Aggregate-Join Query Processing in Parallel Database Systems

D. Taniar
Department of Computer Science
Royal Melbourne Institute of Technology
GPO Box 2476V, Melbourne 3001, Australia
taniar@cs.rmit.edu.au

Y. Jiang, K.H. Liu, C.H.C. Leung
School of Communications and Informatics
Victoria University
Melbourne, Australia
jiang@matilda.vu.edu.au

Abstract

Queries containing aggregate functions often combine multiple tables through join operations. We call these queries "Aggregate-Join" queries. In parallel processing of such queries, it must be decided which attribute to be used as a partitioning attribute, particularly join attribute or group-by attribute. Based on the partitioning attribute, we discuss three parallel aggregate-join query processing methods, namely Join Partition Method (JPM), Aggregate Partition Method (APM), and Hybrid Partition Method (HPM). The JPM and APM models use the join attribute, and the group-by attribute, respectively, as the partitioning attribute. The HPM model combines the other two methods using a logically hybrid architecture.

1 Introduction

Queries involving aggregates are very common in database processing, especially in *On-Line Analytical Processing* (OLAP), and *Data Warehouse* [2, 4]. These queries are often used as a tool for strategic decision making. 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. In this paper, we concentrate on this kind of queries in which the queries contain aggregate functions and join operations. We call this query "*Aggregate-Join*" query.

As the data repository containing data for integrated decision making is growing, aggregate queries are required to be executed efficiently. Large historical tables need to be joined and aggregated each other; 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 aggregate-join queries.

The motivation for efficient parallel query processing is not only influenced by the need to performance improvement, but also the fact that parallel architecture is now available in many forms, such as systems consisting of a small number but powerful processors (i.e. SMP machines), clusters of workstations (i.e. loosely coupled shared-nothing architectures), massively parallel processors (i.e. MPP), and clusters of SMP machines (i.e. hybrid architectures) [1]. We are particularly interested in formulating efficient parallel processing of aggregate-join queries, by exploiting a logically hybrid parallel database architecture, in which a set of processors is logically grouped into clusters.

We present three parallel processing methods for aggregate-join queries, *Join Partition Method (JPM)*, *Aggregate Partition Method (APM)*, and *Hybrid Partition Method (HPM)*. The *JPM* and *APM* methods mainly differ in the selection of partitioning attribute for distributing workloads over the processors. The *HPM* method is an adaptive method based on the *JPM* and *APM* methods.

The rest of this paper is organized as follows. Section 2 explains briefly aggregate queries. Section 3 describes the three methods for parallel processing of aggregate-join queries. Finally, section 4 gives the conclusions and explains future work.

2 Aggregate-Join Queries: A Brief Overview

It is common that an aggregate function query (Group-By query) involves multiple tables. These tables are joined to produce a single table, and this table becomes an input to the group-by operation. We call this kind of aggregate query as *Aggregate-Join* queries, that is queries involving join and aggregate functions. To illustrate aggregate-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)
```

0-7695-0589-2/00 $10.00 © 2000 IEEE

For simplicity of description and without loss of generality, we consider queries that involve only one aggregation function and a single join. The following two queries give an illustration of aggregate-join queries. Query 1 is to "retrieve project numbers, names, and total quantity of shipments for each project having the total shipments quantity of more than 1000".

QUERY 1:

```
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
```

Another example is to "cluster the part shipment by their city locations and select the cities with average quantity of shipment between 500 and 1000". The query written in SQL is as follows.

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
```

The main difference between Query 1 and Query 2 above lies in the join attributes and group-by attributes. In Query 1, the join attribute is also one of the group-by attributes. This is not the case with Query 2, where the join attribute is totally different from the group-by attribute. This difference is particularly a critical factor in processing aggregate-join queries, especially in parallel query processing, as there are decisions to be made regarding which attribute to be used for data partitioning.

When the join attribute and group-by attribute are the same as shown in Query 1 (e.g. attribute *J#* of both Project and Shipment), the selection of partitioning attribute becomes obvious. Therefore, instead of performing join first, the aggregate is carried out first followed by the join. Comparatively, join is a more expensive operation than aggregate functions, and it would be beneficial to reduce the join relation sizes by applying the aggregate function first. Generally, aggregate functions should always precede join whenever possible with an exception that the size reduction gained from the aggregate functions is marginal or the join selectivity factor is extremely small. In real life, early processing of the aggregate functions 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.

However, aggregate functions before join may not always be possible, such as Query 2 above. The semantic issues about aggregate functions and join and the conditions under which the aggregate functions would be performed before join can be found in literatures [3, 5, 7, 13]. In this paper, we concentrate on more general cases where aggregate functions cannot be executed before join. Therefore, we will use Query 2 as a running example throughout this paper.

3 Parallel *Aggregate-Join* Query Processing Methods

Our parallel database architecture consists of a host and a set of working processors. The host accepts queries from users and distributes each query with the required base relations to all processors for execution. The processors perform the query in parallel with possibly intermediate data transmission among each other via the network, and finally send the result of the query to the host. Our previous work has proved the suitability of this architecture in for parallel database processing [8, 9, 10]. Using this parallel architecture, parallel query processing is commonly carried out in three phases:

- *Data partitioning*, the operand relations of the query are partitioned and the fragments are distributed to each processor;
- *Parallel processing,* the query is executed in parallel by all processors and the intermediate results are produced;
- *Data consolidation*, the final result of the query is obtained by consolidating the intermediate results from the processors.

There is an important decision need to be made in processing *aggregate-join* queries, namely the selection of partitioning attribute. Selecting a proper partitioning attribute plays a crucial role in performance. Although in general any attributes of the operand relations may be chosen, two particular attributes (i.e. *join attribute* and *group-by attribute*) are usually considered.

If the *join attribute* is chosen, both relations are partitioned into *N* fragments by employing a partitioning function (e.g. a hash/range function), where *N* is the number of processors. The cost for parallel join operation can therefore be reduced as compared with a single processor system. However, after join and local aggregation at each processor, a global aggregation is required at the data consolidation phase, since local aggregation is performed on a subset of the group-by attribute.

If the *group-by attribute* is used for data partitioning, the relation with the group-by can be partitioned into *N* fragments, while the other relation needs to be broadcast to all processors for the join operation.

Comparing the two methods above, in the second method (partitioning based on the group-by attribute), the join cost is not reduced as much as in the first method (partitioning based on the join attribute). However, no global aggregation is required after local join and local aggregation, because records with identical values of the group-by attribute have been allocated to the same processor.

Based on these two partitioning attribute strategies, we introduce three parallel processing methods for *aggregate-join* queries, namely *Join Partitioning Method* (JPM), *Aggregate Partitioning Method* (APM), and *Hybrid Partitioning Method* (HPM). They are discussed in more details in the following sections.

3.1 Join Partition Method (*JPM*)

Given the two relations *R* and *S* to be joined, and the result is grouped-by according to the group-by attribute and possibly filtered through a having predicate, parallel processing of such query using the *JPM* method can be stated as follows.

Step 1: *Data Partitioning*

The relations *R* and *S* are partitioned into *N* fragments in terms of join attribute, i.e. the records with the same join attribute values in the two relations fall into a pair of fragments. Each pair of the fragments will be sent to one processor for execution.

Using QUERY 2 as an example, the partitioning attribute is attribute *P#* of both tables Parts and Shipment, which is the join attribute. Suppose we use 4 processors, and the partitioning method is a range partitioning, such as part numbers (*P#*) *p1-p99*, *p100-p199*, *p200-p299*, and *p300-399* are distributed to processors 1, 2, 3, and 4, respectively. This partitioning function is applied to both tables Parts and Shipment. Consequently, processor such as processor 1 will have Parts and Shipment records where the values of its *P#* attribute are between *p1-p99* , and so on.

Step 2: *Join operation*

Upon receipt of the fragments, the processors perform in parallel, the join operation on the allocated fragments. Join in each processor is done independently to each other [6]. This is possible because the two tables have been disjointly partitioned based on the join attribute.

Using the same example as above, join operation in a processor like processor 1 will produce a join result consisting of Parts-Shipment records having *P#* between *p1-p99*.

It is worth to mention that any sequential join algorithm (i.e. nested-loop join, sort-merge join, nested index join, hash join) may be used in performing a local join operation in each processor [11].

Step 3: *Local Aggregation*

After the join is completed, each processor then performs a local aggregation operation. Join results in each processor is grouped-by according to the group-by attribute.

Continuing the same example as the above, each city found in the join result will be grouped. If, for example, there are three cities: Beijing, Melbourne, and Sydney, found in processor 1, the records will be grouped according to these three cities. The same aggregate operation is applied to other processors. As a result, although each processor has distinct part numbers, some of the cities, if not all, among processors may be identical (duplicated). For example, processor 2 may have three cities, such as London, Melbourne, and Sydney, where Melbourne and Sydney are also found in processor 1 as mentioned earlier, but not London.

Step 4: *Re-distribution*

A global aggregation operation is to be carried out by re-distributing the local aggregation results across all processors such that the result records with identical values of the group-by attribute are allocated to the same processors.

To illustrate this step, a range partitioning method is again used to partition the group-by attribute, such as processors 1, 2, 3, and 4 are allocated cities beginning with letter A-G, H-M, N-T, and U-Z, respectively. Using this range partitioning, processor 1 will distribute its Melbourne record to processor 2, Sydney record to processor 3, and leave Beijing record in processor 1. Processor 2 will do the same to its Melbourne and Sydney records, whereas London record will remain in processor 2.

Step 5: *Global Aggregation*

Each processor performs an *N*-way merging of the local aggregation results, followed by performing a restriction operation for the *Having* clause if required by the query.

The result of this global aggregate in each processor is a subset of the final results, meaning that each record in each processor has a different city, and furthermore, the cities in each processor will not appear in any other processors. For example, processor 1 will produce one Beijing record in the query result, and this Beijing record does not appear in any other processors. Additionally, some of the cities may then be eliminated through the *Having* clause.

Step 6: *Consolidation*

The host simply consolidates the partial results from the processors by a union operation, and produces the query result.

Figure 1 gives a graphical illustration of the *Join Partition Method* (*JPM*). The circles represent processing elements, whereas the arrows denoted data flow through data partitioning or data re-distribution.

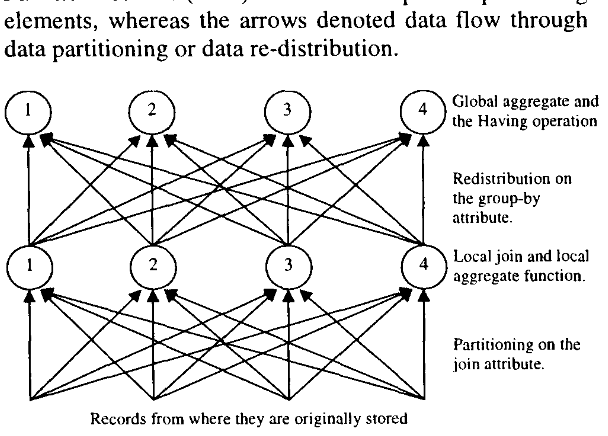
Global aggregate and the Having operation
Redistribution on the group-by attribute.
Local join and local aggregate function.
Partitioning on the join attribute.
Records from where they are originally stored

Figure 1. *Join Partition Method (JPM)*

3.2 Aggregate Partition Method (*APM*)

The *APM* method relies on partitioning based on the group-by attribute. As the group-by attribute belongs to just one of the two tables, only the table having the group-by attribute will be partitioned. The other table has to be broadcast to all processors. This technique is often known as "*Divide and Broadcast*" technique, commonly used in naive parallel join operations [8]. The processing steps of the *APM* method are explained as follows.

Step 1: *Data Partitioning*

The table with the group-by attribute, say *R*, is partitioned into *N* fragments in terms of the group-by attribute, i.e. the records with identical attribute values will be allocated to the same processor. The other table *S* needs to be broadcast to all processors in order to perform the join operation.

Using QUERY 2 as an example, table Parts is partitioned according to the group-by attribute, namely *City*. Assume a range partitioning method is used, processors 1, 2, 3, and 3 will have Parts records having cities beginning with letter A-G, H-M, N-T, and U-Z, respectively. On the other hand, table Shipment is replicated to all four processors.

Step 2: *Join operations*

After data distribution, each processor carries out the joining of one fragment of *R* with the entire table *S*.

Using the same example, each processor joins its Parts fragment with the entire table Shipment. The results of this join operation in each processor are pairs of Parts-Shipment records having the same *P#* (join attribute) and the value of its *City* attribute must fall into the category identified by the group-by partitioning method (e.g. processor 1=A-G, processor 2=H-M, etc).

Step 3: *Aggregate operations*

The aggregate operation is performed by grouping the join results based on the group-by attribute, followed by a *Having* restriction if it exists on the query.

Continuing the above example, processor 1 will group the records based on the city and the cities are in the range of A to G. The other processors will of course have a different range. Therefore, each group in each processor is distinct to each other both within and among processors.

Step 4: *Consolidation*

Since the table *R* is partitioned on group-by attribute, the final aggregation result can be simply obtained by a union of the local aggregation results from the processors.

Figure 2 shows a graphical illustration of the *APM* method. Notice the difference between the *JPM* and the *APM* method. The former imposes a "two-phase" partitioning scheme, whereas the latter is a "one-phase" partitioning scheme.

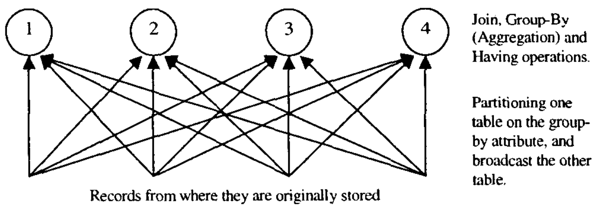
Join, Group-By (Aggregation) and Having operations.
Partitioning one table on the group-by attribute, and broadcast the other table.
Records from where they are originally stored

Figure 2. *Aggregation Partition Method (APM)*

3.3 Hybrid Partition Method (*HPM*)

The *HPM* method is a combination of the *JPM* and *APM* methods. In the *HPM*, the total number of processors *N* are divided into *m* clusters, each of which has *N/m* processors as shown in Figure 3. Based on the proposed logical architecture, the data partitioning phase is carried out in two steps. First, the table with group-by attributes is partitioned into processor clusters in the same way of the *APM* (i.e. partitioning on the group-by attribute and the other table is broadcast to the cluster). Second, within each cluster, the fragments of the first table and the entire broadcast table is further partitioned by the join attributes as in the *JPM*. Depending on parameters such as the cardinality of the tables and the skew factors, a proper value of *m* can be chosen to minimize the query execution time.

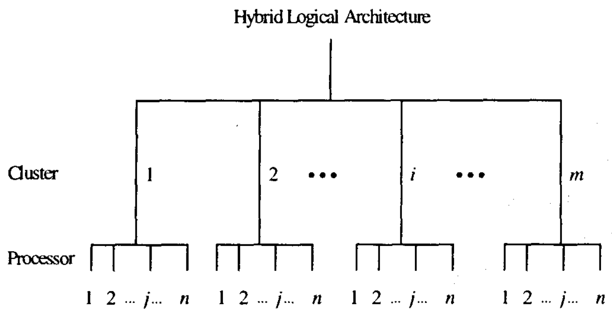

Figure 3. Logical Architecture for *HPM*

The detailed *HPM* method is explained as follows:

Step 1:

a) *Partitioning into Clusters*

Partition the table R on group-by attribute to m clusters, denoted by r_i where $1 \leq i \leq m$. Table S is broadcast to all clusters.

Using QUERY 2 as an example, first partition table Parts into m clusters based on attribute *City*. If there are three clusters, table Parts is divided into three fragments. Table Shipment is, on the other hand, replicated to all the three clusters. As a result, each cluster will have a fragment of table Parts and a full table Shipment.

b) *Partitioning within Clusters*

Within each cluster i, further partition fragment r_i and the full table S on the join attribute to n processors where $n=N/m$. Each fragment is now denoted by r_{ij} and s_j.

Suppose that there are twelve processors in total. Since we use three clusters, each cluster will contain four processors. In each cluster, a Parts fragment and the whole Shipment table are partitioned based on the join attribute *P#* into the four processors.

In practice, steps 1(a) and (b) described above are carried out at once, in order to reduce communication/distribution time. When doing the two partitioning steps as a single partitioning step, each record of the tables is applied a partitioning function that determines into which processor and cluster the record should be sent. The partitioning function takes into account whether or not the table is the group-by table. Figure 4 gives the algorithm of the partitioning step in the *HPM*. The algorithm clearly shows that the table containing the group-by attribute is purely partitioned into all processors, whereas the other table is to some degrees replicated.

```
Let processors be $P$
Let group-by partitioning function be $G$
Let join partitioning function be $J$

For each record $r$ of the group-by table
    Determine cluster $i$ for record $r$ based on
        partitioning function $G$
    Determine processor $j$ for record $r$ based on
        partitioning function $J$
    Distribute record $r$ to processor $P_{ij}$
End

For each record $s$ of the other table
    Determine processor $j$ for record $r$ based on
        partitioning function $J$
    For each processor $k$ in each cluster
        Distribute record $r$ to processor $P_{kj}$
    End
End
```

Figure 4. *Partitioning Algorithm in HPM*

Step 2: *Join operation*

Carry out the join operation in each processor. Join operation is executed like in the other two methods, that is, it is carried out locally and independently in each processor without involving other processors.

Step 3: *Local Aggregation*

Perform local aggregation at each processor. This local aggregation operation is carried out as like in the *JPM* method. As a result, each processor will produce a number of groups and some groups among other processors may be the same, and these groups will be later grouped in the global aggregation stage.

Step 4: *Re-distribution*

Redistribute the local aggregation results to the processors within each cluster by partitioning the results on the group-by attribute. This step is similar to that of in the *JPM* method. The main difference is that in *HPM* the re-distribution is done within each cluster, not globally involving all processors like in *JPM*.

Step 5: *Global Aggregation*

Merge the local aggregation results within each cluster. Then perform the *Having* predicate, if exists, in each cluster. Unlike the *JPM*, again, this process is done locally in each cluster. As a result of this process, each cluster contains a subset of the final query result.

Step 6: *Consolidation*

Transfer the results from the clusters to the host.

Figure 5 shows a graphical illustration of the *HPM* method. By the look at it, the flow of process is similar to that of *JPM*, in which both are a "two-phase" processing scheme. We, however, need to highlight two main differences. *One* is that the initial partitioning in *HPM* is different, and in fact, it is a combination between partitioning in *JPM* and in *APM*, where both group-by and join attributes are being utilized in the partitioning phase. This unified two-step partitioning scheme is not clearly shown in the diagram, but as shown in the algorithm the partitioning uses both group-by and join attributes. The *second* difference is related to the re-distribution phase where re-distribution in *HPM* is a cluster-based, not global in the sense like in *JPM*.

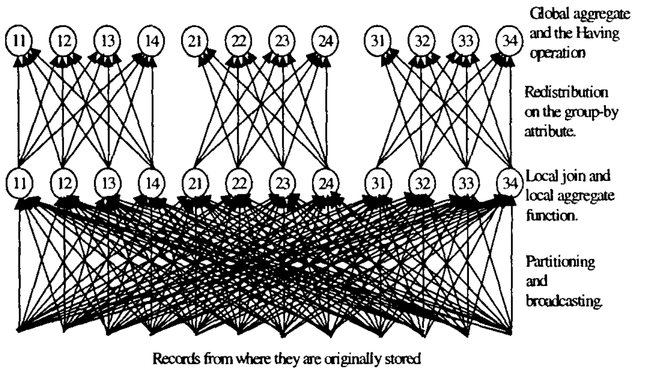
11 12 13 14 21 22 23 24 31 32 33 34
Global aggregate and the Having operation
Redistribution on the group-by attribute.
11 12 13 14 21 22 23 24 31 32 33 34
Local join and local aggregate function.
Partitioning and broadcasting.
Records from where they are originally stored

Figure 5. *Hybrid Partition Method (HPM)*

4 Conclusions and Future Work

Traditionally, join operation is processed before aggregation operation and tables are partitioned on join attribute. In this paper, we study that group-by attribute may also be chosen as the partition attribute and present three parallel methods for aggregation queries, *JPM*, *APM*, and *HPM*. These methods differ in the way of distributing query tables, i.e. partitioning on the join attribute, on the group-by attribute, or on a combination of both.

Our future work is planned to investigate the behaviour of each of the methods. This will include analytical analysis and performance measurements. We also plan to investigate further the *HPM* method in a real hybrid architecture whereby the number of clusters and the number of processors within each cluster are predetermined. We will take into consideration the fact that it is common that processors within each cluster normally share the memory (i.e. shared-memory SMP machines), but different clusters communicate through network (i.e. shared-nothing among clusters) [12]. It will be interesting to see the impact of two levels of partitioning methods like in the *HPM* model in real hybrid architectures.

References

[1] Almasi G., and Gottlieb, A., *Highly Parallel Computing*, Second edition, The Benjamin/Cummings Publishing Company Inc., 1994.

[2] Bedell J.A. "Outstanding Challenges in OLAP", *Proceedings of 14th International Conference on Data Engineering*, 1998.

[3] Bultzingsloewen G., "Translating and optimizing SQL queries having aggregate", *Proceedings of the 13th International Conference on Very Large Data Bases*, 1987.

[4] Datta A. and Moon B., "A case for parallelism in data warehousing and OLAP", *Proceedings. of Ninth International Workshop on Database and Expert Systems Applications*, 1998.

[5] 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.

[6] DeWitt, D.J. and Gray, J., "Parallel Database Systems: The Future of High Performance Database Systems", *Communication of the ACM*, Volume 35, Number 6, pp. 85-98, 1992.

[7] Kim, W., "On optimizing an SQL-like nested query", *ACM Transactions on Database Systems*, Volume 7, Number 3, September 1982.

[8] Leung, C. H. C. and Ghogomu, H. T., "A High-Performance Parallel Database Architecture", *Proceedings of the Seventh ACM International Conference on Supercomputing*, Tokyo, Pages 377-386, 1993.

[9] Leung, C. H. C. and Taniar. D., "Parallel Query Processing in Object-Oriented Database Systems", *Australian Computer Science Communications*, Volume 17, Number 2, Pages 119-131, 1995.

[10] Liu K. H., Jiang, Y. and Leung, C. H. C., "Query execution in the presence of data skew in parallel databases", *Australian Computer Science Communications*, Volume 18, Number 2, Pages 157-166, 1996.

[11] Mishra, P. and Eich, M. H., "Join Processing in Relational Databases", *ACM Computing Surveys*, Volume 24, Number 1, Pages 63-113, March 1992.

[12] Valduriez, P., "Parallel Database Systems: The Case for Shared-Something", *Proceedings of the International Conference on Data Engineering*, Pages 460-465, 1993.

[13] Yan W. P. and Larson, P. "Performing group-by before join", *Proceedings of the International Conference on Data Engineering*, 1994.