3Q－4

Pipeline Stage Based Dynamic Load Balancing for Right-Deep Multi-Joins

Stephen Davis Masaru Kitsuregawa
University of Tokyo, Institute of Industrial Science

1 Introduction

With ever increasing database sizes, parallel processing has become an important technology for improving over all query response time. However, this reduction in response time comes at the expense of the increased complexity required for distributing the work to be done in a query evenly amongst the processors.

This paper presents our dynamic load balancing algorithm for right-deep hash multi-joins executing on a shared nothing architecture, which makes use of a centralized processor to handle all load balancing decisions. The overall goal is to assure that each processor generates nearly the same number of multi-join result tuples, and this is achieved through the dynamic migration of build relation tuples between processors.

2 Right-Deep Hash Multi-Join

A multi-join is a relational join operation where more than two relations are being joined together. The right-deep hash multi-join is a multi-join making use of a right-deep query execution plan for determining the order in which the pair-wise hash joins are performed. This execution plan allows multiple pair-wise joins to be executing simultaneously through the use of pipelining. Each stage of the pipeline performs a pair-wise hash join, with the join result tuples generated at a stage being used to probe the next stage. In a hash join, the tuples of one relation, the build relation, are used to build an in memory hash table, and the tuples of the other, the probe relation, probe against the hash table for matches on the join criteria. In the right-deep hash join, the build relations are all base relations, the first stage of the pipeline's probe tuples are read from disk, and the final stage's result tuples are written to disk.

When the right-deep hash join is processed on a shared nothing architecture, each processor works on each pipeline stage, with each relation partitioned amongst the processors. Ideally, the relation partitioning is performed so that all processors finish join computation at approximately the same time, however determining such a partitioning can be too computationally expensive. As a result, repartitioning is done through heuristics making use of approximations about the relations, hence there are generally variations in the amount of work done by each processor. Also the time required to determine the best repartitioning is overhead since no join processing is being done. Thus, a dynamic load balancing mechanism can be used to reduce variations in the execution times of the processors, while also allowing for reductions in the time required for relation repartition scheduling.

3 Dynamic Load Balancing

The load balancing algorithm works to dynamically redistribute the number of tuples remaining to be produced as a result of the multi-join evenly amongst the processors. It is a centralized load balancing algorithm which makes use of a processor, called the *foreman*, devoted solely to making load balancing decisions. The foreman receives statistics about the multi-join from each of the join processors (JPs), and based on these statistics it determines how tuples stored in the JPs' in memory hash tables should be migrated in order to equalize the number of tuples remaining to be generated by all processors. Since the estimate of future work is based on statistics obtained during join processing, a basic assumption is that the attribute distributions seen so far reflects the distributions of the remaining tuples to be probed.

The initial statistics gathering begins when the first join processor has finished processing a fixed percentage of its probe tuples from disk. The processor which first reaches this threshold sends a message to the foreman, and the foreman then informs all of the JPs that statistics gathering is required. The JPs send the statistics, and continue with join processing. Based on these statistics, the foreman determines whether load balancing is required and if so, how the base relation tuples should be migrated. Once the foreman has generated the load migration plan, it sends an enter load balancing message to all join processors. The join processors respond by sending an acknowledgement to the foreman and stop performing join processing at this time. Once the acknowledgements are received, the foreman sends the load migration plan to each of the JPs. At this point, the JPs perform the necessary migrations and inform the foreman upon completion. After all completion messages are received, the foreman sends an end of load balancing message to the JPs which causes them to resume join processing. If load balancing is performed more than once, then subsequent balancings begin after a fixed interval of time has passed, instead of being triggered by probe tuple consumption.

When making the data migration decisions, the foreman examines each stage to estimate the number of join result tuples remaining to be produced by the stage. Based on this estimate, the foreman determines how the stage's build tuples should be migrated

between the processors in order to balance the remaining number of result tuples to be generated. The unit of migration is a hash line which consists of all the tuples of the stage's build relation that map to the same hash entry in a particular processor's hash table. Hash lines are migrated from highly loaded stage fragments to lightly loaded ones, where a stage fragment is the portion of the stage being processed on a particular processor. A highly loaded stage fragment is one generating more result tuples than the stage average, similarly a lightly loaded one is generating less than the average. A sufficient number of hash lines are migrated to equalize the expected number of result tuples generated by each processor. The tuple migration decisions are made independently for each stage, and since each stage is balanced amongst the processors, each of the processors will be balanced.

4 Estimation of Tuples Remaining to be Generated

In order to estimate the number of result tuples, ERT, remaining to be generated, each processor maintains statistics about the number of tuples probed, TP, and the number of tuples generated, TG, for each of the stage fragments. Assuming that the attribute distribution of the probe tuples probed so far is similar to that of the remaining probe tuples, ERT for a stage fragment is just TG/TP times the estimated number of tuples remaining to be probed at that fragment.

The estimated number of tuples remaining to be probed is determined using statistics about how the probe tuples are being redistributed amongst the processors for joining. For the first stage, a stage fragment's expected input is the total number of base probe tuples remaining times the percentage of the tuples probed so far which were probed at that stage fragment. Similarly for subsequent stages, the stage fragment input is estimated as the expected number of tuples remaining to be generated at the previous stage times the percentage of the already generated output it has consumed so far.

5 Hash line selection

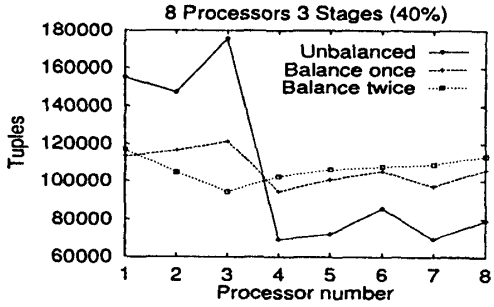
Hash lines are selected for migration so that after migration each stage fragment of a stage will produce roughly the same number of result tuples. Since the unit of load is the number of tuples remaining to be generated, a hash line is selected based on the estimated number of tuples remaining to be generated by that hash line. This estimate is formed by multiplying the expected number of tuples remaining to be generated by the stage fragment by the fraction of the stage fragment's result tuples generated by the hash line so far. In order to determine this percentage, each hash line maintains a count of the total number of result tuples generated by it.

During selection, the foreman determines the expected load of each hash line for each highly loaded stage fragment, sorts them by decreasing load, and assigns hash lines to lightly loaded stage fragments in a first fit decreasing manner. Selection starts by sending hash lines from the highest loaded stage fragments to the lowest loaded ones.

6 Experiments

The experiments were performed through simulation with the actual joins being performed. The relations were artificially generated so that the number of tuples generated by each stage fragment could be varied. The simulator measures the time required to complete the multi-join. Disk and network buffers are 8 kilobytes, a read or write from disk takes a minimum of 1024 time units (TUs), probing against a single build tuple takes three TUs, creating a result tuple takes 256 TUs, and transfers over the network take at least 1280 TUs. Each base relation has 240,000 tuples and the tuples are uniformly partitioned amongst the processors. The initial probe relation is stored to disk, and is repartitioned before join processing begins. The simulation measures only the time required to perform the join and does not include the time for building the hash tables, or partitioning the base relations. The first load balancing begins after a join processor processes 45% of its disk tuples. The second balancing occurs at the same time interval as the first.

The graph below shows the number of tuples generated at each processor for no load balancing, a single load balance, and balancing twice. The table shows the time required, the portion of that time caused by load balancing overhead, and the maximum number of tuples generated by a single processor.



	Time	LB Overhead	Max tuple
No balancing	114,729,197	0	175,55
Balance once	92,325,881	8,481,548	121,23
Balance twice	97,725,838	12,181,298	116,70

7 Conclusion

As can be seen, load balancing evens out the number of tuples generated by each processor, thus reducing execution time.

References

[1] D. Schneider and D.J. DeWitt. "Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines." Proceedings of the 16th International Conference on Very Large Data Bases, August 1990.