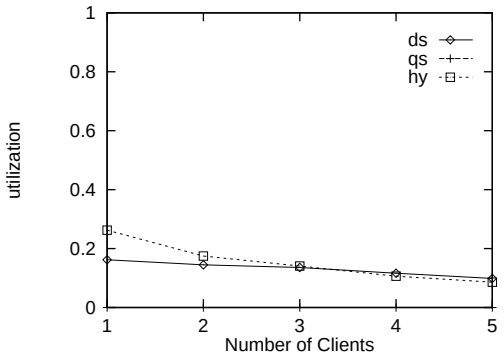

Figure 47: Client Non Device Disk Utilization, Hybrid Hash Join, Shared, No Memory, Server Buffer 12288K, Client Buffer 512 K

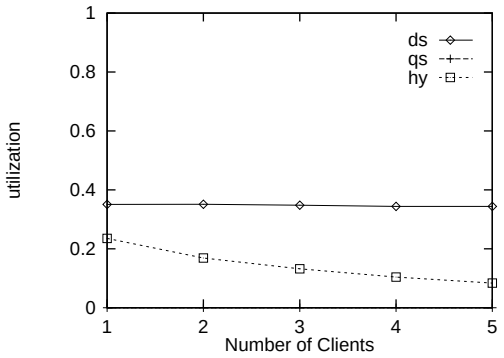

Figure 48: Client Non Device Disk Utilization, Hybrid Hash Join, Shared, No Memory, Server Buffer 12288K, Client Buffer 6144K

Typical disk utilization observed at the client, corresponding to the actual computation of the join (generation of hybrid hash join partitions), can be seen in figures 47 (small client buffer size) and 48 (big client buffer size). As expected, higher buffer size promotes a higher disk utilization as generating the partitions for the join, does not need to wait for server data.

The impact an increase on server I/O has on the data shipping strategy (achieved by forcing each client to use a different set of relations/indices) in terms of response time, can be seen in figure 49 (compare against figure 37). The corresponding server device disk utilization can be seen in figure 50 (compare it against the utilization seen for data shipping in figure 45).

Hybrid Shipping

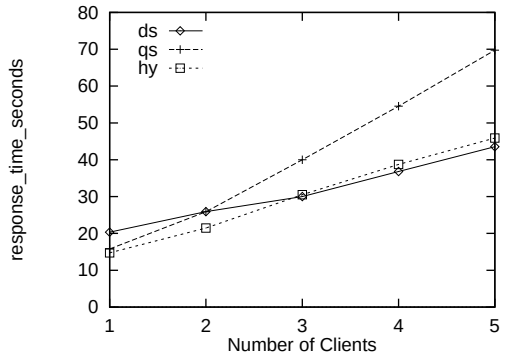

Figure 49: Client Response Times, Hybrid Hash Join, No Shared, No Memory, Server Buffer 1024K, Client Buffer 512K

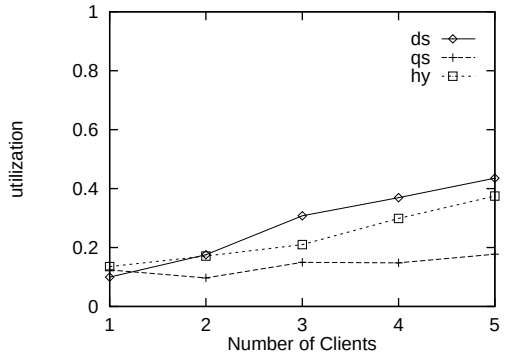

Figure 50: Server Device Disk Utilization, Hybrid Hash Join, No Shared, No Memory, Server Buffer 1024K, Client Buffer 512K

The hybrid shipping plan implemented, tries to divide the computation required for the join, by calculating the selections at the server, and then computing the join at the client. The response times produced by this plan can be seen in the hybrid shipping curves of figures 37, 38, 39, 40.

Performance improvements were observed, when the server buffer was increased for the hybrid shipping alternative (compare the hybrid shipping curves of figures 37 and 39). Extra server buffer space, allow the warm up runs to save more of the relations being selected, what promotes the improvements seen in response time.

The hybrid shipping alternative presented, similarly to the data shipping one, performs the final join at the client what promotes a reduction in the amount of I/O performed at the server, therefore, offloading it. This offloading, of course, is paid by extra work at the client (when compared with query shipping) as the figures for client cpu and disk utilization illustrate (figures 46 and 47 respectively).

When the level of I/O was increased for the hybrid shipping alternative, we observed a behavior similar to the one observed for the other two alternatives: the response times showed a significant increase (compare figures 37 and 49) as the load increased (more than two clients).

Notice that moving the computation of the join to the client alleviates the usage of the server disk, what can improve the access times of data from the database in the case where both the database and the hybrid hash join partitions coexist in the same device (disk).

Network Usage

An example of the network usage the different alternatives generated (for the small client buffer size case), can be seen in figures 51 and 52.

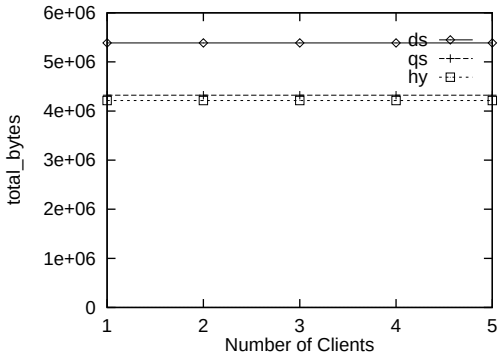

Figure 51: Client Total Bytes Transmitted, Hybrid Hash Join, Shared, No Memory, Server Buffer 1024 K, Client Buffer 512K

4.2.2 Index Nested Loop Join Test

The computation of a join can also be done, by following the index nested loop join algorithm. We are presenting this example to illustrate the performance behavior of the query, data and hybrid shipping alternatives under a method, different from the hybrid hash join, to perform the computation of the join. Trade offs similar to the ones found for the hybrid hash join case were found for this case too.

The join query that the example to be presented computes is the following:

```
select *
from R1, R2
where (R1.unique2 == R2.unique2) and
      (R1.unique2 < 10000)
```

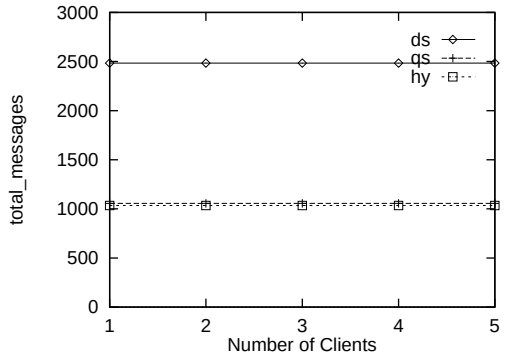

Figure 52: Client Total Messages Transmitted, Hybrid Hash Join, Shared, No Memory, Server Buffer 1024 K, Client Buffer 512K

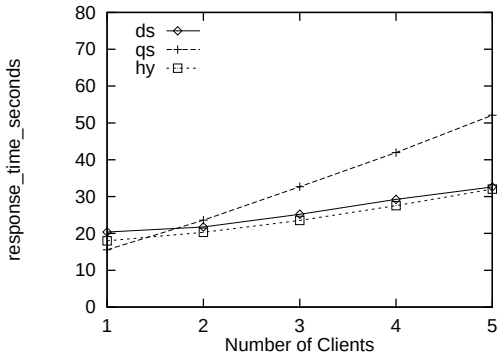

Figure 53: Client Response Times, Index Nested Loop, Shared, Server Buffer 1024K, Client Buffer 512K

The query shipping plan implemented does a selection on the outer relation using an index scan, and then uses this selection for the index nested loop join algorithm. The data shipping plan used is similar, but the selection on the outer relation is done by a remote index scan from the client, which then performs the join by using the index nested loop join algorithm (which also does remote index scans for the inner relation). Finally, the hybrid shipping plan, pushes the outer selection to the server (which does it by using an index scan) and then using the resulting tuples, the client computes the join by using the index nested loop join algorithm (which does remote index scans for the inner relation). All the indices used are clustered indices. As in previous sections, we will look at two cases: one in which all clients shared the same set of relations/indices and another one where each client uses a unique set of them.

The response times for different combinations of server/client buffer sizes, for the case where all the clients use the same relation and associated index can be seen in figures 53, 54, 55, 56.

Query Shipping

Increases in server buffer size, promote some reductions in the response time observed for the query shipping strategy (compare figures 53 and 55). These reductions in response time were more noticeable, when the number of clients in the system was low, as for higher loads, the computation required for the join computation overcomes any benefits of caching.

Figures 57 and 58 illustrate the server cpu utilizations observed for the small and big server buffer size cases respectively, for this index nested loop join example. The reader will notice that the server utilization of the query shipping strategy gets very close to 1 after one client, what implies that the computation of the join when following a query shipping strategy is very computationally demanding.

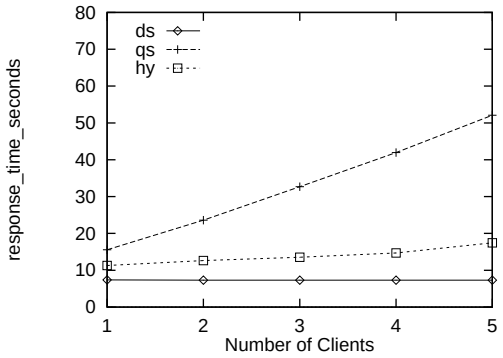

Figure 54: Client Response Times, Index Nested Loop, Shared, Server Buffer 1024K, Client Buffer 6144K

An example of the impact an increase on I/O has on the query shipping strategy, in terms of response time, can be observed in figure 59 (compare against figure 55).

It is interesting to notice the big impact on the server device disk utilization an increase on I/O has on the server as is illustrated in figure 60 (the case for reduced I/O generate almost zero I/O as everything fitted in the cache).

Data Shipping

When comparing the response time results for small and big server buffer sizes (figures 53 and 55) we can see that the server buffer size increase provides some improvement in the response time observed, specially as the number of clients increases. However, as we

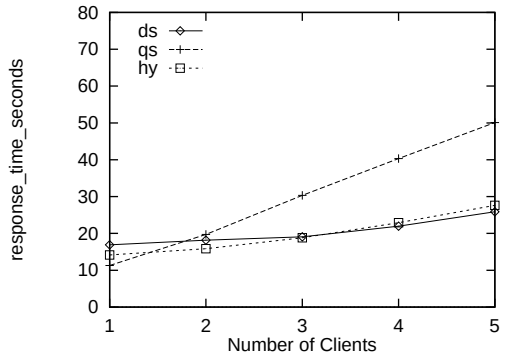

Figure 55: Client Response Times, Index Nested Loop, Shared, Server Buffer 12288K, Client Buffer 512K

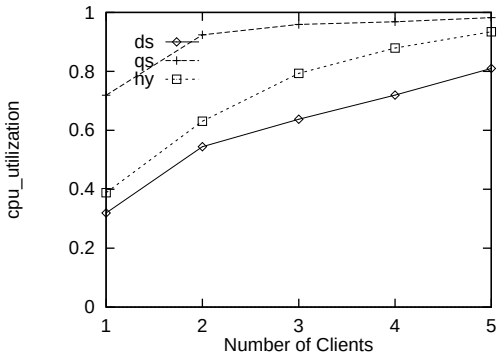

Figure 57: Server Cpu Utilization, Index Nested Loop, Shared, Server Buffer 1024K, Client Buffer 512K

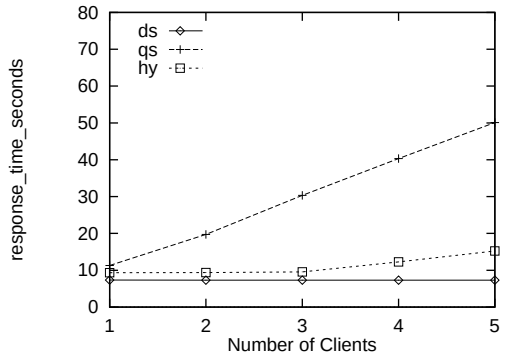

Figure 56: Client Response Times, Index Nested Loop, Shared, Server Buffer 12288K, Client Buffer 6144K

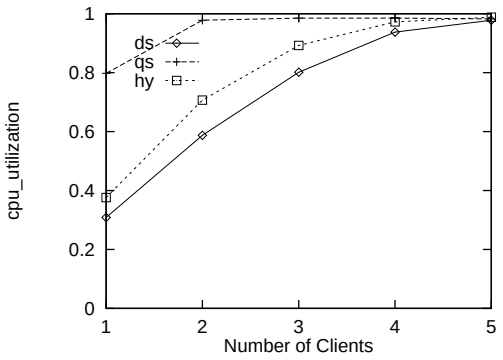

Figure 58: Server Cpu Utilization, Index Nested Loop, Shared, Server Buffer 12288K, Client Buffer 512K

have seen already, the impact a big client buffer size (figures 53 and 54) has on response time is higher than increases in server buffer size.

Typical server and client cpu utilizations observed for the data shipping strategy when a small client buffer size was used, can be seen in figures 58 and 61, respectively. The big client buffer case, generated client cpu utilizations close to one and server cpu utilizations close to zero.

The data shipping strategy was very sensible to an increase on I/O at the server. Figure 59 shows the response time observed when the I/O was increased in the system. The corresponding disk utilization can be observed in figure 60.

Hybrid Shipping

Increases in the server buffer size for a particular client buffer size, generates some improvements in the response time observed for the hybrid shipping alternatives. For example, for a server buffer size of 1024K and client buffer size of 512K (figure 53), executing the join takes a client approximately 32 seconds when five clients are being processed in the system, while when using a server buffer size of 12288K (figure 55) the response time goes to 28 seconds.

The effect of a client buffer size increase was quite significant in the hybrid shipping case as it was for the data shipping case. For example, we can see that given a server buffer size of 1024K, an increase of client buffer size from 512K to 6144K (see figures 53 and 54) reduced the response time observed for a client (when the number of clients in the system is five) by approximately 14 seconds.

An example of the client and server cpu utilizations generated by the hybrid shipping alternative can be seen in figures 62 and 63. Notice that different to the data shipping case, a big client buffer does not generate a client cpu utilization close to one; the interaction required with the server in order to download

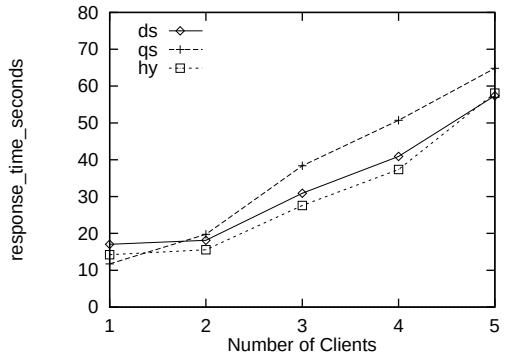

Figure 59: Client Response Times, Index Nested Loop, No Shared, Server Buffer 12288K, Client Buffer 512K

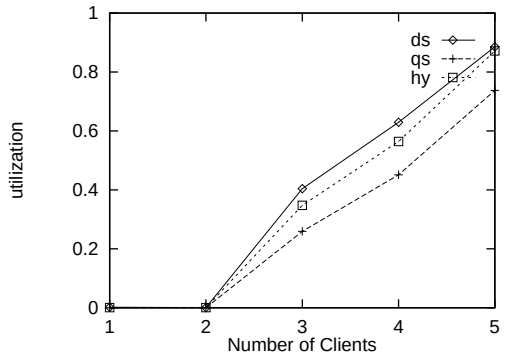

Figure 60: Server Device Disk Utilization, Index Nested Loop, No Shared, Server Buffer 12288K, Client Buffer 512K

the outer selection of the join (which is computed at the server), is the responsible for this.

The hybrid shipping strategy provides a alternative to exploit the client cache and to do load balancing, that would not be possible if strict query and data shipping strategies were followed. When a data shipping strategy determines it is missing some data, it will download it to the client, what could destroy the current content of the cache, avoiding its future use. The hybrid alternative avoids this situation by computing what it needs at the server, avoiding any alteration of the client cache content. In addition, the hybrid alternative reduces the amount of data transmitted, in order to finalize the computation at the client. The hybrid shipping alternative, in this way, provides more choices on how to exploit in a better way the current status of the system.

Network Usage

An example of the network usage the different al-

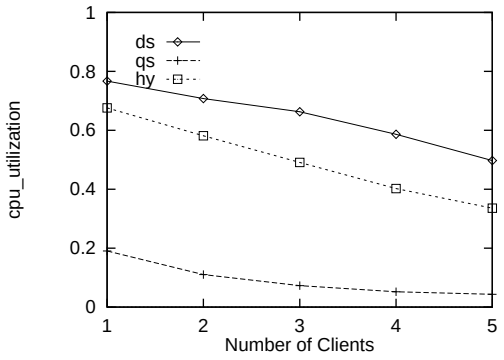

Figure 61: Client Cpu Utilization, Index Nested, Shared, Server Buffer 12288K, Client Buffer 512K

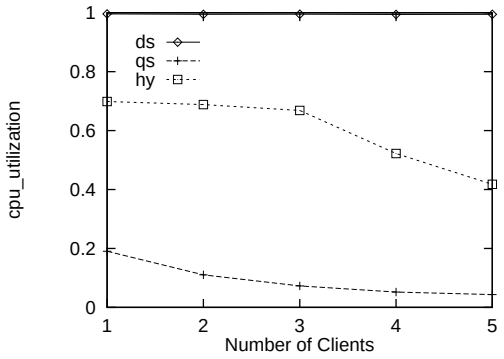

Figure 62: Client Cpu Utilization, Index Nested Loop, Shared, Server Buffer 12288K, Client Buffer 6144K

ternatives generated for the small client buffer size, can be seen in figures 64 and 65.

5 Conclusions

When comparing the query, data and hybrid shipping implementations of the relational operators investigated, we can not claim that one is superior to the other. Reality is that each one can outperform each other. We have seen examples where, for the processing of the same query, each strategy shows a performance advantage against the other depending on the run time environment (cache content, number of clients in the system, etc.). In spite of this, certain trends were observed:

1. Query shipping is the alternative to use for cases where the server resources are plenty, and the load in the system is low (we saw this behavior repeatedly, in the tests performed).

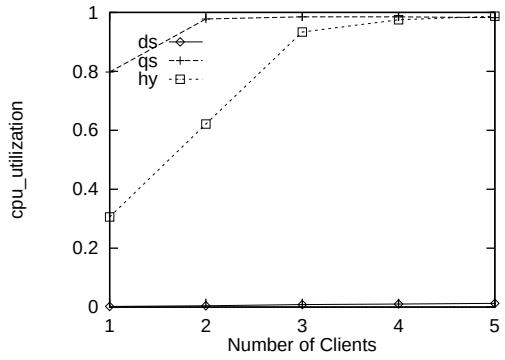

Figure 63: Server Cpu Utilization, Index Nested Loop, Shared, Server Buffer 12288K, Client Buffer 6144K

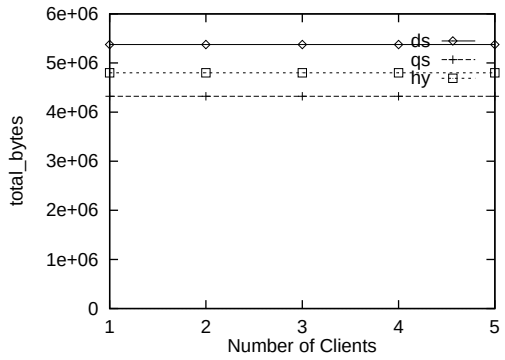

Figure 64: Client Total Bytes Transmitted, Index Nested Loop, Shared, Server Buffer 1024 K, Client Buffer 512K

2. The management of the client buffer can really impact the performance of data and hybrid shipping strategies. We have seen impressive performance gains in cases where the client buffer was used in an efficient (e.g., indices) manner by the client.

3. For computational expensive queries (e.g., joins) data shipping provides a better response time than query shipping (as the load in system increases) given that the client buffer can be partially exploited by the strategy. Data shipping promotes a better performance by moving a big part of the computation task to the client (server offloading) and by reducing the network traffic.

4. Hybrid shipping plans have shown to be able to outperform query and data shipping plans by exploiting in an efficient manner the state (server/client load, cached relations, etc .) of the Client-Server system.

5. The offloading effect that data shipping has on the server promotes scalability in the N-Client 1-Server architecture.

6. The network factor seems to have been overestimated in N-Client 1-Server architectures. As local area networks become faster, it seems that load balancing between the server and clients is the real issue to be addressed.

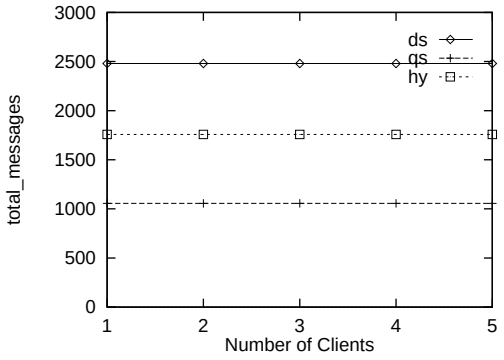

Figure 65: Client Total Messages Transmitted, Index Nested Loop, Shared, Server Buffer 1024 K, Client Buffer 512K

As part of our future work, we will explore the performance of other relational operators, and how updates perform when following the different strategies studied. Experiments in an ethernet local area network will also be performed.

6 Acknowledgments

We would like to thank Michael Zwilling, for providing us with information about how to use the SHORE storage manager facilities, in order to design Tornadito. Also we would like to thanks Markos Zaharioudakis for his support in the remote scans implementation of the system, Amit Shukla for providing us with the hybrid hash join-code used, Michael Chang for his support during the installation of SHORE and the staff members of the University of Maryland Institute for Advance Studies for their help with the IBM rs/6000 machines.

References

[CAREY94] Carey, M., DeWitt, D., Franklin, M., Hall, N., McAuliffe, M., Naughton, J., Schuh,

D., Solomon, M., Tan, C., Tsatalos, O., White, S., and Zwilling, M., "Shoring Up Persistent Applications", *Proc. ACM SIGMOD Conf.,* Minneapolis, MN, May 199 4.

[FRAN95] Franklin, M., Jonsson, B.T., Kossmann, D., "Performance Tradeoffs for Client-Server Query Processing", *Proc. ACM SIGMOD Conf.,* Montreal, Canada, June 1996, to appear.

[GRAE94] Graefe, G., "Volcano-An Extensible and Parallel Query Evaluation System", IEEE Transactions on Knowledge and Data Engineering, vol. 6, no. 1, Feb 1994.

[GRAY93] Gray, J., "The Benchmark Handbook for Database and Transaction Processing Systems, Second Edition", Morgan-Kaufmann, San Mateo, CA, 1993.

[KOSS95] Kossmann, D., Franklin, M., "A Study of Query Execution Strategies for Client-Server Database Systems", Technical Report CS-TR-3512 (also UMIACS-TR-95-85), University of Maryland, College Park, August 1995.

[SHAPI86] Shapiro, L., "Join Processing in Database Systems with Large Main Memories", *ACM Transactions on Database Systems*, Vol. 11, No. 3, (Sept. 1986), 239-264.