OTM SPECTRUM Reports

Open Transaction Management -
Strategies for multi-platform environments in the 1990s

"In our analysis two or three things have to change. The first is that we have to decide that we are serious about distributed processing. In my opinion, the approach that most of the database vendors are taking today — to the issue of distributed database — is revealing. It is as if they are trying to wish distributed processing away."

David Vaskevitch
Director of Enterprise Computing
Microsoft Corporation

Volume 8 Report 1 February 1994

cessing, messaging and queuing and database are needed. But they are needed in ways not currently delivered. He sees this as Microsoft's opportunity.

For a comparison, see page 29 where Keith Hospers of Informix describes how database vendors are assuming a transaction role. That Informix is climbing the transaction ladder is clear. He almost delivers a knockout blow when he remarks how little competitive advantage Informix and its partners have obtained by adding transaction monitor links.

At the same time he confirms that growing up (from PCs and workstations and LANs) — rather than moving off (and down) from hosts — is where the action is. It is no accident that Informix has 3000+ vendors selling applications on its databases, with over 300,000 licences sold. This may be a small percentage of 500M but it is an order of magnitude larger than all mainframe transaction processors ever sold. And this is from only one vendor; think about adding Oracle, Sybase/Microsoft, Borland, ...

Other threats

The Microsoft and Informix views serve notice of intentions to force change. Elsewhere in this Report two different threats are discussed, namely:

- time independence
- transactional OO.

Time independence is hardly novel, either technologically or for management. Nevertheless much of the early 1990s computing model seems to depend on explicit connections — that updates are committed all at once or not at all. In centralized systems this was achievable. In distributed systems, using facilities like two-phase commit, this becomes much harder to deliver. Some even claim it is impossible.

One answer will be messaging and queuing. It enables time independence (or 'almost real time', if that is what is needed). Add this capability to OO and to transaction systems and flexibility can be brought to distributed systems.

Although there is much talk is about OO and transactions (page 41 and 47), the probability is that messaging and queuing, or transactional messaging, will steal the limelight in the late 1990s (page 16). The reason is that it is applicable in most distributed enterprises. Who will exploit this remains to be seen.

Management conclusion

When looking at the computing industry for 1994 some trends emerge:

- traditional OLTP is being marginalized on the mainframe
- the application developers' focus is moving to selecting the right database
- the need for time independent processing in the distributed enterprise will steadily assume greater importance.

Change is needed. It is coming. But not from those with traditional IS skills or products. That is what 1994 currently looks likely to bring.

CONTENTS

Volume 8 Report 1, February 1994

Requirements for today's database server architectures — Part I

Management introduction

Current business and technology trends are driving the database industry toward a certain class of architectural features. Business organizations and computing technologies have traditionally fostered an artificial division between OLTP, batch processing and decision support. Typically Information Systems (IS) departments responded to customer service and business management needs by decomposing business requests until they could be processed in an OLTP environment, a batch processing environment or a decision support environment.

Today's businesses are increasingly information and service oriented. They need the ability to unify these classes of data processing, creating an integrated class of data processing.

This suddenly popular class of data processing is called on-line complex processing (OLCP). OLCP-capable platforms can provide businesses with the ability to:

- answer customer requests
- manage the business directly
- provide this in a single, integrated environment.

The single most critical factor in an OLCP platform is the database server architecture. Part I (of two analyses) describes the architectural features which a database server (and architecture) must possess in order to meet today's trends and opportunities.

Key developments

At the same time that business has begun to recognize the importance of OLCP, technology has been making rapid advances. On the one hand new hardware technologies have made lower cost, higher powered computers readily available plus network speeds have improved and memory costs carry on decreasing. On the other hand, while disk capacities have continued to increase, disk speeds have not kept pace.

Yet by far the most important change in computer architectures has been the move toward using multiple processors to increase computing power. Although platforms today are dominantly uniprocessor, three hardware architectures are becoming increasingly important:

- Symmetric MultiProcessing (SMP) — the most common form of tightly coupled multiprocessor systems; that is, systems which share main memory services and, usually, disk storage
- loosely coupled multiprocessor systems — systems which may share disk storage but have separate main memory services such as are found in clusters
- Massively Parallel Processing systems (MPP) — systems which combine hundreds or even thousands of CPUs.

SMP as the mainstream

Although all three of these systems have had some success — and can be expected to become increasingly important — SMP systems represent the mainstream in the near future. The main reason for this move toward SMP is low-cost scalability.

Various software trends are now beginning to take advantage of SMP capabilities:

- platform downsizing has become commonplace
- communications standards have enabled a wide variety of distributed computing (for example, client server computing takes direct advantage of the ability to interconnect systems and divide the work load)
- independent scaling of the hardware (without adding to the administrative burden or requiring rewriting of applications) has been found to be key to the support of both client and server processes.

All of these are well supported by SMP systems.

Yet, for all the benefits available now from these changes in hardware, software (especially operating systems and DBMSs) must be written to take advantage of the real power of SMP. They must not limit the ability to use uniprocessor systems today or to migrate to loosely coupled, uncoupled or MPP systems tomorrow. When used efficiently, SMP systems can provide scalable, OLCP-capable platforms at relatively low-cost.

Key requirements for database server architectures

Modern database server architectures are driven by four key requirements:

- scalability
- performance

- OLCP
- availability.

Each, along with the key factors which differentiate database server architectures, is discussed below.

Scalability

Scalability is the property of a system which permits predictable support of additional users, higher performance, greater throughput, etc:

- by adding computing resources
- without changing the application or administrative practices.

There are two ways of scaling a database server:

- horizontal
- vertical.

Horizontal scaling can be achieved when multiple servers interoperate transparently and share the workload. This method of scaling will become steadily more popular as loosely coupled systems and distributed databases are better supported. The downside is that it usually requires additional administrative support.

Vertical scaling, in which a single server is scaled up, can be achieved when computing resources — such as faster or additional CPUs — can be added to a platform to improve response time and throughput. Database server support for vertical scaling should not require the addition of extra software modules because this increases administration complexity and decreases predictability.

Whether the horizontal or vertical scaling is used, its effectiveness depends on how well the database server software uses the available resources.

This analysis focuses on vertical scaling because it is the method which is seeing the most rapid growth today.

Scalability is an important goal today for two primary reasons:

- with business requirements changing so rapidly, it is no longer possible to perform the kind of rigid and time-consuming long-term capacity planning that was once promoted in MIS: instead an incremental approach is required
- with technology changing equally rapidly, new capabilities — and lower hardware costs — are always appearing.

Without an incremental approach, businesses cannot afford to take advantage of new technology without depreciating older technology.

Scalability at every level is crucial. In today's business environment, this often means that older businesses cannot compete with newer ones because of outdated capital investments. These facts require that both the hardware and the software, and especially the RDBMS, be scalable.

The key enabling factors applicable to scalability are:

- multiprocessor support
- architectural scalability.

Multiprocessor support

SMP systems are used increasingly to achieve vertical scaling because they permit the addition of CPUs without changing the entire platform. While MPP can also be used to achieve vertical scaling, this approach is currently being used only in specialized applications.

Two key issues for an SMP database server architecture are extensibility and transparency. Extensibility demands that the database server architecture not be specialized to any particular number of CPUs; it should be equally capable of supporting one or half a dozen CPUs:

- without reinstallation
- without changes to off-line configuration parameters
- without additional software options.

Such an architecture will be equally useful and efficient if the platform consists of one CPU (a uniprocessor), multiple processors (SMP) or even many processors (MPP). The user should not have to purchase separate products specialized to a type of platform.

Without an incremental approach, business cannot afford to take advantage of new technology without depreciating older technology.

In contrast transparency demands that the database server architecture be able to hide changes on the platform architecture from applications. In particular, a uniprocessor or SMP system might use shared memory for communication, while a loosely coupled system might use messaging.

Applications should not need to be changed if the platform changes or if the communication mechanism changes. Data manipulation and the database API should remain the same.

Efficient support for multiprocessing requires that the database server be capable of scheduling tasks to use the available resources. This can be achieved by scheduling requests sequentially or by dividing requests into subtasks which can be performed in parallel.

Properly implemented, the latter approach is the most efficient and the benefits will not depend on application type. The granularity at which load balancing and task scheduling occurs has a strong impact on efficiency:

■ if the granularity is too high (for example, per SQL statement), the collection of CPU and memory resources will not be shared efficiently (CPU resources may not be used while waiting for disk I/O)

■ with lower granularity, sharing of resources can occur within a single request and — even more efficiently — for multiple concurrent requests.

Parallel disk I/O enables the database server to make efficient use of multivolume tables and table partitioning.

Architectural scalability

Regardless of the degree of portability or support for standards and parallelism, a database server architecture that has built-in limitations will not scale. Limitations on:

■ table sizes

■ database sizes (in bytes as well as number of tables)

■ log sizes

■ number of concurrent connections

■ memory (buffer) sizes

■ numbers of users

■ etc.

are just as constraining as a DBMS that will not support multiple platform configurations.

Factors which limit query complexity — especially those which control the depth of recursion, such as stack size — should be managed dynamically and not require system shutdown. Replacing server hardware with a more powerful configuration will have no effect if these limits are internal to the DBMS.

Common architectural bottlenecks also appear in the form of the inability to dynamically tune the database server. The ability to tune — the amount of memory, the number of CPUs, the number of concurrent threads of execution (whether actual threads, processes, or virtual processes) and the number of not-necessarily-disjoint table partitions (called fragments by C. J. Date) and their disk distribution — without having to shut down and restart the database server should all make it possible to respond to changing application requirements.

Ideally, each of these will change dynamically within user set limits. Support for subqueries and cascading triggers are obvious examples of the need for recursion.

Performance

Two of the main methods of achieving higher performance — given today's hardware and operating system software — are:

■ support for parallelism and parallelized algorithms

■ multi-threading.

Parallelism and parallelized algorithms

One way of achieving higher performance in a database server is through parallelization of algorithms. There are three types of parallelization which need to be addressed by a modern database server architecture:

■ parallel disk I/O

■ parallel utilities

■ parallel query processing.

Parallel disk I/O enables the database server to make efficient use of multi-volume tables and table partitioning. Support for physical table partitioning is particularly important for parallel disk I/O. It greatly improves efficiency and resource management when properly implemented.

Parallel utility operation (sorting, index building, load, backup, and recovery) all involve parallel processing as well as parallel disk I/O but typically have only a few component operations that can be parallelized.

By contrast with parallel disk I/O and parallel utilities, parallel query processing is much more complex. Processing a query normally involves invocation of a number of atomic database operations.

The composition and sequence of these operations depends on the specific query and the execution plan selected by the query optimizer. Ordinarily, database operations are performed in a strict sequence, with the output of one operation feeding the input of the next.

Essentially, parallelization is a divide and conquer approach, like assigning multiple workers to a single task by dividing the work and thereby completing the task sooner.

For a database server to support parallel query processing, vendors must select atomic database operations which can:

- be replicated and then can process different portions of the data concurrently
- combine the results as though a single thread of execution had performed the operation.

Meeting this goal requires that queries be decomposed and processed independent of the communications architecture necessitated by any particular hardware configuration. (Creating multiple copies of a particular database operation which can run in parallel is called horizontal parallelism.)

If each of the database operations is designed so that it is demand driven (a data flow approach), the database server can also achieve parallelism by running different database operations concurrently. For example, a database operation which selects records based on the value of a field should run concurrently with a database operation which accesses records from disk and need not wait until all requested records have been read.

Similarly, selected records can be fed to a sort routine for initial partitioning according to the sort key while both the selection operation and the access operation continue. In addition to independence of the implementation from the degree of parallelism, a demand driven design also permits transparent incorporation of new intrinsically parallel algorithms. This makes it easier for vendors to take advantage of state-of-the-art algorithm research to improve performance.

This approach, much like instruction pipelining in modern CPUs, is called vertical parallelism. The combination of horizontal and vertical parallelism increases overall throughput and decreases response time compared to the usual sequential processing approach. The degree of improvement is speeded up.

Multi-threading

The method of handling multiple user requests that has the lowest overhead is true multi-threading (as distinguished from simulated multi-threading). A thread is a unit of context management under the control of a single process, and can be:

- either implemented within the process (the database server)
- or via operating system services.

An operating system-level process context switch is considerably more costly (in system terms) than a thread-level context switch. The same is true for process creation and destruction compared to thread creation and destruction.

In principle threads can also perform concurrent tasks by cloning themselves, creating subthreads much like subprocesses. Because the operating system need not create, schedule or terminate multiple processes, overhead for a multi-threaded database server is lower than for other architectures.

Thus a true multi-threaded architecture provides a higher degree of resource sharing — and tends to make performance more stable with respect to the numbers of users enabled — than do shared multi-process architectures. In addition, multi-threading can also provide a higher degree of module independence since the execution of logical operations can be event-based rather than control flow-based. This means that server code can be expected to be more stable when new functionality is added.

By completely managing all the resources needed by the RDBMS — including buffer space, disk space, and locking — a multi-threaded database server architecture is essentially a dedicated operating system which schedules thread execution. The scheduler can be either preemptive or non-preemptive:

- in a non-preemptive scheduler threads execute until they signal the scheduler that they are willing to give up the processor (the scheduler frequently uses a round-robin algorithm which permits each thread to run until it blocks or until a period of time known as a time slice has elapsed); such scheduling is efficient for a dedicated use database server, since the number of types of threads which a process need execute are relatively few, can be tightly controlled and are highly predictable
- in a preemptive scheduler, threads can be interrupted by the scheduler based on, for example, the need for a thread with higher priority to begin executing; such a scheduler is efficient for database servers which must share resources with non-database applications.

On-Line Complex Processing

The evolution of systems toward OLCP is characterized by hybrid environments in which OLTP, DSS, and batch processing share common data processing resources and manipulate the same data (also see Figure 3). However it remains true that resource management, tuning and administration of hybrid environments is inherently more difficult than when a platform is dedicated to a particular type of processing.

In addition the different types of application supported often require conflicting system configurations. When successfully implemented as a hybrid environment, new applications themselves become hybrid. Over time queries and transactions in such environments change character and become more complex. The application design begins to address business concerns directly, and to reflect data processing solutions less.

DBMS vendors often architect versions of their products for different classes of application. For example, a database server which is dedicated to OLTP can depend on several conditions:

- transactions are typically of short duration
- transactions generally do not interfere with each other
- statements generally affect only a few rows
- only a few tables have many rows or are very volatile.

DBMS vendors that focus on OLTP take advantage of these facts by using physical methods to improve performance or by requiring applications to enforce noninterference of transactions. The query optimizer component of the DBMS need not be especially powerful nor need the data access and processing methods be capable of efficient access to and manipulation of larger amounts of data.

Nevertheless, within a hybrid environment, such systems suffer severe performance penalties. They often require unacceptable levels of database and application maintenance.

Managing memory consumption in a database server is as important as the process management overhead that is addressed by a multi-threaded architecture.

From this it can swiftly be seen that key factors involved in a database server architecture capable of supporting OLCP must include:

- optimization
- efficient resource management
- parallel query processing.

Optimization

The capabilities of a database server query optimizer determine, to a large degree, the ability of the RDBMS to perform efficient set processing. In particular the ability of the query optimizer to use an appropriate index for restricting the rows selected, or to manage queries which reference many tables, is important.

Optimizers should not be sensitive to SQL syntax. They should be sensitive to data statistics and data value distributions. In this way they can evaluate the expected cost of the possible query execution plans and then select the one with the lowest cost.

When set processing queries are processed, the various portions of the processing can be interleaved or scheduled to improve concurrency and resource usage.

In order to do this properly an RDBMS must have a scheduler which:

- cooperates with the optimizer
- takes into account the transaction isolation levels of all transactions which are executing concurrently.

Resource management

Efficient resource management consists of addressing two issues:

- transparent support for appropriate resources
- efficient use of particular resources.

Transparent resource support is crucial. For example, in developing a client server application, the designer cannot (and should not have to) make assumptions about how the client and server communicate. Whether a network or shared memory services are used should not affect the design, nor should the design constrain their use.

When a relational application resides on the same physical platform as the database engine, the application and the database engine can synchronize state information efficiently through shared memory. This is particularly important when managing locks and when buffering result sets for a cursor-driven interface.

Among the most important resources is memory. Managing memory consumption in a database server is as important as the process management overhead that is addressed by a multi-threaded architecture. RDBMS applications use memory to:

- maintain the state of a connection
- cache requests and results.

Some servers allocate and manage memory separately for each user. This means that data or index pages in memory for one user are not accessible to another user until they have been reread from disk.

Other servers consume undue amounts of memory for maintaining process context. This contrasts with multi-threading which uses minimal amounts of memory for context management.

Without efficient management of shared memory, a database server architecture cannot address the simultaneous needs of OLTP, DSS, and batch processing — let alone complex queries and transactions. For example, if a task blocks (waiting for some resource such as I/O), it is important that it does not continue to hold resources. Instead, the vehicle of execution (a thread or a process) should move on to execute some unblocked, executable task.

The portion of a task that must complete before it is willing to yield resources is therefore highly important. If it is at the level of a SQL statement, resources will be wasted whenever the task blocks. In an OLCP environment, an entire SQL statement in a batch process might prevent critical OLTP applications from meeting critical response time requirements. This is unacceptable.

Parallel query processing

Parallel query processing offers a solution to the problem of poor performance on complex queries or against large, volatile databases. By accessing and processing portions of the affected data in parallel, parallel query processing can greatly improve performance of DSS and batch processing.

Such improvements in performance make it possible to include complex queries in read write transactions without sacrificing data integrity or transaction isolation. With adequate response time, DSS queries that would not be possible in a traditional system:

- become possible
- need not compete destructively with OLTP applications.

On-line administration

Ideally, administration utilities support continuous operation. The system should reduce or even eliminate both planned and unplanned outages.

In practice, and too often, RDBMSs must be taken off-line in order to run some utility.

To provide high availability, utilities — such as database loading, backup, recovery, integrity validation, index reorganization, etc. — should all be executable on-line.

For example, if a failure occurs during such maintenance operations, the utility should not have to be restarted from the beginning of the operation. The utility might perform periodic checkpoints or even full journaling so that other utilities can continue processing from the point of failure.

Other capabilities such as on-line and automatic archival of full log files, automatic restart (no operator intervention), and controllable system restart times are also important. By minimizing the operations requiring the RDBMS to be taken off-line, availability is improved.

Even so, administration tasks — including monitoring and resource management — frequently interfere with availability. Ideally, utility operations on large tables — such as backup, load, index creation, and index reorganization — should not require that any data unit — such as a table, portion of a table, or the database — be made unavailable for either read or write access.

If it is necessary that the data unit be taken off-line, that fact should not interfere with access to other data units.

Similarly, physical database allocation and reorganization, space management, log archiving, and system restart should have a minimal impact on applications.

The database server should also have flexible monitoring capabilities which permit users to build custom utilities. Mainframe administration functionality — such as unattended and scheduled utility operations — is just as essential for a database server architecture.

Equally the ability to add additional tape drives as necessary and support for parallel utilities (backup, restore, load, etc.) are essential if a database server architecture is to satisfy to the high availability and performance requirements of OLCP.

Robustness

Robustness should reduce the importance of a particular failure and recover from it transparently. The RDBMS should provide both read and write access to data regardless of the circumstances, including hardware system or component failure.

Several different facets of robustness need to be incorporated in a database server architecture, including:

- redundancy
- replication.

Systems which provide such high availability are said to be fault tolerant. Fault tolerant systems often rely on various forms of redundancy. Two of the most important are:

- system hardware redundancy
- controlled data redundancy.

System hardware redundancy may involve completely redundant hardware platforms, standby processors, dual-ported disk drives and the like. A common form is hardware mirroring in which one disk drive is designated as a copy of another and protects against media failures.

System hardware redundancy — although important for fault tolerance — is not integrated with the requirements of a database server. It is unresponsive to transaction boundaries, atomic read/writes, and generally the DBMS cannot take advantage of this redundancy to improve performance.

Controlled data redundancy occurs in two forms:

- software mirroring
- replication.

Software mirroring — also called duplexing or multiplexing — can simultaneously:

- protect against hardware failures
- be used to optimize performance.

It is the process by which the DBMS duplexes a unit of data (table space, dbspace, chunk, or logical device), usually on a different physical device. In particular, the copy is defined as the mirror and any writes are transparently written to the mirror. Read operations can be distributed between the primary device and its mirror, leading to improvement in performance as a side effect of mirroring. (Of course, mirroring may be of little value if the copies are located on the same physical device.)

In the event that a device is corrupted — for example by media failure — the damaged copy should automatically:

- be taken logically off-line
- have all reads and writes directed to the remaining undamaged copy.

The key is that no interruption should be perceived by the user when this occurs. Once the damaged device is repaired or replaced, it can be brought back into synchronization with the undamaged copy by a process known as on-line remirroring.

Replication is similar to mirroring — except that the copy may be remotely located and need not be synchronously updated. It may even copy an entire database. When used to copy an entire database, its purpose is usually to create a 'warm standby'. However, some implementations add another benefit — they make the standby copy available for read-only access.

Replication can provide significant benefits to hybrid environments. For example it can permit DSS, report generation and other read-only batch applications to have access to data on a standby system while OLTP applications update the primary system.

Management conclusion

This analysis examines key requirements for a modern database server architecture. These requirements help identify the key features which a truly OLCP database server architecture must support. They can be summarized as:

- multi-threading — for performance, resource usage, and scalability via support of multiprocessing
- parallelism — for scalability, performance, and resource usage (efficient extensibility across hardware architectures)
- optimization — for higher performance and support of more complex queries and transactions
- controlled data redundancy — for higher availability and improved performance
- system redundancy — for higher availability of the entire business system through fault tolerance
- on-line administration — for higher availability
- OLCP support — for efficient support of hybrid environments with concurrent OLTP, DSS, and batch applications

As users recognize the importance of supporting either a mixed application environment from a single database server or more complex queries (the two predominant characteristics of OLCP), it is probable that the underlying database server architecture will become the dominant factor in product selection.

This is Part I of two analyses; the second will appear in the May 1994 edition of OTM Spectrum Reports *about database server architectures. Part II will examine how the necessary features are provided by three major database server products:*

- *the INFORMIX OnLine Dynamic Server*
- *Oracle 7*
- *Sybase's System 10.*

Both were written by David McGoveran of Alternative Technologies (Boulder Creek, CA 95006). Both are drawn from An Evaluation of Three Database Server Architectures, *published in 1993 by Alternative Technologies.*

Replication is similar to mirroring — except that the copy may be remotely located and need not be synchronously updated. It may even copy an entire database. When used to copy an entire database, its purpose is usually to create a 'warm standby'. However, some implementations add another benefit — they make the standby copy available for read-only access.

Replication can provide significant benefits to hybrid environments. For example it can permit DSS, report generation and other read-only batch applications to have access to data on a standby system while OLTP applications update the primary system.

Management conclusion

This analysis examines key requirements for a modern database server architecture. These requirements help identify the key features which a truly OLCP database server architecture must support. They can be summarized as:

- multi-threading — for performance, resource usage, and scalability via support of multiprocessing
- parallelism — for scalability, performance, and resource usage (efficient extensibility across hardware architectures)
- optimization — for higher performance and support of more complex queries and transactions
- controlled data redundancy — for higher availability and improved performance
- system redundancy — for higher availability of the entire business system through fault tolerance
- on-line administration — for higher availability
- OLCP support — for efficient support of hybrid environments with concurrent OLTP, DSS, and batch applications

As users recognize the importance of supporting either a mixed application environment from a single database server or more complex queries (the two predominant characteristics of OLCP), it is probable that the underlying database server architecture will become the dominant factor in product selection.

This is Part I of two analyses (the second will appear in the May 1994 edition of OTM Spectrum Reports) *about database server architectures. Part II will examine how the necessary features are provided by three major database server products.*

- *the INFORMIX OnLine Dynamic Server*
- *Oracle 7*
- *Sybase's System 10.*

Both were written by David McGoveran of Alternative Technologies, Boulder Creek, CA 95006. Both are drawn from An Evaluation of Three Database Server Architectures, *published in 1993 by Alternative Technologies.*