



A Survey of Query Optimization in Parallel Databases

Max Kremer and Jarek Gryz

Technical Report CS-1999-04

November 1, 1999

Department of Computer Science

4700 Keele Street North York, Ontario M3J 1P3 Canada

A Survey of Query Optimization in Parallel Databases

Max Kremer and Jarek Gryz
York University
Toronto, Canada

Abstract

There has been much research into parallel database query processing. In this paper, we focus on several techniques for query optimization in shared-nothing parallel database systems. We examine several methods of searching for optimal query execution plans, cost models and processor allocation algorithms. We also focus on static and dynamic load balancing techniques.

A survey of the proposed methods reveals their pros and cons. Taking advantage of strengths and eliminating weaknesses is the goal in implementing an algorithm. We describe a design method which fits each algorithm to the environment it is best suited for.

Introduction

The future of large database machines lie in the realm of highly parallel computers. The main reason for this is that parallel machines can be constructed at a low cost without the need for any specialized technology, using existing sequential computers and relatively cheap interconnection networks.

Parallelism demonstrates two properties that make it desirable. The first is possible linear *speedup* where adding n times as many processors reduces response time by a factor of n . The second is linear *scaleup* where adding n times as many processors allows the system to perform a task n times the size in the same amount of time. Three problems exist which hinder parallelism in obtaining linear speedup and scaleup:

Startup: Starting a parallel system may require the creation and coordination of thousands of processes. If this task dominates the actual query processing time then an inefficient system results.

Contention: As more processes are created, contention for shared resources increases. This slows down the system.

Skew: Skewed data can lead to unbalanced load so that only a few processors are working while the rest are idle.

A perfect algorithm on a parallel machine would scale easily at a low-cost, and would demonstrate linear speedup and scaleup even for hundreds or even thousands processors. Three architectures have emerged from the quest for perfect parallel machine; they are shared-memory (SM), shared-disk (SD) and shared-nothing (SN). In the shared-memory architecture, each processor has access to all disks in the system and to a global shared memory.

The main advantage of a shared-memory system is that it has zero communication cost; processors exchange messages and data through the shared memory. This also makes it easier to synchronize processes. Another advantage of such a system is that load-imbalance can be corrected with minimal overhead. The drawback of such a system is the problem of contention. A shared-memory architecture can only be scaled to few processors (30-40) before interference begins to affect the rate at which the memory can be accessed.

Shared-nothing is the extreme opposite: every processor has its own memory and disks. Nodes communicate by passing messages to one another through an interconnection network. The main advantage of shared-nothing is its ability to scale to hundreds and potentially thousands of processors. The drawbacks of such a system are complications created by load-imbalance and its dependence on a high bandwidth interconnection network for effective message exchange. Parallelizing a sort or join operation may gain little in response time if the data is highly skewed, and the operation depends a great deal of coordination among the nodes. Another disadvantage to shared-nothing architecture is the problem of node failure. If a processor fails, then all access to the data owned by that processor is lost. A solution to the node-failure problem is a shared-disk architecture where every processor can read and write to any of the disks in the system but manages its own memory.

Shared-nothing has emerged as the design of choice, mainly due to its high reliability and ease of scalability to hundreds of processors. Of course one of the main issues is cost, and since shared-nothing architectures can be constructed from existing sequential machines using a simple interconnection network, it is the most cost effective means of parallelizing a database system. Most of the current research in the field of parallel databases concentrates on shared nothing architectures; this fact is reflected in this paper where most of the algorithms and solutions discussed apply mainly to shared-nothing systems.

In examining the way in which queries are executed on parallel systems we observe two forms of parallelism:

Intra-operator parallelism: one operation is parallelized over several processors. On a shared-nothing system this is achieved by partitioning or de-clustering the data among the processors. When an operation such as a scan is performed each processor operates on its local data cluster, the results of each processor are then combined for the final result

(which can also be partitioned among processors). There exist several methods for de-clustering data; these include round-robin distribution, range partitioning and hash partitioning.

Inter-operator parallelism: several operations are executed concurrently. Each of these operations is assigned to one or more processor. When more than one processor is used for a given operation, then we are also exploiting intra-operator parallelism. There are two forms of inter-operator parallelism: independent and pipelined. Independent means that no dependencies exist between operations, pipelined refers to a producer – consumer relationship between operators. The result of the producer operator is pipelined to the input of the consumer operator instead of being materialized. This has the obvious advantage of not having to wait for the input data to be completed before starting a new operation on the data.

Notice that in the above discussion we refer to the parallelism of operations while not clearly defining the specifics of an operation. This is because there exists varying degrees of granularity for parallelism, depending on how small or how big are the pieces used to construct a query. On one end of the scale we can parallelize entire queries, referred to as coarse granularity, and on the other end we parallelize each of the components, for example a hash-join: build, scan and probe, this is called fine granularity. As we shall see granularity plays an important role in query optimization for parallel execution.

In this paper we will discuss some of the ways in which queries can be optimized for parallel execution. We will look at various cost models, search algorithms and methods of generating query execution plans (QEPs) [GANG92], [LU91], resource allocation techniques [GAR96], [NIC93], [TAN93] and ways of dealing with load-imbalance[LuTAN94]. We will focus on some of the major issues associated with parallel databases and how well these algorithms address them.

Search Strategies for Finding Optimal QEPs

Optimization strategies for parallel execution come in two flavors: two-phase optimization and one-phase optimization. The first phase in two-phase optimization generates a query execution plan (without scheduling information). A resource allocation algorithm is then applied in the second phase to distribute the query into schedulable components. The one-phase approach takes resources into consideration when generating the QEP. This method seems more likely to generate optimal plans since it is difficult to see how an optimal plan can be constructed without considering preemptable resources such as CPU, disks and the communication network.

Modeling a QEP is the first step in searching for an optimal one. The method used in most approaches ([GAR96], [LU91], [LAN94], [GANG92]) is to model QEPs as

annotated join trees. The level of granularity is thus reflected in the nodes of the join tree, which are base relations at the leaves and components of join operations in internal nodes.

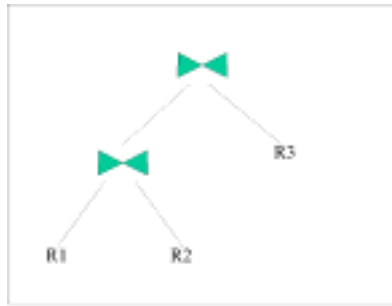


Figure 1. A three way join of R1, R2 and R3

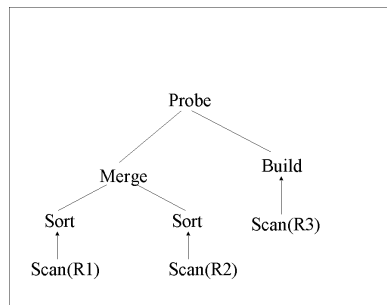


Figure 2. The join graph from Figure 1 one is decomposed into atomic components. From the figure we can see that R1 and R2 are joined using a sort-merge join and the result is joined with R3 using a hash-join.

If pipelined parallelism is to be considered when generating an execution plan, the edges of the tree should be distinguished as either materialized or pipelined. When decomposing the tree operations, sub-trees connected by pipelined edges must be scheduled together. Two problems can arise with pipelining. The first is bursty or unpredictable output from the producer. This leads to irregular patterns of idle time for the consumer. The second problem is long pipeline chains. Long chains can result in high propagation delay from the first producer to the final consumer. As with the first problem, it is the unpredictability of idle times that result in inefficiencies. If idle times are known to be regular, the scheduler can consider such plans.

The problem of determining the optimal join ordering is NP-hard on the number of base relations. So an exhaustive search would be far too time consuming for larger number of joins. The number of possible parallel execution plans greatly exceeds the number of possible sequential plans due to the extra possibilities for exploiting parallelism such as partitioning and pipelining. Single processor optimizers limit the search space by considering only certain tree configurations.

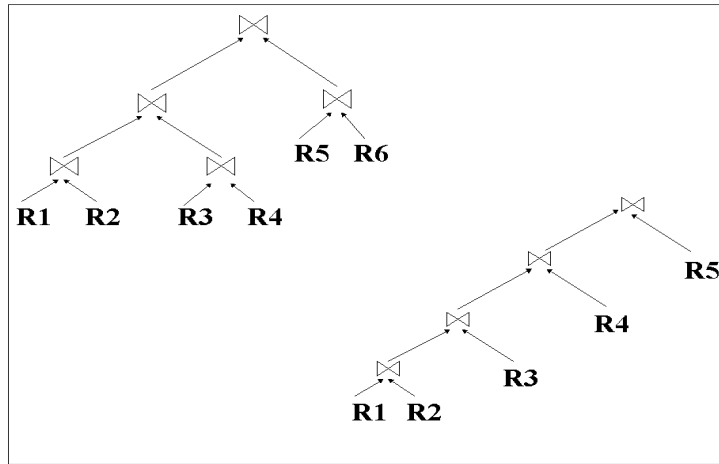


Figure 3. A bushy tree joining 6 relations and a linear left-deep tree joining 5 relations.

The IBM System R optimizer, for example, considered only left-deep trees. This type of limitation is unacceptable for parallel machines, for which bushy trees offer the best opportunity for exploiting independent parallelism. When bushy trees are included in the search space of an optimizer, the number of feasible QEPs increases dramatically. Limiting the size of the search space and using a non-exhaustive search algorithm is critical. Search strategies can be organized as depicted in Figure 4.

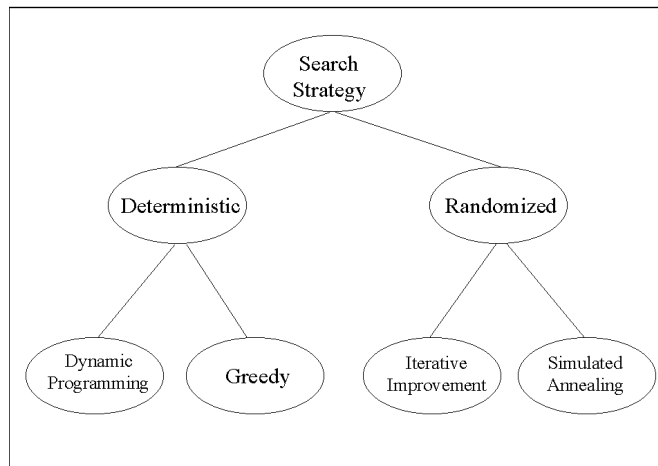


Figure 4. Organization of Search Strategies

Deterministic strategies proceed by building plans, starting from base relations and joining another relation at each step until the complete query is formed. When constructing QEPs through dynamic programming (DP), equivalent partial plans are constructed and compared based on some cost model. Partial plans that are cheaper are retained and used to construct the full plan. An important issue (addressed in [LAN94]) is determining what is meant by equivalent when comparing two plans. Partial plans that produce the same

output of tuples seem to be equivalent, but actually the order of tuples can also be important. If the result of a join is used as the input to a sort-merge join with another relation, then we can save on extra work if the tuples are sorted. The System R optimizer takes this kind of information into account when comparing equivalent plans.

A fundamental property that must hold when using DP techniques is the principle of optimality, which when applied to the context of QEPs says that if two plans differ only in a sub-plan, then the plan with the better sub-plan is also the better plan. Only cost models, which observe the principle of optimality, can be used with DP. In the section **Cost Models** we discuss models that do and do not satisfy this principle.

Greedy algorithms are another deterministic strategy. They work by making the choice that seems best at the current iteration. The parallel multi-way join optimization algorithm GP, considered in [LU91], uses a greedy approach. The algorithm considers bushy QEPs as well as linear ones. To limit the search space QEPs are divided into synchronized and unsynchronized. Synchronized QEPs are those where the whole multi-way join process is divided into synchronized steps. At each step a number of joins are executed concurrently and the joins at the next step will not begin executing until all joins in the previous step are complete. The greedy multi-way join algorithm works by selecting as many relations as possible (this is the greedy part) to be joined concurrently at each synchronized step. In the beginning, all relations make up the working set T. A set of relation pairs, R, is selected for the first step by calling a function to select k pairs of relations to be joined at the current step. The function to select pairs of relations uses a minimum cost function that takes as its input the working set and the number of relation pairs k to be joined concurrently and determines the minimum cost plan that joins those k relation pairs first and determines the join method for each pair of relations. Unfortunately the GP algorithm only makes use of independent parallelism. By considering only synchronized QEPs pipelining between join steps is not put to use.

Randomized strategies work by searching for the optimal plan around some particular points. Such strategies do not guarantee the best plan, but they avoid the high cost of optimization incurred by other methods such as DP or exhaustive search. The first step is finding a start plan, which is usually built using some kind of simple greedy strategy. Next a random transformation is applied to the plan, for example, this may consist of switching two randomly chosen relations in the join order. Two randomized strategies are simulated annealing (SA) and iterative improvement (II). They differ on criteria for replacing the current plan with a transformed one and on the stopping criteria. II works similarly to the greedy algorithm. Starting from an initial state it repeatedly checks random neighboring plans (differed by a single transformation) and accepts those that exhibit a lower cost. By using different initial states, different local minimal plans can be obtained. The best of these local minimal plans is then chosen. It can be shown [reference 12 of LAN94] that as time approaches infinity, the probability of finding the global minimum approaches 1. While II accepts only local minimums, SA can take steps

that worsen the plan, to prevent being trapped in a high-cost local minimum. Such moves are taken based on a probability that diminishes at each iteration.

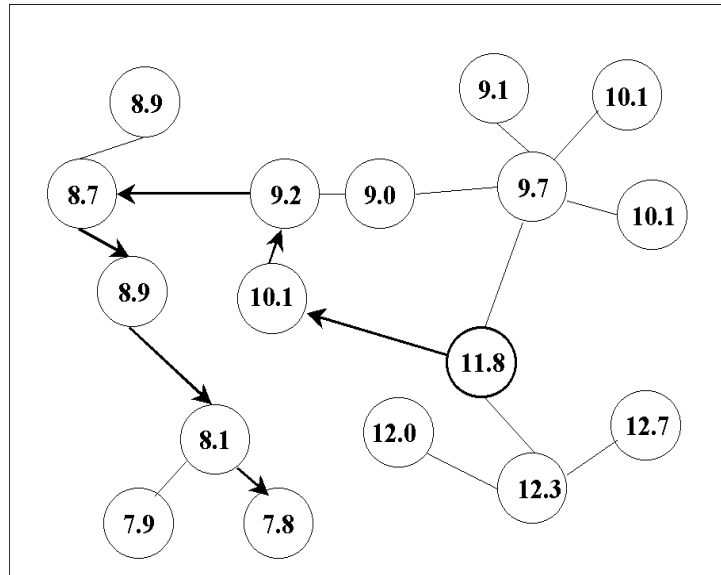


Figure 5. An II path through the search space of plans. Each plan has cost associated with it and the initial plan is shown bolded.

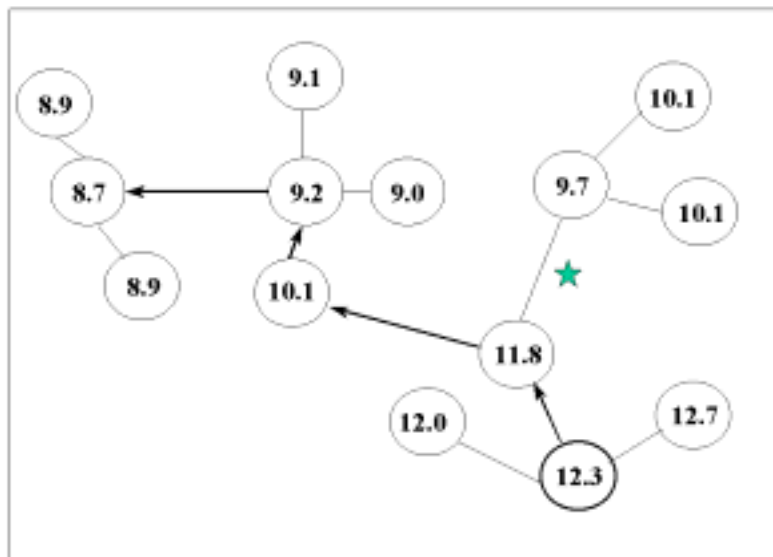


Figure 6. A SA path through the search space of plans. Notice how the algorithm ignores a lower cost plan and chooses a more expensive one (denoted by the star).

Cost Models and Searching

Cost models are used by the optimizer to select the best QEP from among the candidates searched, as well as to allocate resources efficiently. The information included

in a cost model depends on the system architecture, the type of search algorithm, the level of complexity considered acceptable, information describing the data and any assumptions that must be made for simplifying reasons. We will discuss each of these factors.

System Architecture: In a SM architecture all communication is through the shared-memory, hence repartitioning of tuples in SM is simply in-memory hashing, whereas SN would involve transfer of data across the interconnection network. The locality of data in a SD system is not relevant when allocating processor tasks, hence load-balancing is a much simpler issue than it would be with SN. A major factor affecting the SD architecture under heavy load is the increase in communication due to lock request messages, this must be considered when a single process requires many locks. Memory consumption in SN and SD is complicated by inter-operation parallelism. In SM all operations use the same global memory making it easier to test whether there is enough space to execute them in parallel, i.e., the sum of the memory consumption by the individual operators is less than the available memory.

Type of Search Algorithm: As discussed earlier, problems solved using DP must obey the optimality principle. As an example [GANG92] presents a cost model which violates the principle of optimality. The cost model must preserve any fundamental principles that are necessary for the correctness of an algorithm (for instance, the principle of optimality for DP algorithms).

Semantic Data Descriptions: Optimizer cost models should also reflect semantic information about the data to aid in query processing. For example, if two relation R1 and R2 are being joined on a common attribute a , and a is a (non-nullable) foreign key in R2, then we know that all of R2 will join with R1. Another example of incorporating semantic information in cost models is employing information held in integrity constraints. For example, we have the following IC:

Senior Constraint:

Age > 50 \rightarrow Salary > 45,000

The above IC says that if an employee's age exceeds fifty, then their salary must exceed 45k. If the employee table is range declustered on Age we automatically gain information on the locality of Salary. Consider the following query:

```
SELECT *
FROM Employees E
WHERE E.Salary > 45000
```

Given the cardinality of the Age attribute and declustering information, we can predict where the processing load for this query will be concentrated.

Simplifying Assumptions: When designing a cost model, certain factors must be ignored either to simplify calculations or because some things are too difficult to measure. Two assumption made in both [GAR96] and [GANG92] are No Time-Sharing Overhead and Uniform Resource Usage. The first assumption states that dividing a preemptable resource among multiple processes incurs no additional costs. The second assumption states that usage of a preemptable resource by a process is uniformly spread over the execution of the process.

The following three simple algorithms taken from [CHEN92], demonstrate the use of cost models. The first algorithm, GMC greedy, generates a join sequence from a join graph (see Figure 7.) based on minimum cost:

GMC:

1. repeat until $|V| = 1$
2. begin
3. Choose the join (R_i, R_j) from $G = (V, E)$ such that $\text{cost}(R_i, R_j) = \min \{ \text{cost}(R_p, R_q) \}$
 $\forall R_p, R_q \in V$
4. Perform (R_i, R_j) and merge result into G
5. Update profile accordingly
6. end

The above procedure uses the cost function to perform the minimum cost join first, the resultant relation is then added to the join graph and the profile (see figure 9) for the relations is updated. The algorithm continues until a single relation is left in the join graph.

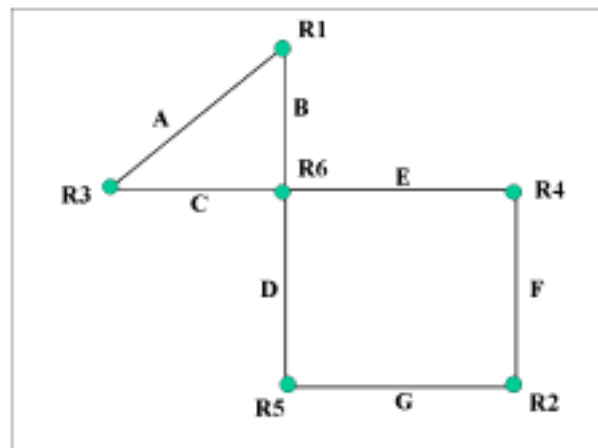


Figure 7. An example query graph where the vertices are relations and edges are labeled with the joining attribute.

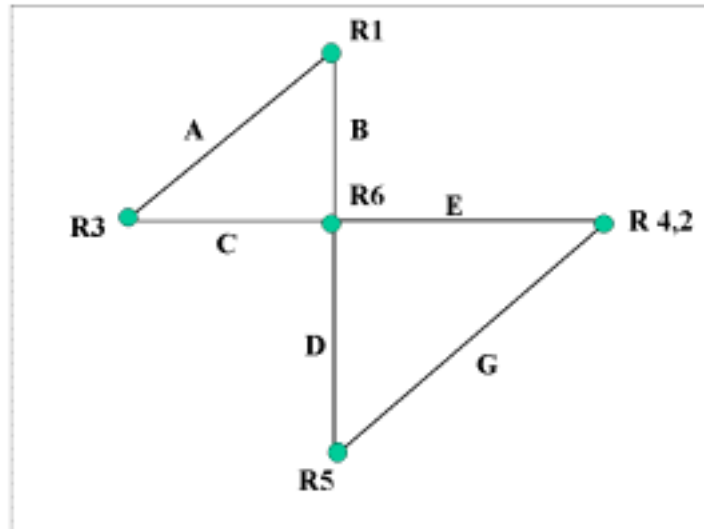


Figure 8. The join graph after joining R4 and R2 on attribute F.

Relation	R1	R2	R3	R4	R5	R6
Cardinality	118	102	106	100	131	120

Figure 9. Profile for relations

In a sequence of joins, the cardinalities of intermediate relations resulting from early joins affect the cost of joins to be performed later. The join procedure is assumed to be a Sort-Merge-Join, hence the *cost* function used by GMC is expressed by $|R_i| + |R_j| + |R_i \times R_j|$. This formula is considered as a general approximation for joining large relations on a shared-nothing multi-processor system. Since the objective is to minimize the total cost required to perform a sequence of joins, we may want to execute joins that produce smaller resulting relations. The following two algorithms are variations of GMC. The first is GMR which uses the minimal resulting relations. The second is GME which uses minimal expansion. The expansion of a join is defined as the size increase of the resulting relation over the two participating in the join.

GMR:

1. repeat until $|V| = 1$
2. begin
3. Choose the join (R_i, R_j) from $G = (V, E)$ such that $|R_i \times R_j| = \min \{|R_p \times R_q|\} \forall R_p, R_q \in V$
4. Perform (R_i, R_j) and merge result into G
5. Update profile accordingly
6. end

GME:

1. repeat until $|V| = 1$
2. begin
3. Choose the join (R_i, R_j) from $G = (V, E)$ such that $|(R_i, R_j)| - |R_i| - |R_j| = \min\{|(R_p, R_q)| - |R_p| - |R_q|\} \forall R_p, R_q \in V$
4. Perform (R_i, R_j) and merge result into G
5. Update profile accordingly
6. end

The resulting join tree from the join graph (Figure 8.) using GMC is $((R2,R4),(R5,R6),(R1,R3))$. The resultant join trees using GME and GMR are equivalent in this example: $((R1,R3),R6),R5),(R2,R4))$. Using an algorithm GOPT (finds the optimal plan) which enumerates all possible join trees we can find the join tree with the lowest cost and compare it to each of the methods described above. Tests of the three algorithms were performed [CHEN92] on queries containing 4, 6, 8 and 10 relations. For each of the four query sizes, 300 queries were randomly generated. Test results for each of three algorithms, measuring average execution costs, were compared to the optimal costs produced by GOPT. The results obtained with GMR are better than the other two for each of the four query sizes. In second place is GME followed by GMC. The results are summarized in the table below (Figure 10).

Number of relations	GMC	GMR	GME
4	107.3 %	106.8 %	107.1 %
6	107.9 %	105.9 %	106.1 %
8	124 %	110.9 %	112 %
10	134 %	124 %	128 %

Figure 10. Percent of optimal cost for each algorithm

The GP greedy parallel multi-way join algorithm described earlier uses an interesting notion of cost. Two types of algorithms are presented; one based on total cost, GP_T and one based on partial cost, GP_P . The total cost of a QEP is computed as the sum of the costs of each synchronized step. It can be quite expensive to compute the total cost at

each iteration, therefore a second version of GP, GP_P computes the partial cost of a QEP by calculating the cost of QEPs which join k pairs of relations concurrently only in the first step. In GP_T , the total cost of a QEP, $Cost_{plan}$, is computed as the sum of the costs at each step i , $Cost_i$. Even when the number of joins is small, enumerating all possible combinations of joins at each step for all possible sequences of steps can be very expensive. An important heuristic used to limit the search space in GP_T is to consider only those QEPs that execute k joins concurrently at the first step and all remaining joins sequentially. For such QEPs the plan cost is given by:

$$Total_k = Par_Join_cost(R_k) + Seq_Join_cost(T - R_k \cup Join_result(R_k))$$

where $Par_Join_cost(R_k)$ returns the cost of joining the k pairs of relations in the set R_k in parallel and $Seq_Join_cost(R)$ returns the cost of joining the relations in the set R sequentially. T is the working set, and initially contains all the base relations.

In the GP_T algorithm, the minimum *total* cost of a QEP is computed. Although the computation is simplified by considering only those QEPs that join k relations in parallel in the first step, it can still be expensive to compute such total costs at each iteration of the GP algorithm, especially as the number of joins increases. To further reduce computation overhead, the second version of GP, GP_P only estimates the *partial* cost of a plan, $Partial_k$, and uses this as the approximation of $Cost_{plan}$. $Partial_k$, the cost of a QEP which joins k pairs of relations in the first step in parallel, is represented by

$$Partial_k = Par_Join_cost(R_k)$$

Finally, we present the GP algorithm below. The function *Minimum_cost*, can employ either the total cost or partial cost metrics described above.

GP:

Input: A join graph $G = (V, E)$ where the set of vertices in V are relations and the edges in E represent joins (see Figures 7 and 8).

Output: S , the join sequence consisting of relation pairs

1. $S \leftarrow \emptyset$
2. while $Size(T) > 3$ do
3. $R \leftarrow Select_Rel_Pairs(G)$
4. $S \leftarrow S \cup R$
5. $G \leftarrow (G \text{ with each pair of relations in } R \text{ replaced by their join result})$
6. end
7. $R \leftarrow Two_Way_Sequential(G)$
8. $S \leftarrow S \cup R$

The function `Two_Way_Sequential` is called to determine the join sequence when the working set `T` contains less than four relations.

Select_Rel_Pairs:

Input: A join graph, `G`

Output: `R`, a set of relations pairs to be joined concurrently

1. `k = 0`
2. repeat
3. `k = k + 1`
4. `Ck = Minimum_Cost(G, k, Rk)`
5. if (`Rk` does not contain all relation in `G`) then
6. `Ck+1 = Minimum_Cost(G, k+1, Rk+1)`
7. until (`Ck+1 > Ck`) or (`Rk+1` contains all pairs in `G`)
8. if (`Ck+1 > Ck`) then
9. return `Rk`
10. else
11. return `Rk+1`

Function `Minimum_Cost` takes as input the reduced join graph `G`, and the number of relations to be joined concurrently in the first step, `k`. It returns the minimum cost plan that joins `k` pairs first and determines those `k` pairs of relations as well as join methods for each of the `k` joins. The two versions of GP, `GPT` and `GPP` differ in the `Minimum_Cost` function.

Simulated testing [CHEN92] of `GPT` and `GPP` on QEPs that joined no more than ten relations showed that both produce plans with cost no more than 120% of the optimal plans. Algorithm `GPT` outperforms `GPP` in most cases. However, the time to generate plans is longer, especially as the number of relations in the QEP is increased. Up to 90% of the results generated by `GPP` are 80% near `GPT`.

Resource Allocation

As discussed in the beginning of the **Search Strategies** section, optimization methods fall into one of two categories, one-phase and two-phase. With a two-phase approach, the second phase is resource allocation, which takes as its input an operator or join tree and divides it into a set of schedulable objects. Most of the work done in this area assumes that input to the resource allocation phase is an optimal join-tree. In [DeWI92], which discusses the Gamma Database Machine, only left-deep trees are considered, as

generated by sequential optimizers. This of course lacks the advantages gained in using bushy-trees for parallel execution.

The resource allocation methods proposed in [NIC93], [GAR96] and [TAN93] uses a tree decomposition method to divide a join tree in schedulable components. These methods differ in the way the join tree is decomposed and the degree of granularity. The first step in each of the algorithms is to somehow decompose the tree. In [NIC93] for example, this is performed by refining each node into a sub-tree of physical operator nodes, each of these nodes is one of the following atomic components:

1. Scan -scans over the entire relation
2. Sort - sorts the input
3. Hash - builds a hash table
4. Probe - uses input to probe a hash table, hits are the output
5. Merge – merges two sorted inputs
6. Nested-Loops – Compares each tuple of one relation against all tuples in another relation

A join-node in query tree becomes expanded based on the join method. For example, a join of R1 and R2 would become $\text{merge}(\text{sort}(\text{scan}(\text{R1})), \text{sort}(\text{scan}(\text{R2})))$.

Once we have an operator tree, the second step determines how a tree is divided into a forest of sub-trees, each of which is a schedulable component. The edges of the operator tree are labeled as either materialized or pipelined. The tree is then broken up, using a bfs-type algorithm, at each materialized edge. The tree is also broken up at pipelined edges when the edge connects a high-variability producer to a consumer. In such a case the cost of idle CPU time may outweigh the cost of materialization. Long pipelines may also be divided at pipelined edges to eliminate large propagation delays. Figures 11. and 12. illustrate the above process.

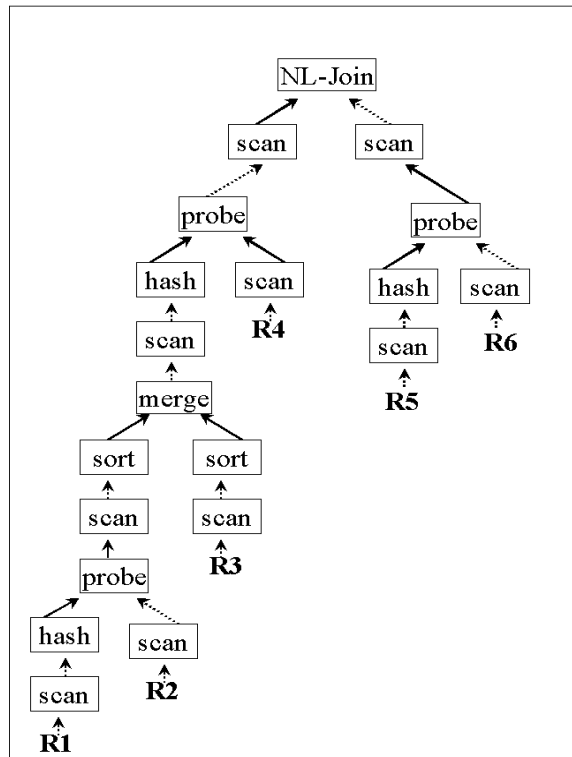


Figure 11. Operator tree with materialized edges shown as solid lines.

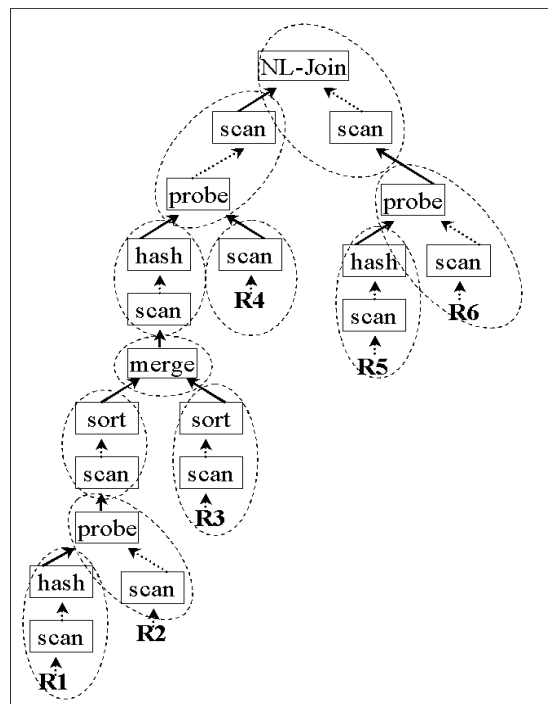


Figure 12. The tree is decomposed in schedulable components called query tasks.

Once the tree is decomposed we have set of query tasks which are essentially maximal sub-trees of the original operator tree containing only pipelined edges. The query task sub-trees demonstrate the three forms of parallelism as follows:

Intra-operator parallelism: Each query task is executed on a set of processors by appropriately partitioning its input data sets.

Inter-operator parallelism: Query tasks with no path between them are executed concurrently on disjoint sets of processors.

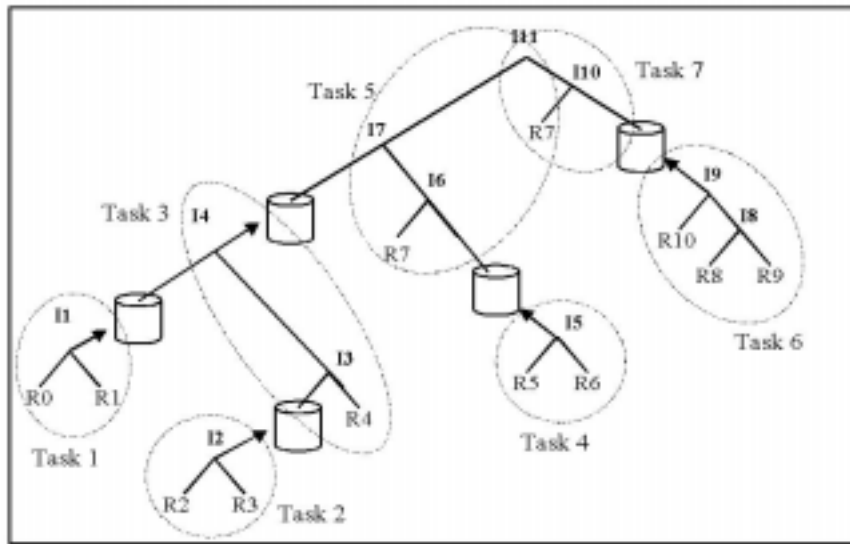
Pipelined parallelism: The atomic components within individual query task sub-trees are executed on a set of sites in a pipelined manner.

The next stage in resource allocation is to create a schedule by allocating resources to query tasks. In this section we describe several cost models for query task sub-trees and several scheduling algorithms.

The basic approach used in [TAN93] begins with a fixed number of *shelves*, each of which will be assigned one task initially. Each shelf will eventually contain one or more tasks that are to be executed concurrently by all processors. For each of the remaining tasks that have not been assigned a shelf, it is either assigned an existing shelf or a new shelf without violating the precedence constraints inherent in the QEP. When a task is added to an existing shelf it will be competing for resources with other tasks. As tasks are assigned, the total number of shelves may increase.

The initial number of shelves is determined by the height of the task tree since each task along the longest path must be processed one after the other due to precedence constraints. The join tree decomposition and task tree formulation steps are similar the method presented in [NIC93] but of a much coarser granularity. Tasks in this context are basically right-deep segments of a join tree.

When there are more tasks than the ones on the longest path, the task with the highest work estimate is selected. The work estimate of a task is the resource consumption of the task and the work estimate of a shelf is the total of the work estimates of tasks on that shelf. The three *shelf-scheduling* strategies we will discuss are *Max-Shelf*, *Min-Shelf* and *Flexi-Shelf* [NIC93], [TAN93]. In the *Max-Shelf* strategy the total number of shelves is maximized, hence each shelf contains a single task. Shelves are processed one at a time and there is no inter-operation parallelism. The *Min-Shelf* method is the other extreme, it produces the minimum number of shelves. Starting with as many shelves as tasks on the longest path, task are assigned to each of these shelves by adding a task to the shelf with the lowest work estimate. This strategy makes full use of inter-operation parallelism.



The *Flexi-Shelf* algorithm, as implied by its name, uses a more flexible method for assigning tasks to shelves. Again, the initial number of shelves are determined by the height of the task tree. A task t , is assigned to an occupied shelf (that satisfies precedence) with a set of tasks T , only if it is cheaper to process T and t in parallel than to process T and t sequentially. When there is more than one candidate shelf available then the one with the most gain over sequential execution is chosen. An illustration of these strategies is shown below (see Figures 13 - 15).

Figure 13. A join tree divided into tasks. Materialized intermediate relations are stored at each node represented by a cylinder.

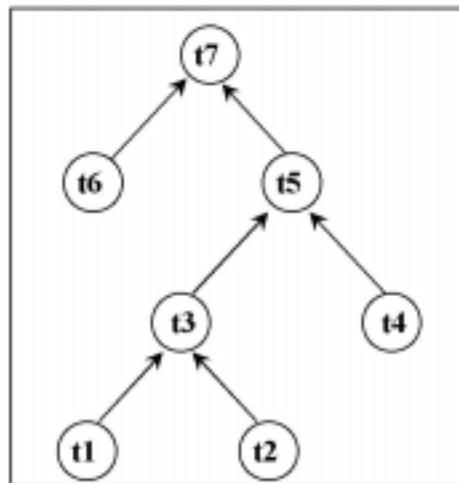


Figure 14. The corresponding task tree

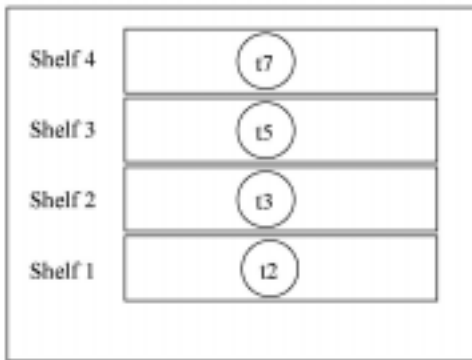


Figure 15(a) Initial schedule

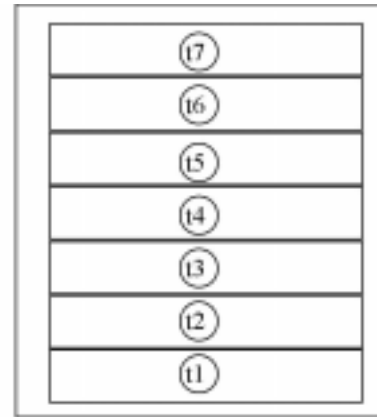


Figure 15(b) *Max-Shelf* schedule

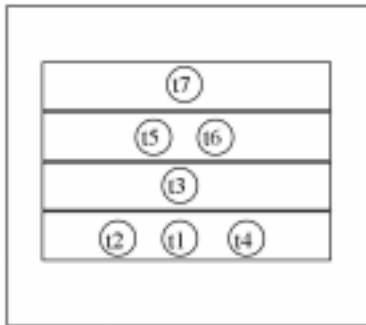


Figure 15(c) *Min-Shelf* schedule

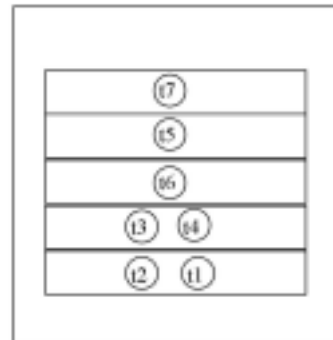


Figure 15(d) *Flex-Shelf* schedule

The *Max-Shelf* algorithm executes tasks serially. This performs well on systems with large memory, especially when pipelining can be employed for several relations that fit into memory. When memory is scarce or when relation and join results are very large, inter-operation parallelism may produce better results. The *Min-Shelf* employs as much inter-operator parallelism as possible; however, it does so without considering reduced resources, which can result in high processing costs. *Flexi-Shelf* is a hybrid of *Min-shelf* and *Max-Shelf* so it has each of their advantages but avoids their shortcomings and performs significantly better than the other two, both on large memory systems as well as those where memory is scarce.

The scheduling technique in [NIC93] also employs a *shelf* scheduling algorithm. The main difference is that the *critical path* is used in computing the number of shelves. In this method, no new shelves are added. Thus, this is a variation of the *Min-Shelf*.

Load Balancing

Skewed data can greatly impact the effectiveness of parallel query processing by creating unbalanced load during execution. Equally dividing the load among processors while minimizing overhead incurred by communication, synchronization and data transfer is the heart of the load balancing problem.

Load balancing strategies fall into one of two categories: static load balancing and dynamic load balancing. Static load balancing algorithms balance the load of processors at the time when tasks are allocated to processors. Once a task is scheduled for execution, it will not migrate from one set of processors to another. The allocation algorithms discussed thus far have been static. Dynamic load balancing algorithms distribute load across the system once execution has begun. Tasks and data can be shipped from overburdened processors to those with lighter load.

The first algorithm we consider is TIJ (Tuple Interleaving Parallel Hash Join) from [HUA91]. This algorithm achieves uniform bucket distribution among all nodes in SN system using a specialized hash-join. The TIJ algorithms has four phases outlined below.

1. Split Phase: The two relations involved in the join are hash partitioned. Each node then divides its local partition of R and S into p buckets where p is considerably larger than the number of nodes. During this process tuples belonging to each of these p buckets are spread among all nodes. The spreading is performed by *interleaving* consecutive tuples from each bucket among the nodes in the system. For each node, the i th tuple of a bucket is sent to the $((i-1) \bmod N) + 1$ th node, where N is the number of nodes in the system. This spreading strategy guarantees that each bucket is spread evenly among nodes and the N sub-buckets of each bucket formed in this way should all be uniform in size.
2. Bucket Tuning Phase: Based on its local distribution of sub-buckets a node can tune the size of its buckets to fit the memory capacity. All nodes have the same distribution of sub-buckets so one only must perform the tuning operation and then transmit the results to all other nodes.
3. Partition Tuning Phase: A coordinating node groups its buckets into N equal partitions. Each partition is allocated to a distinct node. The partition-to-node mapping information is

then broadcast to all nodes in the system. Upon receiving mapping information each node begins sending partitions to their appropriate nodes and receiving incoming partitions that belong to it.

4. Join Phase: Each node performs local joins of respectively matching buckets.

As can be seen from the way in which the algorithm operates, each node will be processing roughly the same number of tuples, hence the load should be relatively balanced. One major disadvantage in TIJ is the large amount of pre-processing and shipping overhead. When joining relations with little or no skew such a procedure would be a tremendous waste of resources.

The ASBJ (Adaptive Static Balanced Join) algorithm presented [LuTAN94] uses a similar yet more inventive method of partitioning work load. ASBJ may also be divided into phases as follows.

1. Task formation phase: Each node divides its local partition of R and S into *logical tasks* as follows: Local sub-buckets of the local R and S partition are stored back onto the local disk and statistics (the cardinality) for each bucket is recorded. Once all the local buckets are formed a coordinator node collects all the statistics and sorts the information in decreasing order of the smaller size of the S or R bucket. Each pair of buckets now make up a *logical task*. Next, for every task whose smaller bucket does not fit into memory, the task is split into k subtasks. Subtasks are formed using a fragment-and-replicate strategy which divides the smaller bucket into k equal sized buckets (whose hash tables fit into memory) and duplicates the larger bucket k times.

2. Load-balancing phase: A greedy algorithm is used to assign the tasks to the nodes to minimize the estimated completion time. The algorithm does this by allocating the most expensive of the remaining tasks to the node with least estimated processing time. At each node, unsplit tasks are sent to node dictated by the allocation algorithm. For the split tasks, the smaller bucket is distributed evenly (round robin) and the larger bucket is broadcast to the appropriate nodes.

3. Join Phase: Each node executes the join tasks which have been allocated to it.

Comparing TIJ and ASBJ we see that ASBJ, as its name implies, is more adaptive to varying skew conditions. The TIJ is good when the degree of skew is constantly mild while ASBJ should be used when there is high degree of skew. Both these algorithms require a high bandwidth network due to the large amount of intercommunication and data shipping that takes place prior to execution.

The problem with static load balancing techniques is that they incur a high pre-processing cost even when the data has little or no skew. Dynamic load balancing algorithms avoid this problem by distributing load once execution has begun. Any imbalance will be corrected during processing and operations with little or no data skew won't incur unnecessary overhead as can be the case with static algorithms.

The task generation and task allocation phases of dynamic load balancing algorithms are similar to those of static ones. However, the task execution phase is different. During task execution, each node maintains certain information about the task currently being processed, including the number of data pages remaining and the size of the result generated so far. Such information is necessary in order to make correct decisions regarding transfer of load across nodes. When a node completes processing all tasks assigned to it, it then makes a request for more load which can come from either one or several nodes depending on the algorithm. The overloaded node(s) and the amount of load to be transferred are then determined. The transfer takes place by shipping buckets from the donor nodes to the idle nodes. Depending on the algorithm, different strategies are used to determine the donor and the amount of data to be transferred. The process of load transfer takes place as follows:

1. The idle node sends a load-request message to the coordinating process.
2. At the coordinator node, requests are queued in a FIFO manner. The coordinator broadcasts a load-information-request message to all nodes.
3. Upon receiving the load-information-request from the coordinator, each node computes its current load and sends it to the coordinator.
4. Based on load information pertaining to each node, the coordinator determines a donor. If there are multiple requests in the queue, the coordinator could select k busy nodes as the donors of the first k requests.
5. The donor determines the load to be transferred and sends the load to the idle node.

As an example of implementing dynamic load balancing, we present the DLPJ (Dynamic and Load Balanced Parallel Join) algorithm discussed in detail in [LuTAN92] and [LuTAN94]. The task generation is *hash-based* as in conventional parallel join-algorithms, but the allocation of tasks to processors does not occur until execution begins.

The task generation phase is similar to those of TIJ and ASBJ discussed above. Each node scans its local partitions of relations R and S. Each node then allocates one output buffer for each of the B buckets, where B is the number of buckets desired (this is usually dependent on the amount of memory and having at least one bucket fit into memory). Each processor then hashes its local partition of R into the B buffers (buckets). The same

hash function is then used to create B buckets of the local S partition. Each processor i ($1 \leq i \leq p$) where p is number of processors, contains bucket j ($1 \leq j \leq B$) of partition R and S. Hence, we may address each bucket across all nodes as S_{ij} or R_{ij} . Therefore, a task is a pair of buckets with equal subscripts S_{ij} or R_{ij} .

Once partitioning is complete and execution has begun, we enter the load-balancing phase. When a node becomes idle, load is transferred using the 5-step protocol defined above. (We will discuss modification to these steps once we have completely defined DLPJ). When the idle node requests load, any working node is a candidate donor. Our goal is to lessen the burden of the most heavily burdened node, thus it is such a node that should be the donor. However it is difficult to provide an exact measure of load for a specific node. Hence DLPJ uses the *estimated completion time* (ECT) as the measure of a node's load. When a transfer request has been made by an idle node, the ECTs of all the busy nodes are computed and the node with the highest ECT will become the donor (ties are arbitrarily broken). Node i becomes the donor when

$$ECT_i = \max_{j=1}^p (EFT_j)$$

When there are k unprocessed tasks at node i , ECT_i at node i is given by

$$ECT_i = \sum_{j=1}^k EET_{ij} + EFT_i$$

The first component, $\sum_{j=1}^k EET_{ij}$, is the total *estimated execution time* (EET) of all the tasks at node i that are awaiting execution. EET is estimated based on the statistics collected during bucket partitioning and the cost formula of the local join method. The second component EFT_i , is the *estimated finishing time* (EFT) of the task that is currently being processed by node i , it is the estimated time required to complete the execution of the current task. The value of EFT is computed similarly to the way in which each EET is computed for each task, except that EFT uses the *remaining* size of each bucket (measured in pages).

Once the donor node has been chosen, it must decide how much data to transfer to the idle node. DLPJ employs the following heuristics:

1. Determine the Task: The transferred load is taken from the unprocessed tasks. The task with the lowest EET is chosen – this is done to avoid *thrashing*[‡]. If no unprocessed tasks exist at the donor node then load must be taken from the task currently being processed.

[‡] Thrashing happens when load is continually shipped back and forth between donors and idle nodes with very little processing occurring. This can happen if an idle node takes too much burden off a donor node so that the donor node very quickly becomes idle and requests more load.

2. Determine the amount of Load: The amount of load transferred must satisfy the following two constraints:

(1) *The amount of load transferred should provide a gain in the completion time of the join operation.*

The ECT of the donor node after the transfer must be less than the ECT of the donor node prior to the transfer,

(2) *After the load is transferred, the ECT of idle node should not exceed that of its donor.*

The first constraint described above ensures that the transfer will be worthwhile. The second constraint aims to minimize thrashing.

In comparing DLPJ to the two static load-balancing algorithms described above we monitor the effect of data skew, the effect of I/O bandwidth and the communication bandwidth. The level of data skew has little effect on the TIJ algorithm since the partitioning strategy is fixed. Unfortunately, high skew could cause this algorithm to perform poorly. A more adaptive strategy in ASBJ ensures more uniform bucket size and load distribution with a cost of more I/Os. Both algorithms have high pre-processing costs, even when the data contains little or no skew. A dynamic strategy like DLPJ uses demand driven task allocation to balance load. Hence, load balancing overhead is proportional to the level of skew. A problem that can arise with DLPJ is the possibility of very high communication once execution has begun. It is possible for DLPJ's performance to degrade to that of a static algorithm.

All three algorithms have scaling issues: Adding processors to DLPJ means more coordination and more transfers. Increasing the communication bandwidth will significantly relieve this effect. The static algorithms generate a large number of partitions to ensure uniform distribution. As processors are added, more buckets need to be generated, requiring more I/Os. Increasing the I/O bandwidth would lessen the cost.

Implementation

We have discussed the main issues affecting the operation of parallel database systems. We now consider a design process that analyzes these issues and investigates possible implementation.

Our first step is to choose an architecture among the possibilities of Shared-Memory (SM), Shared-Disk (SD), Shared-Nothing (SN) or some hybrid. One of the primary goals in the design process is to minimize cost. The lowest cost architecture would use little or

no specialized hardware and would be scaled up (or down) with a minimum of effort. SN has emerged as the winner, providing an easily scaled parallel system at a low cost.

We have seen that the SN architecture can run into problems when message exchange is the dominant portion of query processing. This can occur when using certain load-balancing strategies such as TIJ and ASBJ. The degree to which message exchange and coordination affects processing is highly dependent on the network architecture. A contention based network (i.e. Ethernet) as shown in Figure 16. would not be suited to static load balancing techniques like TIJ and ASBJ. DLPJ would be a better choice for the network architecture in Figure 16., though it too would degrade performance as the level of skew increased.

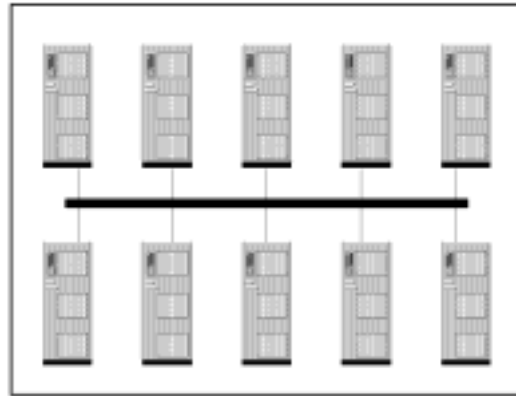


Figure 16. A shared-medium, contention based network (i.e. Ethernet)

To allow for high communication the network should employ a star configuration as shown in Figure 17. This type of setup ensures that only those nodes involved in communication are addressed. For example, load-balancing related communication between a donor node, an overloaded node and a coordinator node will not affect communication between other nodes, such as those involved in producer consumer relationship.

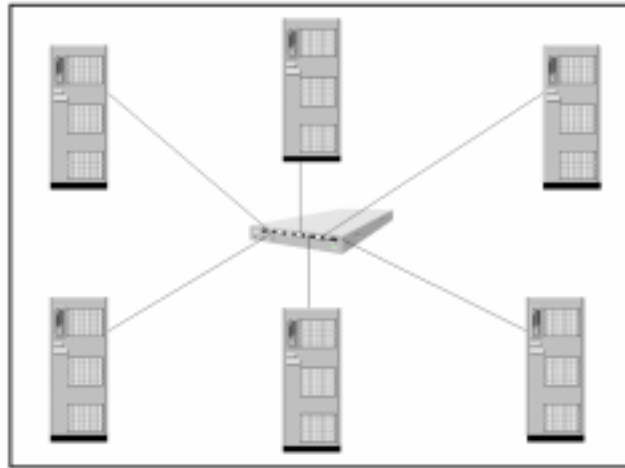


Figure 17. A star network. Each node is connected to a central switch or HUB which directs packets to and from appropriate nodes across the network.

Such a system can be scaled by adding more HUBs that divide processors into groups. Packets from one group to another are transferred by the HUBs which then distribute the packets amongst local nodes. See Figure 18. This configuration allows the system to be scaled without overloading a single HUB.

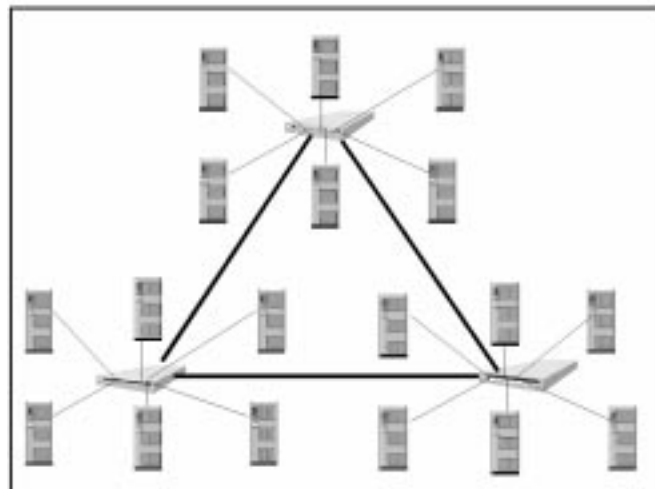


Figure 18. Three interconnected star networks.

The star network architecture is definitely advantageous for query processing algorithms that involve large amounts of message passing. Such a network also poses no major hindrance to a system that depends less on message passing and coordination.

With a star network, we have more flexibility in choosing a task allocation method. But before we do so we must first generate a join tree. Inter-operator, intra-operator and pipelined parallelism should all be exploited, thus we will dismiss the GP algorithm in

[LU91] since it divides QEPS into synchronized steps which ignore pipelines. Of the three algorithms GMC, GMR, and GME [CHEN92] we saw that GMR, which generated join trees based on *minimum resulting relation*, was closest to optimal. Thus, we shall employ GMR as our method for generating QEPs.

The next step in our design is to choose a method by which to decompose the join tree into schedulable components. First we must decide on the granularity with which we form tasks. Coarse granularity is better for a lower bandwidth system where little message passing is done. However, if we are using dynamic load balancing and demand driven task allocation then a finer granularity is better suited to ensuring balanced workload across all nodes. The tradeoff is more communication, most of this communication is among two nodes or a node and the coordinator. Such message interchange is handled gracefully by the star configuration described above.

Our final step is deciding how resources will be allocated. If data is highly skewed, we want to employ a dynamic load balancing algorithm. However, if we are fairly certain that data is evenly distributed and will continue to be so, then we can employ a proven static allocation technique such as *Flexi-Shelf*.

We have shown that many factors affect the choices made in implementing a SN parallel system. Tuning the system is just as important as initial design. As a database shrinks and grows it may have to be periodically repartitioned to ensure balance. Partition size for hash-based join algorithms must also be periodically adjusted until an optimal size is found. Such variable parameters are relatively easy to adjust once the database has been implemented, but other choices such as architecture, network topology and query processing algorithms are not.

Conclusion

In this paper we have discussed some of the major optimization issues currently being addressed in parallel, shared nothing databases. We have seen how each stage in optimizing a query is significantly affected by choice of search algorithm, resource allocation strategy and cost models. Problems caused by data skew were addressed by the three load-balancing techniques covered in the previous section.

We have seen how addressing these issues plays an important role in implementing a parallel database system. As it turns out there is no *best way* to implement a parallel DBMS. There are many different ways, each best suited to a given description of the data and the available hardware.

It seems that parallel database machines are the future, but with this future comes the challenge of devising strategies which fully exploit the potential of these systems. Rising to

this challenge we have attempted to bring into focus some the interesting research problems related to parallel database query optimization.

REFERENCES

[LU91]

H. Lu, M-C. Shan, K-L Tan, "Optimization of Multi-Way Join Queries for Parallel Execution" Proceedings of the 17th International Conference on Very Large Databases, Barcelona, September 1991

[DeWI92]

D. DeWitt, J. Gray, "Parallel Database Systems: The Future of Performance Database Systems" Communications of the ACM, June 1992, Vol.35, No. 6

[HAL97]

G. Hallmark, "Oracle Parallel Warehouse Server", IEEE 1997

[LAN94]

R.S.G. Lancelotte, et al "Industrial-Strength Parallel Query Optimization: Issues and Lessons", Information Systems Vol. 19 No.4, 1994

[GAR96]

M.N. Garofalakis, Y.E. Ioannidis, " Multi-Dimensional Resource Scheduling for Parallel Queries", SIGMOD '96, 6/96 Montreal, Canada

[GANG92]

S.Ganguly, W.Hasan, R. Krishnamurthy, "Query Optimization for Parallel Execution", ACM SIGMOD 6/92 California, USA

[TAN93]

K-L Tan, H. Lu, "On Resource Scheduling of Multi-Join Queries in Parallel Database Systems", Information Processing Letters 48,1993

[LuTAN94]

H. Lu, K-L Tan, "Load-Balanced Join Processing in Shared-Nothing Systems" Journal of Parallel and Distributed Computing 23, 382-398, 1994

[DEetal90]

D. DeWitt, et al, "The Gamma Database Machine Project" Computer Sciences Department, University of Wisconsin, 1990

[NIC93]

T.M Niccum, J. Srivastava, B. Himatsingka, J. Li, "A Tree-Decomposition Approach to the Parallel Execution of Relational Query Plans" Department of Computer Science, University of Minnesota, Technical Report TR93-016, 1993

[CHEN92]

Ming-Syan Chen, Philip S. Yu, Kun-Lung Wu,
"Scheduling and Processor Allocation For Parallel Execution of Multi-Join Queries" 8th International Conference on Data Engineering, 1992

[LuTAN92]

Hongjun Lu, Kian-Lee Tan, "Dynamic and Load-balanced Task-Oriented Database Query Processing in Parallel Systems" Thirst International conference on Extending Database Technology" 357-372, 1992

[HUA91]

Kien A. Hua, Chiang Lee, "Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning", 17th International Conference on Very Large Databases, 525-535, 1991