INFORMIX On-Line Dynamic Server

By David Clay

The INFORMIX On-Line Dynamic Server 7.1 on AIX 4.1 was released in December 1994. This server represents the culmination of Informix's efforts to build a new server architecture aimed at maximizing DBMS performance and utility on Symmetric Multiprocessor (SMP) machines. On-Line Dynamic Server 7.1 features Dynamic Server Architecture with Parallel Data Query (PDQ). Version 7.1 functionality includes parallel scans, joins, aggregates, index builds, backup, restore, and recovery. It also includes a mechanism for horizontal table partitioning and global resource control for decision-support queries.

The Informix SMP capability is not just an extension to an old architecture: it is a fundamental redesign of the server. A look at the evolution of Informix servers, beginning with Version 5.0, provides a better understanding of the advantages of the INFORMIX On-Line Dynamic Server 7.1 architecture.

On-Line 5.0 Server Architecture

The On-Line 5.0 engine is SMP-enabled to benefit many Online Transaction Processing (OLTP) applications. Version 5.0 architecture has one operating system process for each connected client, shown in Figure 1. The SMP operating system can schedule several of these server processes simultaneously, thereby achieving some level of parallelism. The engine processes coordinate their access to shared resources, such as the buffer cache and logging services, using a high-performance mutual-exclusion mechanism. This works well, as shown by the large number of published TPC benchmark results using INFORMIX On-Line 5.0. For example, in October 1993, INFORMIX On-Line 5.0 achieved a TPC-C rate of 726.13 tpm on a RISC System/6000 (RS/6000™) Model 590 running AIX.

David Clay

Of course, some drawbacks to this architecture exist. One is that the number of operating system processes increases as the number of clients increases. This results in an increase in process context switches. A second drawback is that a single-user session cannot use more than one processor to work on a large query.

Dynamic Server Multithreaded Architecture

Version 6.0 of On-Line Dynamic Server addresses the first drawback. Dynamic Server Architecture (DSA) is a multithreaded architecture in which many threads are multiplexed over a few server processes. The number of server processes is typically smaller than the number of physical processors on the SMP machine, as shown in Figure 2. Each user session gets a thread that can be scheduled on one of the Informix server processes rather than getting its own dedicated process.

A thread has a much smaller context than a process, and thread switches can be done without entering the operating system kernel. Thus, a thread-context switch time is a small fraction of a process-context switch time. This is advantageous because Database Management Systems (DBMSs) spend much of their time context switching.

This time savings represents only part of the advantage of multithreading. More significantly, the server can schedule threads according to the best use of the server. For example, suppose 1,000 clients are each supported by one thread in the server. If one thread obtains an exclusive lock on a critical centralized resource such as the log buffer, a DSA server process can guarantee that the thread will not be context-switched while holding the critical resource.

This guarantee cannot be made in a DBMS which implements this using 1,000 server processes. From time to time, the operating

Two Processes Per User

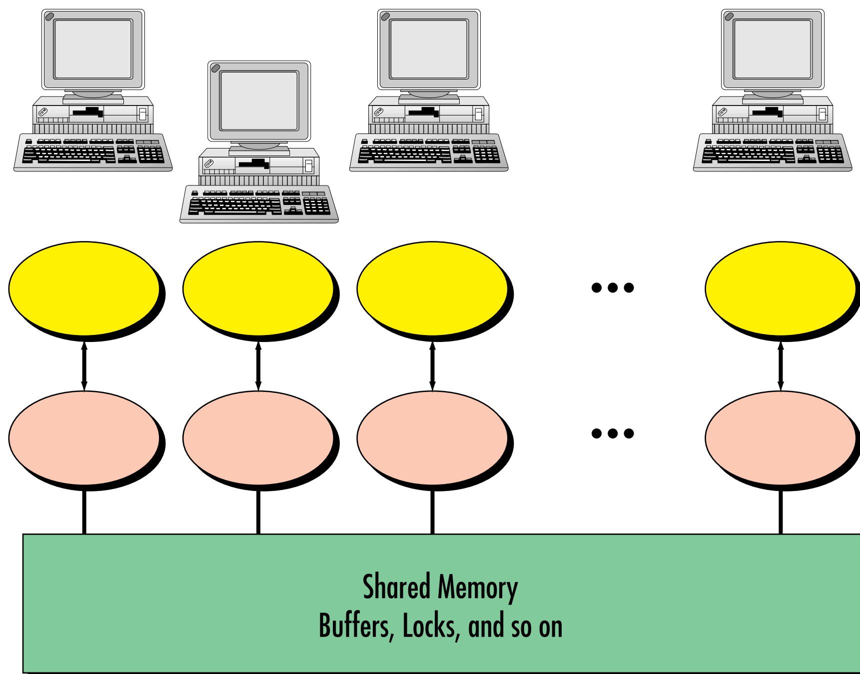

Figure 1. Two processes per user

system will time-switch a process holding a critical lock. The processes allowed to run in its place will simply note that they cannot obtain the lock and give up the processor to the next process, which does the same, and so on. As the number of user sessions increases, the problem becomes worse. Some, but not all, of these problems can be alleviated using a Transaction Processing (TP) monitor in a process-based system. This requires a considerable investment in TP software, additional application development time and complexity, and additional administrative costs.

Figure 3 illustrates the performance of On-Line 5.0 (a process-based server) versus On-Line Dynamic Server 6.0 (a thread-based server) as the number of user sessions increases. With On-Line Dynamic Server 6.0, the throughput remains stable as the number of clients increases.

Informix's Dynamic Architecture has been shown to be scalable for OLTP workloads; that is, physical processors can be added to handle larger workloads so that the processing time remains the same. This helps to fulfill one of the promises of SMP: when a system runs out of horsepower, capacity can easily be expanded by plugging in more processors. SMP scalability is difficult to achieve in a DBMS because of many internal concurrency problems that arise, particularly as the number of physical processors becomes large.

In addition to a radically new architecture, On-Line Dynamic Server 6.0 also included some new performance features: parallel backup, parallel recovery, parallel index build, configurable read ahead, and dictionary cache.

The query engine was completely rewritten for Version 7.1.

PDQ Architecture

The multithreading subsystem that made its debut in Version 6.0 is the ideal platform on which to build a parallel query capability. Thread creation, destruction, and synchronization is much more efficient than the corresponding functions for operating system processes, making it practical to quickly map several threads to a query. However, many other structural changes to queries are required before the thread mechanism can be used effectively on a single query. The iterator model and the exchange iterator are two such structural changes.

The Iterator Model

The query engine was completely rewritten for Version 7.1 to conform to an iterator model. *Iterators* are self-contained software objects that

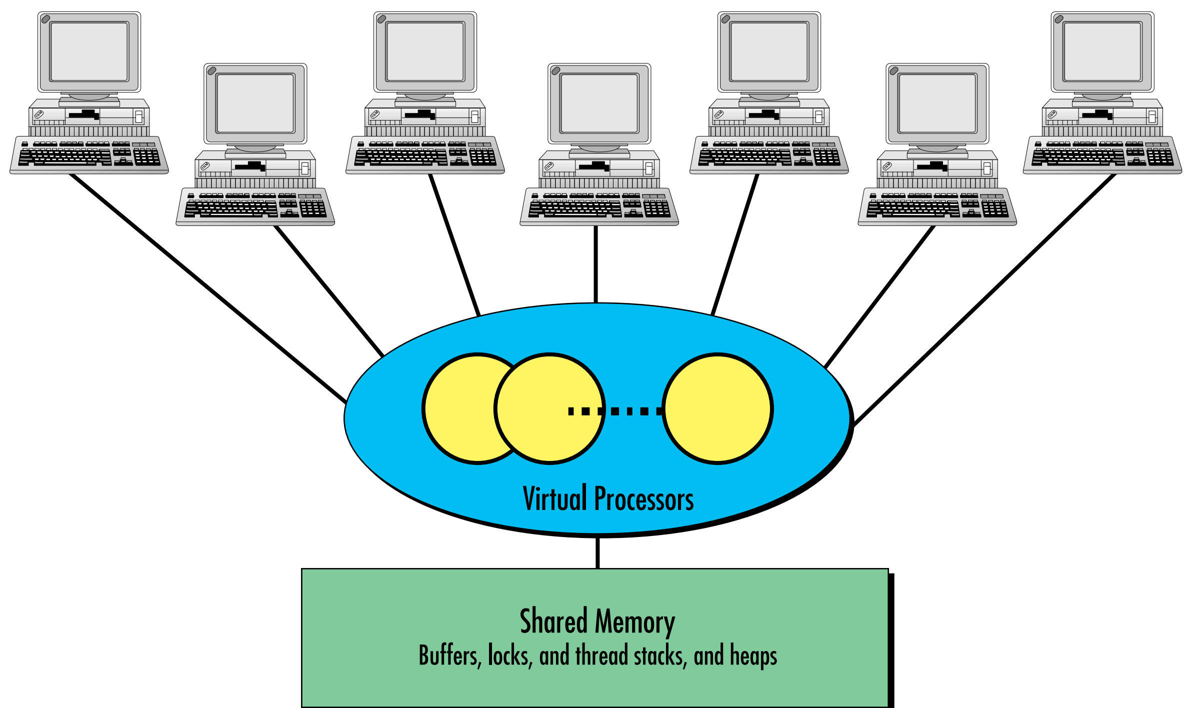

Figure 2. Multiprocess and multithreaded architecture

accept a stream of tuples from one or two anonymous sources, usually another iterator or iterator pair, and produce a transformed stream of tuples. Specific iterators are scan, nested loop join, sort merge join, hash join, union, sort, and aggregate. The iterator model allows complex queries to be structured by stringing together the appropriate iterators into a tree structure as shown in Figure 4. It also facilitates parallelism because iterators can be easily duplicated to make a new instance of the iterator. The OnPerf Graphical User Interface (GUI) monitoring utility that ships with the server actually draws the iterator tree and allows the user to view the flow of tuples through the tree as the query executes.

New iterator algorithms were also introduced in Version 7.1. Aggregation (count, sum, avg, min, max) is done using a unique hashing algorithm that is much faster than the previous algorithm. Hash joins have been slow to appear in commercial products. Hash joins are frequently much faster than sort merge joins, because they only require that a hash table be built on the smaller of the two inputs. The larger input does not need to be organized in a particular way. Sort merge joins require that *both* inputs be sorted, which can be very costly.

The scan iterator, used to read table data from the disk, has many improvements over earlier systems such as “light scan” functionality, which often allows full-table scans to bypass the buffer cache. Predicates are “pushed down” and evaluated directly in the disk buffer. Using these methods, tables can be sequentially scanned at maximum hardware disk rates, typically in the 2 MB to 5 MB per-second range. This is unusually fast for a Relational DBMS (RDBMS).

The Exchange Iterator

The mechanism for launching and managing parallel threads is the *exchange iterator*, first described as part of Goetz Graefe’s experimental Volcano DBMS. A special exchange iterator can initiate new threads and copy iterator instances into the new threads. It can also perform pipe fitting between iterator instances and manage the data flow using buffering and back pressure to equalize the speed between producer and consumer threads. This pipe fitting includes a repartitioning mechanism described below and shown in Figure 5.

Data Partitioning

To parallelize queries, tables and intermediate results must be divided into partitions. The DBA partitions permanent tables using extensions to the `create table` statement. Internal temporary tables are partitioned by the DBMS. The Informix terminology for horizontal partitioning of tables across many disk devices is *fragmentation.* Tables can be fragmented by round robin, a range of key values, or an arbitrary series of expressions. Indexes can be fragmented in the same way as tables. The SQL syntax for creating tables and indexes and altering tables is extended to allow for establishing and changing fragmentation schemes. Figure 6 shows examples of the syntax.

Table or index scans are parallelized by starting a scan thread for each fragment. Another advantage to table and index fragmentation includes logical partitioning by range or expression. This allows the query engine to eliminate certain fragments based on the `where` clause of a query. For example, if a table is partitioned by entry date and a query searches for a range of entry dates, not all of the table's disks will need to be scanned.

A fragment is an ideal unit of backup—another advantage of fragmentation. When a disk crashes, only the affected fragment must be restored, not the entire table. The unaffected fragments remain structurally intact, and queries may proceed against the table even though part of the table is unavailable. SQL syntax is provided to skip unavailable fragments on read-only queries.

Fragmentation can be used to accommodate very large tables. It is not practical to store a single table of several hundred gigabytes as a single structural entity, because it sometimes forces the DBMS to read, update, or restore the table as a unit.

Partitioning of Intermediate Results

Depending on the requirements of an individual iterator, tuples may need to be repartitioned on-the-fly. The exchange iterator does this. For example, when a hash join is parallelized, there will be several instances of the hash join iterator, each with two parallelized inputs. In the simplest case, these inputs will be table scans. The results of the parallel scans should be divided among the join threads so that each unique join key value always goes to the same join thread. This is done by a highly efficient hashing algorithm that is built into the exchange iterator. Exchange takes tuples from the scan thread, then hashes on the join key to

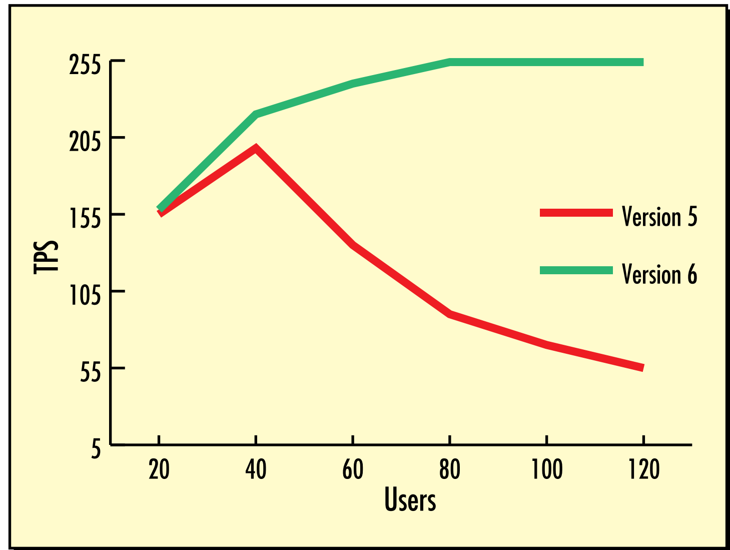

Figure 3. OLTP scalability

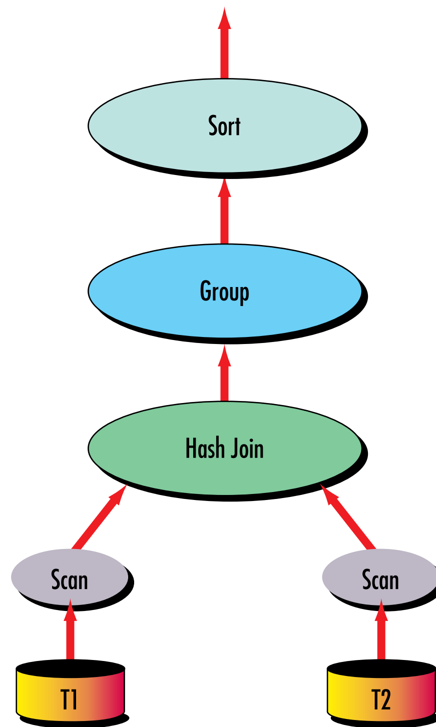

Figure 4. Sample iterator tree

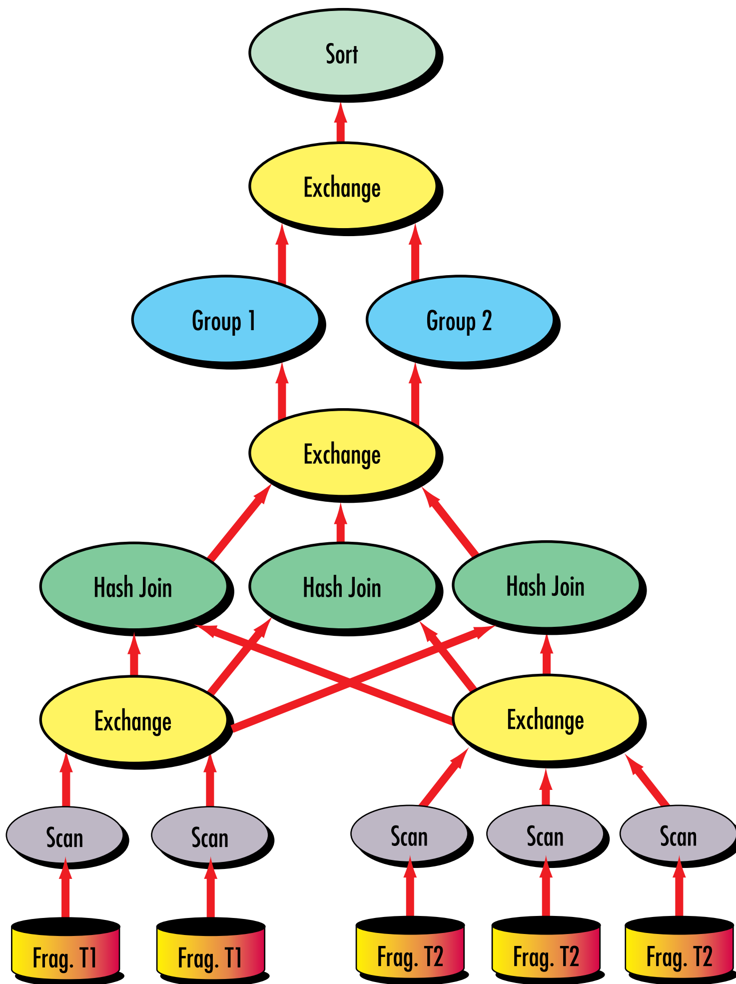

Figure 5. Parallelized iterator tree showing exchange pipelines

```
Create table account fragment by expression
      (column specifications. . .>
       fragment by expression
              account_num <100,000 in account_space1,
              account_num <200,000 in account_space2,
              remainder in account_space_remainder;

Create index account_index on account (account_num)
       fragment by expression
              account_num <50,000 in index_space1,
              account_num <200,000 in index_space2,
              remainder in index_space_remainder;
}
```

Figure 6. Example of syntax for establishing fragmentation

determine which join instance will receive the tuple. Aggregation and sort merge join iterators also require hash partitioning.

Partitioning in this way can cause data skew, which occurs when values of a join or group by key are not uniformly distributed. For example, to count inhabitants of the United States by state, the thread handling New York would be much busier than the thread handling Alaska. The PDQ engine has several unique strategies for dealing with this situation and can adjust to it dynamically.

Resource Management

With PDQ, queries can be assigned a query priority ranging from 0 to 100 to indicate the percent of DBMS resources (such as CPU, memory, and scan bandwidth) that should be allocated to the query. CPU resources are controlled by limiting the number of threads assigned to a query. The number of threads varies with the query priority and the number of server processes, which represents the maximum degree of parallelism.

Performing queries that contain sorts, joins, and aggregates is highly dependent on memory availability. Since most commercial DBMSs (including On-Line 5.0 and On-Line Dynamic Server 6.0) have no way to allot memory among queries, they take the conservative approach of never using too much. Although this allows more queries to make adequate progress without impacting each other, it does not facilitate running large queries that could benefit from large memory allocations. On-Line Dynamic Server 7.1 has mechanisms for adjusting the size of the memory pool used for large queries, the maximum number of concurrent large queries, and the relative priority of the queries, so that the memory can be apportioned among them. Since these mechanisms are dynamic, the availability of resources to large decision-support queries can be dialed down during periods when the OLTP load is high.

Performance

The ultimate measure of a parallel DBMS implementation is its performance. Figures 7 and 8 show speed-up curves for two simple queries. Speed-up is measured by using more CPUs without varying the amount of data. A perfect speed-up curve is a hyperbola (not a straight line) showing that execution time is reduced to $1/n$ x t when resources are increased n times.

Scan Performance

Figure 7 shows the speed-up curve for a parallel select varying the number of disks. The difference between the top On-Line 5.0 line and the lower On-Line Dynamic Server 7.1 line is due to the algorithmic improvements described above. The On-Line 5.0 line does not show speed up because 5.0 was not parallelized for scans. The "Best Possible" line was obtained by running a small UNIX program that performed raw reads in a tight loop. The disks in this experiment could deliver about 2 MB per second, so the scan speeds shown were limited by the disk hardware, not the DBMS software.

Hash Join Performance

Figure 8 shows the speed-up curve for a parallel hash join varying the number of disks and CPUs. Again, speed-up is excellent, and the performance of On-Line Dynamic Server 7.1 surpasses On-Line 5.0. The rightmost point indicates that a speed-up of approximately seven is obtained with eight times the resources.

Future Products

A follow-on product scheduled for release in early 1996 will provide parallel load and unload utilities. These utilities will be capable of I/O to multiple external devices, and come with a GUI interface and built-in conversion routines for handling data from many sources. Numerous server performance enhancements will permit the loading and unloading of data from the database at disk speeds.

Informix's Dynamic Server Architecture is currently being extended to encompass loosely coupled systems such as IBM's SP2™. Many applications will never need a platform this large, but for those that do, DSA provides a growth path from the SMP platform. The new server product, On-Line Dynamic Server 8.0, is planned to be available in mid-1995.

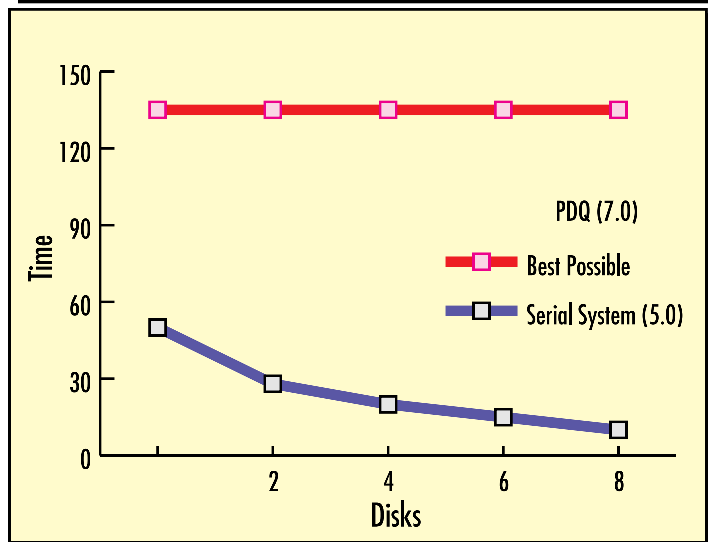

Figure 7. Selects varying number of disks

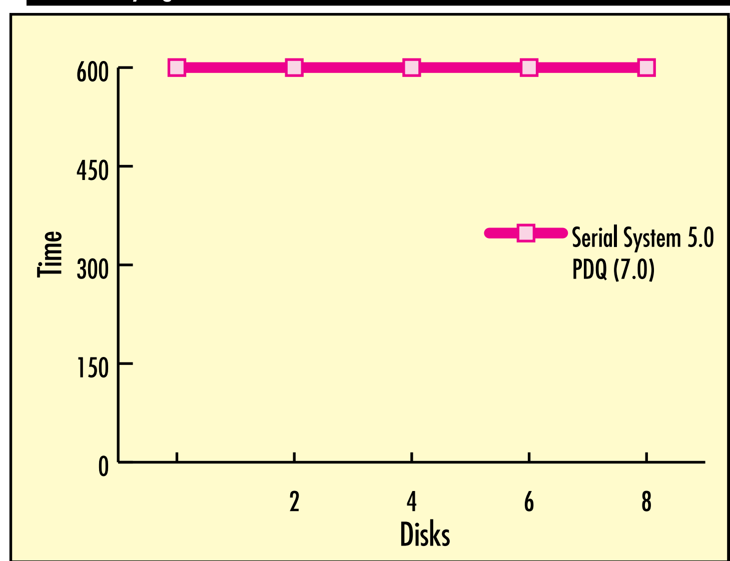

Figure 8. Joins varying number of disks

David Clay, Informix Software, Inc., 921 S.W. Washington Avenue, Portland, OR 97205. Internet: davec@informix.com. Mr. Clay, an engineering manager in the Servers and Connectivity Division, was the project leader of the Informix parallel query project. He is currently extending the Informix engine to massively parallel, shared-nothing architectures such as IBM's SP2. Mr. Clay has a BA in Mathematics from Michigan State University and an MS in Computer Science from Cleveland State University in Ohio.