TANDEM

High Performance SQL Through Low-Level System Integration

Andrea Borr
Franco Putzolu

Technical Report 88.10
June 1988
Part Number 19141

HIGH PERFORMANCE SQL THROUGH LOW-LEVEL SYSTEM INTEGRATION

Andrea Borr

Franco Putzolu

Technical Report 88.10
June 1988
Tandem Part Number: PN19141

HIGH PERFORMANCE SQL THROUGH LOW-LEVEL SYSTEM INTEGRATION

Andrea Borr
Franco Putzolu

June 1988
Tandem Technical Report 88.10
Part Number PN 19141

ABSTRACT: NonStop SQL™ achieves high performance through an implementation which integrates SQL record access with the pre-existing disk I/O and transaction management subsystems, and moves SQL function downward from the client to the server level of these subsystems. System integration and movement of function to the server reduce message traffic and cpu consumption by putting SQL optimizations at the lower levels of the system. Examples of such optimizations are message traffic savings by filtering data and applying updates at the data source, I/O savings by SQL-optimized buffer pool management, and locking and transaction journaling techniques which take advantage of SQL semantics. Achieving message traffic reduction is particularly important in a distributed, non shared-memory architecture such as the Tandem NonStop System. The result of this implementation is an SQL system which matches the performance of the pre-existing DBMS, while inheriting such pre-existing architecturally-derived features as high availability, transaction-based data integrity, and distribution of both data and execution.

TABLE OF CONTENTS

INTRODUCTION

In most implementations, SQL is not fully integrated with the existing record access and transaction management subsystems. Typically, SQL is built on top of an existing access method, which is used to provide blocked I/O, and relies on its own recovery scheme, supported by a proprietary journal. This layered approach results in operational complexity in areas such as system generation, problem isolation, and security administration. Furthermore, there is a performance penalty associated with the non-integrated aspect of this approach. Maintaining a separate journal is more expensive than integrating SQL-related activity with the existing journal. Layering SQL operations on top of an existing access method may also result in performance degradation. SQL implementations of hardware vendors attempt to minimize this penalty by taking advantage of proprietary, privileged, low-level interfaces. Third party vendors do not have this luxury, and are often more heavily impacted.

The *layered* approach has serious performance problems in loosely-coupled (non shared-memory) multi-processor systems such as the Tandem NonStop System. In such architectures, the record access and transaction management functions are split between requester (client) processes, which provide user interfaces, and server processes, which manage disk volumes. Client and server processes typically reside in different processors, or even in different geographical locations, and communicate via messages. The bandwidth limitation inherent in the lack of shared memory requires that management of shared resources (buffers, locks, file structures, etc.) be performed as much as possible on the server side. As a consequence, attempts to port third party SQL products to non shared-memory multi-processor systems without integrating them with the low-level I/O subsystem result in poor performance due to the necessity of running the ported SQL subsystem as user code in the requester process.

Since the disk I/O server is part of the operating system in the Tandem NonStop system architecture, Tandem rejected the idea of porting an existing third party SQL product. Avoiding a design which would layer SQL implementation on top of the existing record access and transaction management subsystems, Tandem instead chose to integrate NonStop SQL with ENSCRIBE™, the pre-existing DBMS. The implementation moves a large part of the new SQL function to the server side of the disk I/O subsystem. It introduces SQL-specific logic into the DBMS subsystems supporting ENSCRIBE. Integration with Tandem's networking and distributed transaction management subsystems has allowed NonStop SQL to inherit pre-existing facilities for high-availability, fault-tolerance, and distribution. The facilities for distribution inherited from the pre-existing architecture, in particular, will allow progressively fuller exploitation of parallelism to produce performance gains in the future.

Pushing SQL support logic to the server side of the record access and transaction management subsystems produces performance gains by reducing path lengths as compared with the layered approach. It furthermore provides the opportunity for significant SQL-specific disk cache management optimizations, resulting in fewer and more efficient transfers of data to and from disk. Given the message-based nature of Tandem's distributed operating system, however, perhaps the most significant performance gains are achieved via message traffic savings -- also in part describable as path-length savings. By introducing a field-level interface to the low-level disk I/O system, and by delegating SQL function such as field projection, predicate evaluation, and set-oriented retrievals, updates, and deletes to the Disk Process (low-level disk file server), message traffic is significantly reduced as compared with the ENSCRIBE record-at-a-time interface. In addition, delegating an update via *update expression* (e.g. `SET ACCOUNT.BALANCE = ACCOUNT.BALANCE - DEBIT`) to the disk process eliminates the extra message which would otherwise be required for the requester to read the record before updating it. These message savings, optimized cache management, and reduced path lengths for I/O and transaction management compensate for increased path lengths at higher levels to support the higher functionality and ease-of-use of the SQL language. The result is the functionality of SQL with performance comparable to that of ENSCRIBE [Benchmark].

OVERVIEW OF TANDEM ARCHITECTURE

The Tandem NonStop architecture consists of up to sixteen loosely-coupled processors interconnected by dual high-speed buses to form a single system, called a *node*, in the Tandem network [Katzman], see Figure 1. Nodes can be connected into clusters via fiber optic links, as well as into long-haul networks via X.25, SNA, or other protocols. The goals of the architecture are fault-tolerance, high availability, continuous operation, and modularity.

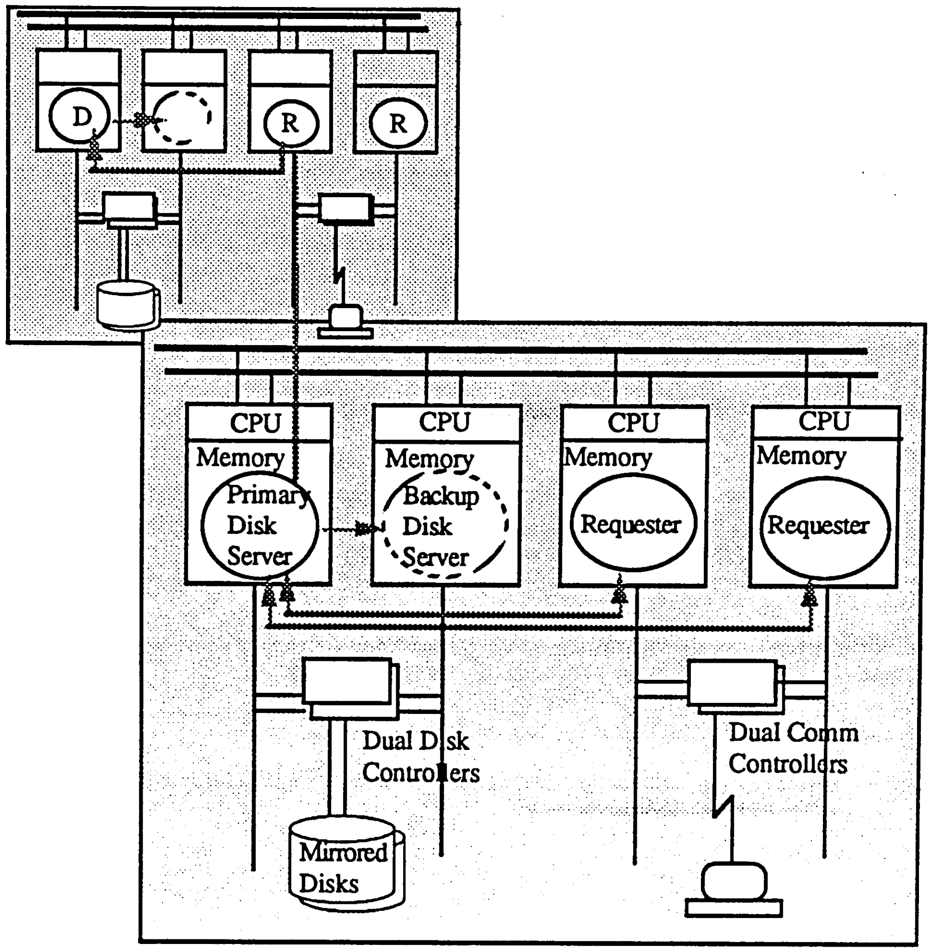

Figure 1: A schematic diagram of two Tandem nodes, each with four processors. Disks and other devices are managed by process pairs. Requestors communicate with local and remote servers via messages.

Hardware and software redundancy provide I/O device availability despite single module failure. Hardware redundancy provides alternate physical paths to I/O devices. Software redundancy provides fault-tolerant device-controlling *process-pairs* , the *primary* process and its hot-standby *backup* process running in two processors physically connected to the device [Bartlett]. A transaction mechanism coordinates the atomic commitment of updates by multiple processes in the network [Borr1].

System resources are managed by a message-based operating system which provides communication between processes executing in the same or different processors. The message system makes the distribution of hardware components transparent [Bartlett]. I/O devices are managed by system-level processes called I/O processes. The *Disk Process* is the I/O process which manages a disk *volume* (optionally replicated on *mirrored* physical drives for fault tolerance).

COMPONENTS OF THE DISK PROCESS

The Disk Process is actually a *group* of cooperating processes which share a message input queue. The process group acts as I/O server for files resident on the volume it manages. These files include code files and virtual memory swap files as well as SQL and ENSCRIBE database files. The Disk Process performs disk I/O by invoking a set of subroutines, collectively called the *Driver*, which run in the process environment of the invoker.

The record management component of the Disk Process implements the access methods supporting the file structures common to ENSCRIBE and NonStop SQL:

- key-sequenced (B-Tree);
- relative (direct access);
- entry-sequenced (direct access for reads, insert at EOF only).

The cache management component of the Disk Process uses a least-recently-used (LRU) algorithm obeying *write-ahead-log* protocol [Gray] to manage a main memory buffer pool for staging data to and from disk. The cache provides transaction-protected database read and write services while attempting to minimize disk I/O accesses. Disk cache management is integrated with the operating system's processor-global virtual memory management mechanism in the sense that the latter implements a globally optimized page replacement algorithm which can, via handshakes with the Disk Processes of the processor, cause the *stealing* of clean database buffers and the *cleaning* (writing) of dirty ones in order to make the underlying physical memory pages available for a higher priority use.

The lock management component of the Disk Process provides concurrency control for both SQL and ENSCRIBE via locking at the file, record, or *generic* (key prefix) level for volume-resident SQL or ENSCRIBE data.

Code supporting transaction management and *auditing* (Tandem's term for journaling or logging) permeates the record management, cache management, and lock management components. Transaction commit and abort are supported by tight integration with the operating system's

Transaction Monitoring Facility, TMF [Borr1]. The dual roles of the backup Disk Process and TMF in maintaining high device availability, fault tolerance, transaction consistency, and robustness to crash are described in [Borr2].

Both SQL and ENSCRIBE share the same TMF audit trail (log), which resides on the audit trail volume, managed by a standard Disk Process. The audit trail writing component of the audit trail volume's Disk Process is highly optimized for long, or *bulk* sequential I/O's using group commit [Gawlick] and audit piggy-backing to maintain a high transaction commit rate with a minimal number of I/O's.

RATIONALE FOR DIVISION OF LABOR BETWEEN FILE SYSTEM AND DISK PROCESS

The *File System* is a set of system library routines which have their own data segment but which run in the process environment of the application (client) program. These routines format and send to various Disk Processes messages requesting database services for files residing on their volumes. Through File System invocations, the application process becomes a requester (client) and the Disk Process a server in the requester-server model.

In the case of ENSCRIBE, the application program invokes the File System explicitly -- calling such routines as OPEN, READ, WRITE, LOCKRECORD -- to perform key navigation and record-oriented I/O.

In the case of SQL, the application program's SQL statements invoke the SQL Executor, a set of library routines which run in the application's process environment. The Executor invokes the File System on behalf of the application. Its field-oriented and possibly set-oriented File System calls implement the execution plan of the pre-compiled query.

Certain aspects of the division of labor between the File System and the Disk Process are mandated by the distributed character of the Tandem architecture. Database files in a Tandem application are typically spread across multiple disk volumes, attached to different processors within a node, or to different nodes within a cluster or network. Base files may have multiple secondary indices (implemented as separate key-sequenced files), and these may be located on arbitrary volumes. Base files and secondary indices may each be horizontally partitioned, based on record key ranges, into multiple fragments residing on a distributed set of disk volumes. Thus, the file fragment managed by the Disk Process as a single B-tree may in fact be merely a single partition of an ENSCRIBE or SQL file, or a secondary index (or partition thereof) for an ENSCRIBE or SQL base file. The file or table is viewed as the sum of all its partitions and secondary indices only from the perspective of the SQL Executor or ENSCRIBE File System invoker. Such an architecture makes the File System the natural locale for the logic which, transparently to the caller, manages access to the appropriate partition based on record key; or manages access to the base file record via a secondary key; or performs maintenance of secondary indices consistent with the update or delete of a base file record.

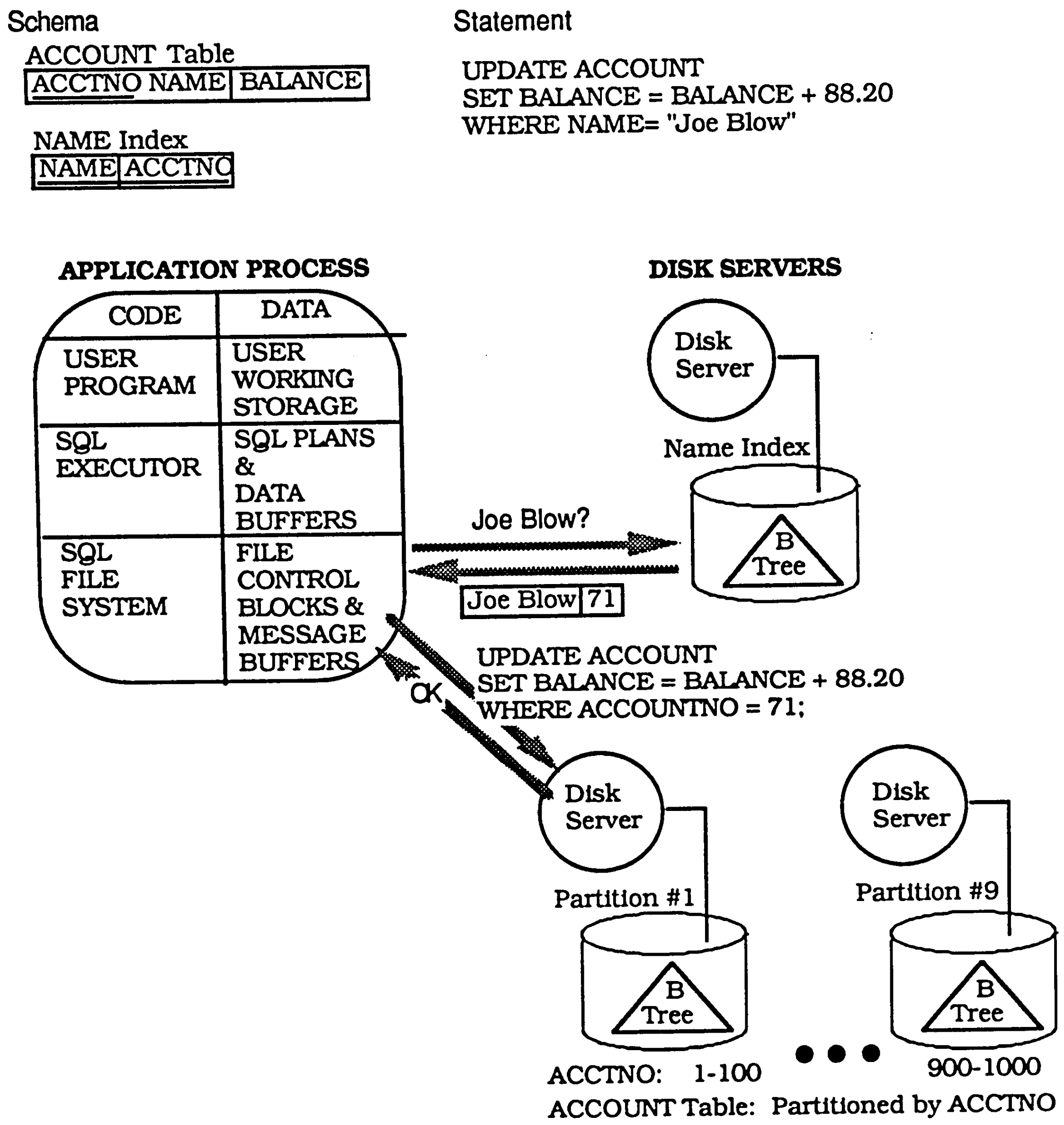

Figure 2: The File System in doing an update via alternate key first sends a request to the disk server managing the index to find the primary key. It then sends the update experssion to the server managing the primary key partition.

For example, to implement a request to read via a secondary index (see Figure 2), the File System first sends to the Disk Process managing the index's volume a read request for the appropriate index record. Having extracted the base file record key from the index record, it then sends a request to the base file's Disk Process to read the base file record having that key. To implement a read or write request to a partitioned file, the File System uses the record key to identify the partition in which that record resides, then sends the read or write request to the Disk Process which manages that partition. These File System functions are common to both SQL and ENSCRIBE, although separate File System procedures perform them for the two systems.

In the following, we describe the nature of the File System-Disk Process interface (FS-DP Interface) for ENSCRIBE, and the reasons for the design of a new FS-DP interface for NonStop SQL.

THE OLD FS-DP INTERFACE MANDATED BY ENSCRIBE

The record-oriented user interface of ENSCRIBE mandates a record-oriented FS-DP interface to support it. The ENSCRIBE user issues requests to read, write, or delete a whole record, specified by its primary or alternate (secondary) key. The only exception to this record-at-a-time interface is a user-controlled sequential read optimization called *sequential block buffering* (SBB). When enabled, SBB for reads causes each FS-DP request message to return a copy of a physical file block. SBB reduces FS-DP message traffic by the file's physical blocking factor (i.e. number of records per block). Once an FS-DP message has returned a block to the File System, multiple record-at-a-time ENSCRIBE READ requests then result in de-blocking by the File System from its local block copy before a message requesting the next block is sent to the Disk Process. SBB under ENSCRIBE has limited utility, however, since no locking other than at the file level is effective when it is in use. The user is therefore required to have an OPEN exclusion mode which excludes other write-access openers.

THE NEW FS-DP INTERFACE TAILORED FOR NonStop SQL

The SQL language is characterized by a field-oriented user interface and set-oriented selection, update, and delete operations [ANSI]. User-specified predicates define selection criteria, update expressions, and integrity constraints. The field- and set-orientation of the user interface have a natural extension down to a field- and set-oriented FS-DP interface, necessitating less total message traffic between the File System and the Disk Process than a record-at-a-time interface. When the selection predicate (e.g. `WHERE ACCOUNT.BALANCE > 0`) involves only one table (actually, one file fragment managed by a single Disk Process), such a *single-variable query* can be evaluated by the Disk Process for each record in a key range and used as a filter limiting the set of records processed or returned in the reply to the FS-DP message. When an update expression (e.g. `SET ACCOUNT.BALANCE = ACCOUNT.BALANCE * 1.07`) specifies a new value for a field in terms of an expression involving only literals and fields of the record at hand, subcontracting the expression evaluation and update to the Disk Process avoids the necessity of returning the record to the File System invoker, which would subsequently request the update via a new message. Where an integrity constraint (`CHECK PART.QUANTITY >= 0`) limits the allowable updates to a table, its enforcement at Disk Process level may likewise obviate the need for a preliminary read by the File System for constraint verification prior to an update request via a second message.

SQL STATEMENT EXECUTION REDUCED TO SINGLE-VARIABLE QUERIES

Although a general SQL predicate can be multi-variable (i.e. involving joins, or expressions in terms of fields of more than one table), the Executor's File System invocations, mandated by the plan produced by the SQL query compiler, are in terms of a single table, with optional access via a secondary index. The File System dynamically decomposes this single-table request into messages to individual Disk Processes managing partitions (if any) and/or secondary indices.

If the SQL statement decomposes in such a way that a single-variable query can be attached to the request message sent by the File System to the Disk Process, then message traffic over the FS-DP interface can be reduced by filtering the data at its source. By pushing SQL selection and projection logic as low as possible in the system, data is filtered early. In a distributed system, this produces important performance benefits due to reduced message traffic, since only selected and projected data is returned to a remote requester.

CONTINUATION RE-DRIVE PROTOCOL FOR SET-ORIENTED FS-DP REQUESTS

The SQL FS-DP interface thus subcontracts selection and projection to the Disk Process wherever feasible. The interface also has a set-oriented option. The Disk Process may be requested to operate on (i.e. to retrieve, update, or delete) a set of records spanning a specified (primary) key range (may include all) and, optionally, satisfying a predicate. To prevent a single set-oriented FS-DP request from monopolizing a Disk Process over a long period of time, limits on the elapsed and processor time spent per request message are set. If exceeded, a continuation re-drive protocol is triggered. The Disk Process then returns to the File System the key of the last record accessed, together with any data selected during the current request execution (retrieval case). The File System then sends a re-drive message supplying as the new (non-inclusive) begin-key of the range the key last processed on the previous execution. Re-drives are also triggered by a full sequential block buffer condition (see next section).

SEQUENTIAL BLOCK BUFFERING

The ENSCRIBE concept of sequential block buffering has been extended for SQL from *real* (RSBB) physical disk block copies to *virtual* (VSBB) blocks built by the Disk Process via selection and projection. In VSBB, data is returned through the set-oriented FS-DP read interface after projected fields have been extracted from key-range-satisfying records which have optionally been subjected to a filtering predicate. This is similar to the concept of portals described by Stonebraker [Stonebraker]. The locking restriction under ENSCRIBE (file locking only) which limited the usefulness of SBB has been removed for SQL. Record locking has been extended to a form of virtual block locking in which the records of the virtual block are locked as a group.

The selection and projection performed by the Disk Process in filling the virtual block buffer, particularly if the predicate is very selective, give VSBB a much reduced message cost over the record-at-a-time interface, and even over the RSBB interface. RSBB gives a factor of three over the record-at-a-time interface. VSBB gives NonStop SQL an additional factor of three over RSBB on many of the Wisconsin benchmark queries [TDBG]. The performance gains of VSBB are attributable to the reduced message traffic resulting from filtering data at its source, and only returning selected and projected data to the requester.

MAPPING SQL TO FS-DP INTERFACE: EXAMPLES

Example (1): Virtual Sequential Block Buffering: The following statement maps into a series of set-oriented read requests involving selection and projection, and returning data via Virtual Sequential Block Buffering (VSBB).

Table EMP has fields: EMPNO (primary key), NAME, HIRE_DATE, SALARY, ...

Statement:
```
SELECT NAME, HIRE_DATE FROM EMP
WHERE EMPNO <= 1000
AND SALARY > 32000;
```

Message types:
```
GET^FIRST^VSBB
GET^NEXT^VSBB
```

The initial FS-DP message is of type GET^FIRST^VSBB. It specifies the projection of the fields NAME and HIRE_DATE (identified by their record descriptor field numbers), the primary key range [LOW-VALUE, 1000] for EMPNO, and the predicate SALARY > 32000. The returned virtual block contains (NAME, HIRE_DATE) from records in the primary key range which satisfy the selection predicate. If a full VSBB condition or a time limit expiration makes a continuation re-drive necessary, message type GET^NEXT^VSBB is used.
It specifies the new key range (LAST-PROCESSED-KEY, 1000] for EMPNO, but does not re-send the predicate or the projection. These latter were saved in the Subset Control Block which was created by the Disk Process at GET^FIRST time.

Example (2): Real Sequential Block Buffering: The following statement maps into a series of set-oriented read requests which return data via Real Sequential Block Buffering (RSBB) since there is no selection or projection.

Statement:
```
SELECT * FROM EMP;
```

Message types:
```
GET^FIRST^RSBB
GET^NEXT^RSBB
```

The initial FS-DP message is of type GET^FIRST^RSBB. It specifies the primary key range [LOW-VALUE, HIGH-VALUE] for EMPNO. Each re-drive, using message type GET^NEXT^RSBB and specifying the new key range (LAST-PROCESSED-KEY, HIGH-VALUE], returns one real sequential block.

Example (3): Update Subset: The following statement maps into a series of set-oriented update requests involving a selection predicate and an update expression.

Table ACCOUNT has fields: ACCTNO (primary key), BALANCE, ...

Statement:
```
UPDATE ACCOUNT
SET BALANCE = BALANCE * 1.07
WHERE BALANCE > 0;
```

Message types:
```
UPDATE^SUBSET^FIRST
UPDATE^SUBSET^NEXT
```

The initial FS-DP message is of type UPDATE^SUBSET^FIRST. It specifies the primary key range [LOW-VALUE, HIGH-VALUE] for ACCTNO, the predicate BALANCE > 0, and the update expression BALANCE = BALANCE * 1.07. If a time limit expiration makes a continuation re-drive necessary, message type UPDATE^SUBSET^NEXT is used. It specifies the new key range (LAST-PROCESSED-KEY, HIGH-VALUE] for ACCTNO, but does not re-send the predicate or the update expression. These latter were saved in the Subset Control Block which was created by the Disk Process at GET^FIRST time.

SET INTERFACE FACILITATES CACHE OPTIMIZATIONS FOR SEQUENTIAL ACCESS

The set-oriented FS-DP requests specify a primary (physically clustered) key range of records to be processed. The begin-key and end-key are specified at the initial FS-DP interaction. From then on, the Disk Process can optimize, reading the blocks containing the required key span from disk into cache using a minimal number of I/O's. Where possible, it reads into cache buffers sequential strings of physical blocks (presently limited to 4K bytes maximum each) using *bulk* I/O's (presently limited to 28K bytes maximum). Of course, where physical clustering of key-sequenced data blocks has been broken due to B-tree splits and collapses, some bulk I/O's may be less than maximal length.

In addition to using bulk I/O to minimize the number of reads, the Disk Process attempts to *pre-fetch* data, i.e. to perform bulk reads asynchronously in anticipation of their need by an active request. Advance knowledge of the required key span, and use of the multi-process structure of the Disk Process group, make asynchronous pre-fetch possible. Asynchronous pre-fetch allows cpu-bound processing using data from the cache to occur in parallel with disk I/O's.

Bulk I/O is also used for asynchronous *write-behind*. This mechanism uses idle time between Disk Process requests to write out strings of sequential blocks updated under a subset. By using its Subset Control Block (created as a result of the initial set-oriented FS-DP interaction), the Disk Process can keep track of strings of sequential blocks which are *dirty* (i.e. have been updated in cache). Once a string of dirty data blocks has aged to the point that the audit related to the blocks of the string has already been written to disk, then the string of dirty data blocks can be written to disk without violating *write-ahead-log* protocol [Gray]. The Disk Process then writes the string to disk using the minimal number of bulk I/O's.

FIELD INTERFACE ENABLES AUDIT RECORD SIZE REDUCTION

The field-oriented nature of the SQL FS-DP interface allows the record management component of the Disk Process to generate SQL-specific TMF audit records containing field-oriented before- and after-images. The resultant *field-compressed* audit records are generally reduced in size as compared with ENSCRIBE audit records, which by default contain full-record before- and after-images. SQL naturally lends itself to audit compression because SQL syntax specifies the fields which are being updated. By contrast, the ENSCRIBE user's unit of update is a record, and while an ENSCRIBE audit-compression user option is available, its implementation is costly since the identity of the updated fields must be computed by comparing the record before-and after-images. Therefore, ENSCRIBE audit records contain full record images by default.

The reduction in SQL audit record size resulting from field compression has performance benefits in many areas. For example, the size of the audit trail data on disk as well as the size of all audit-containing messages throughout the system is reduced. There are fewer sends of audit to the audit trail Disk Process due to audit buffer-full conditions, since the audit buffer fills up less frequently. Less audit per transaction allows each bulk-write of the audit trail to commit a larger group of transactions. Since audit compression reduces the frequency of audit writes due to buffer-full conditions, timers have been introduced to force out pending commits from a partially full buffer. Response times are minimized by dynamically adjusting the timers based on such system statistics as transaction rate and average transaction size [Helland].

OPPORTUNITIES FOR FUTURE PERFORMANCE ENHANCEMENTS FOR NonStop SQL

The previously discussed performance gains due to the use of sequential block buffering for reads point the way to achieving similar results by changing the FS-DP sequential write interface. Presently, the interface for sequential SQL inserts is a message per record inserted. If a blocked interface for inserts were introduced, the message traffic between the File System and the Disk Process could be reduced by the blocking factor. Multiple sequential inserts issued to the File System by the SQL Executor would then be accumulated in a local buffer by the File System, which would, when required, send the buffer of inserted records to the Disk Process using one message. To avoid a late-detected duplicate key condition, however, the Disk Process would have had to keep an empty sequential target key range locked by prior agreement with the File System. Given such an interface, the Disk Process could then maintain an Insert Control Block, similar to the Subset Control Block, which would keep track of strings of sequential blocks previously dirtied. Strings of dirty blocks old enough not to cause write-ahead-log if written to disk would then be written out using bulk I/O.

Similarly, the previously discussed performance gains due to use of set-oriented update and delete request messages point the way to achieving better performance for the construct UPDATE WHERE CURRENT or DELETE WHERE CURRENT. These currently require a message per record updated or deleted. By allowing the updates (deletes) to occur in a buffer local to the File System, and then sending the buffer full of updates (deletes) to the Disk Process in one message, substantial message traffic savings in the FS-DP interface could be realized.

An open-ended area for improving the performance of NonStop SQL is the fuller exploitation of the parallel architecture of the Tandem system. Parallelism is currently exploited in the sense that multiple independent transactions can execute simultaneously [TDBG]. The overlap of I/O and cpu-bound processing inherent in asynchronous pre-fetch and write-behind is also a form of parallelism. There is furthermore a user option which directs the SQL compiler to cause the invocation at execution time of the parallel sorter, FastSort, which uses multiple processors and disks if available [Tsukerman]. Future opportunities for the use of intra-query parallelism involve distributed query optimization, parallel Executor process structure, and no-wait Disk Process message sends in the File System.

SUMMARY

By pushing SQL-specific logic downward in the DBMS support subsystems from the requester to the server level, Tandem has obtained an SQL system which today matches, and is expected one day to surpass, the performance of its pre-existing DBMS. The low-level path length savings, disk cache management optimizations, and reduced message traffic resulting from low-level integration pay for the increased path lengths at higher levels needed to support the high functionality and ease-of-use of the SQL language. Furthermore, system integration allows NonStop SQL to inherit from the pre-existing system the facilities which support high availability, fault-tolerance, and data and execution distribution. In particular, the inherited facilities for distribution make the increased exploitation of parallelism an avenue for major performance gains in the future. Additional benefits can be derived in the future by the introduction of further server-level SQL-specific optimizations which take advantage of SQL semantics.

ACKNOWLEDGEMENTS

Members of the Tandem Database Group too numerous to mention individually contributed ideas to the design of the record access and transaction management architecture for NonStop SQL.

Thanks are due to both Jim Gray and Dieter Gawlick for editorial suggestions whose implementation improved the presentation of this material.

REFERENCES

[ANSI] *Database Language SQL 2 (ANSI working draft)*, ANSI X3H2 87-8. Dec. 1986.

[Bartlett] Bartlett, J. F., *A NonStop Kernel*, Proceedings of Eighth Symposium on Operating System Principles, ACM, Dec. 1981.

[Benchmark] *NonStop SQL Benchmark Workbook*, Part No. 84160, Tandem Computers Inc, Cupertino, Ca., March 1987.

[Borr1] Borr, A. J., "Transaction Monitoring in ENCOMPASS: Reliable Distributed Transaction Processing", Seventh International Conference on Very Large Databases, Sept. 1981.

[Borr2] Borr, A. J., *Robustness to Crash in a Distributed Database: A Non Shared-Memory Multi-Processor Approach*, Tenth International Conference on Very Large Databases, Aug. 1984.

[Tsukerman] Tsukerman, A., et al., *FastSort: An External Sort Using Parallel Processing*, Tandem Technical Report 86.3, Cupertino, Ca., May 1986.

[Gawlick] Gawlick, D. and Kinkade, D., *Varieties of Concurrency Control in IMS/VS Fast Path*, IEEE Database Engineering, June 1985.

[Gray] Gray, J. N., *Notes on Database Operating Systems*, IBM Research Report: RJ 2188, Feb. 1978.

[Helland] Helland, P., et al. *Group Commit Timers and High Volume Transaction Systems*, 2nd International Workshop on High Performance Transaction Systems, Asilomar, Ca., Sept. 1987. Also: Tandem Technical Report 88.1, Cupertino, Ca., March 1988.

[Katzman] Katzman, J. A., *A Fault-Tolerant Computing System*, Eleventh Hawaii International Conference on System Sciences, 1978.

[Stonebraker] Stonebraker, M. and Rowe, L., *Database Portals: A New Application Program Interface*, Tenth International Conference on Very Large Databases, August 1984.

[TDBG] Tandem Database Group, *NonStop SQL, A Distributed, High-Performance, High-Availability Implementation of SQL*, Tandem Technical Report 87.4, Cupertino, Ca., April 1987.

Distributed by
TANDEM
Corporate Information Center
19333 Vallco Parkway MS3-07
Cupertino, CA 95014-2599