Bulletin of the Technical Committee on

Data Engineering

June, 1997 Vol. 20 No. 2 **IEEE Computer Society**

Letters

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems.

Letter from the Editor-in-Chief

Technical Committee Election

The Technical Committee on Data Engineering will have an election to select a new TC Chair. Our current Chair, Rakesh Agrawal, has completed his term. I urge current members of the ICDE to help select a new TC chair by participating in the nomination process and subsequently the election. (Note, however, that you must be a IEEE Computer Society and TCDE member to participate.)

Amit Sheth is chairing the nominating committee for this election and is joined by Nick Cercone and Ron Sacks-Davis, all currently serving on the TCDE Executive Committee. Nominations close on August 30, 1997 so you must act promptly to participate. The election will be held this fall. The new TC chair, upon election, will appoint a new executive committee.

A nomination form is included in this issue of the bulletin on page 44. You may use a paper copy of this form for your nominations, and send it to Amit Sheth at the address listed on the form. Members can also send in nominations electronically via email. An electronic nomination form has been posted to the Data Engineering Bulletin web page(http://www.research.microsoft.com/research/db/debull) for this purpose. This form can be sent via email to Amit at `amit@cs.uga.edu`.

This Issue

Data mining, data warehouses, decision support, these are the terms that our research community uses to describe the quest to query ever larger databases. As fast as our new hardware is, the drive for useable information constantly pushes the query processing envelop toward the need to exploit parallelism.

The current issue is on parallelism in industrial database systems. Betty Salzberg has ”gone the extra mile” in soliciting these papers. Indeed, she has succeeded in gathering papers on parallelism in five systems. This is no small accomplishment given the deadline pressures and implementation focus of development engineers working in industrial settings. This issue is particularly valuable because it exposes what industrial systems are currently exploiting and casts some light on what their development engineers think is important and doable. I think readers should find this extremely useful and I thank Betty for making it happen.

David Lomet
Editor-in-Chief

Letter from the Special Issue Editor

In this issue, we have descriptions of five commercial parallel DBMSs. Four of these papers describe software developed for shared-nothing systems. The fifth describes a shared-memory implementation.

The first paper, “Born to be Parallel” by Carrie Ballinger and Ron Fryer of Teradata, describes a system of virtual processors or *VPROC*s. VPROCs may coexist on a single SMP node and many SMP nodes working together form a single MPP system. Teradata partitions tables by hash functions across all VPROCs in the system. It is able to perform steps of query processing in parallel on different partitions of data, to pipeline operations within a complex step and to do unrelated steps of a query in parallel. Teradata’s query optimizer uses knowledge of the location of data and estimates of number of relevant rows at each VPROC to make query plans.

The second paper, “Parallel Solutions in ClustRa” by Svein Erik Bratsberg, Svein-Olaf Hvassovd and Oystein Torbjornsen, describes a replicated parallel system developed at Telenor, the Norwegian Telephone company. Nodes are grouped into two sites. Each site has a full replica of the database. Primary fragments of tables and hot stand-by fragments of tables are stored at nodes on different sites. A given node can contain primary fragments for some tables and hot stand-by fragments for another. The paper describes how the data is declustered and how the multiple log channels enable parallel updating of different fragments. Take-over, self-repair, backup, and scaling are also covered.

The third paper, “XPS: A High Performance Parallel Database Server,” by Chendong Zou, describes the parallel database server from Informix. XPS also uses a shared-nothing architecture. Data can be partitioned by round-robin, hash or user-defined expressions (e.g. key ranges). Indexes can be partitioned by the same or by different criteria as used for their base tables. Both horizontal parallelism (different parts of the data) and vertical parallelism (different parts of the query) are supported.

The fourth paper, “Query Optimization in DB2 Parallel Edition,” is by Anant Jhingran, Timothy Malkemus and Sriram Padmanabhan. DB2 Parallel Edition (DB2 PE) uses a shared-nothing architecture and function shipping. Most of the work is done by slave tasks executing at the nodes where the table exist. DB2 PE has the notion of a *nodegroup* that specifies which nodes of the system are used for a given table. Two tables may use the same nodegroup. *Collocation* of fragments forming joins is encouraged and is recognized by the query optimizer. Collocated, directed, broadcast and repartitioned joins are possible. The paper describes the cost-based optimizer strategies. An interesting example of processing of a correlated subquery is included.

The last paper, “Parallel Processing Capabilities of Sybase Adaptive Server Enterprise 11.5,” by Eugene Ding, Lucien Domino, Ganesan Gopal, Sethu Meenakshisundaram and T. K. Rengarajan, describes Sybase’s shared-memory parallel product. One important feature of this product is its use of asynchronous read calls provided by the operating system. Multiple disks can be kept busy providing parallel I/O. Multiple threads can scan tables in parallel. This is used in join processing, sorting, and index creation. A parallel database consistency checker is also described which detects violations in consistency in only one pass over the database.

Perhaps what is most interesting is what is *not* in this issue. There is no mention of semi-joins. Several papers suggest that it is a bad idea to use a query plan for a centralized DBMS and “parallelize” it. Hash partitioning seems to be in great favor, whereas using key ranges is discouraged because of poor load-balancing. An emphasis is placed on putting tuples that are likely to be joined on the same node, usually by choosing the join key as the partitioning key.

We have articles here that illustrate the state-of-the-art in parallel DBMSs. We have descriptions of advanced algorithms for shared-memory and shared-nothing systems. We have an example of a highly parallel replicated-data system. I would like to thank all the authors for their efforts on our behalf.

Betty Salzberg
Northeastern University

Born To Be Parallel
Why Parallel Origins Give Teradata an Enduring Performance Edge

Carrie Ballinger and Ron Fryer
NCR/Teradata

Twenty Questions to Ask About Parallel Databases

1. Are some query operations not parallelized?
2. Is parallelism always invoked for every query?
3. How many times does a query undergo optimization?
4. Does the amount of parallelism vary between queries, within queries?
5. Exactly how are the units of parallelism coordinated?
6. How does the data partitioning offered ensure a balanced workload?
7. What effect does growth have on this partitioning?
8. What techniques ensure multiple users can successfully execute?
9. Under what conditions must data or messages traverse the interconnect?
10. Are there any techniques to conserve interconnect bandwidth?
11. How is locking maintained across multiple units of parallelism?
12. Does the optimizer know how many units of parallelism will be used?
13. Does the optimizer cost joins differently in an SMP and MPP configuration?
14. What happens to other units when one unit of parallelism aborts a query?
15. Is the optimizer sensitive to the data partitioning scheme in use?
16. Do the units of parallelism recognize and cooperate with each other?
17. Is the degree of parallelism reduced when the system gets busy?
18. How is transaction logging parallelized across nodes?
19. Are multiple code bases (parallel and non-parallel) being maintained?
20. What are the single points of control and potential bottlenecks within the system?

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Introduction

Parallel processing has become both fashionable and necessary as data warehouse demand continues to expand to higher volumes, greater numbers of users and more applications. Parallelism is everywhere. It has been added underneath, layered on top, and re-engineered into almost all existing data base management systems, and in the process has left some in the user community confused about what it means to be a parallel database. Teradata stands alone in this chaos as a product consciously designed from the base up to be a parallel system for decision support.

This paper reveals some of the techniques architected into the Teradata product that have allowed parallelism in its fullest form to blossom. Being born for parallelism is the single most important fact in establishing Teradata's dominance and success in the data warehouse market.

1 Query Parallelism

What is Query Parallelism?

Executing a single SQL statement in parallel means breaking the request into smaller components, all components being worked on at the same time, with one single answer delivered. Parallel execution can incorporate all or part of the operations within a query, and can significantly reduce the response time of an SQL statement, particularly if the query reads and analyzes a large amount of data.

With a "Just say no!" attitude toward single-threaded operations, designers of Teradata parallelized everything, from the entry of SQL statements to the smallest detail of their execution. The database product's entire foundation was constructed around the idea of giving each component in the system many sister-like counterparts. Not knowing where the future bottlenecks might spring up, developers weeded out all possible single points of control and effectively eliminated the conditions that breed gridlock in a system. From system configuration time forward all queries, data loads, backups, index builds, in fact everything that happens in a Teradata system, is shared across those pre-defined number of VPROCs. The parallelism is total, predictable, and stable.

Teradata's Unit of Parallelism

eradata's unit of parallelism is referred to in this paper as a VPROC. In Teradata V1 these VPROCs were physical uni-processors known as "AMPs." With Teradata V2 they became "virtual AMPs" or "virtual processors" with many co-existing on a single SMP node as a collection of UNIX processes and many SMP nodes working together to form a single MPP system.

Teradata's Dimensions of Query Parallelism

While the VPROC is the fundamental unit of apportionment, and delivers basic query parallelism to all work in the system, there are two additional parallel dimensions woven into the Teradata DBMS specifically for query performance. These are referred to here as "Within-a-Step" parallelism, and "Multi-Step" parallelism.

1. **Query Parallelism.** Query parallelism is enabled in Teradata by hash-partitioning the data across all the VPROCs defined in the system. A VPROC provides all the database services on its allocation of data blocks. Before the database is loaded, the system is configured to support a given number of these VPROCs, in the 5100M that is commonly 4 to 16 per node. All relational operations such as table scans, index scans, projections, selections, joins, aggregations, and sorts execute in parallel across all the VPROCs simultaneously and unconditionally.

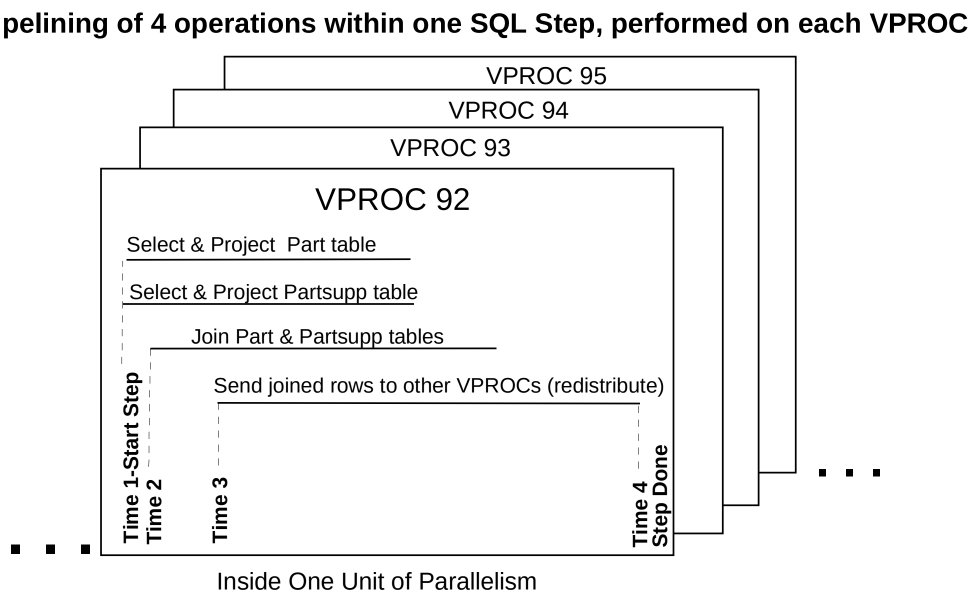
Pipelining of 4 operations within one SQL Step, performed on each VPROC
VPROC 95
VPROC 94
VPROC 93
VPROC 92
Select & Project Part table
Select & Project Partsupp table
Join Part & Partsupp tables
Send joined rows to other VPROCs (redistribute)
Time 1-Start Step
Time 2
Time 3
Time 4
Step Done
Inside One Unit of Parallelism

Figure 1: Pipelining

2. **Within-a-Step Parallelism.** A second dimension of parallelism that will naturally unfold during query execution is an overlapping of selected database operations referred to here as within-a-step parallelism. The optimizer splits an SQL query into a small number of high level data base operations called “steps” and dispatches these distinct steps for execution to the VPROCs in the system. A step can be simple, such as ”scan a table and return the result” or complex, such as ”scan two tables with row qualifications, join the tables, redistribute the join result on specified columns, sort the redistributed rows and place the result in an intermediate table”. The complex step specifies multiple relational operations which are processed in parallel by pipelining. The relational-operator mix of a step is carefully chosen by Teradata to avoid stalls within the pipeline.

3. **Multi-Step Parallelism** Multi-step parallelism, an added level of parallel activity unique to Teradata, is enabled by executing multiple “steps” of a query simultaneously, across all units of parallelism in the system. One or more processes are invoked for each step on each VPROC to perform the actual data base operation. Multiple steps for the same query can be executing at the same time to the extent that they are not dependent on results of previous steps.

Figure 2 is an exact representation of how this 3-dimensional parallelism appears in a query’s execution.

Figure 2 shows a system configured with four VPROCs, and a query that has been optimized into 7 steps. Step 2.2 demonstrates within-a-step parallelism (as does step 1.2), where two different tables (Lineitem and Order) are scanned and joined together (three operations), all three pipe-lined within one step. Step 1.1 and 1.2 (as well as 2.1 and 2.2) demonstrate multi-step parallelism, as two distinct steps are made to execute at the same time, within each VPROC.

And Even More Parallel Possibilities

In addition to the three dimensions of parallelism shown above, Teradata offers an SQL extension called a Multi-Statement Request that allows several distinct statements to be bundled together and sent to the optimizer as if they were one. These SQL statements will then be executed in parallel. When this feature is used, any sub-expressions that the different SQL statements have in common will executed once and the results shared among

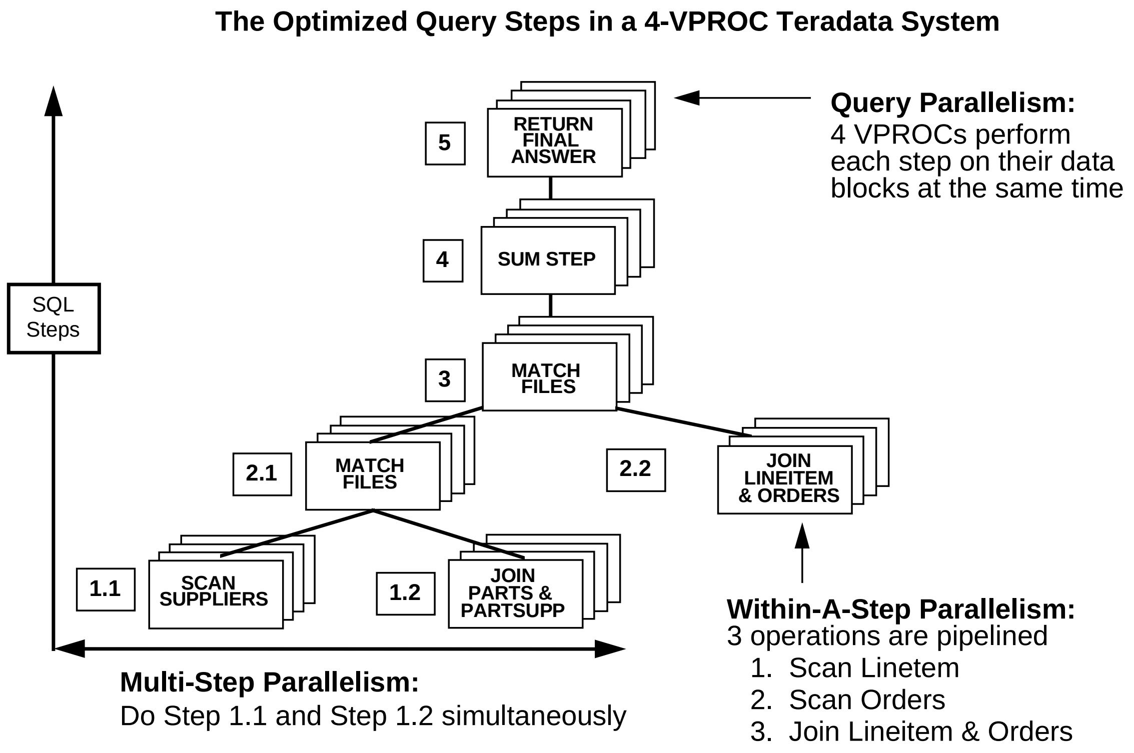
The Optimized Query Steps in a 4-VPROC Teradata System
SQL Steps
5
RETURN FINAL ANSWER
4
SUM STEP
3
MATCH FILES
2.1
MATCH FILES
2.2
JOIN LINEITEM & ORDERS
1.1
SCAN SUPPLIERS
1.2
JOIN PARTS & PARTSUPP
Query Parallelism: 4 VPROCs perform each step on their data blocks at the same time
Within-A-Step Parallelism: 3 operations are pipelined
1. Scan Linetem
2. Scan Orders
3. Join Lineitem & Orders
Multi-Step Parallelism: Do Step 1.1 and Step 1.2 simultaneously

Figure 2: Query Steps

them. Known as "common sub-expression elimination," this means that if six select statements were bundled together and all contained the same subquery, that subquery would only be executed once. Even though these SQL statements are executed in an inter-dependent, overlapping fashion, each query in a multi-statement request will return its own distinct answer set.

This multi-faceted parallelism is not easy to choreograph unless it is planned for in the early stages of product evolution. An optimizer that generates three dimensions of parallelism for one query such as described here must be intimately familiar with all the parallel weapons in the arsenal and know how and when to use them. Only the most rudimentary brand of basic query parallelism is offered today by other database products.

I'll Have My Parallelism Straight Up, Thanks

"Knowledge of Self" has been a valued trait throughout history, usually spoken of in terms of the human spirit. But this concept of self-awareness can also apply to database systems. For example, knowledge of the number and power of the parallel units, along with their predictability, gives Teradata many advantages, including operational simplicity. Teradata operates as a single, integrated, parallel system, with all units intrinsically aware of the entire system. Other parallel databases often operate as discrete copies (called instances) of the data base software with a piece of coordinator software controlling all information flow between the instances.

Knowledge of self means that fewer internal questions need asking or answering. Teradata's persistent parallelism eliminates conversations instances of a DBMS must have with each other to figure out how to divide the work for parallelization of the first step, and then how to re-divide (if necessary) the work for the fewer units available in the next step.

Parallel Systems with More than One User Don't Have to Die

Because parallel systems have the potential of giving a single user a high percentage of total system resources, they also have proven to be very vulnerable in controlling allocation of the same resources when they must be shared. Teradata is an exception to this because developers were given the opportunity of planting techniques deep in the product's base that allow throughput to be maximized while multiple dimensions of parallel activity are being made available for each user on the system. Many other products simply did not have that luxury because they have approached parallelization as transplant surgery, performed after product maturity.

Operating near the resource limits without exhausting any of them and without causing the system to thrash, requires effective work flow management. Teradata provides work flow management by monitoring the utilization of critical resources (such as CPU, memory, interconnect). If any of these reach a threshold value it triggers the throttling of message delivery, allowing work already underway to complete. This internal watchfulness is automatic to the product and is not dependent on a staff of 7 x 24 DBAs. It does not prevent new or lower priority queries from executing or reduce their parallelism. The message subsystem in conjunction with the interconnect software allows the throttling information to be passed upstream. This has the effect of controlling message generation at the origin, and cooling the demand immediately.

2 Data Placement for Parallel Performance

When conceptualizing how Teradata's parallel units could optimally process data within an MPP system, it became clear to the product's designers that physical data partitioning could either help or hinder that parallel advantage. From an MPP perspective, many of the available partitioning choices were no longer useful. One of the most common problems encountered with these traditional data placement schemes was that while the data might be balanced across disks or nodes, this did not guarantee that the actual database work was going to be balanced. Equal processing effort across parallel units became a key criteria as Teradata evolved.

Teradata solved the balance-of-work issue by permanently assigning data rows to VPROCs and using an advanced hash algorithm to allocate data. This decision to marry data to the parallel units led to the selection of a single partitioning scheme, a scheme that enforces an even distribution of data no matter what the patterns of growth or access, while formalizing an equal sharing of the workload–hash partitioning.

Teradata's Hash Partitioning

Data entering a Teradata database is processed through a sophisticated hashing algorithm and automatically distributed across all VPROCs in the system. In addition to being a distribution technique, this hash approach also serves as an indexing strategy, which significantly reduces the amount of DBA work normally required to set up direct access. In order to define a Teradata database, the DBA simply chooses a column or set of columns as the primary index for each table. The value contained in these indexed columns is used to determine the VPROC which owns the data as well as a logical storage location within the VPROC's associated disk space, all without having to perform a separate Create Index operation.

To retrieve a row, the primary index value is again passed to the hash algorithm, which generates the two hash values, VPROC and Hash-ID. These values are used to immediately determine which VPROC owns the row and where the data is stored. There are several distinct advantages in decision support applications that flow from this use of hash partitioning.

- **No Key Sequence.** One dramatic side-effect of using the hash algorithm as an indexing mechanism is the absence of a user-defined order. Most database systems use some variant of balanced trees (BTrees) for indexing, which are constructed based on an alphabetic sequence specified by the user. As new data enters the system, and random entries are added to the BTree indexes, these structures will gradually become unordered, requiring overflow techniques. Either the index quickly becomes more expensive to use, or the entire structure must be made unavailable for re-organization. Hash algorithms do not care about user-defined alphabetic order, they don't use secondary structures that require reorganization, and there is never a need to sort the data before loading or inserting.

- **Ease of Joining.** Hash partitioning of primary index values allows rows from different tables with high affinities to be placed on the same node. By designating the columns that constitute the join constraint between two tables as the primary index of both tables, associated rows-to-be-joined will reside on the same node. Since two rows can only be joined if they reside in the memory of one of the nodes, this co-location reduces the interconnect traffic that cross-node joins necessitate, improving query times, and freeing the interconnect for other work.

- **Simplicity of Set-up.** The only effort hashed data placement requires is the selection of the columns that comprise the primary index of the table. From that point on, the process of creating and maintaining partitions is completely automated. No files need to be allocated, sized or named. No DDL needs to be tediously created. No unload-reload activity is ever required.

Teradata's hash partitioning is the only approach which offers balanced processing and balanced data placement, while minimizing interconnect traffic and freeing the DBA from time-consuming reorganizations.

Why Traditional Data Placement Schemes Conflict With MPP Goals

The traditional data placement choices all fall short of parallel processing requirements because of their inability to provide a balanced workload, their inherent data accessibility weaknesses, and their tendency to flood the interconnect.

1. **"Data Set" Partitioning Guarantees Hot Spots.** The data base administrator, who must batch up data into data sets as it arrives into the system, has total control over where the data physically reside when using this primitive approach to data placement. It is possible to accomplish a balanced data load with data set partitioning, however, it is impossible to establish a balanced processing load. Because the DBA is manually assigning the data one load file at a time, a small subset of the files will contain the most recent data. Because the majority of processing in a data warehouse is comparing the most recent data to some earlier time period, the vast majority of users will be attempting to access the newest data sets at the same time. In MPP or clustered environments the DBMS has no way of telling if rows from two tables which are to be joined are located on the same processing node. Any join activity using this approach is likely to pass enormous amounts of data across the interconnect, reducing overall system throughput.

2. **Range Partitioning is an MPP Teeter-Totter.** Range partitioning appears to be a good fit when there is repetitive access based on the same constraint, because the DBA can partition the table into multiple collections of rows based on the those particular values. But range partitioning offers several difficult challenges for a parallel database and its DBA. First, business data is never politely balanced, so a thorough analysis of the data distribution must be undertaken to start setting up the partitions across nodes. Second, a method to locate commonly joined rows from different tables onto the same nodes to reduce interconnect traffic needs to be considered. Third, a method of balancing the data for each table across multiple nodes must be achieved. Fourth, since data demographics change over time, the partitioning scheme must be regularly revisited, recalculated, and data re-juggled.

   Even assuming this tremendous data balancing effort is successful, processing may remain unbalanced. Most realistic range partitioning strategies are in some way a function of time. Even the seemingly random selection of item color for a retailer will vary greatly by season. Some of the processing nodes will always

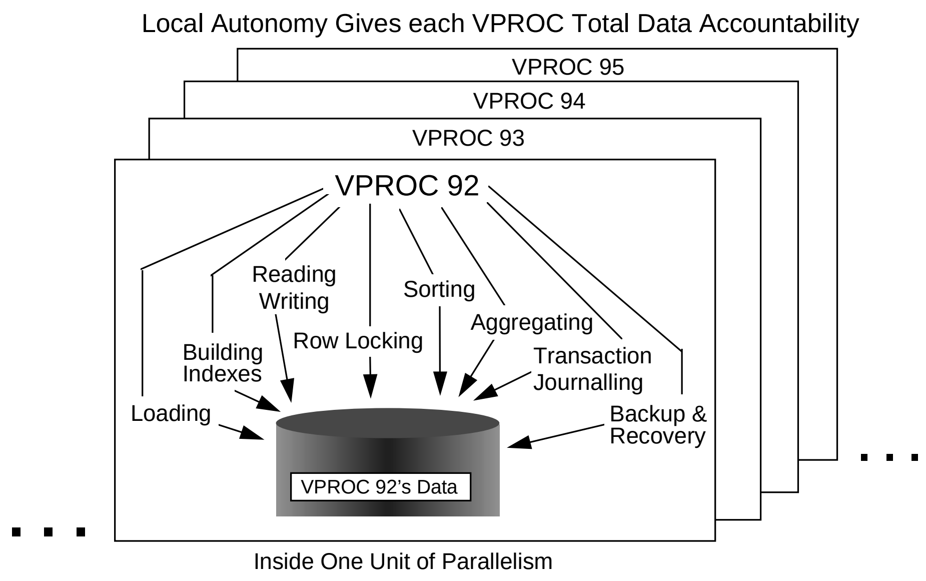

Figure 3: Local Autonomy

contain significantly more current data than others. As with data set partitioning, this will tend towards unbalanced processing.

3. **Random Partitioning Closes the Door on Direct Access.** Random, or round-robin partitioning is the assignment of data to data sets based on a random number calculation. A random partitioning mechanism is non-repeatable, meaning the DBMS never understands where a particular row for a table resides. While this technique will evenly spread the data across nodes, it also means rows will always require redistribution for a join or an aggregation, or even the final sort of an answer set.

4. **Schema Partitioning Can Be Abusive to the Interconnect** Schema partitioning is the assignment of data to specific physical processors or nodes and has proven useful when there is a need to restrict portions of the hardware to handling certain groups of tables, or schemas. While this is generally viewed as a means of increasing performance for specific tables, and can be applied in useful ways on an SMP-only database, it has not proven hugely successful in the MPP world. In most cases to join data from two schema- partitioned tables all rows have to be redistributed across the interconnect for each query. This data movement may significantly impact performance for the affected queries and the increased interconnect traffic may also impact overall system throughput. Further, node imbalances are aggravated if this option is used on anything but very small tables.

Local Autonomy

While balanced processing was a central theme during Teradata's evolution, another goal emerged having to do with ownership of or responsibility for the individual data rows. Without dividing up or parallelizing data integrity responsibility, activities such as locking have to be handled in one central place, or in a unnatural hand-off fashion, which consume large amounts of CPU and interconnect resources and create a bottleneck.

With most systems, any process can take temporary control of an object but some central process must control locking access to ensure data integrity. But the problem with the central locking mechanism is two-fold. First, the locking strategy itself will become a bottleneck unless locking can be completely suspended. Second, a task which wishes to use a row must retrieve a lock. When it is done, it must pass control, usually in the form of the

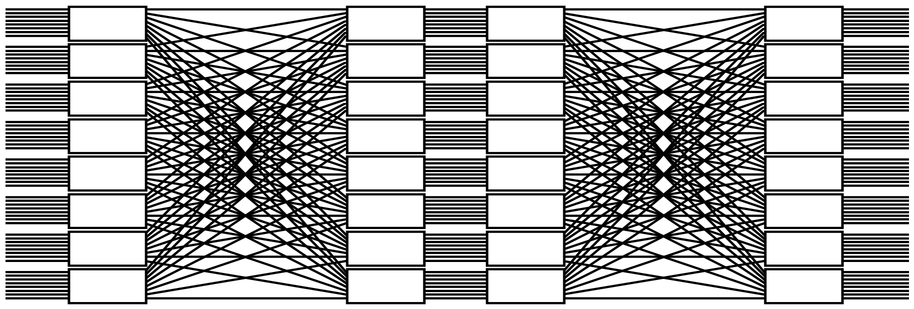

Figure 4: One of the two folded banyan BYNET networks for a 64-Node Teradata configuration. Each node has many paths to each other node in the system.

row itself, to the next process wishing access. While this is quite feasible in an SMP environment where shared memory pointers are available, it will quickly saturate any interconnect on the market.

This is one reason a shared disk approach will never be able to offer high-end scalability. Systems which rely on shared-disk will always use some dynamic algorithm to determine which instance of the database accesses which data rows. The accessing instance may or may not be the same as the instance which owns, and therefore controls the locks for the row. Bottlenecks will develop on all but the most lightly used systems.

Local autonomy within a DBMS defines precisely who owns a particular object (row or index). With Teradata, each row is owned by exactly one VPROC, and this VPROC is the only one which can create, read, update, or lock that data. All transaction logging is under the local control of the VPROC. A local lock manager functions independently in each VPROC and maintains its own set of locks. Because of this both logging and locking are parallelized, control of data is consistent, and interconnect traffic can be reduced.

3 Intelligent Use of the Interconnect

Because of the number of tasks a parallel operation can generate, poor use of any resource will quickly fan out, leading to scalability issues for a parallel database. If taken for granted, no resource can become more tenuous in an MPP system than the communication link between nodes, the interconnect. Teradata architects had the advantage of knowing they were designing an MPP-database system and therefore were able to develop the database software with the issues of inter-nodal communication in the front of their minds.

The BYNET used by Teradata is a fully scaleable folded Banyan switching network. Since the bandwidth increases as nodes are added, a 24 node system will deliver an aggregate bandwidth of nearly 1 GB/second. Teradata attempts to utilize as much BYNET functionality as possible, while at the same time taking care to keep its bandwidth usage as low as possible. When combined with Teradata the BYNET delivers messages, moves data, collects results and coordinates work among VPROCs in the system.

Just as telecommuting frees up traffic lanes by keeping employees off congested roads, Teradata minimizes interconnect traffic by encouraging stay-at-home, same-node activity where possible. VPROC-based object ownership keeps activities such as locking local to the node. Hash partitioning that supports co-location of to-be-joined rows reduces data transporting for a join. All aggregations are ground down to the smallest possible set of subtotals at the local level first. And even when BYNET activity is required, use of dynamic BYNET groups keeps the number of VPROCs that must exchange messages down to the bare minimum. BYNET-driven semaphores provide a short-cut method for coordinating parallel query processing across the interconnect. Teradata's query optimizer knows and takes into account the cost of sending a table's rows across the BYNET and factors this expense into the final join plan that it constructs.

4 Parallel-Aware Optimizer

The optimizer is the intricate piece of database software that functions similar to a travel agent, booking the most efficient travel path through the many joins, selections and aggregations that may be required by a single query.

Recycled Query Optimizers.

Two factors determine how well an optimizer will do its job: First, the quality and depth of environmental knowledge that is available as input; and second, the sophistication and accuracy of the costing routines that allow an efficient plan to be produced. Optimizers created for single-server database are ill-equipped if asked to do double-duty in a parallelized version of the same database. Extensive code changes or optimizer re-writes will be required because the cost of most operations and choices that are available with parallelism will be different.

Multiple Optimizations.

Some parallel implementations require optimization to be performed not just once, but a second time for one query: first by the base product non- parallel optimizer, and then by a second-level optimizer that makes decisions about the parallel activity. This approach suffers from lack of integration and the inflexibility of top- down decisions. The non-parallel optimizer receives the SQL first, not expecting any parallel behavior to be available, and builds a plan that might be reasonable for a single-server database. This single-server plan is passed to the second level optimizer who attempts to add parallelism, but without having the capability of influencing the original plan's formulation. It will be difficult to take full advantage of parallel potential under these conditions.

Double or Nothing.

A further problem with databases that have evolved parallel operations is that vendors will commonly support two different software code bases: one base for the original product which may be widely installed, and a different set of code for the new parallel version. Not only does this add to the development cost and overhead, but one base or the other at any point in time may lag in features and functionality, and the customer will be forced to make a choice between the two.

Teradata's Parallel Knowledge

Teradata's optimizer is grounded in the knowledge that every query will be executing on a massively parallel processing system. It knows the number of VPROCs per node configured in the system and understands that the amount of parallelism is the same for all operations. It has row count information and knows how many of a table's rows each parallel unit will be managing so I/O can be costed more effectively, and considers the impact of the type and number of CPUs on the node. The optimizer calculates a VPROC to CPU ratio, that compares the number of parallel units on a node against the available CPU power. It uses this information to build the most efficient query plan.

For example, the optimizer uses its knowledge of the number of VPROCs in order to decide whether or not to duplicate one table's rows as one alternative in preparing for a join. Table duplication will happen less frequently in a system with hundreds of VPROCs compared to a system with tens of VPROCs, because duplication requires work from all nodes in the system. The optimizer weighs the cost in CPU resources for the duplication activity, and knows that this cost will be relatively more expensive as the number of VPROCs increase.

In simple queries at low volume a parallel-aware optimizer may not greatly out-distance the less intelligent models. Like a multiple choice test with only 1 choice listed for each question, when few decisions are required, the level of skill behind the making of the decision is less important. Today's powerful hardware can easily hide a multitude of software sins at the low end, but no amount of hardware can hide software failures as data volume and environmental complexity increase. With a changing, dynamic MPP application running against a large and growing volume of data, the intelligence and thoroughness of the optimizer will make or break the application.

Conclusion

Watching some of the world's best athletes perform in the Olympics, it's hard not be impressed by how often quality is accompanied by a sense of effortlessness. In this paper we have illuminated some of the processes and underpinnings contributing to Teradata's effortless nature: Automated data placement, multi-dimensional query parallelism, database/interconnect synergy, and a High-IQ optimizer. When a product is born to be parallel, then most wasted energy and unnatural efforts can be eliminated, resulting in a product that is focused, integrated and enduring.

Parallel Solutions in ClustRa

Svein Erik Bratsberg, Svein-Olaf Hvasshovd, Øystein Torbjørnsen
Telenor R&D
N-7005 Trondheim, Norway

1 Introduction

Parallelism is used to achieve scalability, where much focus has been put into query processing and optimization. ClustRa is aimed at telecommunication applications, where transactions are simple, but where there is a demand for high availability and soft real-time response [7]. In addition to catering for increased load and data volume, ClustRa uses parallelism in all situations requiring a highly available database. Parallelism is important for high availability, because failure masking situations must be done as fast and smoothly as possible. To cater for increased load in a highly available database, the reorganization of data must be online and non-blocking.

The ClustRa database system uses a shared-nothing hardware architecture, where each node is an atomic failure unit. Nodes are grouped into sites, which are collections of nodes with a correlated probability of catastrophic failure. The basic idea is that each site has a replica of the database. Logically, the database system consists of a collection of interconnected nodes that are functionally identical and act as peers, without any node being singled out for a particular task. This improves the robustness of the system and supports easy maintenance and replacement of nodes. Furthermore, it enables a straightforward scaling of the system.

The central ideas behind ClustRa are concerned with data declustering and logging and recovery. By having a clever fragmentation and replication strategy, we cater for both scalability and high availability. This article focuses on how the declustering strategy combined with a location and replication transparent logging and recovery method is used to support parallelism at takeover, online self-repair, online backup and online refragmentation.

2 Declustering Strategy

Together, the data placement involved in fragmentation and replication constitute a *declustering strategy*. The main goals of the declustering strategy used in ClustRa are to achieve high availability, scalability, and load balancing, both during normal processing and upon failures.

We are minimizing the storage redundancy by keeping the number of replicas to a minimum, i.e. two, by combining disaster recovery and efficient single component failure masking. To achieve this, the declustering strategy must ensure that the replicas of a fragment reside on different sites. We also want the load to be evenly distributed over the sites by letting all nodes store both primary and hot standby fragment replicas.

Figure 1 illustrates *mirrored declustering* [2, 9]. Pairs of nodes (one node from each site) store replicas of the same fragment. The nodes in the pair divide the primary and hot standby tasks between them so that the load is balanced. The figure shows that the table is divided into six fragments, all existing in two copies. The

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

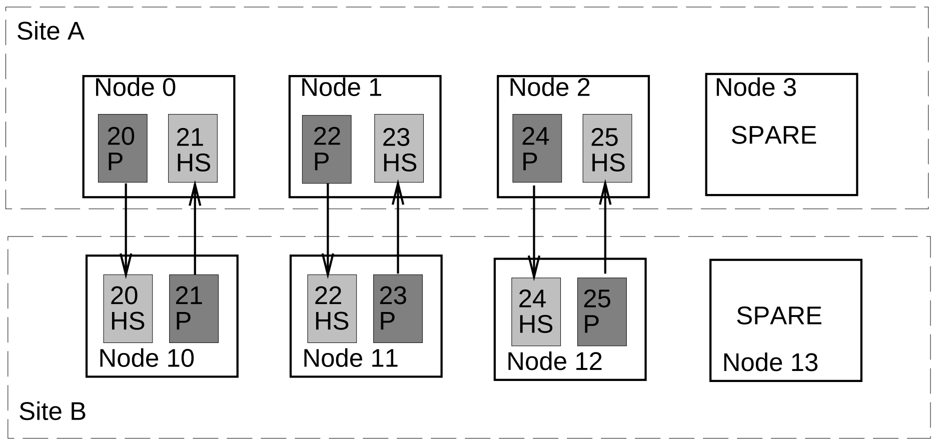

Figure 1: Mirrored declustering with dedicated spares.

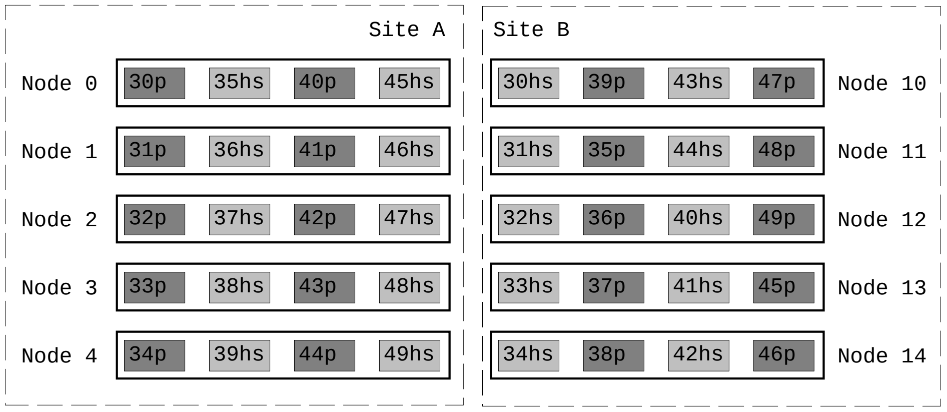

Figure 2: Minimum intersecting sets declustering.

primary fragment replicas are shown with dark boxes, while the hot standby fragment replicas are shown with bright boxes. At each site it is a *spare* node, which is used when a failed node does not recover.

Other declustering strategies lead to more complicated allocation schemes where fragment replicas stored at one node at one site is distributed over multiple nodes at the other. Figure 2 illustrates *minimum intersecting sets declustering* [9]. This strategy is easiest to understand by viewing the identifiers of fragment replicas as columns in matrixes. When comparing the matrixes for the two sites, the first column is equal, the second column is rotated one position down, the third column is rotated two positions down, and so on. The idea of minimum intersecting sets declustering is to balance the load of failure handling, i.e. during takeover and online self-repair. This will be explained in ensuing sections of the paper.

3 Software Architecture and Transaction Execution

Each node is organized as a set of services: Transaction controller, database kernel, node supervisor and update channel. The *transaction controller* receives requests from clients to execute stored procedures, or it receives code which it interprets. It coordinates transactions through an extended two-phase commit protocol. The *kernel* has the main database storage manager capabilities, like locking, logging, access methods, tuple and buffer management. It receives code to be interpreted from a transaction controller. Lock management is performed in the kernel. Thus, there is no central lock server creating bottlenecks. The *update channel* is responsible for reading

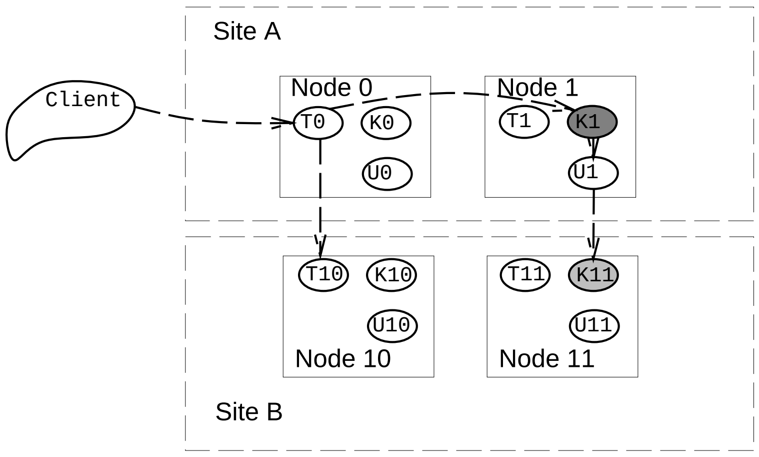

Figure 3: Execution of a simple transaction.

the log of primary fragment replicas, and for shipping the log records to the nodes of the hot standby fragment replicas. The *node supervisor* is responsible for collecting information about the availability of different services, and for informing about changes.

Figure 3 illustrates execution of a 2-safe [5] transaction in ClustRa. The transaction controller (T0) receives the request from the client, and thus become the primary controller. It has a hot standby controller (T10) at the other site, which is ready to take over as primary if T0 fails. T0 consults the locally cached distribution dictionary to find the nodes holding the primary replicas of the tuples in question. For this particular transaction, K1 has the role as primary kernel. T0 sends the start transaction command, the operations to be executed and the prepare-to-commit command to K1. The update channel (U1) reads the log of K1 and consults the dictionary cache to know where to ship the log records. The log records of this transaction are shipped to K11, where they are stored in the log and redone. When K1 has received an acknowledgement of the log records from K11, it responds with “ready” to T0. When T0 has received “ready,” it sends the hot standby controller (T10) the outcome of the transaction. When T0 has received an acknowledgement from the hot standby controller, it responds to the client. The second phase of the commit includes sending commit messages to both kernels involved. When both kernels have responded done to T0, the transaction is removed from T10 and T0.

When a transaction accesses several tuples, there may be several primary and hot standby kernels executing the transaction in parallel.

4 Logical Logging and Update Channels

The logging and recovery system in ClustRa has been designed to facilitate any conceivable declustering strategy based on horizontal range or hash partitioning. ClustRa also supports online self repair and online database reorganization. To achieve this flexibility, we cannot impose any restrictions on the physical allocation and representation of data inside a node. Therefore, the tuples and tuple log records are completely location transparent. This is done by using logical logging and state identifiers connected to tuples [1].

The fragment replicas consist of both the tuples themselves and the associated tuple log records. The log records are at any time only generated at the primary replica and sent to the hot standbys, ensuring sequence preservation in the logs. Any log replica can be applied to any of their corresponding fragment replicas. This is important in a highly available database, because when a table grows, online refragmentation may be necessary. Thus, log records applied to the same primary fragment replica, may be redone to different, *subfragmented* hot standby replicas. Thus, each subfragmented hot standby replica will receive log records with increasing sequence numbers, even though some of the log records may have been shipped to other subfragments. The log received at each subfragment reflects the tuple operation order per tuple, which is required to obtain a consistent hot standby replica.

An *update channel* is responsible for shipping log records from one primary fragment replica to the corresponding hot standby fragment replicas. Figure 1 shows this as an arrow. Log records are sent as asynchronous parallel streams from primary fragment replicas to the hot standby replicas. One node will therefore host the same number of update channels as primary fragment replicas. Since the state identifiers are generated and maintained per fragment, *parallel update channels* become possible. This ensures that the log transport capability will scale linearly with the number of nodes. In Figure 2 we see that each node at site A has two outgoing update channels to different nodes at site B. Commercial replication systems used for disaster recovery purposes restrict scalability by having one log stream between sites.

Online refragmentation needs to merge fragments into one consolidated fragment replica. This type of operation is needed when restarting a crashed node with a fragment replica having a subfragmented hot standby. The operation may also be needed at scaleup, when a new node is included and some fragments are to be gathered and merged at this node. The log from each fragment will be kept separate. After a following takeover, all new log records are entered into a separate log and they are given log sequence numbers that are higher than the log sequence numbers in all the original fragment logs. The old logs will be removed as part of the checkpointing policy [6].

5 Parallel Takeover

When a node has failed, *takeover* will happen. This means that one or more nodes at another site will take over the primary responsibility for the fragments which resided at the failed node. If node 11 in Figure 1 failed, node 1 takes over its primary responsibility (for fragment 23). However, by using minimum intersecting sets declustering, the takeover will be performed in parallel at different nodes. If node 11 in Figure 2 failed, node 0 and 3 will take over its primary responsibility (for fragments 35 and 48). This balances the extra load between two nodes.

Before takeover, the *virtual node set protocol* [3] informs the other nodes that a node has failed. The update channels from the failed node will be closed and all enqueued log records from the failed node will be redone. For each fragment replica turning primary due to the takeover, the next log sequence number to be allocated will be increased by a certain interval. This interval of unused log sequence numbers may be needed for undo by the failed node when it recovers. Prior to the crash, it may have produced some log records which were not replicated. The new primary nodes will undo all transactions which were active at the failed node. The change in the current virtual node set will redirect all tuple requests to the new primary fragment replicas.

There will be some redo and undo activity going on during takeover. Therefore, parallel takeover makes the unavailability interval shorter. However, ClustRa does eager redo at hot standbys so that takeover may be performed as fast as possible. It has been measured to be on the order of 1/3 second using off-the-shelf workstations [7].

6 Online Self-Repair

When a node has failed and is not able to recover, online self-repair is started. The purpose of online self-repair is to reestablish the availability level without degrading the response time and the current availability of the database. Thus, the repair is done while other transactions are doing both reads and updates to the database.

There are several phases of a repair transaction. First, fragment replicas are created at the spare node and log shipping is started from the current primary nodes of these fragment replicas. The spare node should replace the failed node, so it will include the same fragment replicas. Second, fuzzy copy is started. This creates a copy of the fragment replicas at the spare node while other transactions are running. Third, the distribution dictionary

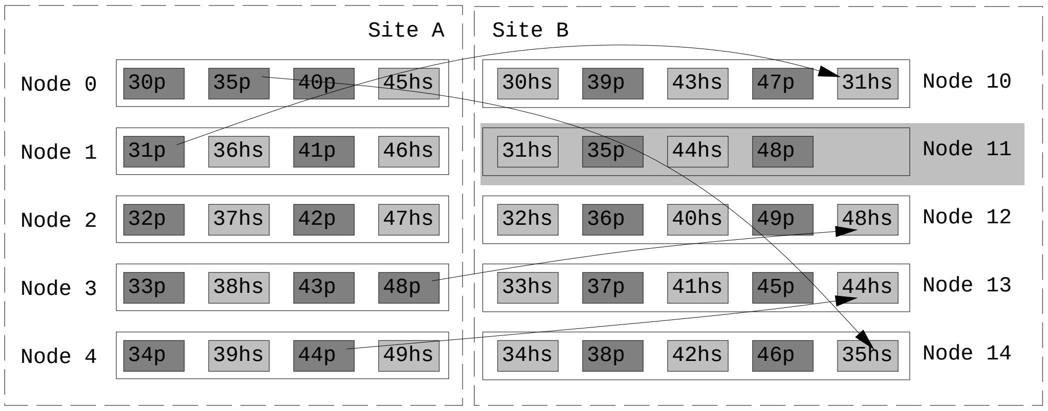

Figure 4: Distributed spare using minimum intersecting sets declustering.

caches at all nodes are reflected with the "move" of the fragment replicas. Fourth, the node dictionary is updated with the state of the failed node and the spare node, which is turning into an active node.

The repair is non-blocking because it does not stop other transactions in executing both reads and updates to the primary copy. The copy is done fuzzily, because the fragment replicas being copied are not in a transaction consistent state during the whole copy. The fuzzy reader does not care about tuple locks kept by other transactions. Thus, the new copy at the spare node is updated with the use of the log. The normal redo activity can not be applied to fragment replicas being copied, because it expects that all tuples are present. The redo activity is divided into the redo that can be done at once, because the fragment replicas are copied and the redo activity that has to wait until the fragment replicas are copied. When the fuzzy reader starts to fill a chunk of data, the current log sequence number is read and put into the message going to the spare node. Thus, for each chunk of received data the spare node knows where in the log to start the redo activity.

Online self-repair will put some load on the current primary nodes of the system. This might suggest giving the fuzzy copy low priority. However, it is also important to have the period when the availability is degraded be as short as possible. To cater for this, we can use parallel copying of data, both at the sender and the receiver side. By using minimum intersecting sets declustering we may have both multiple senders and multiple receivers participating in fuzzy copy. In Figure 4 we see that the repair of node 11 creates one new fragment replica at each of the other active nodes at site B. When there is no dedicated spare node, but the extra load is balanced between active nodes in the system, this is called the *distributed spare strategy*. This strategy requires the active nodes to have spare storage and processing capacity.

7 Online Backup

Backup production is a vital area for a continuously available parallel database. To distribute the load the backup production is performed in parallel.

ClustRa uses dedicated sites to run as backup sites. ClustRa's backups will be produced on disk. A backup site requires less processing power than an active site, but it needs comparable disk space to hold the required number of backup versions of the entire database. A backup site can be instructed to dump a backup version on tape.

The production of a backup is performed similarly to online non-blocking repair. A fragment is backed up by producing a fuzzy copy of it at a backup node, and by producing a copy of the log records created during the backup. The method is designed to add minimum load on the involved active nodes. To cater for this, a maximum number of fragments to be copied in parallel is specified. After start of copy of a fragment, all log records produced for that fragment must be copied to the backup node to guarantee transaction consistent backup database.

The first backup version requires the number of backup fragments to correspond to the number of primary fragments at the start of the backup production. This is achieved by establishing the same number of logical nodes at the backup site as at the active site. If a node or site crashes during a backup production, the backup will be rolled back. In future versions, it will be allowed to establish a different number of fragments for a backup table than for the primary table. This will require concatenation of subfragments and their corresponding logs.

Creating a transaction consistent image of a backed-up database is a problem quite similar to obtaining a 1-safe consistent database after a site crash. We use the single mark variant of the epoch algorithm [4] method to obtain consistency. The same problem occurs between fragments of a table in case a table consistent backup is to be produced.

8 Scaling and Refragmentation

It is of great importance for continuously available database servers to be able to do management operations like scaleup without interrupting services. There are two main reasons why a system needs to be scaled up. Firstly, the transaction load has increased so that more processing power is needed. Secondly, the data volume has increased so that more memory and disks are needed to store the data. Scaling up a shared-nothing database means adding more operational nodes to the system. Scaleup is an operation which may be performed as a low priority background activity.

ClustRa supports parallel, online, non-blocking scaleup. The scaling is performed by gradually redistributing the database so that all the available nodes are taken into operation. ClustRa performs scaleup by redistributing a table at a time. Within a table, a few fragments are refragmented in parallel to avoid overload of the sender and the receiver nodes. Scaleup involves subfragmenting a table, i.e. the number of fragments increases as part of a scaleup. This gives a high number of fragments. This strategy will in addition keep the additional load on the system low and minimize the extra storage needed as part of the subfragmentation. Only the fragments under redistribution will need an extra replica.

When the subfragmentation of a fragment is completed, the subfragmented replica will take over as the hot standby replica for the fragment and the old hot standby replica will be deleted. The storage for the old replica will then be freed. The new hot standby replica will then take the role as the primary replica for the fragment. Another subfragmented replica is then produced based on the current primary replica. When this replication is finished, the new subfragmented replica takes the role as the hot standby and the old non-subfragmented replica is deleted. The subfragmentation of a fragment will be run as a transaction with savepoints per replication.

Scaledown is like scaleup performed through parallel replication. Parallel scaledown requires that multiple fragments are merged. The same techniques as for scaleup is used with respect to minimizing additional storage needed.

Fuzzy copying, as used in both online repair and online scaling, creates a compact version of the database, because the new copy is created by sequential insertion of the tuples.

9 Fully Replicated Distribution Dictionary

Parts of the dictionary tables are in frequent use by the transaction controllers of ClustRa. Since we use a peer-to-peer architecture, where transactions controllers may run on all ClustRa nodes, the distribution dictionary and stored procedures are cached at all nodes. The distribution dictionary describes how tables are fragmented and replicated. Thus, to know where to send tuple operations, the transaction controller does a lookup in the dictionary cache. This means that all transaction controllers can access the distribution dictionary in main memory in parallel.

Transactions updating the distribution dictionary need to grab the dictionary lock, which blocks the distribution dictionary for other updates. An update to the distribution dictionary is replicated to the dictionary caches at all nodes. During an update of the distribution dictionary, the dictionary cache will exist in two versions. One version is the current version of the dictionary, the other is the one being updated. Only the dictionary change transaction sees the new version during the update. When the dictionary change transactions commits, the new version becomes the current one. However, old transactions which started prior to the commit of the dictionary change transaction, will still use the old dictionary. If a new dictionary change transaction tries to start when there still are transactions using the oldest dictionary version, these transactions will be aborted.

During a dictionary change, the update channel, which uses the distribution dictionary to know where to ship log records for replication, will not see the dictionary change before it commits. To prevent update channels from becoming direct participants in two-phase commit, all transaction which change the dictionary will generate dummy change operations, which are “executed” at primary fragment replicas. The update channel reads these operations in the log, and thus, they know the dictionary change. This technique is used both at online self-repair and during schema update operations.

10 Comparison

Some commercial systems use inter site redundancy by replicating sites for disaster recovery purposes. These systems are geared towards transaction throughput by buffering the replication log records at either the sender or the receiver site. This gives either a 1-safe solution, where committed transactions may be lost, or bad transaction response time. The takeover time is also quite high for these systems, since vast amount of log records needs to be redone at takeover. ClustRa, on the other hand, uses 2-safe replication with an eager redo policy, which leads to fast takeover time.

Other commercial solutions use intra site redundancy (e.g. RAIDs) and shared disk solutions to mask single component failures. DB2 for MVS/ESA moves in the direction of supporting self-repair, since it allows for online reorganization to restore clustering of tables [8]. Both reads and writes are allowed during copy. The new copy is made up-to-date by replaying the log created during the copy phase. This is similar to ClustRa’s method of fuzzy copy and catchup, which, unlike DB2, is used for online backup and self-repair as well as for online refragmentation. Since DB2 uses a physiological identification of tuples within log records, log records must be translated according to new locations of tuples prior to being applied. ClustRa uses logical identification of tuples within log records, and thus does not need any translation.

11 Conclusions

Parallel solutions are important to allow for scalability. ClustRa supports scalability by using a shared nothing architecture with horizontal fragmentation. Some applications demand scalability to be combined with high availability. This creates a need for online maintenance, like repair, refragmentation and backup. ClustRa’s declustering strategy combined with the logging and recovery method allows for parallel solutions aimed at improving the availability of the system.

References

[1] Bratsberg, S.E., Hvasshovd, S.-O., and Torbjørnsen, Ø. Location and replication independent recovery in a highly available database. In 15th British National Conference on Databases. Springer-Verlag LNCS, July 1997.

[2] Copeland, G., and Keller, T. A comparison of high-availability media recovery techniques. In proceedings of the ACM SIGMOD Conference, pages 98–109, June 1989.

[3] El Abbadi, A., Skeen, D., and Cristian, F. An efficient, fault-tolerant protocol for replicated data management. In M. Stonebraker, editor, Readings in Database Systems, pages 259–273. Morgan Kaufmann, 1988.

[4] Garcia-Molina, H., Polyzois, C.A., and Hagmann, R.B. Two epoch algorithms for disaster recovery. In proceedings of the 16th International Conference on Very Large Databases, pages 222–230, August 1990.

[5] Gray, J., and Reuter, A. Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1992.

[6] Hvasshovd, S.-O. Recovery in Parallel Database Systems. Vieweg, 1996.

[7] Hvasshovd, S.-O., Torbjørnsen, Ø., Bratsberg, S.E., and Holager, P. The ClustRa telecom database: High availability, high throughput, and real-time response. In Proceedings of the 21st International Conference on Very Large Databases, pages 469–477, September 1995.

[8] Sockut, G.H., and Iyer, B.R. A survey of online reorganization in IBM products and research. IEEE Data Engineering Bulletin, 19(2):4–11, 1996.

[9] Torbjørnsen, Ø. Multi-Site Declustering Strategies for Very High Database Service Availability. PhD thesis, The Norwegian Institute of Technology, January 1995.

XPS: A High Performance Parallel Database Server

Chendong Zou
Informix Software, Inc.
921 S.W. Washington Street, Suite 670
Portland, OR97205
email: zou@informix.com

1 Introduction

In recent years, as more businesses start using massive databases as their main source of information, more emphasis is placed on the performance of the database system. Because of the size of these databases and the nature of the applications (for example, data mining and data warehousing), a traditional single-threaded relational system is not good enough to meet customer needs. On the other hand, hardware has become cheaper and cheaper each year [Sch96], making a system that consists of hundreds of disks and CPUs not that expensive anymore. Under these circumstances, parallel database systems have begun to gain ground in the business world, and are considered a viable (sometimes the only) choice for commercial information systems.

Informix OnLine XPS is a high performance parallel database server. It can run on either a MPP (massively parallel processing) system or a *cluster* of stand-alone computers that are connected with a high-speed network. We refer to each computer within a MPP system or cluster as a *node*. XPS uses “shared-nothing” architecture [Sto86] [DG90], i.e., each node runs a separate *coserver* that is equivalent to a single INFORMIX-OnLine Dynamic Server 7.x instance. An XPS server includes multiple coservers communicating directly to share data and performing parallel execution. It provides:

- Near-linear scalability.
- Near-linear speedup.
- Fast data loading utilities.
- Single point of administration.

In this paper, we describe the XPS system. Section 2 gives an overview of the system architecture. Different ways of partitioning data and indexes are discussed in section 3, followed by a description of query execution in section 4. Other interesting features in XPS are described in section 5. Results of our TPC-D benchmark and some concluding remarks are also in section 5.

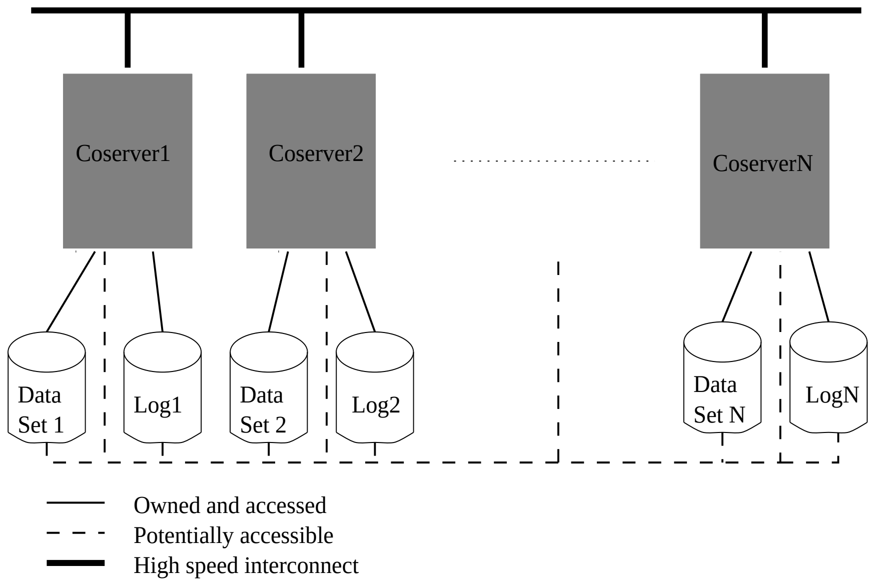

Figure 1: XPS shared-nothing architecture

2 XPS Overview

As we mentioned in section 1, XPS uses a shared-nothing architecture. Each coserver in the system manages its own logging, locking, buffer and recovery. Coservers are connected through a high speed network. Figure 2 shows the architecture of the system, where Data1 ... DataN are partitioned data sets. This shared nothing architecture allows XPS to have:

- **Data partitioning.** Each coserver has its own disk. Data can be partitioned to reside on those disks. Different data partitioning strategies will be discussed in section 3.
- **Control partitioning**. Each coserver manages its local log, lock, and buffer. There is no centralized locking and buffer management. This avoids the potential bottleneck problems of centralized control.
- **Execution partitioning.** Because of the data partitioning and the control partitioning, execution can be done in parallel *all* the time.

2.1 XPS System Components

We will describe the internal components of the XPS system in this section. Figure 2 shows the different components of the XPS system.

There are four global services in Figure 2:

- **Configuration manager:** maintains the identities and locations of all coservers that exist in the XPS system, and what portions of the database are owned by those coservers.
- **Metadata manager (MDM):** manages the catalog information (cached table schema) across multiple XPS coservers. It maintains identical copies of the schema information on each coserver. There is a MDM service thread on each coserver. However, there is only one MDM, called the *primary* MDM for each database. The rest are called *secondary* MDMs. The primary MDM is responsible for sending requested dictionary table entries to the secondary MDM. It is also responsible for sending requests to secondary

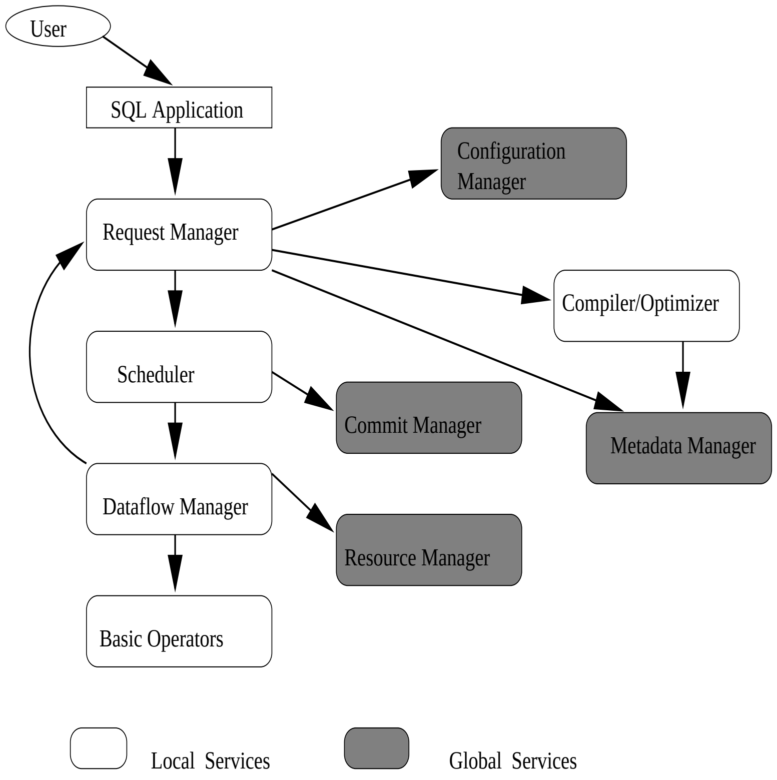

Figure 2: XPS Internal Subsystems

MDMs to invalidate their private copies when the master copy is invalidated. A MDM service thread can be the primary of one database and a secondary of another database.

- **Resource manager:** is used by scheduler to determine what resources are available to execute a query.
- **Commit manager:** manages global transactions. It is also used during recovery to decide the status of a global transaction. XPS uses a patent-pending commit protocol instead of the standard two phase commit protocol for global transactions.

The *request manager* is responsible for managing the context of a user's query. It initiates the start, commit or abort of a user transaction. It also initiates savepoints during a transaction. There is a request manager service thread on each coserver.

The *dataflow manager* is responsible for pipelining data between iterators and provides flow control during execution. (The execution model is explained in Section 4.) It tries to match the speed of producer iterators and consumer iterators. Because of the management of data flow, unnecessary spooling to disk during execution can be avoided.

A query is optimized by the *optimizer*, and the optimized execution plan is managed by the *scheduler*. Our optimizer uses a dynamic programming technique to do optimization. It considers disk I/O cost and CPU cost. It builds the execution plan using the system metadata. Metadata is globally visible and cached on each coserver through MDM.

In addition, subsystem XMF (XPS Messaging Facilities) handles communication between XPS subsystems and threads. XMF provides reliable communications on top of a platform-specific datagram service. Because XMF is connectionless, and doesn't have the overhead of TCP/IP, it provides efficient communication between XPS subsystems.

3 Data Partitioning

One of the key reasons of XPS's high performance is its data partitioning techniques. In this section, we will briefly describe some of them.

Data partitioning (fragmentation) in XPS allows a user to fragment a table across disks that belong to different coservers. It can also store the results of a query in a temporary table and dynamically fragment that temporary table to different dbspaces. (A *dbspace* is the basic logical storage unit that XPS manages. It comprises one or more chunks of disk space.)

XPS supports the following *basic* data fragmentation strategies:

- **Round Robin.** The system is responsible for distributing the data in a even fashion among all the dbspaces. This is ideal for load balancing.
- **Hash.** XPS uses a system-defined hash function to decide where to put rows that are being inserted. The system-defined hash function is carefully chosen so that reasonable load balancing can be achieved. Hash-based fragmentation is good for equijoins and exact matches. It is reasonable for load balancing.
- **Expression-based fragmentation.** In this strategy, the user will define the fragmentation rule and supply this rule as part of CREATE TABLE and CREATE INDEX statements. Users can define any arbitrary expression as the fragmentation strategy. Range expressions are the most often used ones. Expression-based (range expression) fragmentation is good for range queries, equijoins and group-bys.

A potential problem with range-expression-based fragmentation is data skew. For example, if most of the data falls into a particular range that is on a single coserver, then often that coserver will do most of the work for queries because the other coservers don't have much data to work on. Parallelization is hard for this kind of data skew. In order to avoid this problem, a new fragmentation strategy called *hybrid* fragmentation has been invented.

The basic idea of hybrid fragmentation is to use two levels of fragmentation. It first uses expression-based fragmentation to partition data set to different dbslices. (A *dbslice* is a named set of dbspaces that XPS can manage as a single storage object, the set of dbspaces can belong to different coservers in the system.) Then it uses the hash fragmentation strategy shown above to do further fragmentation. In the case of range-based fragmentation, this avoids the data skew problem because data can be distributed by hash among different dbspaces (on different coservers) in a dbslice.

3.1 Index Partitioning

XPS supports index fragmentation as well as data fragmentation. Index and data can be fragmented into the same coserver using the same fragmentation strategy. They can reside in the same dbspace or different dbspaces. Moreover, in order to improve performance, index and data can be fragmented using different strategies. For example, one can hash on column A to fragment a table, and use a range expression on column B to fragment an index on column B of the table. Thus, range queries on column B can be performed more efficiently.

3.2 Fragment Elimination

Fragmenting data and indexes allows different CPUs and/or coservers to participate on the same query, thus improving the performance of the system. On the other hand, if a coserver does not contain data that is of interest to the query, it does not need to participate in the query.

For example, suppose there are two coservers in the system, and table T is fragmented by range expression:

```
CREATE TABLE ttt(a INTEGER, b INTEGER)
        FRAGMENT BY EXPRESSION a < 10 in dbspace1, a >= 10 in dbspace2;
```

and the query is:

SELECT * FROM ttt where a > 12;

Then we don't need to scan data in dbspace1 on coserver1, thus improving the query response time.

Fragment elimination is the technique that XPS uses to avoid unnecessary work for query processing. It is part of query optimization. Currently, XPS supports fragment elimination for range-expression-based fragmentation and hash-based fragmentation.

Because of data and index fragmentation, single-user response time is greatly improved. Data fragmentation also makes parallel data loading possible in XPS.

4 Query Execution

When the request manager receives a user query, it first calls the query compiler/optimizer, which generates an optimized tree-structured query plan. The request manager then registers the query plan with the scheduler. The activation of a query plan usually involves the request manager, the resource manager, and the scheduler.

Query execution in XPS follows the *iterator* model that is similar to that of the Volcano [Gra90]. The iterator model in XPS follows a simple *create-open-next-close-free* protocol. Basically, iterators go through the following stages: creation, initialization, iteration (loop through the input data set by calling *next*), termination, and free(deletion). The iterator model has the following advantages:

- iterators are I/O type independent and provide encapsulation of functionality.
- the *exchange* iterator provides an easy way to do parallelization.
- demand-driven dataflow control between iterators within a query tree is easy.

XPS provides iterators for scan, nested-loop join, merge join, hash join, group by, sort, merge etc. In addition, insert, delete, update, aggregation, and index build can also be executed in parallel.

Because of XPS's data partition technique and its use of the iterator execution model, it provides both vertical parallelism and horizontal parallelism. Vertical parallelism means that different iterators in the query plan are executed in parallel. When different coservers execute an different instance of the same iterator in the query plan in parallel, we call this horizontal parallelism. For example, in order to scan a table, each coserver will execute a scan iterator for the part of the table that is fragmented to its disk. Data partitioning is the key to horizontal parallelism.

5 Other Features

We will briefly describe some other interesting XPS features in this section. These include:

- **Virtual processor and multithreading.** XPS uses a configurable pool of database server processes called virtual processors to schedule and manage user requests that are represented by lightweight threads. This is significantly cheaper than managing user requests at the operating system process level.
- **High performance parallel loader(Pload).** Pload/XPS views external data as an external table which only supports sequential read (loading) and write (unloading). Treating external data as a table provides a powerful method of moving data into or out of the database and specifying transformations on the data. This is because many of the XPS parallel SQL operators can be applied on the external table, which speeds things up greatly.
- **High availability.** XPS will do automatic switchover to recover a failed node by letting a surviving coserver take over the log and data on the failed node and assume the workload of that node.

- **Backup/restore.** XPS provides utilities for backup, archive, and restore of an XPS system.
- **Ease of use/administration.** XPS provides a single-system view to users and system administrators. For example, users can query a table without knowing whether the table is fragmented or how the table is fragmented.

Table 1: TPC-D Result

Configuration	Database Size	Database Load Time	TPC-D Power	TPC-D Throughput
HP/Informix-XPS	300GB	7:47:7	3416.4 Qppd	1673.7 Qthd

In a recent TPC-D benchmark, XPS achieved impressive numbers as shown in Table 1. It is interesting to note that XPS took less than 8 hours to load the benchmark database. (Interested readers can look up TPC-D numbers at TPC Council's Web site at *http://www.tpc.org*.)

XPS adopts a true, shared-nothing architecture. Through data partitioning, multithreading, parallel query execution and its unique transaction management protocol, it outperforms competitive databases in terms of overall performance and scalability.

Acknowledgement

I would like to thank all the people who make XPS possible. I would also like to thank Scott Shurts, whose comments greatly improved this article.

References

[DG90] D. DeWitt and J. Gray. Parallel database systems: The future of database processing or a passing fad? *sigmod*, 19(4):104–112, December 1990.

[Gra90] Goetz Graefe. Volcano, an Extensible and Parallel Query Evaluation System. Technical Report CU-CS-481-90, University of Colorado at Boulder, 1990.

[Sch96] Bob Schaller. The origin, nature, and implications of ”moore's law” – the benchmark of progress in semiconductor electronics, 1996 http://131.107.1.182/research/BARC/Gray/Moore_Law.html

[Sto86] M. Stonebraker. The case for shared-nothing. *IEEE Data Eng. Bull.*, 9(1), March 1986.

Query Optimization in DB2 Parallel Edition

Anant Jhingran
Timothy Malkemus
Sriram Padmanabhan

IBM TJ Watson Research Center,
Hawthorne, NY 10532

Abstract

This paper describes query optimization and processing in DB2 Parallel Edition – a full function, parallel, scalable database system from IBM. We give a brief overview of the components, and delve deeper into query optimization. In particular, we discuss the use of data partitioning as a property of operators and issues related to "SQL Completeness", wherein some SQL constructs like correlated subqueries need to be optimized, as well as processed at run-time.

1 Introduction

Considerable research in the area of parallel database systems is available in the literature. For example, papers on prototypical shared-nothing database systems like GAMMA [5], and BUBBA [3], and shared-memory systems like XPRS [9] have published results on several aspects of parallel database systems. In addition, the technology for parallel algorithms for execution of important database operations [2, 14], query optimization techniques [11, 13], data placement [4, 7, 10, 12] etc. form a basis of our knowledge of parallel database issues today and have found their way into commercial systems.

DB2 Parallel Edition [1] (DB2 PE) is a parallel database system that runs on AIX parallel machines such as IBM's SP.[1] Its Shared-Nothing (SN) architecture model and Function Shipping execution model provide two important assets: *scalability* and *capacity*. Its query optimization technology considers a variety of parallel execution strategies for different operations and queries and uses Cost (CPU + IO + Messaging) in order to choose the best possible execution strategy. The execution time environment is optimized to reduce process, synchronization, and data transfer overheads. The ACID properties [8] are enforced efficiently in the system to provide full transaction capabilities. Utilities such as Load, Import, Reorganize Data, and Create Index have been efficiently parallelized. We also provide a parallel reorganization utility called *Redistribute* which effectively corrects data

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1] Its future releases will also support other platforms. The technology described in the paper refers only to the current release.

and processing load imbalance across different nodes of the system. In sum, DB2 PE is a comprehensive, full-fledged, parallel database system.

By using function shipping, database operations are performed where the data resides. This minimizes network traffic by filtering out unimportant data, as well as achieves good parallelism. So a major task in a shared-nothing implementation is to split the incoming SQL into many subtasks – these subtasks are then executed on different processors (if required, interprocess and interprocessor communication is used for data exchanges). Typically, a coordinator serves as the application interface – it receives the SQL and associated host variables, if any, and returns the answers back to the application. The bulk of the work is performed by slave tasks, which, if they access base tables, need to execute at the nodes where the table exists, and in other cases (such as joins) could be "free" to be instantiated based on run-time parameters such as load. One of the advantages that we realized from using function shipping was that we could leverage a lot of the existing DB2/6000[2] code – the scans, sorts, joins etc in DB2 PE are identical to the operators in DB2/6000 – they are transparently parallelized because their inputs are. The real fundamental technology is in the mechanism that glues the nodes together to provide a single system view to outside world. In addition to function shipping, other technologies required are (1) generation of parallel plan, (2) streaming of data and control flow, (3) process management, (4) parallel transaction and lock management, and (5) parallel utilities.

DB2 PE provides extensions to SQL in the form of new data definition language (DDL) statements which allow users to control the placement of database tables across the nodes of a parallel system. DB2 PE supports partial declustering (a table can exist on a subset of nodes), overlapped assignment (multiple tables can exist on the same node), and hash partitioning of database tables using the notion of *nodegroups*. A nodegroup is a named subset of nodes in the parallel database system. The following example illustrates the use of the nodegroup DDL statement:

CREATE NODEGROUP **GROUP_1** ON NODES (1 TO 32, 40, 45, 48)

When creating a table, it is possible to specify the nodegroup on which the table will be declustered. The cardinality of the nodegroup is the degree of declustering of the table. In addition, it is possible to specify the column(s) to be used for the partitioning key. The following example illustrates the use of DDL extensions to the CREATE TABLE statement:

CREATE TABLE **PARTS** (*Partkey* integer, *Partno* integer) IN **GROUP_1**
PARTITIONING KEY (*Partkey*) USING HASHING;
CREATE TABLE **PARTSUPP** (*PartKey* integer, *Suppkey* integer) IN **GROUP_1**
PARTITIONING KEY (*Partkey*) USING HASHING;

For each row of a table, the hash partitioning strategy applies an internal hash function to the partitioning key value to obtain a partition (or bucket) number. This partition number is used as an index into an internal data structure associated with each nodegroup, called the *partitioning map (PM)*, which is an array of node numbers. Each nodegroup is associated with a distinct partitioning map. If a partitioning key value hashes to partition i in the map, then the corresponding row will be stored at the node whose node number appears in the i^{th} location in the map, i.e. at $PM[i]$. If two tables share the same nodegroup, and their data types of the partition keys are *compatible*, then the tables are said to be *collocated*. DB2 PE provides a simple set of rules that define compatibility of unequal data types. The partitioning strategy ensures that rows from collocated tables are mapped to the same partition (and, therefore, the same node) if their partitioning key values are the same. This is the primary

[2]DB2/6000 V1 is the single node (serial) database engine that DB2 PE is based on.

property of collocated tables. Conversely, if rows from collocated tables map to different nodes then their partitioning key values must be different. Collocation is an important concept since the equi-join of collocated tables on the respective partitioning key attributes can be computed efficiently in parallel by executing joins locally at each node without requiring inter-node data transfers. Such joins are called *collocated joins* and have the property of being highly scalable (perfectly scalable in the ideal case). Thus, in the above example, the following is a collocated join:

$\mathit{PARTS} \bowtie_{(Partkey=Partkey)} \mathit{PARTSUPP}$.

If the collocated property does not hold, alternative strategies based on repartitioning or broadcasting one or more input relations will be considered.

2 Query Optimization for DB2 Parallel Edition

The Compiler component of DB2 Parallel Edition is responsible for generating the Parallel Query Execution Strategies for the different types of SQL queries. The DB2 PE compiler is implemented on the basis of a number of unique principles:

- **Full-fledged Cost based optimization -** The optimization phase of the compiler generates different parallel execution plans and chooses the execution plan with the best cost. The optimizer accounts for the inherent parallelism of different operations and the additional costs introduced by messages while comparing different strategies.
- **Comprehensive usage of data distribution information -** The optimizer makes full use of the data distribution and partitioning information of the base and intermediate tables involved in each query while trying to choose parallel execution strategies.
- **Transparent parallelism -** The user applications issuing Data Manipulation SQL statements do not have to change in order to execute on DB2 PE . Hence, the investment that users and customers have made already in generating applications is fully protected and the migration task for the DML applications is trivial. Application programs written for the DB2/6000 product do not even need to be recompiled fully when they are migrated to DB2 PE ; the application only requires a **rebind** to the parallel database which generates the best cost parallel plan for the different SQL statements, and, if appropriate, stores them.

2.1 Operator Extensions

For the most part, parallel processing of database operations implies replicating the basic relational operators at different nodes. Thus, the basic set of operators (such as **Table Access**, **Join**, etc.) are used without much change. However, the function shipping database architecture introduces two new concepts that are not present in a serial engine such as DB2/6000:

- Query execution over multiple logical tasks. Consequently, we need operators that the coordinator task can use to control the run-time execution of slave tasks. This operator, called distribute sub-section, is described in more detail in a later section.
- As a consequence of multiple processes, interprocess communication operators (notably send/receive) are required in DB2 PE. These operators can have attributes (e.g., send can be broadcast or directed; receive could be merging (when its inputs are sorted), or first-in-first-out).

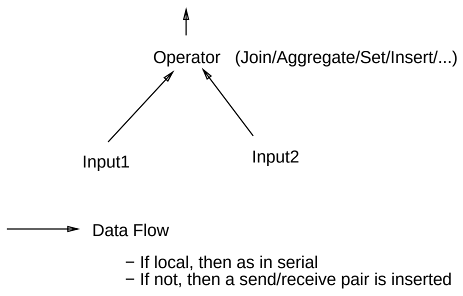
Operator (Join/Aggregate/Set/Insert/...)
Input1
Input2
Data Flow
− If local, then as in serial
− If not, then a send/receive pair is inserted

Figure 1: A Typical Operator subgraph in DB2 PE

2.2 Partitioning Knowledge

In DB2 PE, we are conscious about partitioning in the DDL, Data Manipulation SQL, and at run-time. DB2 PE's partitioning methodology can be viewed simply as a load balancing tool (by changing the key and partition map, we can adjust the number of tuples on any node); however by making the compiler and the run-time systems understand it, we have succeeded in improving SQL performance beyond simply load balancing. As mentioned before, an example of this is collocated joins. The compiler, being fully cognizant of partitionings, node groups etc., can evaluate the costs of different operations (collocated vs. broadcast joins for example, as described later) and thus choose the optimal execution strategy for a given SQL statement. In the case of certain directed joins, the run-time system uses the knowledge of partitioning to correctly direct tuples to the appropriate nodes.

2.3 Query Optimization and Execution Plan Generation

The compiler is responsible for generating the *optimal* strategy for parallel execution of a given query. In this section, we will describe the query execution plans as trees of operators separated into tasks. The query execution can be viewed as a data flow on this tree, with send/receives being used for inter-task communication.

In Figure 1, a generic binary operator which takes two inputs is shown. It can execute in the same task as one or both of its input(s), or different. In the latter case, the cost of one or two send/receive pair(s), has to be factored in during optimization. The properties of the send/receive (in addition to decision on whether it is required) depends on the partitioning properties of the inputs and the operator in question. For example, if the operator is a UNION ALL and the inputs exist on the same nodegroup, then the operator can execute without requiring a send/receive pair on either input stream.

A query optimizer typically chooses (a) the optimal join order and (b) the best way to access base tables and to compute each join. This task is inherently exponential ([15, 6]) and many optimizers use heuristics like postponing of cross products, left-deep trees, etc. in order to prune the search space. In the case of a parallel database, this operation is further complicated by (c) determining the nodes on which operations need to be done (this is called the *repartitioning strategy* and is required because the inner and the outer may not be on the same set of nodes) and (d) choosing between system resources and response time as the appropriate metric for determining the cost of a plan.

In DB2 PE, we have made a few simplifying assumptions in order to keep the query optimization problem tractable:

- We keep track of, on a per node basis, the total system resources accumulated during the bottom-up generation of a query plan. The maximum across all the nodes is a measure of the response time of a query. This ignores the costs associated with the coordinator, as well as the response time complications associated with multiple processes.
- Of all the possible subsets of nodes that can be used to execute a join, we typically consider only a few –

all the nodes, the nodes on which the inner table is partitioned, the nodes on which the outer is partitioned, and a few others. Since this is a constant number of "interesting partitionings", by using effective pruning techniques, our overall complexity is also within a constant factor of the serial optimization problem.

In DB2 Parallel Edition, the join operation can be performed by a variety of different strategies. See [1] for a description of collocated, directed, broadcast and repartitioned joins. In brief, a directed join involves execution of join on the nodes of one of the inputs and directing the tuples from the other to the correct nodes (this will happen when the join is an equi-join on the partitioning key of one of the input); repartitioned join involves re-distributing both the inputs (and can be used in any equi-join); and finally in a broadcast join, one of the input relations is broadcast to the nodegroup of the other before performing the join (this can be used for any join). Cost determines which is the best join strategy for the query.

Cost Based Optimization

One of the most important features of the optimizer is that it uses *cost estimates* when deciding between different execution choices. This is to be contrasted with an optimization technique which heuristically decides to go with a particular strategy. For example, given a join operation, the optimizer estimates the cost of each of the join strategies described above and chooses the one with the least cost estimate for a given join step. The cost basis makes the optimizer decisions more robust when choosing between strategies such as broadcast or repartitioned joins.

Cost estimation also enables the optimizer to choose the best query execution plan in a parallel environment. It accounts for the messaging costs incurred by operations. Most importantly, estimation tries to influence *parallel processing* of different parts of the query whenever possible. Unlike XPRS [9], in a shared-nothing environment we discovered that a two step approach (serial optimization followed by parallelization) could lead to arbitrarily bad plans, since data partitioning and collocation have a profound impact on the execution cost of the query.

2.4 Parallelism for all operations

Parallel database systems have to be "SQL complete". This poses challenges in both optimization and processing, because a lot of SQL constructs were simply not designed with parallelism in mind. In addition, using a parallel database system underneath a serial application poses some interesting changes in the semantics, that each vendor must address and at least explicitly document, if not solve.

The challenges are the most in the so called shared-nothing systems because they must necessarily, by virtue of data partitioning, do all SQL operators in parallel. For example, some shared-disk parallel systems have an easier task of at least implementing some of the SQL constructs like searched updates – they can, in the worst case, execute it in a single stream. However, in shared nothing systems that utilize function shipping, the operation must happen across all the nodes that contain the table being updated.

A guiding principle in our compiler design has been to enable parallel processing for all types of SQL constructs and operations. For the sake of brevity, we only list the other operations and constructs where we apply parallelism while generating the execution strategies. Also note that based on our partitioning properties, strategies similar to collocation, repartitioning, direction and broadcast can be used for all operators.

- Aggregation: Ability to perform aggregation at individual slave tasks and, if required, at a global level. In addition, we can do a "repartitioned" strategy when the degree of useful parallelism for the "group by" clause is higher than that of the input.

- Set operations: Collocated or repartitioned strategies are considered. Note that the UNION can be an n-ary operation.

- Inserts with subselect, updates, deletes: Inserts can be collocated with subselects, or may use a "directing" strategy; all updates and deletes are collocated with the search.
- Outer Joins: They are similar to joins, except that special care is needed when executing in parallel to prevent generation of multiple non-matching rows in the answer set, one from each node.
- Subqueries: We consider collocated, directed, and broadcast methods of returning subquery results to the sites where the subquery predicate is evaluated.

Because of space limitations, we describe only one, namely subqueries.

3 Subquery Processing in DB2 Parallel Edition

Past work on parallel SQL has often concentrated on the abstract problem of joins and has ignored complex constructs like subqueries. The subquery construct in SQL has been known to pose several challenges to query optimization. In serial engines, various optimizations such as de-correlations have been used, and in addition, when forced to execute subqueries (e.g., when de-correlation does not work), optimizations such as execute an uncorrelated subquery only once etc. have been used. For example, in

```
select * from t1 where
t1.a in (select b from t2 where
         t2.c = t1.a and t2.d in (select b from t3 where t3.a = 10))
```

the innermost query block can be evaluated once for the entire query, whereas in

```
select * from t1 where
t1.a in (select b from t2 where
         t2.c = t1.a and t2.d in (select b from t3 where t3.a = t1.f))
```

the innermost query block must be evaluated *once for every distinct value generated by the outer query block.*

While in serial systems, such optimizations are straightforward to execute, in parallel systems, the run-time gets considerably more complicated. In particular, outer tuples are being produced in parallel, yet each node wants to execute some version of the subquery without trampling on the other nodes. Thus, (i) we should not set up threads for each outer tuple (i.e. execute the subquery as if it were a new query – that will be very expensive) (ii) must make sure that different nodes which are producing outer tuples do not get serialized in the execution of the subquery (iii) guarantee that the correct outer node receives the result of the execution (iv) push down predicates wherever possible so that minimal traffic flows etc. Consider the following simple one level correlated subquery.

```
select * from t1 where
t1.a > (select avg(t2.b) from t2 where t2.c = t1.d)
```

Figure 2 shows how DB2 Parallel Edition might process the query in a two node system (with both tables partitioned across both nodes). Slave tasks #1 generate outer tuples and feed "t1.d" to the slave tasks #2 as a broadcast (shown as light arrows going from #1's to #2's). Each slave task #2 is in a loop, listening to the "t1.d" value it receives. It computes the local sum and count and sends the result back to the node it received the correlation from (shown as dark arrows connecting #2's to #1's). The sending node, after it has received partial sums from all nodes, can now compute the global average associated with the subquery, and determine if the corresponding tuple of t1 satisfies the predicate. Every tuple of t1 that satisfies the clause is then shipped to the coordinator

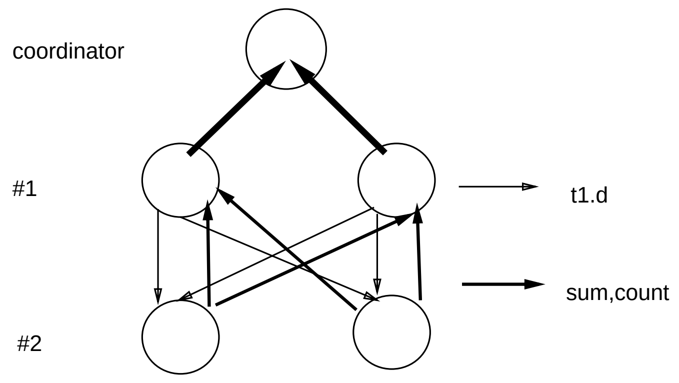

Figure 2: Execution of a simple correlated subquery

(shows as heavy arrows connecting #1's to the coordinator). *The interesting aspect of this mode of processing is that at the same time, different nodes of slave task #2 can be executing different correlation values – i.e. there is no need to synchronize the execution of the subquery!*

In addition to this model of correlated subquery processing, in DB2 Parallel Edition, we consider other options (such as purely local, which is possible when the correlation predicate (t2.c = t1.d) and/or the predicate connecting the outer query block to subquery could guarantee locality). Again, this is based fully on the partitionings associated with the outer and inner query blocks.

Optimizing correlated subqueries when de-correlation is not an option, is a challenge. This is mainly due to the violation of the principle of optimality. In general (assuming no suitable indexes), serial query optimization can optimize the subquery (substituting "t2.c = constant" and then optimize the outer query block and still achieve an optimal plan. However, in a parallel execution environment, the optimal plan for a complex subquery evaluation can depend on "where" it is used (i.e. where the subquery predicate is applied), and from where the correlation value flows (i.e. which task generates the correlation value). In DB2 Parallel Edition, we use some simple heuristics to make this problem tractable.

4 Summary

DB2 Parallel Edition provides a flexible infrastructure for designing the layout of tables using nodegroups and partitioning keys. It also provides very efficient function-shipping and data exchange operators. Query optimization is the crucial component that "completes the puzzle" by using the layout and partitioning knowledge of base and intermediate tables to generate efficient query execution plans. DB2 PE's query optimizer considers parallelism for all SQL statement constructs in a uniform fashion using the basic infrastructure that includes collocation, repartitioning, or broadcast strategies. Also, cost is the basis of all query plan decisions. Hence, we believe that we have succeeded in implementing a robust optimizer that can generate very efficient query plans for complex queries on large scale database systems.

References

[1] C. K. Baru et al. DB2 Parallel Edition. *IBM Systems Journal*, 34(2):292–322, 1995.

[2] C.K. Baru and S. Padmanabhan. Join and Data Redistribution algorithms for Hypercubes. *IEEE Transactions on Knowledge and Data Engineering*, 5(1):161–168, February 1993.

[3] H. Boral et al. Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):4–24, March 1990.

[4] G. Copeland et al. Data placement in Bubba. In *Proceedings of the 1988 ACM SIGMOD Conference*, pages 99–108, Chicago, IL, June 1988.

[5] D.J. DeWitt et al. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990.

[6] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *Proceedings of the 1992 ACM SIGMOD COnference*, May 1992.

[7] S. Ghandeharizadeh and D.J. Dewitt. Performance analysis of alternative declustering strategies. In *Proceedings of 6th Intl. Conference on Data Engineering*, pages 466–475, Los Angeles, CA, February 1990.

[8] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4), 1983.

[9] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in XPRS. In *Proceedings of the 1991 PDIS Conference*, 1991.

[10] K. A. Hua and C. Lee. An adaptive data placement scheme for parallel database computer systems. In *Proceedings of the 16th VLDB Conference*, pages 493–506. Morgan Kaufman, August 1990.

[11] S. Khoshafian and P. Valduriez. Parallel execution strategies for declustered databases. In M. Kitsuregawa and H. Tanaka, editors, *Database Machines and Knowledge base Machines*, pages 458–471. Kluwer Acad. Publishers (Boston, MA), 1988.

[12] S. Padmanabhan. *Data Placement in Shared-Nothing Parallel Database Systems*. PhD thesis, EECS Department, The University of Michigan, Ann Arbor, MI 48109, 1992.

[13] D. Schneider and D.J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 469–481, Brisbane, Australia, 1990. Morgan Kaufman.

[14] D.A. Schneider and D.J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the ACM SIGMOD Conference*, pages 110–121, Portland, Oregon, May 1989.

[15] P. Selinger et al. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD Conference*, pages 23–34, 1979.

Parallel Processing Capabilities of Sybase Adaptive Server Enterprise 11.5

Eugene Ding, Lucien Dimino, Ganesan Gopal, T.K. Rengarajan
Sybase, Inc.
1650 65th Street
Emeryville, CA 94608
eugened, dimino, gopal, ranga@sybase.com

Abstract

Parallelization allows database management systems to take full advantage of the widespread use of multi-processor computers [1]. Sybase Adaptive Server Enterprise 11.5 parallelizes a number of DBMS operations for the SMP environment. The parallel intra-query processing facility enables the server to process a query in parallel, and thereby reduce the response time considerably. The database consistency checker utility takes full advantage of IO parallelism to check the consistency of a database in a fraction of the time. Database recovery also exploits IO parallelism to achieve up to 500% performance improvement. The asynchronous pre-fetch technique is used widely throughput the database for query processing and recovery as well as utilities.

Introduction

The widespread availability and use of multi-processor computers make parallel database management systems (DBMSs) an attractive, cost-efficient solution for meeting the increasing demand of business data processing. The parallel processing capabilities of Sybase Adaptive Server Enterprise 11.5 (ASE 11.5) are designed to utilize the potential power of multi-processor computers with shared memory. The parallelization significantly improves the performance of many DBMS operations. This paper presents a number of parallel processing capabilities of Sybase ASE 11.5. We shall discuss parallel query processing, parallel database consistency checking, parallel recovery and the common asynchronous pre-fetch mechanism.

1 Parallelism Overview

The parallel features of ASE 11.5 are targeted at SMP systems across popular hardware and OS platforms. Shared memory is assumed to be present. The algorithms are applicable to shared memory clusters, NUMA machines etc. The fundamental goal for the parallel features is to achieve scale up and speed up with increasing numbers of processors in an SMP system. Parallelism in CPU as well as disk IO is exploited.

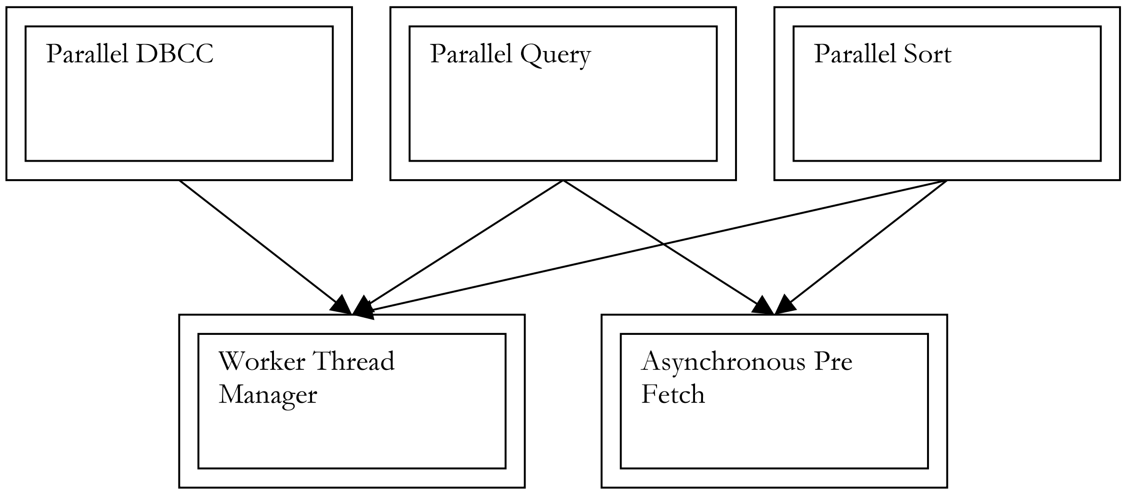

Fig 1. Parallel system components

The core software infrastructure for the parallel features is the Worker Thread Manager (WTM). Figure 1 shows the interaction between software components discussed in the paper. Sybase ASE 11.5 is a multithreaded server consisting of multiple engines (OS processes) managing a number of user threads. The WTM enables the server to create internal threads that perform various functions inside the server. The parallel query, parallel DBCC as well as parallel sort features use the WTM. The Sybase scheduler schedules the worker threads just like any user thread. As a further enhancement, ASE 11.5 allows the database administrator (DBA) to schedule database threads on specific processors, control priorities of queries etc.

Partitioning of tables in the database provides the basis for parallelism for parallel query and parallel sort. Each table in the database can be partitioned into many slices (i.e., partitions). At present the partitioning is random. The query optimizer considers all parallel plans to determine the one that minimizes cost. Parallelism is not orthogonal to query optimization. Users are recommended to place slices of tables on independent devices. However, the slices are logical and do not require such hard partitioning. This provides a smooth upgrade path for existing customers so that they can slice the tables first to realize partial benefits of parallelism and improve it further by moving the slices to independent disks.

The asynchronous pre-fetch (APF) feature exploits the parallelism in the IO subsystem via the use of asynchronous read calls provided by the operating system. We refer to this feature as "parallel IO." This basic infrastructure is exploited by many policies to keep multiple disks busy as much as possible. In parallel query execution, each of the worker threads itself issues multiple parallel reads to keep multiple disks busy. The disk writes by ASE 11.5 already are asynchronous and exploit the parallelism in the disk subsystem to the maximum. The disk writes are accomplished by the buffer washing mechanism and by the Housekeeper thread ([2,3]). Parallel query and parallel sort use the APF mechanisms. However, the parallel DBCC uses a specially optimized form of APF to maximize performance for bulk sequential reads.

The parallel query design is optimized for common operating environments. Shared memory is exploited to optimize inter-thread communication. The focus is on partitioned parallelism and not on pipelined parallelism. This is based on expectation of multi-user workloads and tens of processors in one SMP system. ASE 11.5 also supports some parallel execution methods that are not based on slices.

All parallelism methods steal buffers from the cache in order to satisfy memory needs. The IO size and the APF mechanism are orthogonal to the parallel execution. These are controlled by independent configuration parameters that affect the entire server.

2 Parallel Query Processing

ASE 11.5 introduces intra-query parallel query processing capabilities into the server. For each query that the client submits, the optimizer shall consider processing the query in parallel as an alternative strategy. It decides whether to execute the query in parallel and the optimal way to parallelize the query execution. A parallel processing strategy is chosen only if it is determined to be beneficial through a cost based analysis.

Parallelism is achieved through data partitioning. Sybase ASE users can horizontally partition a table, and distribute data rows over multiple partitions of the table. Furthermore it is possible to bind each partition to a different disk. This allows the server to exploit the available I/O bandwidth by fetching data from multiple disks in parallel.

While a one-to-one mapping between partitions and disks is normally recommended, it is not required. Users can place multiple partitions of a given table on a single disk. This flexibility is useful in situations where I/O parallelism is already built into some specialized hardware, such as RAID systems.

Parallel execution of queries on a partitioned table is effective only if the partitions are roughly equal in size, i.e., without data skew. In practice, a partitioned table can become unbalanced over the time as data rows are inserted and deleted. This could negatively affect the performance of parallel query execution, as the response time is dominated by the thread that retrieves the largest amount of data. ASE 11.5 provides means to rebalance a partitioned table and distribute rows evenly among partitions.

One can realize CPU parallelism without partitioning a table a priori. The server dynamically divides data rows among multiple buckets with an internal hash function. The technique can be applied to scan an index in parallel as well. While it does not take full advantage of I/O parallelism, the mechanism leads to finer granularity of parallelism and can be beneficial for some multi-table join.

A join is processed in parallel as long as any table in the join is scanned in parallel. Given a query that joins two tables, say A and B, where A has 2 partitions, B has 3 partitions, and there is no useful index. The optimizer may choose any of the following four strategies:

- Serial execution of join between two tables, join(A, B).
- Parallel scan on A only, with a parallel degree of 2. Each thread scans a single partition of A and the entire table B. The 3 threads perform join(A1, B), and join(A2, B), respectively. It is shown graphically in Figure 2.
- Parallel scan on B only, with a parallel degree of 3. Each thread scans the entire A and a single partition of B. The 2 threads perform join(A, B1), join(A, B2), and join(A, B3), respectively. It is shown graphically in Figure 3.
- Parallel scan on both A and B, with a parallel degree of 6. Each thread scans a single partition of A and a single partition of B. The 6 threads perform join(A1, B1), join(A2, B1), join(A1, B2), join(A2, B2), join(A1, B3), and join(A2, B3), respectively. It is shown graphically in Figure 4.

The technique can be generalized to joins with more than two tables. For an N-way join, it is possible to achieve a very high degree of parallelism (P1 * P2 * ... * Pn), where Pi is the parallelism degree for scanning the (i)th table.

Parallel aggregates are processed in two steps. First, multiple threads compute the local aggregate concurrently, where each thread covers a portion of the data. Second, the local aggregate results are merged to produce the global result.

Although the second step has to be serialized, it should not constitute a bottleneck as the cardinality of aggregate is typically low, relative to the size of raw data. The first step would have already reduced the size of data significantly, and the second step needs only to merge a small number of groups.

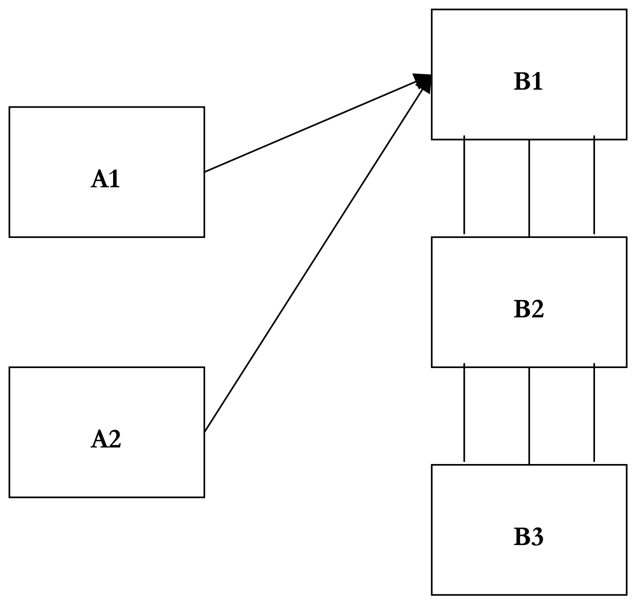

Figure 2.

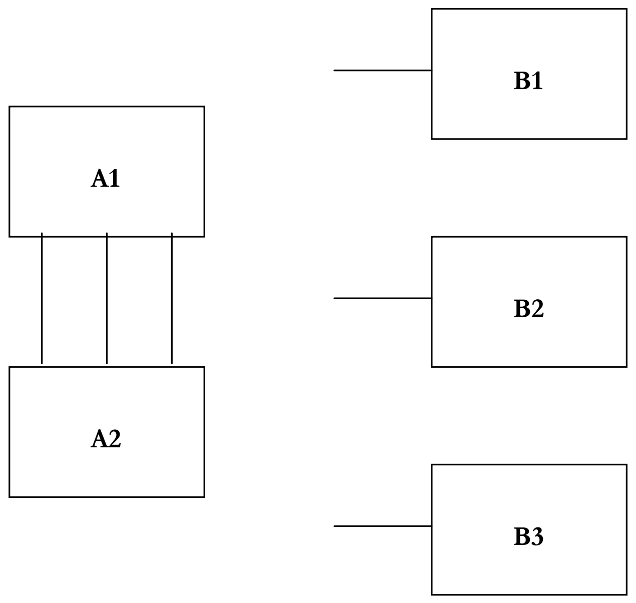

Figure 3.

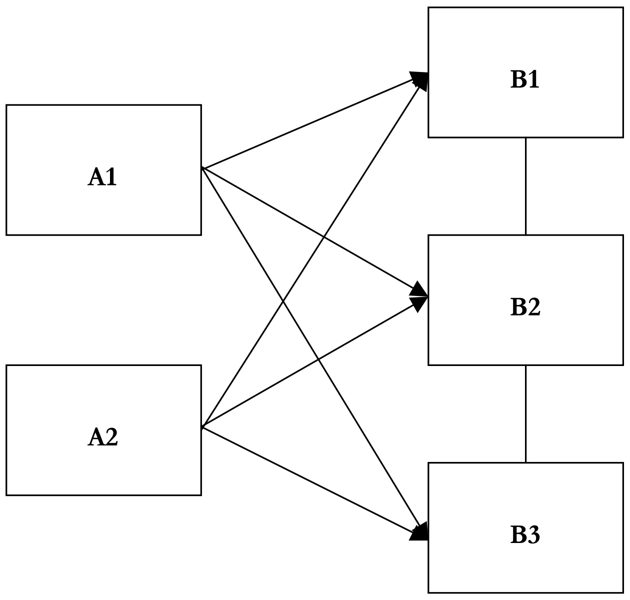

Figure 4.

During the execution of a query, there is frequently a need to store intermediate results in a temporary work table. It is normally followed by a sort of the work table, either to eliminate the duplicates or to arrange the data in some sorted order. ASE 11.5 supports partitioned work table to facilitate parallel insertion into it. The number of partitions for a work table corresponds to the parallelism degree of the query. In addition, the work table can be sorted in parallel as well.

While parallel query processing could significantly reduce the response time of a query, it also increases the resource consumption. ASE 11.5 is designed to operate in a multi-user, mixed work load environment. There are a number of mechanisms to prevent a single user (or application) from monopolizing the server resource:

- The server administrator can set up a limit on the amount of resources a user (or application) can consume per query. This limit cannot be violated whether the query is processed in parallel or not. Suppose the resource limit is L, and a query is processed in parallel by 10 threads. The parallel execution does not automatically increase the limit to 10*L. Instead, the total resource consumption of all 10 threads cannot exceed L.

- The server administrator can specify a set of engines that a user (or application) can run. It is possible to set aside a subset of the engines for OLTP applications, while the rest of the engines can be used by decision support applications. A complex query in a decision support application may be processed in parallel and consume a lot of resources. But it cannot prevent other OLTP applications from being serviced.

3 Parallel Database Consistency Checker (DBCC)

Providing data integrity is a prime function of a database management system. In order to provide the integrity of critical data, it is necessary to ensure the integrity of the on-disk database structures. It is expensive in time and money to diagnose and repair a single on-disk corruption after observing symptoms from application failure.

Database consistency checking software can provide the required confidence in the database integrity. However, the cost of consistency checking is so high that it becomes impractical. As a result, either confidence in the database integrity is compromised, or extraordinary lengths are taken to maintain the desired level of confidence. For example, one may generate a copy of the database solely for the purpose of consistency checking, or one may employ periodic restructuring of the data to refresh the on-disk structures.

ASE/11.5 introduces a new consistency check that addresses the problem of ensuring database integrity in a VLDB environment. The new parallel dbcc typically scales linearly with the system aggregate throughput, combines many of the checks of the five current functions, does not lock tables or indexes for extended periods, and locates and reports errors accurately despite concurrent update activity. It also separates the three objectives of report generation, checking, and repairing. This provides for custom reporting and evaluation, and quick, minimally intrusive repairs. Recording dbcc activity and results within the server provides new capabilities such as trend analysis, and a new source of accurate diagnostic information for support.

A practical consistency check must perform well, and it not sufficient for the operation to scale with the database size. The time available for performing the operation does not increase significantly as the size of the database expands to the VLDB range. Consequently, the consistency check must be able to apply the full aggregate system capacity to execute in the absolute minimum time. Parallel dbcc employs CPU parallelism, data caching, IO queuing, IO parallelism, device modeling, and load leveling to minimize the execution time.

The ASE 11.5 worker thread manager enables dbcc to subdivide the checks to be performed and execute each of the subdivisions in parallel. Parallel dbcc uses the available knowledge of data placement relative to the devices, table fragmentation, data cache availability, and the performance and interaction between logical devices to determine the best subdivision of checks. Dbcc then optimizes the parallel work assignment for minimal execution time with minimal resource usage. Dbcc itself determines the optimal level of parallelism to use. The result is a consistency check that can use the full system capacity.

A practical consistency check must avoid repeatedly visiting the same disk pages. The many consistency checks require correlating information stored in widely dispersed disk locations. These checks need to be restructured so that they can be effectively performed in a single pass over the database. Parallel dbcc uses a special form of table, called a workspace, to permit these checks to be performed in a single pass over the database.

A practical consistency check must have consistent and repeatable performance. Generally there will be a fixed window of time in which the check can be executed, and large or unpredictable variations in performance can not be tolerated. Parallel dbcc always performs a single pass over the entire database, and it carefully schedules and optimizes the sequence of checks. This insures that performance is not sensitive to uncontrollable factors such as table sizes, or table fragmentation. Performance is very consistent and repeatable.

A practical consistency check should also be executable online with minimal impact on applications sharing the database server. Parallel dbcc achieves this by using dedicated resources, and by using unique ”order from chaos” checking strategies. Parallel dbcc does not block concurrent activity, not even activities such as reindexing, bulk load, or create and drop table. Parallel dbcc is designed to have the capacity to use the entire system capacity. However, parallel dbcc’s usage of worker threads (processes?) and data cache is configurable so that the impact on the concurrent activity can be reduced. In addition, the ASE/11.5 Application Queues feature can be used to lower the priority of parallel dbcc, or restrict it to execute on a fraction of the CPUs in a SMP server.

A consistency check is an ideal place to gather information about disk usage, and the placement of tables and indexes. In the past this has led to voluminous reports that need to be post-processed to extract the desired information, including such basic information as the number and density of faults discovered. Parallel dbcc remedies this situation, by recording all it’s results in a management database. You can then use the full power of the SQL language with that data. Storing the results in a database enables dbcc to record much more detail It also allows one to investigate how the database is changing over time, and to provide a historical analysis of detected faults.

Supplied with parallel dbcc are a set of basic reports for presenting the parallel dbcc configuration parameters, detected faults, table and index placement, and the dbcc activity log. Also supplied as a set of stored procedures to aid in configuring the parallel dbcc configuration parameters, and managing the dbcc database.

4 Parallel IO : Asynchronous Pre-Fetch

Traditional disk and buffer cache management are reactive; disk accesses are initiated and buffers are allocated in response to applications demands for data. The following things make proactive I/O management desirable and possible:

1. under utilization of storage parallelism
2. growing importance of data access performance
3. ability of I/O-intensive applications to offer hints about their future I/O demands.

Storage parallelism is increasingly available in disk arrays and striping device drivers. These hardware and software arrays promise the I/O throughput needed to balance ever-faster CPUs by distributing the data of a single disk over many disk arms. Trivially, parallel I/O workloads benefit immediately; very large accesses benefit from parallel transfer, and multiple concurrent accesses benefit from independent disk actuators.

Unfortunately, many I/O workloads are not at all parallel, but instead of consist of serial streams of non-sequential accesses. In such workloads, the service time of most disks accesses is dominated by seek and rotational latencies. Moreover, these workloads access one disk at a time while idling the other disk in an array. Disk arrays, by themselves, do not improve I/O performance for these workloads any more than multiprocessors improve the performance of single-threaded programs.

Pre-fetching strategies are needed to ”parallelize” these workloads. The second factor encouraging Parallel IO is that ever-faster CPUs are processing data more quickly and encouraging the use of even-larger data objects. Unless cache miss ratios decrease in proportion to processor performance, the overall system performance will increasingly depend on I/O- subsystem performance. The problem is especially acute for read-intensive applications.

Access patterns for many applications are predictable. This predictability can be used directly by the application to initiate asynchronous I/O accesses. This sort of explicit pre-fetching may sometimes cripple the resource management. First, the extent to which an application needs to pre-fetch depends on the throughput requirements of the application. Second, asynchronously pre-fetched data may eject useful data from the data cache. We will discuss how this management is done.

LOOK-AHEAD SET

We have enhanced certain access methods and utilities in SQL Server, in such a way a task will be able to guess with reasonable confidence the set of pages that it will access in the near future. We denote this set as the current ’look-ahead’ set. This prediction is not required to be exact. It is not an error to include a page in the look-ahead set and never access it or to exclude a page that is accessed from the look-ahead set. But the efficiency of Parallel IO is positively correlated to the accuracy of this predication.

There are two major components in the current implementation of Parallel IO: (1)Governor and (2)policies like recovery, non-clustered index scans and sequential scans.

PARALLEL IO GOVERNOR

The main purpose of the Parallel IO Governor is to have basic asynchronous read infrastructure. The Governor eliminates duplicate IO requests on behalf of multiple users or multiple APF policies for one user. This frees the code that implements the APF policies to be liberal in issuing APF calls.

Parallel IO is on by default. Parallel IO is controlled by limiting the number of pre-fetched pages in each buffer pool. This limit can be expressed as a percentage of the size of the buffer pool. The Governor also ensures that the system never exceeds configuration limits.

PARALLEL IO POLICIES

Recovery

Recovery scans the log sequentially to determine the last log page. We use Parallel IO at this stage to do some speculative pre-fetch to bring all the pages belonging to the log to cache. Since the log page chain is most likely to be sequential in nature, we expect the speculative pre-fetch to be really effective.

For each complete transaction, the Parallel IO look-ahead scan issues a Parallel IO request for the page referenced by every log record. Not all log records generate Parallel IO requests, since a Parallel IO request is only a hint and not a directive. In particular, log records corresponding to infrequent operations are not pre-fetched to reduce implementation effort. Initial results show more than 10 times performance improvement during recovery, in some cases.

Non-clustered index scan

In the case of non-clustered index scan where the query is not covered by the index, the access methods compute the list of qualifying rowid's from the leaf level of the non-clustered index. The Parallel IO policy for non-clustered index scan uses the set of pages referenced in these rowid's as the look-ahead set(hint) and issues Parallel IO requests on the corresponding pages. In addition, scanning of the leaf-level of the index can itself benefit from Parallel IO. However, we consider that a special case of Parallel IO policy for Sequential Scans.

Sequential Scans (Uses Speculative Pre-fetch)

Sequential scans of page chains are involved in (1)table scans (2)joins that scan the inner table in ascending order of the join index and (3)clustered index scan (for the leaf level data pages).

Tables are stored in pages that are chained in a linked list. Thus, any look-ahead set of size greater than one must involve some speculation. The Parallel IO policy for sequential scans computes the look-ahead set every time the page chain moves from one allocation unit to another. (There is one allocation page every 256 pages). It adapts to crooked page chains by maintaining statistics on the physical reads and hops between allocation units seen by the current scan. The purpose of this adaptation is to compute a large look-ahead set when there is no fragmentation and compute a small look-ahead set when there is fragmentation.

PARALLEL IO CHARACTERISTICS

Parallel IO is on by default. Parallel IO is controlled by limiting the number of pre-fetched pages in each buffer pool. This limit can be expressed as a percentage of the size of the buffer pool. Execution does not depend on the completion of pre-fetch, since it is only a hint. Parallel IO will increase the number of buffer manager operations. The efficiency of Parallel IO is positively correlated to the accuracy in the prediction of the look-ahead set. A too-aggressive Parallel IO policy may cause an overload. This will be detected by the Parallel IO governor. On the other hand, an Parallel IO policy that does not issue Parallel IO requests far enough in advance will end up blocking on completion of physical reads. Parallel IO is intended to be self-tuning.

APF improves performance whether or not the IO system is saturated. When the IO system is saturated, APF can batch IO requests and minimize disk head seeks. When the IO system is not saturated, APF can drive multiple disks concurrently.

5 Summary

Sybase Adaptive Server Enterprise 11.5 release includes a number of parallel features. We have described parallel query, parallel DBCC and parallel recovery in this paper. These parallel features work in the SMP environment.

They share the worker thread manager and asynchronous pre-fetch, which are implemented as common software components.

Acknowledgements

We'd like to acknowledge the extensive contributions of a number of past and present participants of the SQL server development team that built this parallelism release of Sybase SQL Server.

References

[DG] D.J.DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6):85-98, June 1992.

[Syb] Sybase System 11 SQL Server documentation.

[RDC] T.K.Rengarajan, Lucien A. Dimino, Dwayne Chung. Sybase System11 Online Capabilities. *Data Engineering Bulletin* Volume 19(2):19-24, June 1996.

CALL FOR NOMINATIONS

IEEE COMPUTER SOCIETY

TECHNICAL COMMITTEE ON
DATA ENGINEERING (TCDE)

The Technical Committee on Data Engineering (TCDE) is holding an election for Chair in **September 1997.** The current Chair, Rakesh Agrawal, has completed his term. In preparation for the election, TCDE is conducting a ***call for nominations*** for the position. After nominations are received, a ballot will be sent to all TCDE members to vote for the next TC chair, based on the slate of candidates. Candidates will be asked to submit a brief biography and position statement (up to one page in length) for inclusion with the election ballot.

CALL FOR NOMINATIONS
Term: 1998-2000

Please list those colleagues who you wish to nominate to run for the position of TC Chair for TCDE. (Those running for TC Chair must be members of the Computer Society.) Please provide a contact number for us to reach them. You may nominate yourself.

	NAME	PHONE/EMAIL
1.	______________________	______________
2.	______________________	______________
3.	______________________	______________

Your Name:____________________ Phone:________________
Email:________________

Please email or mail your nomination to arrive by August 30, 1997 to:
Prof. Amit P. Sheth
[TCDE Election Committee Chair]
LSDIS Lab, University of Georgia
415 GSRC, Athens GA 30602-7404
e-mail: amit@cs.uga.edu, http://lsdis.cs.uga.edu/~amit

RETURN BY AUGUST 30, 1997

VLDB '97

Preliminary Call for Participation

23rd International Conference on Very Large Data Bases

Astir Hotel, Athens, Greece

August 26-29, 1997

PROGRAM CHAIRS
GENERAL PC CHAIR
Matthias Jarke, RWTH Aachen, Germany
EUROPE
Klaus Dittrich, Univ. of Zurich, Switzerland
AMERICAS
Mike Carey, IBM Almaden Res. Center, USA
MIDDLE- AND FAR-EAST
Fred Lochovsky
Hong Kong U. of Sc. and Tech.,Hong Kong

INDUSTRIAL/COMMERCIAL TRACK CHAIR
Pericles Loucopoulos,UMIST,United Kingdom

TUTORIAL CHAIRS
Yannis Ioannidis, Univ. of Wisconsin, USA
Tamer Ozsu, Univ. of Alberta, Canada

PANEL CHAIRS
Ramez Elmasri,U. of Texas at Arlington, USA
Arie Segev, Univ. of California, USA

GENERAL CONFERENCE CHAIR
Yannis Vassiliou, NTUA, Greece

ORGANIZING COMMITTEE
John Mylopoulos, U. of Toronto, Canada
Nick Roussopoulos, U. of Maryland, USA
Timos Sellis, NTUA, Greece

AREA COORDINATORS
EUROPE
Joachim Schmidt, U. of Hamburg, Germany
AMERICAS
Jim Gray, Microsoft, USA
MIDDLE- AND FAR-EAST
Ron Sacks-Davis, CITRI, Austialia

VLDB ENDOWMENT LIAISON
Stefano Ceri, Politecnico di Milano, Italy

PUBLICITY CHAIR
Panagiotis Georgiadis, Univ. of Athens, Greece

For more information, please send email to vldb97@dbnet.ece.ntua.gr, or visit the home page of VLDB'97 at http://www.dbnet.ece.ntua.gr/VLDB97/ or http://SunSITE.Informatik.RWTH-Aachen.DE/VLDB97/

You can also contact the VLDB Secretariat at the following address:

VLDB Secretariat (c/o Prof. Timos Sellis)
Department of Electrical and Computer Engineering,
National Technical University of Athens
Zografou 15773, Athens
GREECE

Conference History

For 23 years, VLDB has served the database community as its major truly international conference all over the world. Each year the conference is held in a different location. In the recent past, VLDB conferences have been held in locations such as Bombay, India (1996), Zurich, Switzerland (1995), Santiago, Chile (1994), Dublin, Ireland (1993), Vancouver, Canada (1992), Barcelona, Spain (1991) and Brisbane, Australia (1990).

From an arena where just research results are presented, VLDB has evolved to a forum where researchers and practitioners exchange ideas and experiences, without sacrificing quality ensured by rigorous refereeing. The VLDB '97 conference will continue and strengthen this trend.

Conference Highlights

The conference will cover several tracks.

- **Research Track.** A high-quality scientific program consisting of the presentations of the best in recent research, selected from a large submission from leading researchers across the world.
- **Tutorials.** Tutorials on challenging developments in databases conducted by leading scientists. These are targeted at practitioners as well as researchers.
- **Invited Talks.** Invited talks are given by leaders in the field illuminating recent trends, and in some cases providing enlightening retrospective accounts as well.
- **Industry and Applications Track.** Industrial sessions identifying problems and describing solutions from a practical point of view. Experts from the leading database companies take part in these sessions.
- **Panel Discussions.** These provide a forum for industry and academic experts to discuss current issues in database technology.

The VLDB conferences offer a lot to everyone in the area of databases, including *practitioners, implementors, researchers,* and *students*. The tutorials are of particular interest to practitioners and students.

Conference Location

VLDB '97 will be held near Athens, Greece, a place equally famous for its historical attraction, beautiful landscape and weather, lively database community, and hospitality. This is the first time that VLDB comes to a European country in the South East Mediterranean region - the cross-roads of Western and Eastern Europe, Middle-East and Africa. The conference will serve as a catalyst for the development and dissemination of database technology in this region and will also offer inroads into new markets in Eastern Europe and the Middle-East.

Registration Information

The conference fee is US$ 450 (regular, after Aug. 1, 1997), US$ 400 (regular, before Aug. 1, 1997), and US$ 225 (student). For regular participants, the fee includes: admission to all conference sessions, including all tutorials, conference proceedings and tutorial notes, conference banquet, and lunches and coffee breaks on all days. The reduced student fee does not include the conference banquet.

Hotel Information

The conference hotel is Astir Hotel, one of the most famous hotels in the Athens area. Room rates are approximately US$ 140 for a single room and US$ 160 for a double. In other nearby hotels, prices are between US$ 90 and US $130 depending on the type of room.

The gods, who, as they say, never left Athens, are waiting for you in Greece!!

More detailed information about the conference as well as registration forms, hotel reservation forms and travel information will be available soon through the conference home page.