Parallel and Concurrent Operations in Relational Databases

Ph.D. Eng. Georgios KAPENEKAS, Ph.D. George A. THANOS
Technological Education Institute Halkida Greece

1 Operations in the relational data model

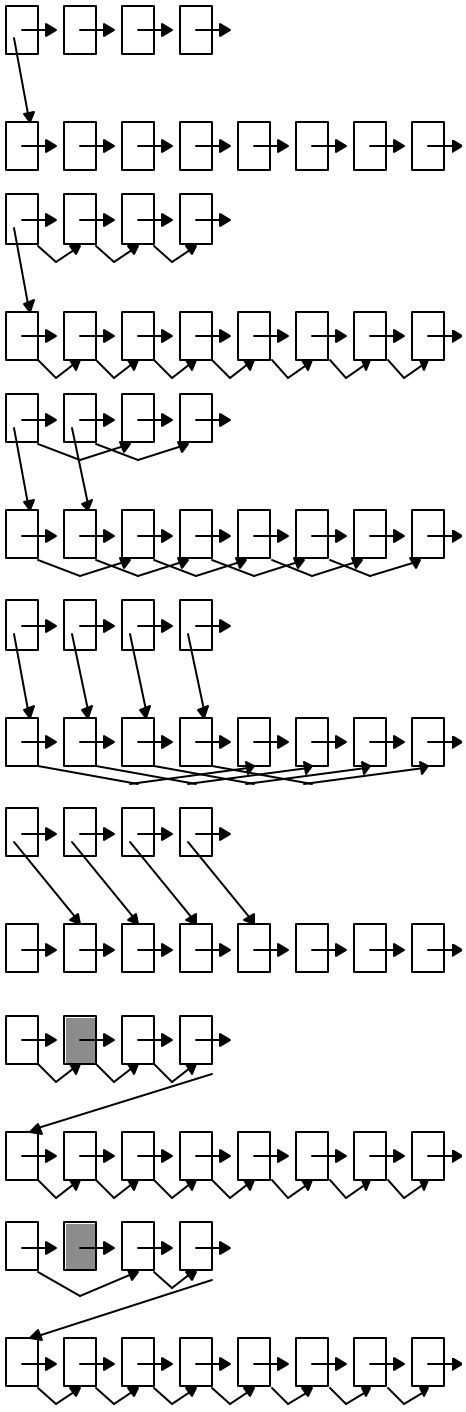
The relational model is one of the most used data model, it is a good choice for the implementation of databases. This is a powerful model, it supports simple and declarative languages, and it is value oriented. It has the ability to define operations on relations whose results are themselves relations, and the operations can be combined and cascaded easily The first variant we can implement operations in the relational model is to allow any program to be an operation on relations. The main disadvantage of this case is that the programs have to know everything about the physical structures of data used, and the code depends on the particular structure selected at the design time. As a result it is preferred in database system to design query languages, that deals only with data model, not of a particular physical implementation of the model[1,6,7]. Supplementary to choose of a query language that operates only on relations, it is important that operations to have implementations that are efficient, since we want fast response to query on large databases.

2. Union

The union of relations R and S, denoted $R \cup S$, is the set of tuples that are in R or S or both[4,5]. We may only apply the union operator to relations of the same arity, so all tuples in the result have the same number of components. Recall that the attribute name for the operand relations is ignored and to the result relation can be given attributes arbitrarily. The order of attributes in the operands is respected when taking the union. We consider two relations R and S that have the tuples stored in two lists. Each tuple of the two relations is associated with a processor that keeps the data. Every processor has four local variables: ***next[k]*** pointer to the next processors from list; ***asociat[k]*** pointer between two tuples from different relations; ***urm[k]*** auxiliary variable and ***test[k]*** a Boolean variable that indicates if the tuple must be added to the result relation[11].

```
for all k in parallel do ç
        asociat[k]:=null
        test[k]:=true
        asociat[list1]:=list2
for all k in parallel do ç
        urm[k]:=next[k]
        while urm[k]≠null do |
                if asociat[k]≠null then |
                        asociat[urm[k]]:=urm[asociat[k]]
                        urm[k]:=urm[urm[k]]
        |
|
next[end_list2]:=list2
for all k in parallel do ç
        if data[k]:=data[asociat[k]] then |
                test[k]:=false
        asociat[k]:=next[asociat[k]]
        while asociat[k]≠list2 do |
                if data[k]:=data[asociat[k]] then |
                        test[k]:=false
                asociat[k]:=next[asociat[k]]
        |
|
next[end_list2]:=null
for all l in parallel do ç
        urm[l]:=next[l]
for j:=1 to log2k do |
        for all k in parallel do ç
```

```
                    if test[k]=false then ç
                           urm[k]:=urm[urm[k]]
              |
        |
    urm[end_list1]:=list2
```



The algorithm to compute union of two relations begins by initialization of the local variables of processors that store relation R. Initially, we assume that does not exist any tuple from a relation that is friend with a tuple for other relation and in result relation we will keep all tuples from R.

In the second part of algorithm we achieve an association in ascendant order of processors that store tuples from relations R and S, at the end of loop for each tuple from relation R exist an friend tuple in relation S. We consider that the number of tuples in R is less that number of tuples in S. For this loop the compute time is $O(\log_2 T_R)$.

In the third loop we follow the pointer from circular list of tuples from relation S and compare with each tuple from R to find another occurrence of an tuple from relation R that is in relation S to eliminate it from result relation. The compute time for this loop is $O(T_S)$.

In the last loop we get the union relation after we exclude the duplicate. Union relation is keep in a list store by pointers urm[l]. The compute time for this loop has a logarithmic value of number of tuples from R $O(\log_2 T_R)$.

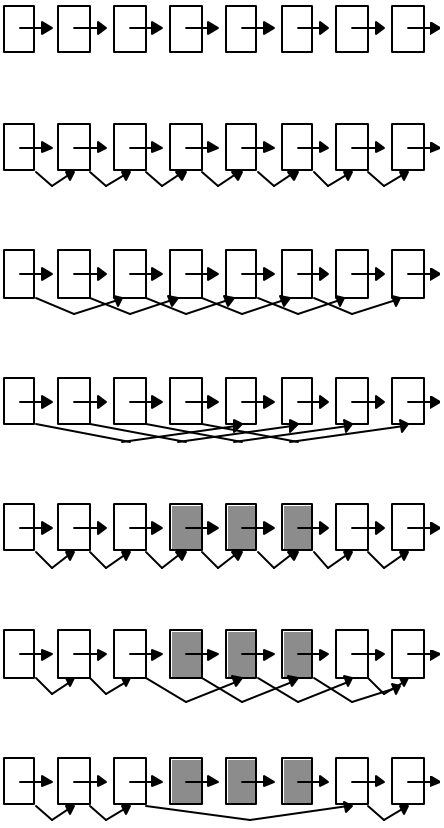
3. Selection

Let F be a formula involving

- operands that are constants or component's attribute of R,
- the arithmetic comparison operators $<$, $=$, $>$, $\leq$, $\neq$, $\geq$;
- the logical operators $\wedge$, $\vee$, NOT.

Then $\sigma(R)$ is the set of tuples μ in R such that when we substitute for all i the i^{th} component of μ for any occurrences of &i in formula, it becomes true. To demonstrate how we can calculate selection in a parallel system we consider a relation S, which each tuple is stored by an unique processor from an array of processors. Each processor have the following variables: ***data[k]*** to keep the initial tuple; ***next[k]*** pointer to next processor that have an tuple, ***test[k]*** a Boolean variable that is true if the local tuple is not a duplicate of another tuple from relation R, ***sel[k]*** to keep tuple of result selection relation and ***urm[k]***, pointer to next processor that keep a tuple from result relation[11].

```
      for all k in parallel do ç
        sel[k]:=selection(data[k])
        urm[k]:=next[k]
|
urm[end]:=list
for j:=1 to log2k do |
        for all k in parallel do ç
                if sel[k]:=sel[urm[k]] then |
                        test[k]:=false
                |
        |
        urm[k]:=urm[urm[k]]
|
for all k in parallel do ç
        urm[k]:=next[k]
|
for j:=1 to log2k do |
        for all k in parallel do ç
                if test[k]:=false then |
                        urm[k]:=urm[urm[k]]
                |
        |
|
```



each tuple from relations R and S is associated to a processor. All tuples from relation R and S are keep in two different lists. Each processor have the following variables: ***next[k]*** - pointer to next processor that keep a tuple from relation; ***urm[k]*** - auxiliary variable keeping a pointer to tuple from other relation to which we compare the local tuple; ***test[k]*** - a Boolean variable that is true if the tuple must be in result relation; ***data[k]*** - to keep the local tuple. Finally relation R-S result in list keeps by pointers ***urm[k]***.

The algorithm [8] is based on simultaneous read of data by the processors. In the initialisation phase all processors which keep tuples of relation R attribute to variable urm[k] the address of processor that keep the first tuple from relation S After comparison of tuples the variable test[k] is set according to necessity of keeping the tuple in result relation we pass to the next tuple from relation S. This process need an compute time of O(l), where l is the number of tuples of relation S. Finally are actualised pointers urm[k] to obtain relation R-S. The execution time is of O(log k), where k is the number of tuples from relation R.

4. Set difference

The set difference of relations R and S, denoted R-S, is the set of tuples in R but not in S. The arity of relations R and S must be the same. In our implementation we consider that

```
for all k in parallel do ç
          urm[k]:=list 2
          test[k]:=true
   |
   while urm[k] ≠ null do |
```

```
        for all k in parallel do ç
                if data[k] = data[urm[k]] then |
                        test[k]:=false
                |
        |
        urm[k]:=urm[urm[k]]
|
for all k in parallel do ç
        urm[k]:=next[k]
|
for j:=1 to log2k do |
        for all k in parallel do ç
                if test[k]:=false then |
                        urm[k]:=urm[urm[k]]
                |
        |
```

5. Cartesian product

Let R and S be two relations of arity k1 and k2, respectively. Cartesian product of R and S, denoted RxS, is the set of all possible (k1+ k2) tuples whose first k1 components from a tuple in R and whose last k2 components from a tuple in S.

Each tuple of the two relations R and S is associated with a processor from system. Tuples of relation S are stored in a list, to pass from a tuple to next tuple is used variable ***next[k]***. Last tuple of relation has ***next[k]*** = null. The algorithm uses simultaneous read of data by more processors: all processors that have an tuple of first relation read the same tuple of second relation to calculate a new tuple of Cartesian product. Execution time is of $O(T_S)$ where T_S is the number of tuples of relation S.

```
for all k in parallel do ç
        urm[k]:=list 2
|
for j:=1 to Ts do |
        prod[j]:= prod-cart(data[k] ,data[urm[k]])
        urm[k]:=urm[urm[k]]
|
```

6. Computing the join by selection from product

The obvious way to compute join R ∞ S is to compute product R x S and keeps the tuple μν, where μ is in R and ν in S, only for those tuples that have the same value for common attributes in R and S.

6.1. Join using two indices

A better cost we obtain if exist an index on attribute B in both relations. Presume that both indices are clustering. We can find the set of values of B by examining one of the indices. We use the index with the smaller size, assume that is the index on attribute B of relation S and has the size $I_{S,B}$. Using the index on B of S we retrieve all value of B. It is not necessary to find all values for B of R because those absent from S not appear in the result join. Once we have the set values of B we can retrieve the needed tuples of relations R and S. The algorithm is:

```
to each value b of B do
   join tuples of σB=b(R) with tuples of
σB=b(S)
```

7. Studying the communication in parallel architectures

In database systems we keep information on the extern support. To process it, we must read and send the data to the processors. Generally, the number of extern storage units is less than the number of processors. First strategy to get the data needed by processors is to assure access for each processor to the secondary storage. The main disadvantages of this solution are:

-when we grow the number of processors appears a bottleneck in communication network, because only one processor can access an unit at once;

-the duration of input/output operations is big and the same information can be read by many processors at different moment of time.

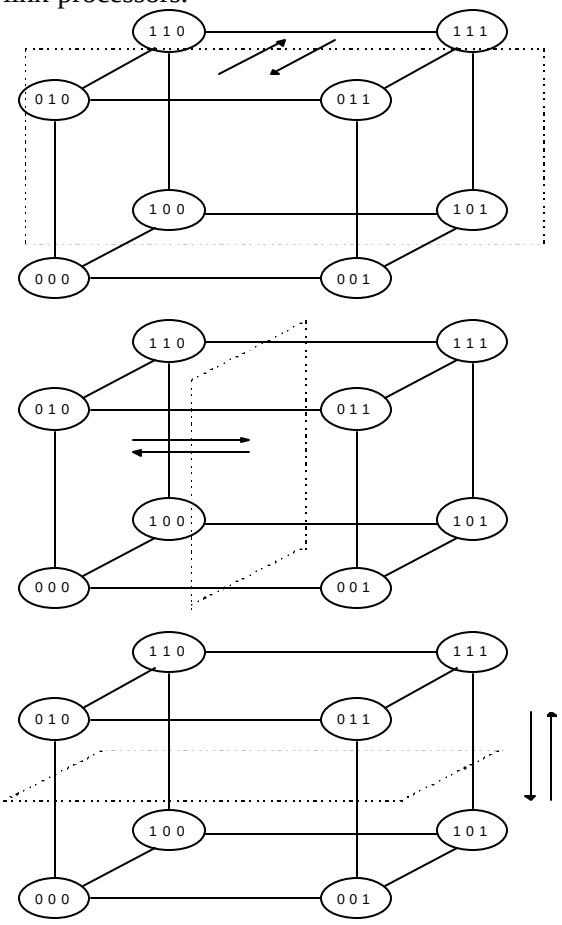
A second strategy is to dedicate processors to each storage unit, the information of such disk can be read only by the assigned processor and send to other processors to process. The processors that realize the input/output operation when are not asked to read or to write can be added to the set of processors that process data. The method of specialize some processors in input/output operations has the advantage that uses a simple management of data, and if the same information must be sent to many processors needs only one input operation, then information is sent faster in intercommunication network to different destinations. In this case we mention that the distribution of input data to processors is more flexible and is done in a shorter time. As an example, assume that processor 1 reads the information. In the first step it sends to the processor 2. Iterative, on each step, all processors which obtained data send it to other processors that not receive it yet. In this way, on second step processor1 send data to processor3, and processor 2 sends to processor 4. Generally, we can say that if exist p processors that must receive the information, this algorithm needs to distribute the input data log(p) steps, not in p-1 steps if the same processor sands information to all destinations.

Another case that appears in parallel processing is when data is distributed on a set of processors and must communicate data they hold to all other. At the end of communication all processors have the same data. To solve this problem we must have in mind the hardware structure of machine, thus the main categories are: a ring structure and a hypercube structure.

For ring structure we select a direction on which we send data and each processor sends its data to immediately adjacent processor on this direction. On next stage each processor continues to send data it received during the previous stage.

If we have p processors connected in a ring, to communicate data from each to all other processors it needs (p-1) steps of communication.

Assume that we have $p=2^d$ processors and the connecting structure of processors hypercube. In this case we have d directions that link processors.



Communication between processors in hypercube

On first stage of algorithm we select a direction that splits the processors in two groups of equal size, each processor in the first group is paired with a processor in the second. The paired processors exchange all data they hold. After this stage we have p/2 pair of processors that contain the same data. On next stage we chose another direction that split all processors in pairs that will communicate directly all data. This way, in each stage each processor will double the amount of data it contains. The total number of stage for this method is $d=\log_2(p)$.

8. The parallelism of input/output operations

The parallelism of input/output operations refers to reducing the time needed to retrieve relations from disk by partitioning the relations on multiple disks of storing information. The most common form of data partitioning in a parallel database environment is horizontal partitioning. In this case, the tuples of a relation are divided among many disks, such that each tuple is entirely stored on one disk. To exemplify several data partitioning strategies we assume that there are n disks, D0, D1...Dn-1, that can keep the information[2,3].

Once a relation has been partitioned among several disks, we can retrieve it by reading in parallel, using all the disks. Similarly, a relation can be written to multiple disks in parallel. The transfer rates for reading or writing an entire relation are much faster with input/output operations in parallel than in sequential operation. On the other hand, when we use the partitioning of relations on many disks we must take into account that exist many access types to the database, in function of the amount of information needed to retrieve:

- access to the entire relation by scanning all the tuples
- locating a tuple or look up tuples that have a specified value for a specific attribute
- locating all tuples that have for a given attribute a value that lies within a specified range

First possible technique for partitioning the tuples on disk is to scan the relation in any order and the each i^{th} tuple is sent to disk $D_{i \bmod n}$. This scheme ensures an even distribution of tuples across disks; each disk has approximately the same number of tuples as do the others. The method is useful for applications that need to read the entire relation sequentially for each query. However the queries that locating one tuple or tuples with values that lied in a range are complicated to process, since each of the n disks must be used for the search.

Another technique that can be take into account is the hash partitioning. For this strategy we can choose one or more attributes of the given relation as the partitioning attributes. We chose a hash function whose values are in the range {0, 1,..., n-1}. Each tuple of the original relation is sent to a disk find by the result of hash function on the partitioning attributes. If the hash function returns i, then the tuple is placed on disk Di. This method is best suited for queries that access a tuple based on the partitioning attribute. We start from the attribute values of needed tuple and we apply the hash function to locate the disk that keeps the tuple. Directing a query to a single disk reduces the start-up cost of initiating a query on multiple disks, and leaves the other disks free to process other queries. This strategy is useful for sequential scans of the entire relation. If the hash function is a good randomising function, and the partitioning attributes form a key of the relation, then the number of tuples in each of the disks is approximately the same, without much variance. The time taken to scan the relation is approximately 1/n of the time required to scan the relation in a single disk system.

This strategy is not well suited for queries that retrieve tuples on non-partitioning attributes, and or for queries that have as results a range of tuples in relation, since hash functions do not preserve proximity within a range. Therefore, all the disks need to be scanned for range queries to be answered.

The last proposal strategy is the range partitioning. This method distributes contiguous attribute-value ranges to each disk. It is chosen a partitioning attribute A that is a partitioning vector. Let this vector $[v_0, v_1,...,v_{n-2}]$ in increased order, such that if $i < j$, then $v_i < v_j$. In this case the relation is partitioned as follows. For each tuple of relation if the value of attribute chosen as partitioning attribute is less than v_0, the tuple is placed on disk D_0. If the value is greater than v_{n-2}, then the tuple is placed on disk D_{n-1}. If the value respect the relations $v_i \le x < v_{i+1}$, then the tuple is placed on disk D_{i+1}. This partitioning way is well suited for queries that locate a tuple or a range of tuples on the partitioning attribute. For queries that locate a record we can consult the partitioning vector to locate

the disk where the tuple resides. For queries that locate a range of tuples we consult the partitioning vector to find the range of disks on which the tuples may reside. In both cases, the search is narrowed to exactly those disks that might have any tuples of interest. This feature is both an advantage and a disadvantage. The advantage is that, if there are only a few tuples in the queried range, then the query is typically sent to one disk, as opposed to all the disks. In this case other disks can be used to answer other queries, range partitioning results in higher throughput while maintaining good response time. On the other hand, if there are many tuples in the queried range, many tuples have to be retrieved from a few disks, resulting in an input/output bottleneck at those disks. In this case the other two partitioning strategies would engage all the disks for such queries, giving a faster response time.

Into a computer system with many disks, the numbers of disk on which the relation is partitioned depend on the number of tuples. If a relation has a small number of tuples, that can fit in a single block on disk; it is preferable to keep all relation on a single disk. For relations with a large number of tuples is preferable to partitioning tuples across the all disks. If relation has m disk blocks and the system has n disks then the relation is stored in min(m,n) disks.

A special attention must be give to the distribution of tuples when a relation is partitioned, except for first method, because it can appear a skew in distribution, with a high percentage of tuples placed in some partitions and fewer tuples in other partitions. The skew can appear for two causes: attribute-value skew and partition skew.

Attribute-value skew refers to the fact that some values appear in the partitioning attributes of many tuples. All the tuples with the same value for the partitioning attribute end up in the same partition, resulting in skew. Attribute-value skew can result in skewed partitioning whether range partitioning or hash partitioning is used' since the partition vector is not chosen carefully, or in the second case the hash function is not good. Partition skew refers to the fact that there may be load imbalance in the partitioning, even when there is no attribute skew.

If the partitioning attributes form a key for the relation, a good range-partitioning vector can be constructed by means of sorting. The relation is first sorted on the partitioning attributes, then the relation is scanned in sorted order. After every 1/n of the relation has been read, the value of the partitioning attribute of the next tuple is added to the partition vector. This way the partitioning on n disk is uniformly. The disadvantage of this method is the extra input/output overhead needed for the initial sort.

Bibliography

[1]. S. Aditedonl, R. Hull, V. Viann - *Foundations of databases* - Adisson Wesley 1995;

[2]. D. De Witt, J. Cray - *Parallel Database Systems: The Future of High Performance Database Systems* - Communications of the ACM Memory Multiprocessors Kluwer Academic Publisher;

[3]. D. Bell, J. Grimson - *Distributed Database System* - Adisson Wesley;

[4]. E. F. Codd - *The Relational Model for Database Management* - Adisson Wesley

[5]. P. Atseni, V. De Antonellis - *Relational Database Theory* - Benjamin Cummings;

[6]. M.Petrescu, *Proiectarea bazelor de date* - note de curs;

[7]. C.J.Date - *An Introduction to Database Systems* - Adisson Wesley 1995;

[8]. C.J. Date, G. Darwen - *A Guide to the SQL Standard* - Adisson Wesley 1993;

[9]. Sillerschatz, Korth, Sudarshan - *Database System Concepts* - McGraw-Hill 4th Edition;

[10]. Buchanan - Distributed Systems and Networks – McGraw-Hill 2000;

[11]. G. Kapenekas - *Transfer optimisation for calculation of relational expression in parallel systems and distributed databases* - Computer science The Proceedings of the 3-rd International Symposium of Economic Information - may 1997;