

Using J2EE/.NET Clusters for Parallel Computations of Join Queries in Distributed Databases

Yosi Ben-Asher, Shlomo Berkovsky¹, Eduard Gelzin, Ariel Tammam, Miri Vilkhov
 Computer Science Department, Haifa University, Haifa, Israel
 Edi Shmueli
 I.B.M. Research Center, Haifa, Israel

Abstract: In here we consider the problem of parallel execution of the Join operation by J2EE/.NET clusters. These clusters are basically intended for coarse-grain distributed processing of multiple queries/business transactions over the Web. Thus, the possibility of using J2EE/.NET clusters for fine-grain parallel computations (parallel Joins in our case) is intriguing and of practical interest. We have developed a new variant of the SFR algorithm for parallel Join operations and proved its optimality in terms of communication/execution-time tradeoffs via a simple lower bound. Two variants of SFR algorithm were implemented over J2EE and .NET platforms. The experimental results show that despite the fact that J2EE/.NET are considered to be platforms that use complex interfaces and software entities, J2EE/.NET clusters can be efficiently used to execute the Join operation in parallel.

Categories and Subject Descriptors

C.2.1 [Network Architecture and Design]: [Network communications]; H.2 [Database Management]

General Terms

Net Clusters, Distributed Databases

Keywords: J2EE/.NET clusters, NET Clusters, distributed databases, Parallel Computations

Received 30 Oct. 2004; Reviewed and accepted 30 Jan. 2005

1 Introduction

Server-side technologies have grown up dramatically over the past years. Various Web-based application servers have emerged as a convenient mean for numerous financial, E-Commerce and business applications. Commercial products supporting these technologies are developed by IBM (WebSphere [2]), BEA (WebLogic [17] [1]), SUN (Sun One [10]) and Oracle (Oracle application server [9]). These products support high level programming architectures such as J2EE (Java2 Enterprise Edition) [5] developed by SUN and .NET [8] framework developed by Microsoft. Both of them simplify enterprise applications development by basing them on standardized, modular components, by providing a complete set of services to those components, and by handling many details of application behavior automatically, without complex programming. In particular, J2EE/.NET platforms are designed to support distributed Web applications allowing a set of multiple clients to perform concurrent queries to a remote database over the Web.

In this work we show that the J2EE/.NET architectures can be used not only to easily implement a concurrent set of queries but also to improve parallel processing of a single query. In particular we show that the parallel computation of Join [24] query (one of heaviest types of queries) can still obtain high speedups over J2EE/.NET platforms. We remark that due to the complexity of J2EE/.NET (support many high-level features and standards) it is not clear in advance that they can be also used to efficiently parallelize processing of queries. The implemented algorithm has some novel concepts in respect to previous works on parallel algorithms for executing the Join operation (pipeline of messages, binary filtering trees and random distribution of new rows). The optimality of this algorithm is proved via a simple lower bound. Thus, beside the proof of concept for parallel Joins over J2EE/.NET clusters, this work also contributes to

the theoretical knowledge regarding the parallel algorithms for the Join operation.

A J2EE/.NET application can be viewed as a collection of independent components (Java Beans in J2EE context [4], or .NET components in .NET [13]), communicating each other using various types of messages. Each component can be accessed remotely over the Web by any other component that has a "handle" to the target component. For the sake of simplicity we will describe the principal architecture of J2EE platform only. Although the name of the architectural layers in .NET are different, their capabilities basically remain the same.

There are several types of beans (figure 1), each implementing a different functionality in the J2EE application. The application can be viewed as a distributed multi-tiered structure. We will shortly describe the principal tiers:

- Client-tier: contains the application clients, initiating queries using applets or HTML pages, through the Web browser of the client's machine.
- Web-tier: contains Servlets [4] or JSP/ASP pages [4] [13] that receive the clients requests (as HTTP or XML), parse them and activate the appropriate methods of the Business tier components. Later, the components of the Web tier also deliver the result of the client's query back to the client. The Web tier components are activated by the Web server on the machine that hosts the relevant components (figure 1).
- Business-tier: contains the implementation logic of the query. It is based on a distributed asset of components that can communicate each other via messages and Message Driven Beans (MDB [4]) in J2EE, Microsoft Message Queue (MSMQ in [13]), or directly by activating remote methods. These components are managed and executed by special servers called Application servers.
- Database-tier: contains components that can access to the set of Databases which act as a persistent storage of the application data. The components allow the J2EE/.NET application to perform database queries via a set of database servers (figure 1).

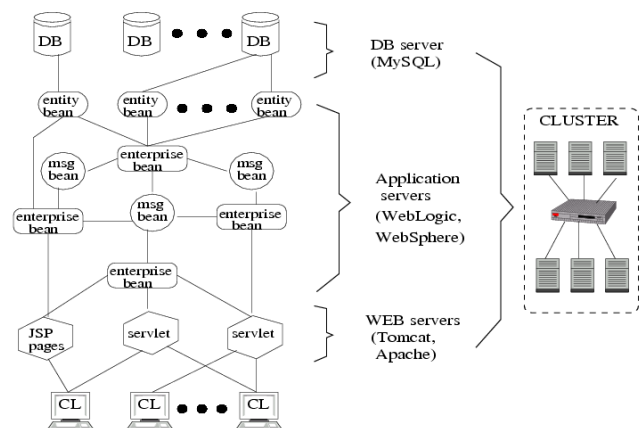


Figure 1 The J2EE Architecture

¹ This work was supported by a research grant from Caesarea Edmond Benjamin de Rothschild Foundation Institute for Interdisciplinary Applications of Computer Science (C.R.I.).

A J2EE/.NET application should serve a dynamic set of Web clients obtaining reasonable response times. Consequently, the components of the platform (including the database servers) can be partitioned between a set of machines constructing a given cluster. Most of the modern Application Servers and the database servers support multi-transaction mode of execution [15]. Thus transactions updating the same data in the databases are executed atomically. In addition, in case of an error in one of the components, the set of databases is automatically rolled back to a persistent state before the last transaction started. Thus, it is possible to comprehend J2EE and .NET platforms as a novel convenient environments and programming methodologies for the implementation of distributed applications over a cluster.

It is a natural question if J2EE/.NET clusters can be used not only to process multiple queries but also to speedup the execution of a single query. This motivation is supported by the fact that large clusters can be easily built using common PCs and regular communication medias. Thus it is not unlikely to foresee a growing use of large J2EE/.NET clusters in academia and industry.

For such a large cluster, we may also require true parallel processing of a single query. Basically, current servers do support concurrent execution of multiple queries over a cluster, but fail to support parallel execution of a single query. To achieve a parallelization, the query should be parsed, recompiled, and be later executed in parallel over a number of computation resources. We remark that parallel computations of this kind are not supported yet neither by J2EE and .NET, nor by the leading servers' manufacturers.

2 The Parallel Join Algorithms

In here we describe the parallel Join algorithm used by the proposed cluster and prove its optimality via a simple lower bound. The Join operation computes a Cartesian Product of two database tables A Join B returning only the tuples $\langle A_{row_i}, B_{row_j} \rangle$ that satisfy a given condition. For instance, if the tables A and B contain the lists of students in the university, we would like to find the pairs of students, where $\{ \langle A_{row_i}, B_{row_j} \rangle \mid A_{row_i}.age > B_{row_j}.age \}$. We focus on the parallel execution of the Cartesian Product part of the Join, assuming that no preceding optimization stages such as sorting or filtering can be operated to reduce the amount of tuples that are generated by the product (see [23] for a discussion about such possibilities).

Let n be the number of rows in each table (A , or B) and p the number of machines in the cluster. We assume that the two tables A and B are partitioned between some $k < p$ machines in any arbitrary way. Thus, if all of the combinations are to be created, then the best possible parallel computation time is n^2/p . A naive solution will be to partition A between the p machines and to replicate B in each machine, in total p times. In this way each machine will compute a different part of the product (n^2/p in each machine). This solution, though optimal in the number of combination each machine computes, requires minimal of n communication steps, as B will be broadcasted to every machine. The problem we consider is to find a parallel algorithm that minimizes both the communication and the computation requirements. In J2EE clusters, due to the fact that components communicate through Message Driven Beans, the communication minimization is particularly important.

Now we will briefly review the related works. The Fragment and Replicate (FR) algorithm [14] for distributed computation of Join operation is one of the earliest algorithms using parallelization in distributed systems. FR algorithm fragments one of the tables between the p machines and replicates the other table across the rest of the machines. This results in a relatively high communication costs. The Symmetric Fragment and Replicate (SFR) algorithm proposed in [21] reduces the communication costs by fragmenting both the tables to \sqrt{p} equal parts and replicating each fragment across machines. This significantly decreases the amount of data communicated, and does not change the amount of computation in each machine. SFR [21] also proves the communication minimality, however, it is proved under initial assumptions which are not necessarily true.

A number of recent works refer to the issue of distributed query processing optimizations on the Web. For instance, [12] presents a distributed query processing using asynchronous messaging. In this approach a query computation path is shortened when a part of it is satisfied by a specific machine. Another kind of computation is described in [22]. There the query is sent to the sites, processed locally, and finally the results are sent back for a recombination. Many of the works in distributed processing of database queries (see [16], [20] and [19]) address the issue of how to optimally compute a complex query distributing the sub-queries between the different machines of the cluster. For distributed databases two-step optimization is generally used. In the first stage, at the compile time, a plan, specifying the Join order, methods and access path is executed. In the second stage, just before the actual execution, the machines to perform the operation are selected. In here we focus only on the parallel execution of the Cartesian Product part of the Join operation.

As observed in previous works, there is a trade-off between the communication and the efficiency of a parallel Join algorithm, as the whole Cartesian Product can be computed in one machine without any communication overhead.

We turn now to the proof of the optimality of the SFR algorithm. Though we basically used the SFR algorithm, we have to prove its optimality. The reason is that in [21] the proof for optimality is based on the assumption that the sub-part of the Cartesian Product that is executed by some machine is "rectangular" (i.e., the number of columns in each row is equal). Since we consider only two-way Join, in our case the problem can be formulated as follows:

Definition 2.1: Let the Cartesian Product of two tables, A and B , each of size n , be represented by an $n \times n$ mesh (G), where each unit represents a different combination of rows in the product. We are interested to find a partition of G to rectangles such that each rectangle is colored by one of p colors. The problem is to find an optimal partition that minimizes both the overall circumference of each color and the overall area of the rectangles of each color.

The intuition behind the above definition is that each rectangle colored by p_i represents a sub-part of the product that will be computed by the machine p_i in the cluster. The circumference of this rectangle models the communication needed to transmit the relevant part of A and B to the machine p_i .

Figure 2 depicts the differences between the FR, SFR and another alternative solution. The FR algorithm (figure 2 left) copies B to each machine, thus the communication overhead, circumference of the rectangle allocated to p_i , is 10. The SFR algorithm (figure 2 middle) improves this factor and the circumference of the rectangle allocated to p_i is 8. However, theoretically other non-rectangular partitions are also possible and we need to prove that any other partition, different from that of the SFR, will increase the total circumference of each p_i . For example, the circumference in the partition of some alternative algorithms (figure 2 right) is 16 for any p_i .

	A1	A2	A3	A4		A1	A2	A3	A4		A1	A2	A3	A4
B1	p1	p2	p3	p4	B1	p2	p2	p3	p3	B1	p1	p2	p3	p2
B2	p1	p2	p3	p4	B2	p2	p2	p3	p3	B2	p3	p4	p1	p3
B3	p1	p2	p3	p4	B3	p1	p1	p4	p4	B3	p4	p2	p3	p1
B4	p1	p2	p3	p4	B4	p1	p1	p4	p4	B4	p1	p4	p2	p4

Figure 2 Three Possible Partitions of a Cartesian Product Computation of 4 Machines

Theorem 2.1: The optimal partition of the structure defined in 2.1 is obtained using the SFR algorithm by allocating a square of $(n / \sqrt{p}) \times (n / \sqrt{p})$ to each machine (assuming that \sqrt{p} is a natural number).
Proof: Assume that there is a non-square partition \acute{a} of the mesh G such that each machine / color has a lower circumference than in

the SFR partition. Clearly, the total circumference of \acute{a} can only decrease if we "glue" all the sub areas of each color to one shape (true for any form of gluing). Thus after gluing we have p shapes each with a smaller circumference than $4n / \sqrt{p}$. Since G is a mesh, the edges of each shape in \acute{a} are straight lines (parallel to one of the axes of G). It is now possible to improve the circumference of any shape of \acute{a} by "smoothing" corners. This process, depicted in figure 3, evolves by selecting a "convex" corner, "peeling" a prominent sub-part of this corner and filling it in another "concave" corner of the shape. Clearly, after the peeling, the circumference of each shape gets smaller. This process is repeated until each shape has no more concave corners. Finally, a shape whose edges are straight lines, with no concave corners and minimal circumference, must be a square. This contradicts the fact that \acute{a} has a smaller circumference than the squares of the SFR algorithm.

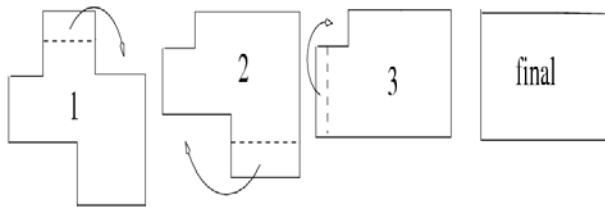


Figure 3 Smoothing a convex corner by peeling.

This proof can be easily extended to a higher dimension and be applied to an n -way Join. Note, that using the SFR algorithm implies that each table will be partitioned to n / \sqrt{p} parts. Thus, out of the p machines of the cluster, we will use only \sqrt{p} databases, letting each of these \sqrt{p} machines hold one fragment of each table. We remark that the duplication of each fragment to \sqrt{p} machines can be done in a pipeline manner. Thus, a fragment containing n / \sqrt{p} rows of a table, can be duplicated to \sqrt{p} machines in $\sqrt{p} + n / \sqrt{p}$ steps, letting each machine to resend the accepted data in a pipeline manner to the next neighbor. In this respect working in a pipeline mode is reasonable when using a cluster, since at any given time there will be at most p messages in the cluster. Assuming that the communication network can support p messages, the communication should not be a bottleneck in such a cluster. The main obstacle for obtaining speedups can only be the result of significant overheads involved with the three types of servers used in the cluster (Web servers, Application servers and database servers).

3 Structure of the J2EE server

In here we describe the J2EE implementation of the SFR algorithm. It is clearly not sufficient to implement only the SFR algorithm, as the Join operation is always part of a query whose overall execution could potentially dominate the possible speedups of parallel Joins. Thus, the J2EE server supports the execution of relatively simple queries. The queries have the following syntax

$$T^A \times T^B / F \Rightarrow U_{T_A}, U_{T_B}, D_{T_A}, D_{T_B}, I_{T_A}, I_{T_B}$$

where T^A and T^B are regular JQL [6] (a Java interface to execute SQL) commands, F is a general Java function implementing a condition on the tuples (e.g., $T^A.x + T^B.y < 100$), and U, D and I are user-defined methods handling respectively Update, Deletion and Insertion operations.

Basically the server performs the following sequence of actions: (1) Computes a Cartesian Product of two tables $A \times B$; (2) Restricts the resulting combinations only to those satisfying the function F ; (3) Filters the combinations in the product which will result in updates,

deletions, or insertions of the same rows in A or B ; (4) Updates the databases by modifying, deleting or inserting rows in A or B .

Figure 4 depicts the proposed bean configuration of the server where $p=4$ and two tables T^a and T^b are partitioned between two databases. The figure shows the beans that are relevant for the query:

$$Q = T^A \times T^B / F \Rightarrow \{T^a.x = T^a.x + T^b.y, T^b.y = T^a.x * T^b.y\}$$

The above query can be explained as "Choose a pair of rows from table A and B satisfying the restriction F . Then update the field x of the row from table A the field y of the row from table B with the sum and the product (respectively) of their values prior to the query execution".

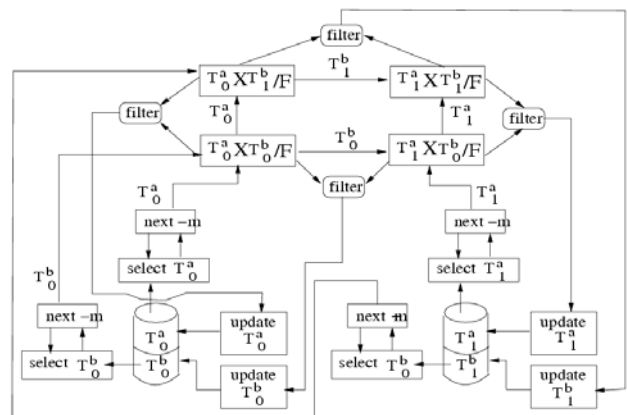


Figure 4 Bean configuration for the i^{th} column of the mesh.

Now we will describe the basic stages of query execution in our system:

1. A client activates a servlet in one of the cluster's machines chosen at random (this distributes equally the requests between the machines).
2. The servlet invokes in parallel four *next-m* beans, associated with $T^a_0, T^b_0, T^a_1,$ and T^b_1 (figure 4), to extract m rows from the table. This operation is done in quanta of m rows to cover the overhead related to the accesses to database.
3. The *next-m* beans create entity beans to select the appropriate rows from $T^a_0, T^b_0, T^a_1,$ and T^b_1 tables in the database.
4. The collections of database rows, returned by the entity beans, are packed in messages and sent to the suitable "product bean" of the mesh.
5. Messages contain a fixed number (m) of rows. This allows the pipeline mode of operation and covers the overhead related to messages sending and receiving.
6. In the mesh each "product bean" (say, $T^a_0 \times T^b_0 / F$) accumulates the incoming messages. During the accumulation, each bean sends in a pipeline fashion the incoming messages to its neighbors along the column and the row, using Java Message Driven Beans (not shown in figure 4).
7. In fact, initial filtering occurs already in a "product bean", as already at this stage it is possible to filter updates, deletions and insertions to the same row.
8. The tuples, produced by the Cartesian Product $T^a \times T^b / F$, are sent to a suitable filter bean using Message Driven Beans.
9. The filter beans eliminate multiple updates, deletions and insertions to the same rows of T^a or T^b and activate the entity beans, responsible for updating the tables.
10. In each filter bean the filtering is done separately for a relevant table. Thus, concurrent updates, deletions and insertions to the tables can be handled.
11. New rows are inserted to one of the \sqrt{p} databases, chosen at random. Thus, increases the probability of

keeping the size of tables roughly equal after a number of queries.

- As well as the Select operation, the Update operation on the database rows is also done in quanta of m rows.

Note, that pipeline mode of work is essential to get low execution times. For example, letting each process that computes a sub-part of the Cartesian Product, to select its data directly from the suitable database will result in a sequential bottleneck and will prevent us from getting the right execution times. In addition, even though JMS supports broadcast, it is very inefficient to use it, since broadcasting to/from one location results in sequential execution times.

All the beans of the cluster are handled through one container and are thus working in a multi-transaction mode. No method can update some database table while a concurrent invocation of this method accesses that table. As well, recall that Insert/Delete/Update operations are executed in J2EE by applying the Create()/Remove()/Set() methods of the corresponding entity beans and the changes are rolled back in case of failure. Thus, both the properties of query atomicity and failure resilience are achieved.

Allowing many clients to access the servlet simultaneously yields possible concurrent activation of many queries. We assume that the container will handle all these queries in multi-transaction mode, so that they will not interfere each other.

4 Experimental results in J2EE

The above algorithm was implemented over a cluster of 8 machines. The hardware configuration of the machines was similar: PentiumIII-1800 CPU with 256MB of RAM memory. BEA (Weblogic) [1] application server was used as the underlying deployment infrastructure of the application. One of the machines was chosen to act as an Admin server which controlled the cluster, and the rest 7 machines were used as slaves ("managed" in BEA terminology). The Admin machine was also used to remove and add machines to the cluster (i.e. to change the cluster size) and also to deploy the application to the corresponding BEA application servers. The Admin machine itself also ran a managed BEA instance, and thus the full cluster size was composed of one admin instance and 9 managed instances spread on 8 different machines.

All the k databases holding different parts of each table should have been distributed between the k machines, however, this was impossible, as BEA application server did not allow to work with a distributed database. Technically, MySQL server does not have a JDBC [4] driver supporting distributed transactions (XA driver). Consequently, we were forced to use different tables, located in a single database to simulate the distribution of the databases and insert a dimension of concurrency to the database operations. This problem has clearly dominated the execution times. We actually believe that optimal speedups would have been obtained if we could distribute the database server. This is supported by the observation that during the experiments the Cartesian Product was completed long before the actual update to the database has completed.

The database contained two tables of integers (each row corresponds to one integer). The Join operation was used to implement the following queries:

- Q_{double} - doubles the range of numbers in the table.
- Q_{prime} - marks each number as prime / non-prime by checking if it is can be represented as a multiplication of two other number in the database.
- Q_{sum} - checks whether each number can be represented as a sum of two prime numbers.

A transaction began when submitting a query on the Web page and ended when all data was written down back to the database. To measure the transaction time, log4j [7] package was used. It recorded the timestamps of different phases of the computation from submitting the query to the completion of all writes.

Two main parameters were modified during the experiments: cluster size (p) and tables size ($n=30, \dots, 300$). We ran multiple tests and measured the execution time on different table and cluster size. We started with a cluster of size 1 and gradually increased its size to 8 machines. When the cluster contained a single machine, all the beans were deployed to the same machine. To enlarge the cluster to

2 machines, we activated another managed BEA instance and relocated approximately half of the beans to the recently connected machine and so forth. When enlarging the cluster, we always tried to deploy the application with equal as possible number of beans on every participating machine. By doing this, we attempted to equally distribute the computation operations between the cluster machines.

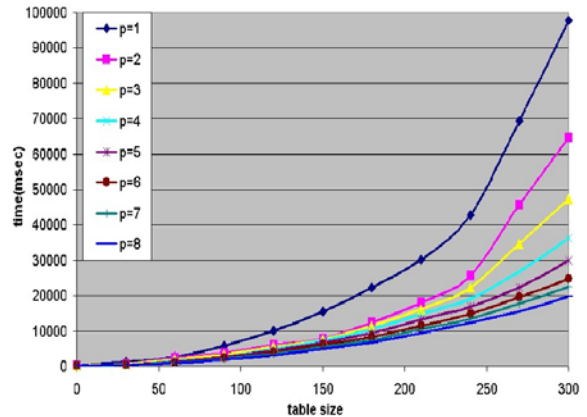


Figure 5 Execution time as a function of the table size n .

Figure 5 summarizes the results of this set of experiments. Note, that the best speedup, as expected, is achieved when the cluster contains all 8 machines. The time of 300×300 product computation with 1 machine was 97.643 sec, while with 8 machines it took only 19.849 sec. The total speedup in this case was over 4.9. This is a very good result, taking into account sequential access to the database and communication overheads. As the table size increases, the speedup improves due to the fact that the relative part of the database accesses, compared to the whole transaction computation time, decreases. Note that there is no point in using well-known benchmarks (e.g., TPC-H) as these do not include extensive Cartesian Product experiments.

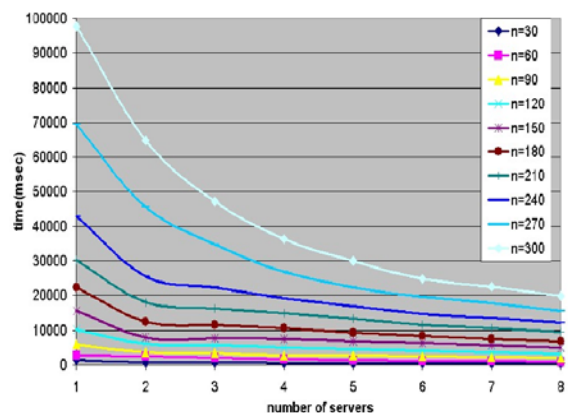


Figure 6 Execution time as a function of the cluster size p .

The results in figure 6 demonstrate the effect of changing number of machines in the cluster for various values of n . Note, that the improvement in the speedup is not linear as the number of machine increases. We believe that this is also the result of the sequential bottleneck in the database accesses.

5 .NET implementation and experimental results

In this section we shortly describe the results of implementing the parallel query engine using Microsoft .NET platform instead of J2EE platform. .NET is an equivalent technology to J2EE with similar functionality and parallel software components to those available in J2EE. Thus, it was important to see that speedups can be also obtained using .NET technology. We do not give here a detailed description of the .NET implementation, but simply indicate the different components that were used. Moreover, in .NET we were able to distribute the database servers between different machines

- Distributing the databases allows concurrent access to the database. Operating the sequential Cartesian Product algorithm obtains a minor speedup of $24.7/17.2=1.43$.

compared to our J2EE setting where due to technical limitations the databases were located on the same machine. The results of the .NET experiments thus indicate what can be the impact of distributing the databases to different machines.

Each of the components was implemented as a Dynamic Link Library (DLL). The main component is an ASP.NET project which gives us the opportunity to invoke the system using a Microsoft Internet Explorer browser. Microsoft Message Queue (MSMQ) [13] was used for communication and COM+ services [13] were used to implement distributed transactions over a set of SQL-DBs. A schematic description of a .NET cluster is given in figure 7, where IIS stands for Internet Information service [3]. IIS is a Web server providing a highly reliable, manageable and scalable infrastructure for .NET platform.

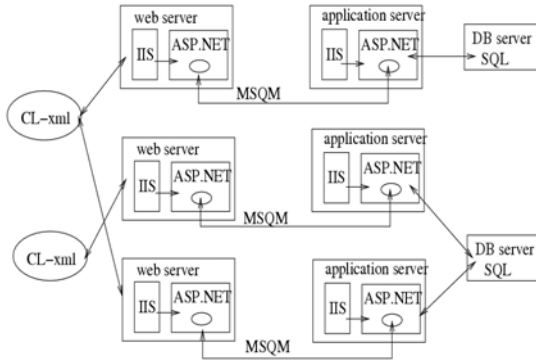


Figure 7 General scheme of a .NET cluster.

The parallel Cartesian Product computation was used to implement a SISC (Similarity-based Soft Clustering) algorithm for documents clustering [18]. The algorithm uses an iterative approach to merge clusters according to a similarity function between documents representing clusters. The similarity is proportional to the number of common key-words in two documents. This algorithm is implemented by repeatedly applying a query (called the main-query) over a set of databases where every document is represented by a single entry in a large table. Thus, the Cartesian Product part of the main query is used to compare every two documents in the databases in parallel. The merging of clusters is implemented by updating a part of the main query. Finally, the implementation produces a set of clusters, such that each document belongs to several potential clusters, where each cluster is characterized by a set of keywords and a topic, as shown in the following table.

Cluster	Files in cluster	Words of cluster	Cluster topic
1	51127.txt, 64830.txt	void, stack, socket	Hardware, X - Windows
2	52766.txt, 52768.txt	logarithm, parallel, rate, gain	electronics, cryptograph
3	20540.txt, 20571.txt, 51131.txt	wall, disappoint, confess	Religion, christian, atheism
4	102590.txt, 102655.txt, 20506.txt, 20540.txt	cheat, strict, adapt	baseball
5	20554.txt, 20571.txt, 20602.txt, 82796.txt, 82804.txt	christ, match, check, teach	Religion, Christian, talk about religion

200 documents from the UCI KDD corpus [11] were used as an input. This set contains documents from 20 different news groups (10 documents in each group). Auxiliary external software tools were used to parse and process the documents and build the database entries. The experiments were performed on a cluster of eight machines, similar to the machines of J2EE experiments. The following table shows the execution times (in seconds) for distributed / centralized database and operating/not operating the parallel Cartesian Product algorithm. *na* values in the table indicate that this result is not available.

centralized/distributed	1 - DB	2 - DB	3 - DB	4 - DB
sequential algorithm, centralized DB	24.7	na	na	na
sequential algorithm, distributed DB	19.5	18.6	17.9	17.2
parallel algorithm, centralized DB	11.52	na	na	na
parallel algorithm, distributed DB	na	6.59	6.23	5.87

The results show that:

- Parallelizing the Cartesian Product algorithm without distributing the access to the databases obtains a speedup of $27.7/11.52=2.14$.
- Distributing the centralized database to four databases and also parallelizing Cartesian Product computation algorithm obtains a speedup of $24.7/5.87=4.2$.

Thus, it is possible to conclude that the effect of distributing the databases is relatively minor to the effect of parallelizing the algorithm in computation intensive tasks such as the Cartesian Product computation.

6 Conclusions

In here we considered the problem of parallel Joins in cluster, based on J2EE/.NET technology. The SFR parallel algorithm, implemented in our experiments, showed that in spite of the considered J2EE/.NET complexity, it still can be used in fine-grain parallel computations. To reach satisfactory experimental results in the current implementation, the following enhancements were used:

- Usage of message pipelining as a crucial technique to reduce communication delays and overlap database accesses with computations.
- Reduction of memory size usually needed by J2EE/.NET to compute large-size products. This was achieved, for example, by replacing the use of entity beans with direct access to the database using JDBC (for J2EE implementation).
- Usage of probabilistic load balancing scheme enabling us to ensure that the number of rows in the sub-tables roughly remains equal.
- Usage of multi-transaction mode of execution to preserve both the database consistency and recoverability (in spite of the concurrent execution of multiple queries through the Web).
- J2EE/.NET overall communication performance is known to be quite low, so that messages containing $m>1$ rows of the database tables were used to hide the overhead associated with messages in J2EE.

We believe that optimal speedup could have been obtained in J2EE implementation if the underlying application server could support transaction on a distributed set of databases. This is verified by the results over .NET platform, showing that the distribution of the database allows to obtain additional speedups of the parallel algorithm.

We found that J2EE/.NET significantly shortens software development times. However, it increases the execution time due to the associated overhead of using the beans and internal scheduling. Thus, there is no point in comparing the execution time of the proposed J2EE cluster with a naive sequential implementation, working directly over the databases. The proposed system is useful only in the context of J2EE/.NET applications, where J2EE/.NET is used due to its ability to quickly build distributed Web applications over databases.

References

- [1] BEA, WebLogic. <http://www.bea.com/products/weblogic> server
- [2] IBM, WebSphere. <http://www.ibm.com/software/info1/webSphere>
- [3] Internet Information Services. <http://www.microsoft.com/windowsserver2003/iis/default.mspx>
- [4] J2EE 1.4 Tutorial. <http://java.sun.com/j2ee/1.4/docs/tutorial/doc>
- [5] Java2 Platform, Enterprise Edition. <http://java.sun.com/j2ee>
- [6] JQL Query Language. <http://www.jbase.com>
- [7] Log4j project. <http://logging.apache.org/log4j/docs>
- [8] Microsoft .NET Platform. <http://www.microsoft.com/net>
- [9] Oracle Application Server. <http://www.oracle.com/appserver>
- [10] Sun Microsystems, SUNTM Open Net Environmen. <http://www.sun.com/sunon>.
- [11] UCI KDD archive. <http://kdd.ics.uci.edu/databases/20news/20news.html>
- [12] S. Abiteboul and V. Vianu. Regular path queries with constraints. In Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, pages 122-133. ACM Press, 1997.

- [13] D. Chappell. *Understanding .NET: A Tutorial and Analysis*. Addison-Wesley Publishers Inc., 2002.
- [14] R. Epstein, M. Stonebraker, and E. Wong. Distributed query processing in a relational database system. In *Proceedings of the 1978 ACM SIGMOD international conference on management of data*, pages 169-180. ACM Press, 1978.
- [15] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem. Coordinating multi-transaction activities. Technical report, Princeton, NJ, Feb 1990.
- [16] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 102-111. ACM Press, 1990.
- [17] D. Jacobs. Distributed computing with bea weblogic server. In *Proceedings of the International Conference on Innovative Data Systems Research*, Asilomar, CA, 2003.
- [18] K. I. Lin and R. Kondadadi. A similarity-based soft clustering algorithm for documents. In *Proceedings of the Seventh International Conference on Database Systems for Advanced Applications*, Hong Kong, China, 2001.
- [19] J. Smith, A. Gounaris, P. Watson, N. W. Paton, A.A. Fernandes, and R. Sakellariou. Distributed query processing on the Grid. *International Journal of High Performance Computing Applications (IJHPCA)*, 17(4), 2003.
- [20] J. Srivastava and G. Elssesser. Optimizing multi-join queries in parallel relational databases. In *Proceedings of PDIS*, pages 84-92, 1993.
- [21] J. W. Stamos and H. C. Young. A symmetric fragment and replicate algorithm for distributed Joins. *IEEE Transactions on Parallel and Distributed Systems*, 4(12), pages 1345-1354, 1993.
- [22] D. Suci. Distributed query evaluation on semi-structured data. *Database Systems Journal*, 27(1), pages 1-62, 2002.
- [23] B. Vance and D. Maier. Rapid bushy Join-order optimization with Cartesian Products. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of data*, pages 35-46, ACM Press, 1996.
- [24] C. T. Yu and W. Meng. *Principles of database query processing for advanced applications*. Morgan Kaufmann Publishers Inc., 1998.