ELSEVIER

Parallel Computing 25 (1999) 2177–2188

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Methods for parallel execution of complex database queries[☆]

Andreas Reuter[*]

International University in Germany, gGmbH, 76646 Bruchsal, Germany

Received 2 April 1999; received in revised form 9 April 1999

Abstract

During the last decade, all commercial database systems have included features for parallel processing into their products. This development has been driven by the fact that databases grow in size at considerable rates. According to the results of the 1998 ‘very large database contest’ the world’s largest databases, which have reached a size of over 10TB, double in size every year. At that speed, they outgrow the increase in processor speed and memory size, so additional measures are required to accommodate the effects of rapidly growing volumes of data. Parallelism is one of those options. It helps to keep processing times constant, even if the size of the database increases. That effect, which is often referred to as ‘scaleup’ is important for loading, index creation, all kinds of administrative operations on the database, and of course for long batch-type applications. Parallelism is also employed to speed-up queries that otherwise would take days or weeks to process and thus would be useless for the application. This type of requirement: fast results of complex queries on large data sets is characteristic of decision support applications. In this overview we will explain how parallelism in databases can help to solve such problems. © 1999 Elsevier Science B.V. All rights reserved.

1. Introduction

Database systems have exploited parallelism from the very beginning (in the early 1960s), in a way similar to operating systems. An operating system allows many

[☆] This overview uses material from a survey paper by the same author that was published under the title ‘Parallel database techniques in decision support and data mining’ in the Proceedings of the 1997 ParCo Conference.

[*] Tel.: +49-(0)6221/533-100; fax: +49-(0)6221/533-298.
E-mail address: andreas.reuter@villa-bosch.de (A. Reuter)

PII: S 0 1 6 7 - 8 1 9 1 (9 9) 0 0 0 6 6 - 6

users to use the same hardware concurrently, i.e., in parallel, and a database system allows the same database to be concurrently accessed by many application programs. The basic idea is the same: each time an application is blocked because of a slow operation such as disk I/O, communication with a remote node, interaction with the user, the processor is re-dispatched to another application that is ready to execute. There is an important difference, though, between the way an operating system uses time sharing and what a multi-user database system does: whereas the operating system gives each user its own private resources (processes, address spaces, files, sessions, etc.), the database is a shared object. Different applications can access the same records or fields concurrently, and the problem of automatically maintaining correctness in such a parallel execution environment was one of the great challenges in database research in the 1970s. The problem was solved by the transaction concept with its sophisticated interplay of consistency checking, locking and recovery, and the resulting functionality proved an ideal basis for large distributed online information systems. The fact that all technical aspects related to locking and recovery can be completely hidden from the application makes modern SQL-style databases the first commercially used programming environments that automatically control parallelism – inter-query parallelism, that is.

For a long time, this was the only use of parallelism in databases – to the exception of a few special systems. However, in the 1980s users increasingly started developing applications that were radically different from classical online transactions: they ploughed through their databases (which, as was mentioned above, keep growing at a very high rate) in order to identify interesting patterns, trends, customer preferences, etc. – the kind of analysis that is called ‘decision support’, ‘data mining’, ‘online analytic processing’ or many other names. Those applications use databases in a different way than online applications do. An online transaction is short; it typically executes some 100k instructions, touches less than 100 objects in the database, and generates less than 10 messages. However, there are many concurrent transactions of the same kind, each transaction wants to experience short response time, and the system should run at high throughput rates. Parallelism in this case means scaleup: one wants to be able to do the same on larger databases, without affecting the response times of the online transactions.

Decision support applications, on the other hand, touch millions of objects, execute 10^{12} instructions or more, and they typically do not execute in parallel to each other or normal online transactions. Running such applications with reasonable elapsed times requires the use of parallelism in all parts of the database system, which has significant impact on the overall architecture of the database engine and its interfaces to the operating system.

2. Levels of parallelism in a database system

Parallelism can be employed in database systems at different levels, with granules of different sizes. The most relevant types of parallelism can be classified as follows:

- Inter-transaction parallelism,
- Parallelism among database operations inside a transaction,
- Parallel execution of a single database operation,
- Parallel access to the stored data.

This list illustrates that besides data parallelism at the lower layers, database systems also have a potential for function parallelism.

Let us now consider each level of parallelism in turn.

2.1. Inter-transaction parallelism

This is the concurrent execution of different online transactions mentioned in Section 1. Since they operate on the same (shared) database, concurrent read/write access to the same data element must be mediated automatically by the database system in order to keep all aspects of parallelism transparent to the application program, which is a sequential program operating under the assumption of unrestricted access to a single-user database. This transparent handling of parallelism with certain formal consistency guarantees (serializability) is based on the transaction concept [2,8], which gives all executions the so-called ACID properties (atomicity, consistency, isolation, durability). It guarantees that the parallel execution of n transactions on a shared database will produce a result that is identical to the result of at least one of the $n!$ possible serial execution sequences of those transactions. In other words: from each transaction's perspective the system behaves like a single user system; the application will not experience any anomalies caused by parallel processing. This technique was essential for the wide-spread use of multi-user databases, but it is efficient only if transactions are short and small. Analytic models indicate that the probability of deadlock among concurrent transactions increases with the 4th power of transaction size [8], so transaction oriented parallelism in its simplest form breaks down in an environment of long computations – such as data mining. In large online systems, the degree of parallelism of this type, often called multi-programming level, is in the order of 100. This implies that there are at least as many tasks or processes, because at any point in time, an active transaction needs a process in which to execute.

Since this type of parallelism is only marginally relevant to decision support applications, we shall not discuss it in detail.

2.2. Parallel execution of database operations inside a transaction

Each transaction is a sequence of statements written in some host language, which can be ignored for our purposes, and of statements of some database access language such as SQL [16]. The example in the following table assumes an order entry application: a new order is entered from some terminal, and the transaction that processes the input has to update all the relations in the database affected by that order. Quite obviously, there are no data dependencies among these statements, so in principle they could be executed in parallel.

```
update customer_relation      /* customer has placed order */
update inventory_relation     /* reduce quantity on hand for item */
update sales_statistics       /* compute totals, averages etc. */
update sales_rep_relation     /* increase total sales for person */
insert back_order             /* if QOH too low: back order */
```

Since SQL is a non-procedural language, the detection of data dependencies (or the lack thereof) is straightforward, so database systems could easily exploit this type of parallelism. However, it is not supported in commercial systems, for reasons beyond the scope of this paper.

We will not discuss this type of parallelism either, because it is not what is required in a decision support application. Decision support is characterized by the need to execute a single SQL operator, the select statement, which has very complex parameters, nested sub-selects etc. Fig. 1 shows a sample query, which is taken from the so-called TPC-D benchmark [10].[1]

2.3. Parallel implementation of database operations

Syntactically, the query shown in Fig. 1 is a single SQL-operation, a select. SQL being a declarative language, it requires a compiler that translates an SQL statement into a procedural program (plan) implementing that statement by using lower-level operators such as file scan, index scan, sorting, etc. The compiler creates an abstract representation of the plan, which is the starting point for the optimizer, which turns it into a set of specific plans, which the optimizer then evaluates, trying to pick the most efficient implementation.

In that process of compilation and optimization, data dependencies have to be analyzed anyway, and it is a natural generalization to exploit that information in order to derive plans describing parallel execution. Consider the following simple statement:

```
select u,v,w
from X,Y,Z
where X.a = const1 and
      Y.b = const2 and
      Z.c = const3 and
      X.k = Y.k and
      Y.m = Z.m;
```

We have to join three relations X, Y, and Z and have to do restrictions on each of them. For simplicity, let us assume restrictions and joins are 'elementary' operators. The compiler will then generate something like the operator tree shown in Fig. 2.

This obviously is a data flow graph, which can be processed in parallel according to the rules in [3]. All restrict operators can be started in parallel, although the one

[1] For current information on TPC benchmarks, see their web site at: www.tpc.org.

```
select P_partkey, P_mfgr, S_name, S_acctbal, S_address,
      S_phone, S_comment
from parts P, suppliers S, partsupp PS
where P_partkey=PS_partkey and PS_suppkey and P_size=15 and
      P_type="BRASS" and  S_nation= :nation  and
      PS_supplycost = (select MIN(PS_supplycost)
                       from partsupp PS1, suppliers S1
                       where P.P_partkey=PS1.PS_partkey and
                             PS1.PS_suppkey=S1.S_suppkey an
                             S1.S_nation= :nation);
```

Fig. 1. Example of a complex query: Q2 from the TPC-D benchmark.

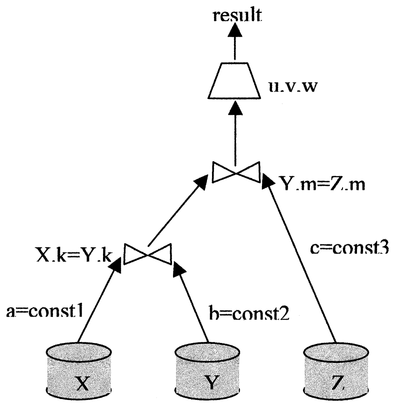

Fig. 2. Operator tree for the sample query.

on relation *Z* will stop soon, because the join operator for relation *Z* and the result of the join between *X* and *Y* is not ready to accept input records. This will happen only after the preceding join operator starts producing the first result tuples.

So at this level, there is the potential for data flow parallelism, which in reality will be more complex than what is shown here. For example, we have assumed the join operator to be a basic operation, which of course it is not. It will be implemented using more primitive operations such as sorts, hash transformations etc. This will introduce additional intermediate steps, additional dependencies – and additional potential for parallelism.

Depending on how complex the query is, how many joins and sub-queries there are, the actual degree of parallelism at that level is in the order of 10–100.

2.4. Data parallel execution of primitive operators

The lowest level according to our initial classification exploits data parallelism, which is the most 'natural' one for database systems. Again, SQL lends itself easily to data parallelism, because it is a set-oriented language. The SQL operators always have tuple sets as operands, and the operator is applied in a strictly uniform way to all tuples in the set. So by definition, there are no data dependencies between the tuples of one relation with respect to the execution of SQL-operator, which means the operator can be applied in parallel to each tuple in the operand set. From that perspective, the maximum degree of parallelism is limited by the number of tuples in the database – which usually is large in data mining applications (10^6–10^8).

In reality, parallelism will not be scheduled at tuple granularity. The relation would rather be partitioned such that each physical or logical processor has one partition to work on. This assumes that the individual processors can actually access the data without interfering with each other. This in turn depends on how many disk drives are used for the database, or how many nodes comprise the parallel system.

As a simple example consider the scan operator, which is needed for the restrict operation. It simply reads the tuples of a relation (the records of a file) in some order and passes them on to the next operator. At the tuple interface, a database system achieves a throughput of ca. 1 MB/s for that operator. Scanning a database of 1 TB in this way (which is a fairly typical size for decision support applications) would take almost 12 days, which is clearly unacceptable. To get this down to less than 1 h, one needs 1000-fold parallelism and thus the same number of data partitions, such as disks, extents, files, etc.

Note that this means more than just having at least 1000 discs. I/O parallelism is only one component of the problem. Driven at their maximum speed with sequential I/O, a disk can deliver data at a rate on the order of 10 MB/s. If 1000 such disks work in parallel, further processing has to happen at a rate of 10 GB/s, which means two things: processor parallelism and efficient reduction of data before the next steps of processing.

A hardware-level implementation of the disk part of this scheme is what has come to be known as RAIDs [7,15].

3. Parallel algorithms for special database operators

Let us now sketch the methods for executing two important operations in parallel. One is the join, and the other one is the creation of an index on a relation for a group of attributes. We assume that all relations involved are partitioned into n files and that there is one processor per file.[2] It is assumed that the reader is at least somewhat familiar with the two operations under consideration.

3.1. Parallel join

There are various ways of joining two relations in parallel; we will discuss two that are particularly important for commercial systems.

[2] This is a simplification we will give up later on.

3.1.1. Parallel join by sorting

An obvious way of joining two relations is the following: first sort both relations by the join attribute(s) and then merge the temporary results. The joined tuples will be produced in the order of the join attribute(s). With respect to a parallel implementation, two cases must be distinguished:

- Both relations are partitioned along the join attribute(s): This means both relations are (at the partition level) sorted by the join attribute, so producing the join simply requires the parallel merge of all corresponding partitions.
- At least one relation is not partitioned along the join attribute(s): in that case a sort phase must precede the merge. First, all partitions can be sorted in parallel. The temporary result has to be re-partitioned. While sorting, one can collect statistics on the value distributions in order to get partitions of equal sizes (load leveling). Repartitioning can also be done in parallel, and the last step is the parallel merge.

3.1.2. Parallel hash join

As was discussed in the previous section, relations that are not partitioned according to the join criterion need to be re-partitioned, and the resulting partitions should be roughly equal in size. It turns out that hashing is a good way of doing this.

Let us assume there are n processors. The left table R is stored in r partitions, the right table S is stored in s partitions. The join criterion is: $R.k = S.f$. Both k and f are defined over domain $\boldsymbol{K}$. Now let us consider a (good) hash function that is defined as:

$$h : \boldsymbol{K} \rightarrow \{1, \ldots, n\}.$$

There are no assumptions with respect to the partitioning or sort order of the left and right table. Fig. 3 illustrates the basic idea of (parallel) hash joins.

The join is performed in two phases.

Building phase: All r partitions of R are read in parallel. For each tuple, $h(k)$ is computed. The result is the number of the processor that is responsible for joining the tuple, so it is sent to that processor. In the above figure, we assume $n = 5$ processors. The hash functions defines a (dynamic) partitioning of the tuples of the left

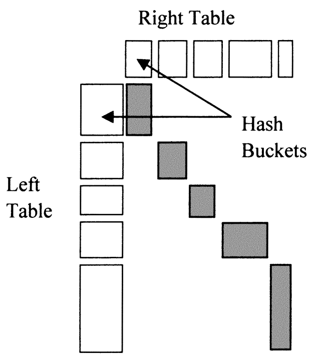

Fig. 3. Hash joins: the basic idea.

table, R. The shaded rectangles represent the processors that receive the tuples from both tables that fall into the same hash class. When all the tuples have been re-mapped, the building phase is complete.

Probing phase: Now the tuples of S are mapped using the same hash function. As soon as S-tuples arrive at processor i, this processor can start finding the matching tuple(s) of R. This can be done with indexes, via sorting, or by applying another, local hash function. Of course, all n processors work on matching their assigned tuples in parallel.

If the results of the join are the input for a subsequent join, it is easy to see that the probing phase of the first join coincides with the building phase of the second join.

Hash joins can be used for equi-joins (including outer joins) only, but those represent the vast majority of all joins in real system – particularly those in decision support systems. There is a problem in cases where the joins attribute has a highly skewed value distribution, because then some processors will receive large numbers of tuples, while other processors will idle. For more information on that see [4,19].

3.2. Parallel index creation

Since an index must be consistent with the base relation at any point in time, the traditional implementation of index creation operations locks the base relation against updates while the index is being created. Now for large relations creating an index can take many hours, so locking the relation is not acceptable.

Here is a sketch of how to do it in parallel to normal processing: at the beginning, the start of index creation is recorded in the database log. Then the index is created by sorting all partitions in parallel (index creation basically means sorting attribute values). The base relation is not locked, so new tuples can be inserted, other tuples can be modified or deleted. After this first step of index creation is completed, the index and the base relation are obviously not consistent with each other. The difference is caused by all tuple modifications since the index creation started. Now this set of tuples is much smaller than the total relation, so the following trick is quite feasible: lock the relation against updates, process the log from the point marking the start of index creation, apply all tuple modifications to the index. This can be done using parallelism at the tuple level (with some extra considerations). At the end of this pass over the log, index and base relation are consistent. Now the lock on the relation can be removed, and the index can be used for creating query plans.

4. Parallel database architectures

Historically, the database community distinguished three types of system architecture as platforms for parallel database systems:

Shared nothing: Each processor has its own memory and its own peripherals; communication is through messages only. Examples are: Tandem, NCR, workstation clusters, IBM SP2. This design has great fault isolation and scales well. The problem is the communication bottleneck.

Shared disk: Each processor has its own memory, but it shares the disk subsystem with other processors. In addition, there is a communication network. Example are: VAX Cluster, Oracle, IBM's IRLM. This design allows for an easy migration from a single node configuration, it offers fairly good fault isolation, and it goes well with electronic disks for performance improvement.

Shared everything: This is what has come to be known as an SMP (symmetric multiprocessor). Memory and peripherals are shared among many processors. Examples are: Convex, IBM DB2 UDB, Sequent, Sequoia, SGI, Sun. The advantages are easy programming and load balancing. The problem is potentially low fault isolation because of the shared memory.

Since shared disk and SMP architectures are very similar in many respects, the comparison between shared nothing systems (mostly referred to as 'massively parallel processing' (MPP-architecture) on one hand and SMP systems on the other hand represent the architectural alternative for modern parallel database systems. Let us briefly compare them from the perspective of handling parallel database operators.

The properties of *MPP-architectures* can be characterized as follows:
- Such systems scale to hundreds and, maybe, thousands of processors.
- No data are shared among processors.
- Data living on different processors must be combined using messages.
- Data in the database are partitioned statically.

The properties of *SMP-architectures* can be characterized as follows:
- Such systems scale to tens, maybe hundreds of processors.
- Data are shared among all processors.
- Data are combined via shared memory.
- Data can be partitioned among processors dynamically, depending on the load situation.

The main disadvantage of MPP systems for parallel databases is the necessity of static partitioning. This means it is difficult to vary the number of partitions, it is hard to size the partitions such that they are approximately equal for a wide range of operations, and there is basically no way to exercise control over difficult value distributions. As an example, consider hash joins in Section 3. Re-partitioning the tuples based on the hash function in an MPP-system means that each tuple is sent via messages to the processor that represent the respective hash class.

Dynamic partitions, which can be created in SMP-architectures, allow to assign data to processors on demand, one can vary the number of logical processors (tasks) at run-time, and heavily skewed value distributions have the only effect of not creating the same load for each task. Hash joins in such a system are performed such that the shared memory is partitioned into n hash classes, and tuples are simply put into the partition corresponding to their hash class.

So SMP systems are more flexible in terms of load balancing, the number of options for parallel algorithms is higher because of the possibility of data sharing, and the need for reorganization is less than with MPP systems. Manageability is also easier for SMP systems. The big advantage of MPP systems, on the other hand, is their scalability over wide ranges of processor numbers.

5. A typical DB-operation for decision support

To illustrate the way parallelism is employed for typical, complex decision support operations, let us consider the so-called star join [14].

The best way to introduce the problem is a simple example. Assume chain-store application, for which we have the following relations:

orders (orderID, custID, artID, storeID, quantity, date)
customer (custID, name, type, industry, size, region)
article (artID, name, price, weight, category)
store (storeID, name, city, region, size)

The 'orders' table is called the fact stable and typically is very large, because it contains a tuple for each sales transaction. The other tables are called the dimension tables.

A typical decision support query tries to analyze the order from customers in a certain industry and a specific region, concerning articles in a certain category, sold via stores of specific size range. Translated into SQL-terms that means we have to apply a large number of restrictions to the dimension tables and then join all of them to the facts table. This pattern: restrictions on a number of dimension tables and subsequent join with the large facts table, is called a 'star join'. The metaphor is that the fact stable is at the center, and the dimension tables surround it. In order to appreciate the way of processing such queries in parallel, we have to make the following assumptions:

- The facts table is very large compared to the dimension tables.
- There is an index on the key attribute and each of the secondary key attributes of the facts table.
- There is an index on the key attribute of each dimension table.

In addition, we assume an SMP system with shared memory for simplicity.

The *first* step of the algorithm assigns *t* parallel tasks to the first dimension table, let us assume: 'customer'. Each task processes a partition of 'customer', doing the following:

- Check the restriction condition on the tuple. If the tuple is not qualified, proceed to the next tuple. If it is qualified:
- Use the index for custID on the fact stable. Each orderID gets encoded in a bit filter that is kept in main memory.

The *second* step is started after the first one is completed. It basically does the same as the first one for the second dimension table, 'article'. But for each orderID that is retrieved during the index scan on the fact table, it is checked whether this orderID is encoded in the bit filter created by step 1. If not, the orderID is dropped. If so, it is encoded in a new bit filter in shared memory.

The *third* step is started after the second one is completed. It proceeds like the second step, this time using the third dimension table, 'store'. However, the output of this step is not a bit filter, but a list of orderID values, i.e., the keys of those tuples in the facts table for which the joins with the dimension tables have actually to be performed.

These joins are then performed using any of the algorithms described earlier.

The important observation here is that before starting the actual parallel joins, three preparatory steps (using moderate parallelism in each of them) are executed. The purpose of this is to reduce the number of tuples of the large facts table for which the expensive join operation has to be executed.

This principle will be found over and over again in all types of parallel query plans in database systems.

6. Summary

Decision support applications push the performance limits of today's database systems. There are many queries users would like to run routinely that take a couple of hundred hours to complete. If somebody manages to reduce the elapsed time for these problems to a few minutes, i.e., gain two orders of magnitude, he will have a big advantage over his competitors. Parallelism is the obvious way towards this goal. Databases lend themselves naturally towards massive data parallelism. In principle, one could employ a processor per tuple – and the larger the problem, the more tuples there are. Thus the scalability over wide ranges is feasible. The question of what exactly the best overall architecture looks like is not yet answered. All vendors are working on this problem, and it is possible that more than one will arrive at the right answer at about the same time. Then none of them will have an advantage over the competition, but the users will have a big advantage over their previous situation.

7. For further reading

[1,5,6,9,11–13,17,18].

References

[1] W. Becker, Dynamische Lastbalancierung im HiCon-System, Stuttgart University, IPVR, Technischer Bericht, 1994 (in German).

[2] P.A. Bernstein, V. Hadzilacos, N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley, Reading, MA, 1987.

[3] T. Bräunl, Parallel Programming – An Introduction, Prentice-Hall, Englewood Cliffs, NJ, 1993.

[4] D.J. DeWitt, R.H. Gerber, Multiprocessor hash-based join algorithms, in: Proceedings of the 11th International Conference on VLDB, 1985.

[5] D.L. DeWitt, et al., A single user evaluation of the Gamma Database Machine, University of Wisconsin-Madison, Computer Science Technical Report, Report No. 712, 1987.

[6] M. Sherman, Architecture of the Encina distributed transaction processing family, in: Proceedings of the ACM SIGMOD Conference, 1993.

[7] J. Gray, B. Horst, M. Walker, Parity striping of disc arrays: low-cost reliable storag with acceptable throughput, in: Proceedings of the 16th International Conference on VLDB, 1990.

[8] J. Gray, A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, San Mateo, CA, 1992.

[9] D.J. Dewitt, J. Gray, Parallel database systems the future of high performance database systems, Communications of the ACM 35 (6) (1992).
[10] J. Gray, The Benchmark Handbook for Database and Transaction Processing Systems, Morgan Kaufmann, San Mateo, CA, 1993.
[11] J. Gray et al., Loading databases using dataflow parallelism, Sigmod Record 23(4) 1994.
[12] T. Härder, E. Rahm, Mehrrechner-Datenbanksysteme für Transaktionssysteme hoher Leistungsfähigkeit, in: Informationstechnik it 28(4) 1986 (in German).
[13] T. Härder, E. Rahm, Hochleistungs-Datenbanksysteme – Vergleich und Bewertung aktueller Architekturen und ihrer Implementierung, Informationstechnik it 29(3) (1987) (in German).
[14] B. Lindsay, SMP Intra-query parallelism in DB2 UDB, IDUG'98, Amsterdam, 1998.
[15] D.A. Patterson, G. Gibson, R. Katz, A case for redundant arrays of inexpensive disks (RAID), in: Proceedings of the ACM SIGMOD Conference, 1988.
[16] J. Melton, A. Simon, Understanding the New SQL: A Complete Guide, Morgan Kaufmann, San Mateo, CA, 1988.
[17] M. Stonebraker, The case for shared nothing, IEEE Database Engineering 9 (1) (1986).
[18] H.-J. Zeller, Parallel query execution in NonStop SQL, in: Proceedings of the Spring CompCon Conference, San Francisco, 1990.
[19] H.-J. Zeller, Adaptive Hash-Join-Algorithmen, Ph.D. thesis, University of Stuttgart, 1991.