Parallel Selection Query Processing Involving Index in Parallel Database Systems

J. Wenny Rahayu
Department of Computer Science & Comp. Eng.
La Trobe University
Bundoora, Victoria 3083, AUSTRALIA
Email: wenny@cs.latrobe.edu.au

David Taniar
School of Business Systems,
Monash University,Clayton Campus
Victoria 3800, AUSTRALIA
Email: David.Taniar@infotech.monash.edu.au

Abstract

Index is an important element in databases, and the existence of index is unavoidable. When an index has been built on a particular attribute, database operations (e.g. selection, join) on this attribute will become more efficient by utilizing the index. In this paper we focus on parallel algorithms for selection queries involving index – that is data searching on indexed attributes. In this paper, we propose two categories of parallel selection queries using index: parallel exact match and range selections; depending on the type of selection conditions. As parallel algorithms for these selection queries are very much influenced by indexing schemes, we will also describe various index partitioning methods for parallel databases, and discusses their efficiency in supporting parallel selection query processing.

1. Introduction

Most of the work on parallel database processing has been focused on parallel join processing and optimization [5, 10, 11]. This is very much motivated due to the fact that join processing is considered as one of the most expensive operations in database processing [6], and parallelism of join is critically needed.

In contrast, parallelism of selection operation is often neglected and overlooked, despite the fact that selection operation is one of the most common operations not only in relational databases, but also in other type of databases, such as image databases, text databases, etc. Data retrieval in these databases is basically a selection process. In other term, selection operation is also known as searching operation [4]. It is the aim of this paper to focus on parallel selection processing for high performance database systems, especially selection operations on indexed attributes.

Index, together with table, is an important element in database systems, particularly because index provides an efficient data structure for database processing (e.g. search, join operations). Therefore, when an index exists on a particular attribute, database processing will be more efficient by utilizing this index. Queries not taking indexes into account are certainly more general, however, in many situations, it is quite common to expect that the searched attributes are indexed. This is especially true in the case where a selection operation is based on a primary key (PK) or secondary key (SK) on a table. Primary keys are normally indexed to prevent duplicate values exist in that table, whereas secondary keys are indexed to speed up the searching process [7].

The work presented in this paper is actually part of a larger research project on *Parallel Indexing in Parallel Database Systems*. This project consists of three stages: (*i*) taxonomy of parallel indexing schemes, (*ii*) parallel join query algorithms using index, and (*iii*) parallel selection query algorithms using index. The research results from the first two stages have been reported in the Distributed and Parallel Databases – an International Journal [8], and Parallel and Distributed Computing Applications and Technology PDCAT'2000 conference [9], respectively. In this paper, we focus on the third and final stage of the project.

2. Background

2.1 Parallel Indexing Schemes

Our previous work [8] has presented various indexing schemes for parallel database architectures. Three parallel indexing schemes were considered, namely:

(i) Non-Replicated Index (**NRI**),
(ii) Partially-Replicated Index (**PRI**), and
(iii) Fully-Replicated Index (**FRI**).

There were three variations for NRI and PRI, depending on two factors, namely *index partitioning attributes* and *table partitioning attributes*. The first variation is where the index partitioning attribute is the same as the table partitioning attribute. The second variation is where no index partitioning attribute is used. And the third variation is where the index partitioning attribute is different from that of the table. For the FRI scheme, only two variations are available, that is the first and the third variations of the above.

2.1.1 Non-Replicated Indexing (NRI) Schemes

A *Non-Replicated Indexing* (**NRI**) scheme, as the name suggests, is where the global index is partitioned into several disjoint and smaller indices. Each of these small indices is placed in a separate processing element.

The first model of NRI, abbreviated as **NRI-1**, is where the index partitioning attribute is the same as the table partitioning attribute. In our example, Assume the table has been partitioned based on the ID field, based on the following range partitioning: processor 1 = IDs 1 to 30, processor 2 = IDs 31 and 60, and processor 3 = IDs 60 and 100.

Using NRI-1, after the table is partitioned according to this range partitioning strategy, each processing element then builds its local index on the ID field. Figure 1 shows the composition of each processing element with its local index. We use a B+ tree structure for the index [1, 2]. The index tree has a maximum number of node pointers from any non-leaf node of 4, and a maximum number of data pointers from any leaf node of 3.

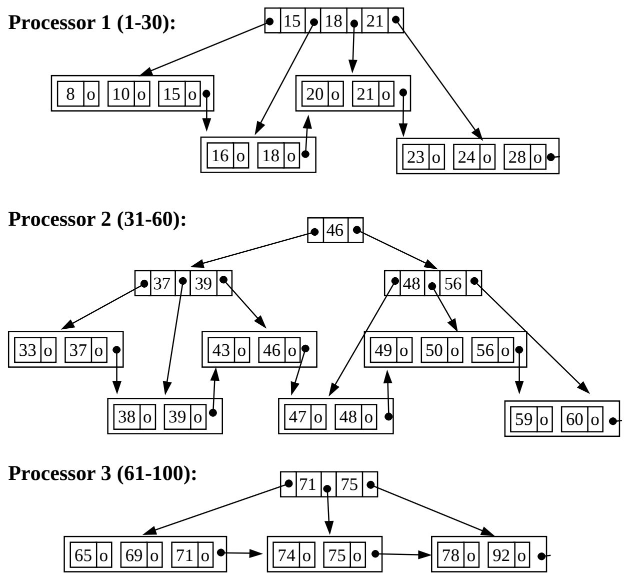

Figure 1. NRI-1 scheme

The second model of NRI, abbreviated as **NRI-2**, is where local indices are built on whatever data already in each processing element. The table partitioning attribute can be unknown, or a different attribute is used in table partitioning, or even a non-range partitioning applied to the indexed attribute. For example, the table is partitioned based on attribute Name, and a certain partitioning function is used on this attribute. Once the table is partitioned using these rules, local indices based on the ID field are then built. NRI-2 scheme assumes that each processor is like an independent single processor, and an index is built on the local data without considering the global picture of a multi-processor environment.

The last model for NRI (abbreviated as **NRI-3**) is where there is an attribute used in the index partitioning, but it is different from that of the table partitioning. For example, the index is partitioned based on the ID field, whereas the table is partitioned according to the Name field. Because of the difference in attribute partitioning, it becomes impossible to locate all of the records at the same place as its indices. In case where the index entry is located at a different processor from where the record is, there will be necessary to have a data pointer from the leaf node in one processor to the actual record in the other processor.

2.1.2 Partially-Replicated Indexing (PRI) Schemes

A *Partially-Replicated Indexing* (**PRI**) scheme has two major differences from the NRI scheme. First is suggested by the name itself, where PRI has some degree of replication while NRI has not. The second difference is related to the composition of the index itself. Unlike in NRI where the global index is physically partitioned, in PRI, the global index is maintained. In other words, each processing element has a different part of the global index, and the overall structure of global index is still preserved. The ownership rule of each index node is that the processor owning a leaf node also owns all nodes from the root to that leaf. Consequently, the root node is replicated to all processors, and non-leaf nodes may be replicated to some processors. Additionally, if a leaf node has several keys belonging to different processors, this leaf node is also replicated to the processors owning the keys.

An example of **PRI-1** is exhibited in Figure 2. In this example, PRI-1 uses the ID field as the index partitioning attribute, which is the same as for the table partitioning. Notice that some non-leaf nodes are replicated whereas others are not. For example, the non-leaf node 15 is not replicated and located only in processor 1, whereas non-leaf node 18 is replicated to processors 1 and 2. It is also clear that the root node is fully replicated.

PRI-2 scheme has a similar concept with NRI-2. As an example, the table is already partitioned according to some partitioning rule on non-ID attribute. The global index is subsequently partitioned based on the location of the partial table. As a result, more replication can be expected even at the leaf node level, because a leaf node consists of *k* keys and each key may be located at a different processor. For example, Suppose records (8, Agnes) and (10, Mary) are located at processor 1 and record (15, Peter) is at processor 2, then the first leaf node of (8, 10, and 15) is replicated to both processors 1 and 2.

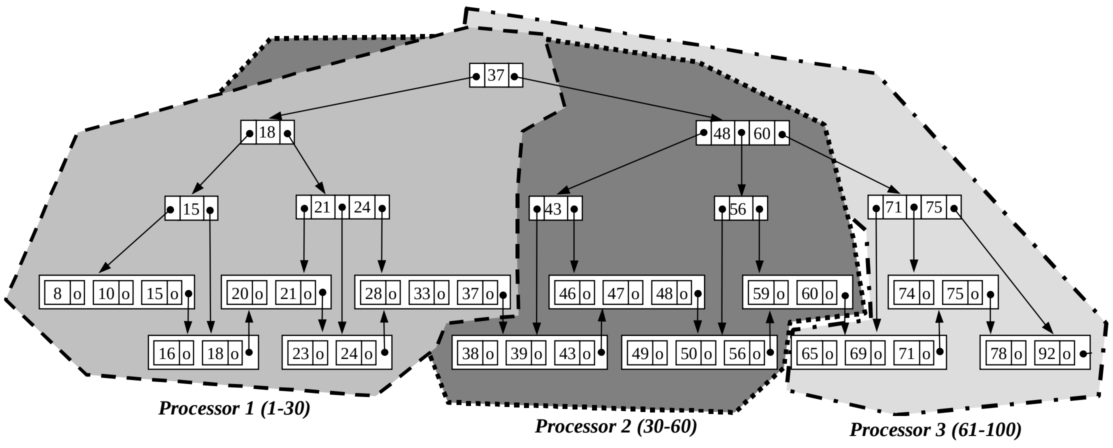

Figure 2. PRI-1 scheme

PRI-3 scheme is analogous with NRI-3. As example the table is partitioned based on the Name field, whereas the index is based on the ID field. Therefore, the index tree will look like PRI-1, but the data pointers will look like those of PRI-2 in which they cross the boundary of processors.

2.1.3 Fully-Replicated Indexing (FRI) Schemes

A *Fully-Replicated Indexing* (**FRI**) scheme is where the global index is fully replicated to all available working processors. Due to its simplicity of this parallel indexing scheme, there are only two different variations, particularly the table partitioning attribute is the same as the indexed attribute, and the table partitioning attribute is different from the indexed attribute. In the context of NRI and PRI, only variations 1 and 3 are available to the FRI scheme. To make the naming convention uniform across the three parallel indexing schemes, the two variations for the FRI scheme are numbered as 1 and 3, leaving variation 2 as not applicable.

FRI-1 has a similar concept with the other two variations 1 (i.e. NRI-1 and PRI-1). For example, table is partitioned on the ID field, and the index is built on the same field. Since the global index is fully replicated, leaf nodes that do not have the base data located at the same place must have their data pointers crossing the processor boundaries. As a result, all records will have *n* incoming data pointers from the leaf nodes where *n* is the number of processors.

FRI-3 is quite similar to PRI-1, except that the table partitioning for FRI-3 is not the same as the indexed attribute. Since the index is fully replicated, and each of the record will also have *n* incoming data pointers where *n* is the number of replication of the index.

2.2 Selection Query

Selection is one of the most common *Relational Algebra* operations [3]. It is a unary operation in which the operator takes one operand only, that is a table. Selection is an operation that selects specified records based on given criteria. The result of the selection is a horizontal subset (records) of the operand. Depending on the selection predicates (conditions), we categorize selection queries into

(i) *Exact Selection Match*, and

(ii) *Range Selection* (*Continuous* and *Discrete*).

An *Exact Match Selection Query* is a query where the selection predicate is to check for an exact matching between a selection attribute and a given value. The resulting table of an exact match query can contain more than one record, depending on whether duplicate values of selection attribute exist or not.

A *Range Selection Query* is a query where the selection attribute value in the query result may contain more than single unique values.

In the *Continuous Range Selection Query*, the selection predicates contain a continuous range check, normally with continuous range checking operators, such as <, ≤, >, ≥, !=, `Between`, `Not`, and `Like` operators. On the other hand, the *Discrete Range Selection Query* uses discrete range check operators, such as `In` and `Or` operators.

The main difference between the two range queries – the continuous range selection query checks for a particular range and the values between this range is continuous, whereas the discrete range selection query checks for multiple discrete values which may or may not be in a particular range. Both these queries are called range queries simply because the selection operation checks for multiple values, as opposed to a single value like in the exact match queries.

3 Parallel Selection Query Algorithm

3.1 Parallel Exact Match Selection Query

There are three important factors in parallel exact match selection query processing, especially *processor involvement*, *index tree traversal*, and *record loading*.

3.1.1 Processor Involvement

For an exact match queries, ideally parallel processing may isolate into the processor(s) where the candidate records are located. Considering that the number of processors involved in the query is an important factor, there are particularly two cases in parallel processing of exact match selection queries.

- Case 1 *(selected processors are used):*

This case is applicable to all indexing schemes, except for the NRI-2 scheme. If the indexing scheme of the indexed attribute is NRI-1, PRI-1, or FRI-1, we can direct the query into the specific processors, since the data partitioning scheme used by the index is known. It is also the same case with NRI-3, PRI-3, and FRI-3. The only difference between NRI/PRI/FRI-1 and NRI/PRI/FRI-3 is that the records may not be located at the same place as where the leaf nodes of the index tree are located. However, from the index tree searching point of view, they are the same, and hence it is possible to activate selected processors that will subsequently perform an index tree traversal.

For PRI-2 scheme, since a global index is maintained, it becomes possible to traverse to any leaf node from basically anywhere. Therefore, only selected processors are used during the traversing of the index tree.

The processor(s) containing the candidate records can be easily identified with NRI-1/3, PRI-1/3, or FRI-1/3 indexing schemes. With PRI-2 indexing scheme, it will ultimately go to the desired processor.

- Case 2 *(all processors are used):*

This case is applicable to the NRI-2 indexing scheme only, because using NRI-2 scheme, there is no way to identify where the candidate records are located without searching in all processors. NRI-2 basically builds a local index based on whatever data it has from the local processor without having a global knowledge.

3.1.2 Index Tree Traversal

Searching for a match is done through index tree traversal. The traversal starts from the root node and finishes either at a matched leaf node or no match is found. Depending on the indexing scheme used, there are two cases:

- Case 1 *(traversal is isolated to local processor):*

This case is applicable to all indexing scheme, but PRI-2. When any of the NRI indexing schemes is used, index tree traversal from the root node to the leaf node will stay at the same processor.

When PRI-1 or PRI-3 is used, even though the root node is replicated to all processors and theoretically traversal can start from any node, the host processor will direct the processor(s) containing the candidate results to initiate the searching. In other words, index tree traversal will start from the processors that hold candidate leaf nodes. Consequently, index tree traversal will stay at the same processor.

For any of the FRI indexing schemes, since the index tree is fully replicated, it becomes obvious that there is no need to move from one processor to another during the traversal of an index tree.

- Case 2 *(traversal from one processor to another):*

This case is applicable to PRI-2 only, where searching starting from a root node at any processor may end up on a leaf node at a different processor. For example, when a parent node at processor 1 points to a child node at processor 2, the searching control at processor 1 is passed to processor 2.

3.1.3 Record Loading

Once a leaf node containing the desired data is found, the record pointed by the leaf node is loaded from disk. Again here there are two cases:

- Case 1 *(local record loading):*

This case is applicable to NRI/PRI/FRI-1 and NRI/PRI-2 indexing schemes, since the leaf nodes and the associated records in these indexing schemes are located at the same processors. Therefore, record loading will be done locally.

- Case 2 *(remote record loading):*

This case is applicable to NRI/PRI/FRI-3 indexing schemes where the leaf nodes are not necessarily placed at the same processor where the records reside. Record loading in this case is performed by trailing the pointer from the leaf node to the record and by loading the pointed record. When the pointer crosses from one processor to another, the control is also passed from the processor that holds the leaf node to the processor that stores the pointed record. This is done similarly to the index traversal, which also crosses from one processor to another.

3.2 Parallel Range Selection Query

For *continuous range* queries, possibly more processors need to involve. However, the main importance is that it needs to determine the lower and/or the upper bound of the range. For open-ended continuous range predicates, only the lower bound needs to be identified, whereas for the opposite, only the upper bound of the range needs to take into account. In many cases, both lower and upper bound of the range need to be determined. Searching for the lower and/or upper bound of the range can be directed to selected processors only due to the same reasons as those for the exact match queries.

With the selected attribute being indexed, once these boundaries are identified, it becomes easy to trace all values within a given range, by traversing leaf nodes of the index tree. If the upper bound is identified, leaf node traversal is done to the left, whereas if the lower bound is identified, all leaf nodes to the right are traversed. We must also note that record loadings within each processor are performed sequentially. Parallel loading is only possible among processors, not within a processor.

For *discrete range* queries, each discrete value in the selection predicate is converted into multiple exact match predicates. Further processing follows the processing method for exact match queries. Since discrete range queries can be transformed into multiple exact match queries, from this point onward, we consider only the exact match and continuous range selection queries.

3.3 Parallel Algorithms for Selection Query Processing

The algorithm for parallel selection query processing consists of four modules: (*i*) *initialisation*, where variables are initialized and discrete range query is transformed, (*ii*) *processor allocation*, based on the cases explained above, (*iii*) *parallel searching*, and (*iv*) *record loading*. The algorithm is presented as follows.

Algorithm: Parallel-Selection (Query *Q* and Index *I*)

```
Initialization - in the host processor:
1   Let $P$ be all available processors
2   Let $P_Q$ be processors to be used by query $Q$
3   Let $V_{exact}$ be the search value in $Q_{exact\_match}$
4   Let $V_{lower}$ and $V_{upper}$ be the range lower and upper
    values
5   If $Q$ is discrete range Then
6       Convert $Q_{discrete}$ into $Q_{exact\_match}$
7       Establish an array of $V_{exact}$ []
Processor Allocation - in the host processor:
8   If index $I$ is NRI-2 Then
9       $P_Q = P$                  -- use all processors
10  Else
11      Select $P_Q$ from $P$ based on Q -- use selected proc
Parallel Search – using processor $P_Q$:
12  For each searched value $V$ in query $Q$
13      Search value $V$ in index tree $I$
14      If a match is found in index tree $I$ Then
15          Put the index entry into an array of index entry
              result
16          If $Q$ is continuous range Then
17              Trace to neighbouring leaf nodes
18      Put the index entry into the array of entry result
Record Loading – using processor $P_Q$:
19  For all entries in the array of entry result
20      Trace the data pointer to actual record $r$
21      If record $r$ is located at a different processor Then
22          Load remote record $r$ through a message passing
23      Else
24          Load the pointed local record $r$
25      Put record $r$ into query result
```

4 Comparative Analysis

As there are different kinds of parallel indexing schemes and consequently various parallel algorithms for selection queries involving index, it becomes important to analyze the efficiency of each parallel indexing scheme in the context of parallel selection query processing. In this section, we compare the complexity involved in parallel selection query processing, which is imposed by each parallel indexing scheme.

Based on the three key factors in parallel selection query processing, we draw a matrix to show a comparison among parallel indexing schemes. This is shown in Figure 3. The shaded cells show more expensive operations in comparison with others within the same operation, whereas the non-shaded cells indicate cheaper operations.

	NRI Schemes			**PRI Schemes**			**FRI Schemes**	
	NRI-1	**NRI-2**	**NRI-3**	**PRI-1**	**PRI-2**	**PRI-3**	**FRI-1**	**FRI-3**
Processor Involvement	Selected processors	All processors	Selected processors	Selected processors	Selected processors	Selected processors	Selected processors	Selected processors
Index Traversal	Local search	Local search	Local search	Local search	Remote search	Local search	Local search	Local search
Record Loading	Local record load	Local record load	Remote record load	Local record load	Local record load	Remote record load	Local record load	Remote record load

Figure 3. A Comparative Table for Parallel One-Index Selection Query Processing

Based on this table comparison, each parallel indexing scheme has advantages and disadvantages in supporting parallel index selection query processing. It is clear from the table that NRI/PRI/FRI-1 indexing schemes provide more advantages than others, since only selected processors are used, and index traversal and record loading are locally done.

Less attractive indexing schemes offered by NRI/PRI/FRI-3 where record loading may be done remotely. The efficiency of remote data loading is very much determined by the selectivity factor of the query. The higher the selectivity, the more records to be loaded, and there is a great chance the records to be loaded remotely and this incurs overhead. In contrast, if the selectivity ratio is very small, overheads for record loading can be minimal. Consequently, NRI/PRI/FRI-3 can be as good as NRI/PRI/FRI-1 indexing schemes particularly for parallel one-index selection query processing.

The other option is NRI/PRI-2 indexing scheme. NRI-2 requires all processors to be used. If the selectivity ratio is very small, most processors will not produce any results. From a elapsed time point of view (speed up), it may not be a problem, but from a throughput point of view (scale up), these processors that do not bring any results may waste a lot of unnecessary processing time. PRI-2 on the other hand may isolate into selected processors, but traversal may need to move from one processor to another – increasing communication overhead.

In wrapping up the comparison, we can clearly see that NRI/PRI/FRI-1 offer much benefit for parallel selection query processing involving index. These indexing schemes clearly offer the best performance. The main difference between these three indexing schemes is the structure of the index, where NRI-1 is purely local index, PRI-1 maintains a global index which is spread among processors, FRI-1 replicates the whole index. Since extra benefits of PRI-1 and FRI-1 are not clearly seen, maintaining global index as in PRI-1 and replicating the whole index as in FRI-1 do not offer extra benefits. In fact, the drawback of PRI-1 and FRI-1 is quite clear, where PRI-1 needs to maintain the link from one index node of one processor to another node in another processor, and FRI-1 needs enormous extra space to maintain the index. On the other hand, NRI-1 is sufficient enough to provide support for parallel selection query processing.

5 Conclusions and Future Work

In this paper, we have proposed a parallel selection query processing algorithm involving index. Discussing parallel selection query algorithms cannot be separated from that of the index structure used in a parallel environment. We have described three parallel indexing schemes, namely non-replicated indexing (NRI), partially-replicated indexing (PRI), and fully-replicated indexing (FRI) schemes. A number of variations to these parallel indexing schemes were also discussed.

The comparison among these parallel indexing schemes particularly in supporting the efficiency of parallel selection query algorithms shows clearly that NRI/PRI/FRI-1 is the most supportive. Others offer various advantages and disadvantages. Looking into this matter more details, it appears that PRI/FRI schemes do not add extra benefits. The decision to choose another index for other attributes is determined by the balance between storage versus performance.

Our future work includes parallel selection query processing involving multiple indexes on different selection attributes.

References

[1] Bayer, R. and McCreight, E.M., “Organization and Maintenance of Large Ordered Indices”, *Acta Informatica*, volume 1, number 3, pp-173-189, 1972.

[2] Comer, D., “The Ubiquitous B-Trees”, *ACM Computing Surveys*, volume 11, number 2, pp 121-137, 1979.

[3] Elmasri, R. and Navathe, S.B., *Fundamental of Database Systems*, Third Edition, Addison-Wesley, 2000.

[4] Knuth D. E., “The art of computer programming”, Volume 3, *Addison-Wesley Publishing Company, INC.*, 1973

[5] Lakshmi, M.S. and Yu, P.S., "Effectiveness of Parallel Joins", *IEEE Transactions of Knowledge and Data Engineering*, volume 2, number 4, pp. 410-424, December 1990.

[6] Mishra, P. and Eich, M.H., "Join Processing in Relational Databases", *ACM Computing Surveys*, volume 24, number 1, pp. 63-113, March 1992.

[7] Ramakrishnan, R. *Database Management Systems*, McGraw Hill, 1998.

[8] Taniar, D. and Rahayu, J.W., "A Taxonomy of Indexing Schemes for Parallel Database Systems", *Distributed and Parallel Databases: An International Journal*, 2001 (in press).

[9] Taniar, D. and Rahayu, J.W., "Chapter 17: Parallel Join Query Algorithms Involving Index", *Parallel and Distributed Computing Applications and Technologies*, C.S.Leung, J.Sum, C.L.Wang, and G.H.Young (eds.), ISBN: 962-85887-1-0, The University of Hong Kong, pp. 133-140, 2000.

[10] Wolf J. L., Dias D. M., and P. S. Yu, "A parallel sort-merge join algorithm for managing data skew", *IEEE Transactions On Parallel And Distributed Systems*, volume 4, number1, January 1993.

[11] Wolf J. L., Yu P. S., Turek J. and D. M. Dias, "A parallel hash join algorithm for managing data skew", *IEEE Transactions On Parallel and Distributed Systems*, volume 4, number 12, December 1993.