Performance Evaluation of Parallel GroupBy-Before-Join Query Processing in High Performance Database Systems

Conference Paper · June 2001

DOI: 10.1007/3-540-48228-8_25 · Source: DBLP

3 authors, including:

Wenny Rahayu
La Trobe University
397 PUBLICATIONS **5,866** CITATIONS

SEE PROFILE

Performance Evaluation of Parallel *GroupBy-Before-Join* Query Processing in High Performance Database Systems

David Taniar[1], J.Wenny Rahayu[2], and Hero Ekonomosa[3]

[1)] Monash University, School of Business Systems, Vic 3800, Australia
David.Taniar@infotech.monash.edu.au
[2)] La Trobe University, Department of Computer Science and Engineering, Australia
wenny@cs.latrobe.edu.au
[3)] University Tenaga Nasional, Department of Computer Science & IT, Malaysia
hero@uniten.edu.my

Abstract. Strategic decision making process uses a lot of GroupBy clauses and join operations queries. As the source of information in this type of application to these queries is commonly very huge, then parallelization of GroupBy-Join queries is unavoidable in order to speed up query processing time. In this paper, we investigate three parallelization techniques for GroupBy-Join queries, particularly the queries where the group-by clause can be performed before the join operation. We subsequently call this query "*GroupBy-Before-Join*" queries. Performance evaluation of the three parallel processing methods is also carried out and presented here.

1 Introduction

Queries involving group-by are very common in database processing, especially in data for integrated decision making process like in On-Line Analytical Processing (OLAP), and Data Warehouse [1,3]. Queries containing aggregate functions summarize a large set of records based on the designated grouping. The input set of records may be derived from multiple tables using a join operation. As the data repository containing group-by queries normally are very huge, consequently, effective optimization of aggregate functions has the potential to result in huge performance gains. In this paper, we would like to focus on the use of parallel query processing techniques in queries, whereby group-by clauses is performed before join operations. We refer this query as "*GroupBy-Before-Join*" query.

The work presented in this paper is actually a part of final stage of a larger project on parallel aggregate query processing. Parallelization of "*GroupBy-Before-Join*" queries is the third and the final stage of the project. The first stage of this project dealt with parallelization of group-by queries on single tables. The results are reported at PART'2000 conference [8]. The second stage focused on parallelization of GroupBy-Join queries where the GroupBy attributes are different from the Join attributes with a consequence that the join operation must be carried out first and then the group-by operation. We have published the outcome of the second stage at HPCAsia'2000 conference [9]. In the third and final stage, which is the main focus of

B. Hertzberger et al. (Eds.) : HPCN Europe 2001, LNCS 2110, pp. 241-250, 2001.

this paper, concentrates on GroupBy-Join queries (like in stage two), but the join attribute is the same as the group-by attribute resulting that the group-by operation can be performed first before the join for optimization purposes.

2 GroupBy Queries: A Background

GroupBy-Join queries in SQL can be divided into two broad categories; whereby the first is group-by is carried out after join and secondly is before join. In either category, aggregate functions are normally involved in the query. To illustrate these types of GroupBy-Join queries, we use the following tables from a Suppliers-Parts-Projects database:

```
SUPPLIER (S#, Sname, Status, City)
PARTS (P#, Pname, Colour, Weight, Price, City)
PROJECT (J#, Jname, City, Budget)
SHIPMENT (S#, P#, J#, Qty)
```

For simplicity of description and without loss of generality, we consider queries that involve only one aggregation function and a single join. The queries on Figure 1 give an illustration of GroupBy-Join queries.

QUERY 1	QUERY 2:

`Select PARTS.City, AVG(Qty)`
`From PARTS, SHIPMENT`
`Where` **`PARTS.P# = SHIPMENT.P#`**
`Group By` **`PARTS.City`**
`Having AVG(Qty)>500 AND`
`AVG(Qty)<1000`
	`Select PROJECT.J#,`
	`PROJECT.Jname, SUM(Qty)`
	`From PROJECT, SHIPMENT`
	`Where` **`PROJECT.J# = SHIPMENT.J#`**
	`Group By` **`PROJECT.J#,`**
	`PROJECT.Jname`
	`Having SUM(Qty)>1000`

Figure 1.Types of GroupBy-Join Queries.

In query 1, two tables are joined to produce a single table, and this table becomes an input to the group-by operation. The purpose of Query 1 is to "group the part shipment by their city locations and select the cities with average quantity of shipment between 500 and 1000". Another example is Query 2, in which it "retrieves project numbers, names, and total quantity of shipments for each project having the total shipments quantity of more than 1000".

The main difference between Query 1 and Query 2 above lies in the join and group-by attributes. In Query 2, the join attribute is also one of the group-by attributes. This is not the case with Query 1, where the join attribute is totally different from the group-by attribute. This difference is especially a critical factor in processing GroupBy-Join queries, as there are decisions to be made in which operation should be performed first: the group-by or the join operation.

When the join attribute and the group-by attribute are different as shown in Query 1, there will be no choice but to invoke the join operation first, and then the group-by operation. However, when the join attribute and the group-by attribute is the same as shown in Query 2 (e.g. attribute *J#* of both Project and Shipment tables), it is

expected that the group-by operation is carried out first, and then the join operation. Hence, we call the latter query (e.g. Query 2) "*GroupBy-Before-Join*" query.

In Query 2, all Shipment records are grouped based on the *J#* attribute. After grouping this, the result is joined with table Project. As known widely, join is a more expensive operation than group-by, and it would be beneficial to reduce the join relation sizes by applying the group-by first. Generally, group-by operation should always precede join whenever possible. Early processing of the group-by before join reduces the overall execution time as stated in the general query optimization rule where unary operations are always executed before binary operations if possible. The semantic issues about aggregate functions and join and the conditions under which group-by would be performed before join can be found in literatures [2,4,6,10].

In this paper, we focus on cases where group-by operation is performed before the join operation. Therefore, we will use Query 2 as a running example throughout this paper.

3 Parallel Algorithms for "GroupBy-Before-Join" Query

In order to study the behavior of the algorithms, in this section we describe three parallel algorithms for *GroupBy-Before-Join* query processing, namely: *Early Distribution* scheme, *Early GroupBy with Partitioning* scheme, and *Early GroupBy with Replication* scheme.

3.1 Early Distribution Scheme

The *Early Distribution* scheme is influenced by the practice of parallel join algorithms, where raw records are first partitioned/distributed and allocated to each processor, and then each processor performs its operation [5]. This scheme is motivated by fast message passing multi processor systems.

The Early Distribution scheme is divided into two phases: *distribution* phase and *group-by-join* phase. Using Query 2, the two tables to be joined are Project and Shipment based on attribute *J#*, and the group-by will be based on table Shipment. For simplicity of notation, the table which becomes the basis for group-by is called table *R* (e.g. table Shipment), and the other table is called table *S* (e.g. table Project). For now on, we will refer them as tables *R* and *S*.

In the *distribution* phase, raw records from both tables (i.e. tables *R* and *S*) are distributed based on the join/group-by attribute according to a data partitioning function. An example of a partitioning function is to allocate each processor with project numbers ranging on certain values. For example, project numbers (i.e. attribute *J#*) *p1* to *p99* go to processor 1, project numbers *p100-p199* to processor 2, project numbers *p200-p299* to processor 3, and so on. We need to emphasize that the two tables *R* and *S* are both distributed. As a result, for example, processor 1 will have records from the Shipment table with *J#* between *p1* and *p99*, inclusive, as well as records from the Project table with *J# p1-p99*. This distribution scheme is commonly

used in parallel join, where raw records are partitioned into buckets based on an adopted partitioning scheme like the above range partitioning [5].

Once the distribution is completed, each processor will have records within certain groups identified by the group-by/join attribute. Subsequently, the *second* phase (the *group-by-join* phase) groups records of table *R* based on the group-by attribute and calculates the aggregate values on each group. Aggregating in each processor can be carried out through a sort or a hash function. After table *R* is grouped in each processor, it is joined with table *S* in the same processor. After joining, each processor will have a local query result. The final query result is a union of all sub-results produced by each processor.

Figure 2 shows an illustration of the Early Distribution scheme. Notice that partitioning is done to the raw records of both tables *R* and *S*, and aggregate operation of table *R* and join with table *S* in each processor is carried out after the distribution phase.

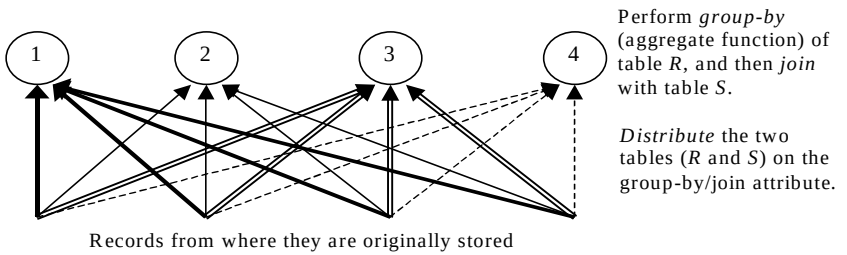

Figure 2. Early Distribution Scheme.

There are several things need to be highlighted from this scheme. *First*, the grouping is still performed before the join (although after the distribution). This is to conform with an optimization rule for such kind of queries that group-by clause must be carried out before the join in order to achieve more efficient query processing time. *Second*, the distribution of records from both tables can be expensive, as all raw records are distributed and no prior filtering is done to either table. It becomes more desirable if grouping (and aggregation function) is carried out even before the distribution, in order to reduce the distribution cost especially of table *R*. This leads to the next schemes called *Early GroupBy* schemes for reducing the communication costs during distribution phase. There are two variations of the *Early GroupBy* schemes, each of which will be discussed in the following two sections.

3.2 Early GroupBy with Partitioning Scheme

As the name states, the *Early GroupBy* scheme performs the group by operation first before anything else (e.g. distribution). The *Early GroupBy with Partitioning* scheme is divided into three phases: (*i*) *local grouping* phase, (*ii*) *distribution* phase, and (*iii*) *final grouping and join* phase.

In the *local grouping* phase, each processor performs its group-by operation and calculates its local aggregate values on records of table *R*. In this phase each

processor groups local records *R* according to the designated group-by attribute and performs the aggregate function. Using the same example as that in the previous section, one processor may produce, for example, (*p1*, 5000) and (*p140*, 8000), and another processor (*p100*, 7000) and (*p140*, 4000). The numerical figures indicate the SUM(Qty) of each project.

In the second phase (i.e. *distribution* phase), the results of local aggregates from each processor, together with records of table *S*, are distributed to all processors according to a partitioning function. The partitioning function is based on the join/group-by attribute, which in this case is attribute *J#* of tables Project and Shipment. Again using the same partitioning function in the previous section, *J#* of *p1-p99* are to go to processor 1, *J#* of *p100-p199* to processor 2, and so on.

In the third phase (i.e. *final grouping and join* phase), two operations are carried out, particularly; final aggregate or grouping of *R*, and join it with *S*. The final grouping can be carried out by merging all temporary results obtained in each processor. The way it works can be explained as follows. After local aggregates are formulated in each processor, each processor then distributes each of the groups to another processor depending on the adopted distribution function. Once the distribution of local results based on a particular distribution function is completed, global aggregation in each processor is simply done by merging all identical project number (*J#*) into one aggregate value. For example, processor 2 will merge (*p140*, 8000) from one processor and (*p140*, 4000) from another to produce (*p140*, 12000) which is the final aggregate value for this project number.

Global aggregation can be tricky depending on the complexity of the aggregate functions used in actual query. If, for example, an AVG function was used instead of SUM in Query 2, calculating an average value based on temporary averages must taken into account the actual raw records involved in each node. Therefore, for these kinds of aggregate functions, local aggregate must also produce number of raw records in each processor although they are not specified in the query. This is needed for the global aggregation to produce correct values. For example, one processor may produce (*p140*, 8000, 5) and the other (*p140*, 4000, 1). After distribution, suppose processor 2 received all *p140* records, the average for project *p140* is calculated by dividing the sum of the two quantities (e.g. 8000 and 4000) and the total shipment records for that project. (i.e. (8000+4000)/(5+1) = 2000). The total shipments in each project are needed to be determined in each processor although it is not specified in the query.

After global aggregation results are obtained, it is then joined table *S* in each processor. Figure 3 shows an illustration of this scheme.

There are several things worth noting. *First*, records *R* in each processor are aggregated/grouped before distributing them. Consequently, communication costs associated with table *R* can be expected to reduce depending on the group by selectivity factor. This scheme is expected to improve the *Early Distribution* scheme. *Second*, we observe that if the number of groups is less than the number of available processors, not all processors can be exploited; reducing the capability of parallelism. And *finally*, records from table *S* in each processor are all distributed during the second phase. In other words, there is no filtering mechanism applied to *S* prior to distribution. This can be inefficient particularly if *S* is very large. To avoid the

problem of distributing *S*, we will introduce another scheme. This is introduced in the next section.

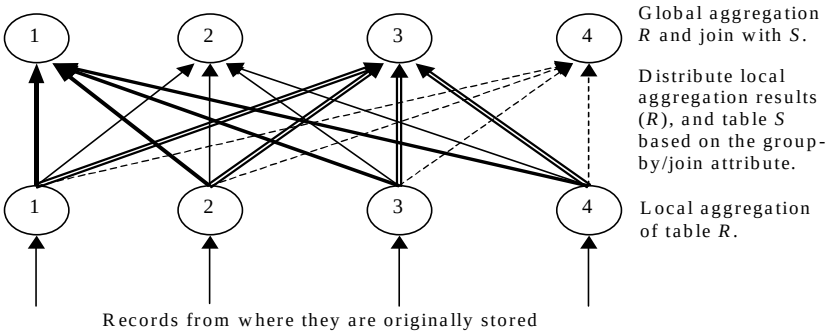

Figure 3. Early GroupBy with *Partitioning* Scheme.

3.3 Early GroupBy with Replication Scheme

The *Early GroupBy with Replication* scheme is similar to the Early GroupBy with Partitioning scheme. The similarity is due to the group-by processing to be done before the distribution phase. However, the difference is pointed by the keyword "*with Replication*" in this scheme, as opposed to "*with Partitioning*". The Early GroupBy with Replication scheme, which is also divided into three phases, works as follows.

In the first phase, that is the *local grouping* phase is exactly the same as that of the Early GroupBy with Partitioning scheme. In each processor, local aggregate is performed to table *R*.

The main difference is in phase two. Using the "*with Replication*" scheme, the local aggregate results obtained from each processor are replicated to all processors. Table *S* is not at all moved from where they are originally stored.

In the third phase, the *final grouping and join* phase, is basically similar to that of the "*with Partitioning*" scheme. That is local aggregates from all processors are merged to obtain global aggregate, and then joined with *S*. Looking into more details we can find a difference between the two Early GroupBy schemes. In the "*with Replication*" scheme, after the replication phase, each processor will have local aggregate results from all processors. Consequently, processing global aggregates in each processor will produce the same results, and this can be inefficient as no parallelism is employed. However, joining and global aggregation processes can be done at the same time. First, hash local aggregate results from *R* to obtain global aggregate values, and then hash and probe the fragment of table *S* to produce final query result. The wasting is really that many of the global aggregate results will have no match with local table *S* in each processor.

Figure 4 gives a graphical illustration of the scheme. It looks very similar to Figure 3, except that in the replication phase, the arrows are shown thicker to emphasize the fact that local aggregate results from each processor are replicated to all processors, not distributed.

Apart from the facts that the non-group by table (table *S*) is not distributed and the local aggregate results of table *R* are replicated, assuming that table *S* is uniformly distributed to all processors initially (that is round-robin data placement is adopted in storing records *S*), there will be no skew problem in the joining phase. This is not the case with the previous two schemes, as distribution is done during the process, and this can create skewness depending on the partitioning attribute values.

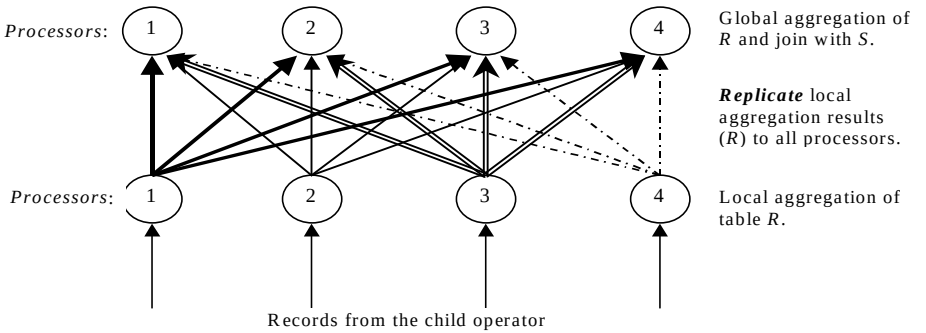

Figure 4. Early GroupBy with *Replication* Scheme.

4 Performance Evaluation

In order to study the behavior of the three methods described in this paper and to compare performance of these methods, we carried out a sensitivity analysis through a simulation. A sensitivity analysis is performed by varying performance parameters. The parameters that were varied consist of faster CPU, faster disk, faster network, faster bigger memory and number of processors. The parameters used include processor speed of 450 *Mips*, disk speed which is estimated to 3.5 *ms*, and message latency per page of 1.3 *ms*.

4.1 Result of Experiment

The graphs in Figure 5 show a comparative performance between the three parallel methods by varying the Group By selectivity ratio (i.e. number of groups produced by the query). The selectivity ratio is varied from 0.0000001 to 0.01. With 100 million records as input, the selectivity of 0.0000001 produces 10 groups, whereas the other end of selectivity ratio of 0.01 produces 1 million groups. The machine in the experiment consists of 64 processors. The graphs also show the results when variation on parameters was applied.

Using the *Early Distribution* method, more extensive data processing occurs during the first phase. In the first phase, the raw records are scanned and then distributed equally to each processor based on certain arrangement. Therefore the major cost of the method is on the scanning, loading and transferring data to every processor, and after that each processor is loaded with equal task and grouped data.

Therefore, in the second phase the total cost is minor, unless the maximum capacity of each processor exceeded. As the consequence, although the process of data transfer, aggregation and join and other processes occur after that, there are just resulting minor to the total cost. This conforms to the graph that shows the total performance is not much affected. Even when faster processors (4 times), faster communications network (4 times) and bigger memory (10 times) and less processors than the original configuration are used in the experimentation, because these factors have influences more on the second phase of the method (see Figure 6).

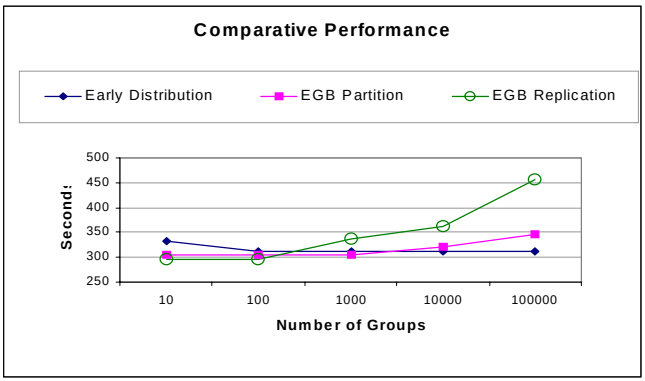

Figure 5. Comparative Performance.

In the *Early Group-By Partitioning* method, the major cost still holds by data scanning and loading. Besides that, the second major data cost is data transfer and reading/writing of overflow of bucket process. This is due to the process of early grouping occurs here. In general, almost similar to Early Distribution Method, by applying faster processors, network and bigger memory in this method does not give much impact to overall performance. However the method's performance is almost steady given number of groups increasing, except when the group number reaches 1000 then the performance starts to decline. This is conforming to the logic, that when number of group produced increasing, meaning that data volume of second and third phase to process is also increasing (also see Figure 6).

Using the *Early Group By Replication (EGB Replication)* method, the result is different compared to two previous methods. In this method, the major costs exist in all three phases of the method. The major costs are the data scanning and loading, data transfer and aggregate and join processes. During the first phase, the data scanning and loading is the major cost. This is as the result of the early grouping process. Then during the second phase, data transfer contributes in decreasing the performance, as all data is sent to all processor for replication. Finally during the third phase, also pooling all data to all processor is decreasing the performance, especially if number of groups growing.

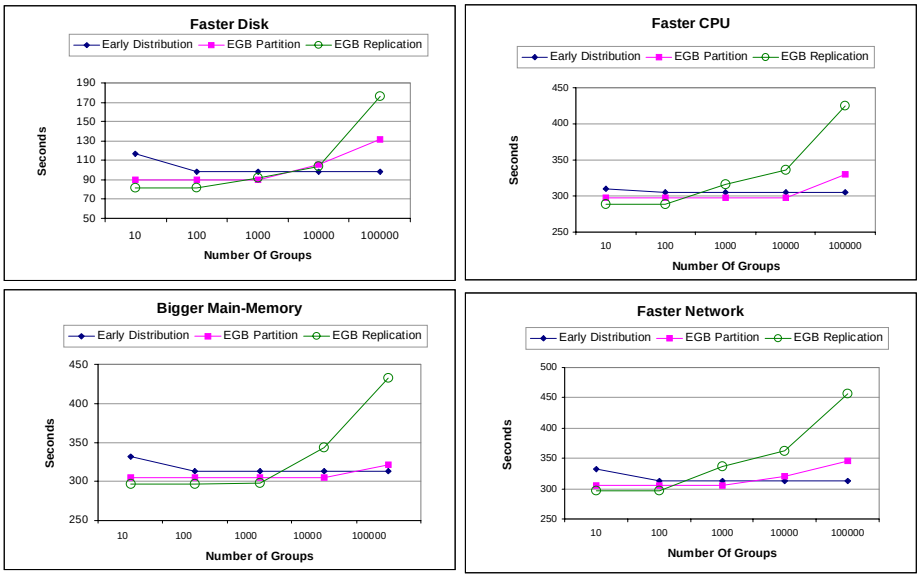

Figure 6. Varying By Selectivity Ratio.

4.2 Discussions

Comparing these three methods, there should be a combination solution for some certain situations. Early distribution shows the best performance if the number of groups produced growing to be large, especially above 1000 groups. However if the number of groups is smaller than 1000, then the choice should go for Early-Group-By Replication method. This is a logic consequence as more or huge number of groups produced, then the cost of, firstly, transferring the data to each processor and secondly the data replication process in second to third phases becomes crucial factors.

As the variation applied, when faster disk is applied, overall performance of the all methods improves significantly (almost 3 times faster). Figure 6 also shows the factors that can potentially improve performance when the number of groups produced increasing. This can be seen in particular there is a delay in the decrease of the performance when the CPU and bigger memory is applied. The delay is from number of groups 1000, as mentioned earlier, to 10000 when the performance starts to decrease. Difference case occurs when faster disk is applied, it improves up to 3 times of overall performance, which is quite substantial. There may be a good attempt to try, by combining of faster disk and either faster CPU or bigger memory to improve overall performance while also delay the decrease as the number of groups involved increasing.

5 Conclusions and Future Work

In this paper, we have investigated three parallel algorithms for processing GroupBy-before-join queries in high performance parallel database systems. These three algorithms are *Early Distribution* method, and *Early GroupBy with Partitioning Method and Early GroupBy with Replication* method. From our study it is concluded that the Early Distribution method is the preference one when the number of groups produced is growing to be large, but not in favor when the number of groups produced small. On the other hand, the Early-GroupBy with Replication is good when the number of groups produced is small, but it suffers serious performance problem once the number of groups produced by the query is large.

Our performance evaluation results show that variation in faster disk is the main potential manner to obtain most efficient performance in all situations taken by this investigation. As additional, consecutively increasing number of processor, speeding up the CPU and adding bigger memory are some other techniques suggested to be applied as the number of groups produced growing.

Our future work is being planned to investigate high dimensional Group By operations, which is often identified as *Cube* operations [7] and are highly pertinent to data warehousing applications. Since this type of applications normally involves large amount of data, parallelism is necessary in order to keep the performance level acceptable.

References

1. Bedell J.A. “Outstanding Challenges in OLAP”, *Proceedings of 14th International Conference on Data Engineering*, 1998.
2. Bultzingsloewen G., “Translating and optimizing SQL queries having aggregate”, *Proceedings of the 13th International Conference on Very Large Data Bases*, 1987.
3. Datta A. and Moon B., “A case for parallelism in data warehousing and OLAP”, *Proc. of 9th International Workshop on Database and Expert Systems Applications*, 1998.
4. Dayal U., "Of nests and trees: a unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers", *Proceedings of the 13th International Conference on Very Large Data Bases*, Brighton, UK, 1987.
5. DeWitt, D.J. and Gray, J., "Parallel Database Systems: The Future of High Performance Database Systems", *Communication of the ACM*, vol. 35, no. 6, pp. 85-98, 1992.
6. Kim, W., "On optimizing an SQL-like nested query", *ACM TODS*, 7(3), Sept. 1982.
7. Ramakrishnan, R., *Database Management Systems*, McGraw Hill, 1998.
8. Taniar, D., and Rahayu, J.W., “Parallel Processing of Aggregate Queries in a Cluster Architecture”, *Proc. of the 7th Australasian Conf. on Parallel and Real-Time Systems PART’2000*, Springer-Verlag, 2000.
9. Taniar, D., Jiang, Y., Liu, K.H., and Leung, C.H.C., "Aggregate-Join Query Processing in Parallel Database Systems", *Proc. of The 4th Intl Conf on High Performance Computing in Asia-Pacific HPC-Asia2000*, vol. 2, IEEE CS Press, pp. 824-829, 2000.
10. Yan W. P. and P. Larson, "Performing group-by before join", *Proceedings of the International Conference on Data Engineering*, 1994.