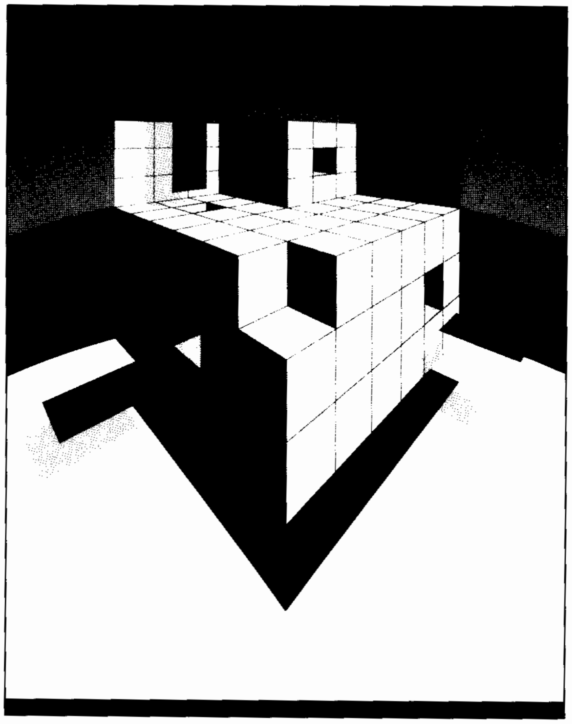
TANDEM

SYSTEMS REVIEW

VOLUME 4, NUMBER 2

JULY 1988

BRANDIFINO



An Overview of NonStop SQL

The NonStop SQL Optimizer

NonStop SQL Reliability

The NonStop SQL Data Dictionary

High-Performance SQL

Debugging TACL Code

Correction:
Susan Wayne Thompson was inadvertently omitted as co-author of the technical paper, ''Tandem's Approach to Fault Tolerance,'' which appeared in the February 1988 issue.

TANDEM SYSTEMS REVIEW

Volume 4, Number 2, July 1988

Articles Editor
Susan Wayne Thompson

Managing Editor
Ellen Marielle-Tréhoüart

Associate Editors
Steven Kahn
Wendy Osborn

Assistant Editors
Sarah Rood
Jodi Steiner

Technical Advisors
Mark Anderton
Bart Grantham

Art Director
Stephen Stavast

Cover Art
David Thompson

Production and Layout
Corporate Graphics

Typesetting
Tandem Typography

The *Tandem Systems Review* is published by Tandem Computers Incorporated.

Purpose: The *Tandem Systems Review* publishes technical information about Tandem software releases and products. Its purpose is to help programmer-analysts who use our computer systems to plan for, install, use, and tune Tandem products.

Subscription additions and changes: Subscriptions are free. To add names or make corrections to the distribution database, requests within the U.S. should be sent to Tandem Computers Incorporated, *Tandem Systems Review,* 18922 Forge Drive, LOC 216-05, Cupertino, CA 95014. *Requests outside the U.S. should be sent to the local Tandem sales office.*

Comments: The editors welcome suggestions for content and format. Please send them to the *Tandem Systems Review*, 18922 Forge Drive, LOC 216-05, Cupertino, CA 95014.

Tandem Computers Incorporated makes no representation or warranty that the information contained in this publication is applicable to systems configured differently than those systems on which the information has been developed and tested. It also assumes no responsibility for errors or omissions that may occur in this publication.

Overview of NonStop SQL

Tandem's NonStop SQL is an implementation of ANSI SQL on Tandem™ computer systems. In addition to the ease of use implicit in SQL, NonStop SQL is a high-performance distributed SQL that can be used both in the information center and in production on-line transaction processing applications. It has the performance, integrity, administrative, and utility features necessary to operate databases that run hundreds of transactions per second.

Development of NonStop SQL

Tandem is a leader in providing highly reliable computer systems for on-line transaction processing (OLTP). As the use of OLTP solutions continues to grow faster than the rest of the industry, database management systems (DBMS) must satisfy two classes of requirements. First, OLTP applications require a robust DBMS for their large production environments; this DBMS must provide high performance, distributed processing, and high reliability. Second, users require a relational database management system (RDBMS) for decision making in an information-center environment; this RDBMS must be easy to use and offer highly functional access to the OLTP database.

NonStop SQL, Tandem's implementation of the American National Standards Institute (ANSI) specification for the SQL database language, is unique in the industry because it satisfies both objectives. While other vendors have implemented SQL exclusively as a decision-support product, NonStop SQL is the first fully distributed, high-performance RDBMS that can operate in a production OLTP environment.

NonStop SQL is compatible with B40, C10, or later versions of the GUARDIAN 90™ operating system. Because it exploits Tandem system architecture and is fully integrated with other Tandem products, NonStop SQL provides the strengths traditionally associated with Tandem, including fault tolerance, linear expandability, and distributed processing.

What Is SQL?

SQL (Structured Query Language) is a language that is based on the relational model (Codd, 1982) and is used for specifying Data Manipulation Language (DML), Data Definition Language (DDL), and Data Control Language (DCL). It was derived from IBM research prototypes (Astrahan, 1986) and later appeared commercially from third-party vendors and from IBM. SQL is now an ANSI and ISO (International Standards Organization) standard. The Gartner Group predicts, with 90% confidence, that 60% of application tools will generate SQL statements by 1990 (Braude, 1986).

An SQL RDBMS is a database system that acts as a base for applications and fourth-generation productivity tools. SQL is a more productive development base than traditional file systems because it features data independence, simpler programs, integrity constraints, and the active dictionary. This productivity is most noticeable during the costly maintenance phase of an application's life cycle. NonStop SQL beta customers attest to significant and measurable benefits.

Although good, logical database design will not change with SQL, application programs will change because of SQL semantics. For example, SQL semantics include the use of set operations rather than record-at-a-time interfaces. This means that the database system (i.e., SQL) will evolve toward doing more and more of what has been, traditionally, application logic. Hence, looping constructs for modifying multiple records will disappear, and editing of data values will be replaced by integrity constraints, referential integrity, and triggers.

Implementation Decisions

The Tandem Database Group had several fundamental decisions to make before the development of the SQL system began. They are dicussed briefly below.

Abandoning Compatibility with ENCOMPASS. Perhaps the most controversial decision of the NonStop SQL project was to abandon compatibility with ENCOMPASS™, Tandem's existing group of database products, and adopt an SQL interface. ENCOMPASS was the first commercial distributed database system, and it has many strong features and a loyal following. There were, however, two major reasons for making this decision.

First, when the NonStop SQL project began in late 1983, SQL had already appeared commercially in three products: DB2, SQL/DS, and ORACLE. Though the official standard was not approved until 1986, SQL was rapidly becoming a de facto standard.

Second, customers were asking for an integrated and active dictionary (that assures consistency between the dictionary and applications), support of views, and integrity constraints. The ENCOMPASS dictionary is passive and proprietary. Like most such systems, ENCOMPASS (e.g., DDL, ENFORM™ query language) was built as a separate system layer on top of the "file-and-security" system, rather than being integrated with it. Also, the ENSCRIBE database record manager provides a record-at-a-time interface for programmers and little data independence. In contrast, SQL provides views and a standard data definition and manipulation language. In addition, NonStop SQL has added an active, distributed, and integrated dictionary.

Build vs. Buy. Tandem also had to decide whether to develop its own SQL system or buy it from a software house. Several software houses were willing to port their SQL systems to Tandem hardware. While this alternative was less expensive, it did not meet Tandem's goal of providing an integrated, fault-tolerant, high-performance, and distributed RDBMS. So, the developers decided to "start from scratch."

Key Objectives. Before the release of NonStop SQL, other vendors marketed SQL as an RDBMS for decision support and marketed a second (non-SQL) system for OLTP applications. The second system was needed because SQL could not perform well enough to handle production applications. Tandem rejected this "dual database" strategy as too expensive, both for Tandem and for Tandem customers.

The primary goal in building NonStop SQL was to create a system that could be used on large and small systems, for decision support as well as for production OLTP applications. This led to several corollary objectives:

- Full integration with the Tandem networking and transaction processing system.
- Continuous access to data.
- Support for modular hardware growth and for tens of processors executing hundreds of transactions per second.
- Distribution of data and execution in local and long-distance networks.

NonStop SQL design allows the software to be expanded into areas such as Teradata-style transaction parallelism (Shemer and Neches, 1984) and heterogeneous database connectivity and, also, to support user interfaces such as QMF (*Query Management Facility: General Information*) or fourth-generation languages.

ANSI Status

In October 1986, ANSI approved the SQL standard. An addendum to that standard (*Overview of SQL/Extension-1*, 1986), covering features such as integrity constraints, is expected to be approved in 1988. Tandem is participating in the review of SQL 2 (*Database Language SQL 2*, 1986), which encompasses major extensions to the standard.

The 1986 standard identifies two levels of compliance (Level 2 is a higher level of compliance than Level 1). NonStop SQL supports Level 2 ANSI DML with the exception of unions and nulls. The data manipulation interface was extended for concurrency control and dynamic SQL. NonStop SQL supports Level 1 DDL; Level 2 exceptions are nulls, naming, and security. Most exceptions were deliberate. The objective of full integration with the Tandem operating environment was judged to be more critical than full ANSI compliance. DDL extensions include partitioned and distributed data.

NonStop SQL Architecture

The ANSI standard for SQL specifies the syntax and behavior for the statements, but the implementation of the RDBMS is left up to each vendor. NonStop SQL's implementation is geared toward on-line transaction processing. To explain how NonStop SQL provides OLTP capabilities, it is first necessary to describe its architecture. The basic elements of that architecture are the NonStop SQL objects, components, and run-time environment.

NonStop SQL Objects

The basic NonStop SQL objects are the dictionary, tables, indexes, views, and programs.

Dictionary. The NonStop SQL dictionary comprises sets of catalogs (each catalog is a set of tables) and file labels, all protected by the Transaction Monitoring Facility (TMF). These objects contain the descriptions of all the database objects (catalogs themselves, tables, indexes, views, programs, columns) and are used by the SQL compiler to get accurate descriptions for producing an access plan.

The NonStop SQL dictionary is said to be "active" because the system modifies both versions of descriptive information (catalogs and file labels) whenever the physical or logical database is changed. It is not possible to have descriptions that do not match the actual objects they describe.

Compiled versions of the table descriptions are stored in the file labels as part of the file manager's disk directory. All necessary run-time information about a table can be read from the file label as part of the file system's OPEN step. Consequently, the catalogs are only examined at compile time.

Tables. Each table has rows and columns of data and corresponds to one physical file. A file can have one of three organizations: key-sequenced, relative, or entry-sequenced. Tables can be horizontally partitioned by primary key and can have secondary indexes.

Views. A view is a logical table derived by selecting a subset of the columns or rows from one or more tables or other views. NonStop SQL has two types of views: *protection views* and *shorthand views*.

A protection view is derived from a single table by taking either a projection of the columns of the table or a selection of the rows of the table, or both. This view inherits the organization, indexes, and partitioning characteristics of its underlying table. Also, a protection view can be updated and secured. Because protection views are implemented by the SQL kernel, access can be granted to the view without granting access to the underlying tables. Another benefit of integrating SQL into the operating system is that there are no "back doors" to NonStop SQL protection views.

By contrast, a shorthand view is derived from one or more tables or other views and defined without the protection attribute. This macro definition, therefore, can only be read; it cannot be updated or secured independently of a base table. When the view name is referenced, the system acts as though the macro body had been entered directly. Thus, the user of a shorthand view must be authorized to its underlying protection views and tables. Shorthand views are very general. They allow combinations of tables and views using projection, selection, joins, and aggregates. Although not protectable outright, shorthand views constructed from properly secured protection views provide security on joined data. For data independence, shorthand views can be used to denormalize the database or sort it in convenient ways.

Programs. A NonStop SQL program is a standard T16 object program that contains SQL statements in source form. It has been run through the NonStop SQL compiler to create and store executable access plans and is registered in a catalog (see Figure 1).

Components

The basic components of NonStop SQL are the COBOL preprocessor, C and Pascal, SQL compiler, catalog manager, SQL conversational interface (SQLCI), SQL executor, SQL file system, SQL disk process, SQL utilities, and TMF. Figures 1 and 2 illustrate the relationships of the components. The components are briefly described in the following paragraphs.

COBOL Preprocessor. The COBOL preprocessor transforms a COBOL program containing embedded SQL statements so that:

- Calls to the SQL executor replace the SQL statements.
- Address pointers connect host variables and SQL run-time structures.

C and Pascal. C and Pascal also support embedded SQL but without the need for a preprocessor. The output of each of these compilers is an object program containing SQL source code. To simplify customer use, the effects of preprocessing are integrated directly into these compilers, thus eliminating the extra steps and work files that occur with preprocessing.

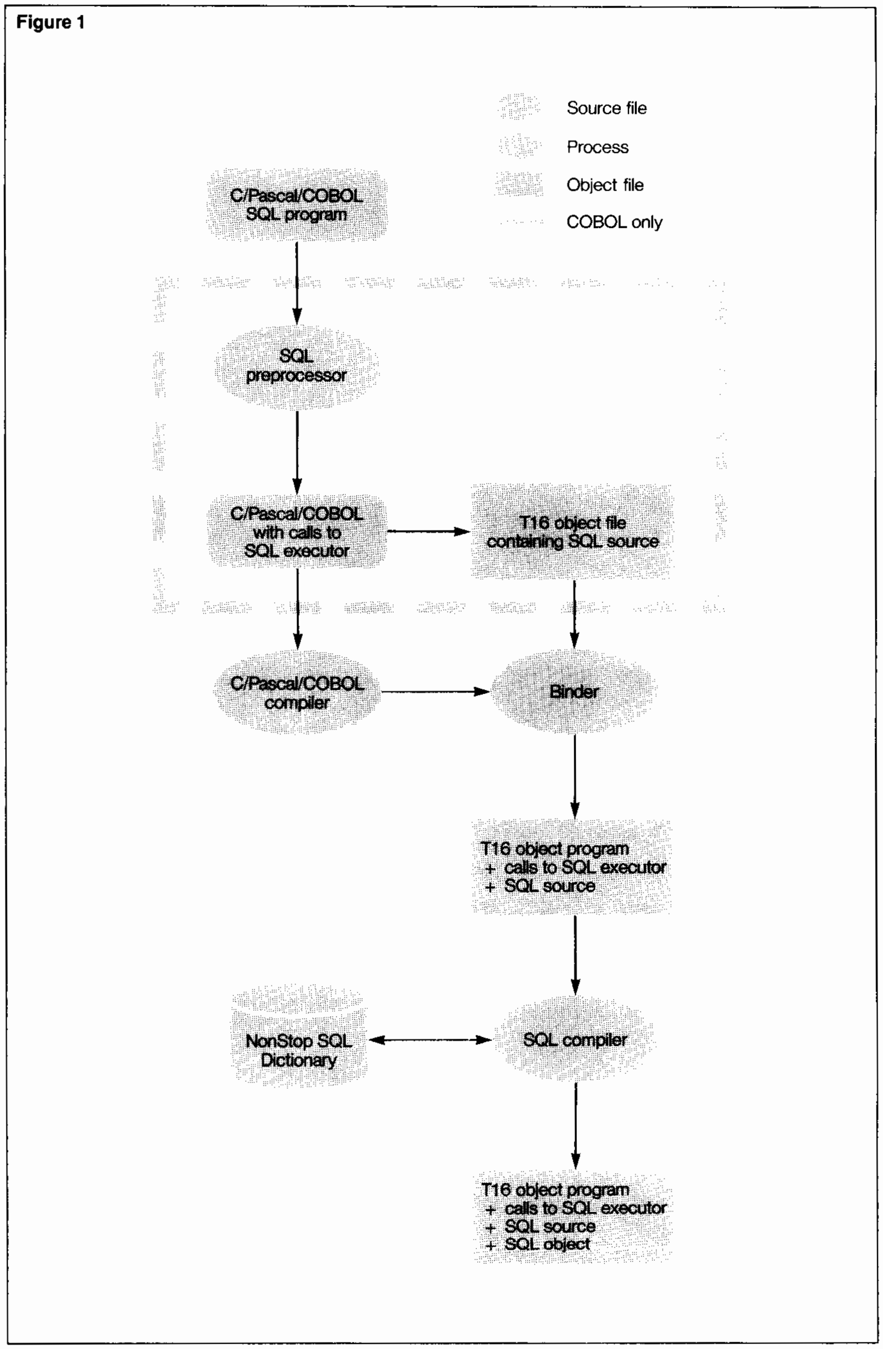


Figure 1
Source file
Process
Object file
COBOL only
C/Pascal/COBOL SQL program
SQL preprocessor
C/Pascal/COBOL with calls to SQL executor
T16 object file containing SQL source
C/Pascal/COBOL compiler
Binder
T16 object program + calls to SQL executor + SQL source
NonStop SQL Dictionary
SQL compiler
T16 object program + calls to SQL executor + SQL source + SQL object

Figure 1.
Preparation of an SQL program.

Figure 2

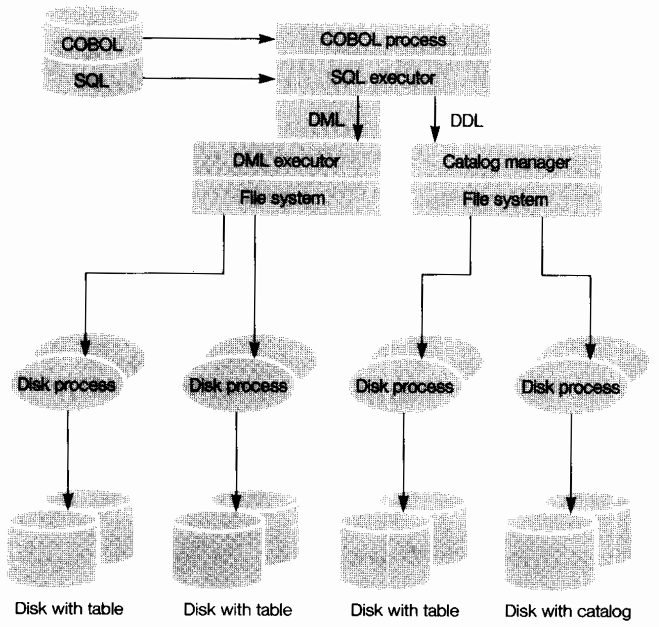


Figure 2.
SQL program execution.

The SQL Compiler. When reading an object program, the NonStop SQL compiler transforms each SQL statement into an optimal access plan that minimizes a cost function based on I/O, CPU, and messages. The SQL executor implements this access plan, and the NonStop SQL compiler accepts directives from the user.

Catalog Manager. The catalog manager handles all changes (i.e., DDL commands) to the NonStop SQL distributed dictionary (see Figure 2). The catalog manager is implemented as a separate process for authorization reasons—only the catalog manager process can write to catalog tables.

SQL Executor. The SQL executor is a set of procedures residing in the system library that executes compiled SQL statements against database tables, views, or the database catalogs (see Figure 2). It can execute both DML and DDL statements. DML statements use the database access plan formed by the SQL compiler, and DDL statements generate a request to the catalog manager to update the appropriate catalog tables. In either case, the executor manages the logical names, collects records from various tables using the file system, joins them, sorts where required, and returns the results to the host-language variables in the user program. The executor calls the file system with single-variable requests.

The File System. The file system manages the physical schema. Residing in the system library, the file system handles OPENs of files and indexes, partitioning, sending requests to appropriate disk processes, and buffering the replies. When a table is updated, the file system manages the updates to a table and all its secondary indexes. If a retrieval can be entirely satisfied by the index, the base table is not accessed. Typical uses of this feature are simple selects, such as Palermo's semi-joins (Palermo, 1974) and determination of minimum values.

Disk Process. Each disk volume is managed by a set of disk processes, which have a common request queue and a shared buffer pool. Disk processes implement file fragments and manage disk space, access paths, locks, log records, and a main memory buffer pool of recently used blocks. Each disk process authorizes the application process to the table when the file system sends the OPEN request. An OPEN to a protection view is authorized by the disk process.

SQL Conversational Interface. SQLCI is a dynamic SQL application, written in TAL (Tandem Application Language), that not only allows static and dynamic SQL statements to be executed interactively but also includes extensive documentation via on-line help text, on-line diagnostic messages, and a report writer. Report writer commands are separated from the SQL commands so that the user defines the answer set with SQL and the report with the report writer syntax. The report definition is an iterative process in which the report format can be altered and the report regenerated.

Program Preparation

As is standard with most SQL systems (*Database Language SQL 2*, 1986), NonStop SQL statements are embedded in the host language and bracketed by EXEC SQL and END-EXEC keywords. For a COBOL85 program, a NonStop SQL preprocessor scans the program text and produces a host-language program (COBOL) with the SQL statements replaced by calls to the SQL executor.

COBOL85 then compiles this new program. (Because their compilers recognize SQL statements directly, C and Pascal do not require separate preprocessors.) The NonStop SQL compiler transforms source SQL statements into a set of execution plans (one execution plan for each SQL statement in the source program) and registers the program in a catalog; this is called "explicit" compilation. After compilation, the program is ready for execution.

After it has been compiled once, a NonStop SQL program automatically recompiles when the database changes or the plans become invalid. For example, dropping an index or overriding logical names causes automatic recompilation. Updating statistics used for access-path selection will, optionally, force automatic recompilation. In addition, if a needed access path is inaccessible (e.g., the network is down), the program will be recompiled to work with the available data. (As the name suggests, automatic recompilation is transparent to the application program.)

Executing a NonStop SQL Program

The Tandem system is designed for on-line transaction processing. After the operator compiles, installs, and brings up the system, the system might run for several months without change or interruption of service. To eliminate extra instructions in the normal operating path, a rule for OLTP systems is to perform checking (e.g., opening files, checking addresses) at startup. Therefore, the NonStop SQL executor "OPENs" tables when the application first references them, and keeps the tables open until the execution plan is invalid and the application needs a new OPEN with a new redefinition time. Subsequent references to the table by another SQL statement in the same process will share this single OPEN.

The OPEN serves three purposes: It covers the redefinition/invalidation issue, authenticates the requester, and provides a virtual circuit between the SQL requester and the SQL disk server. When a transaction commits, all its locks are released and all its cursors are invalidated, but the OPENs continue to support the next transaction.

Key Innovations

NonStop SQL introduced key innovations in the areas of performance, integrated architecture, distributed database design, and fault-tolerant production features.

Performance

A single-variable query is a selection and/or projection on a single table involving only literals, host variables, and database columns. A SET operation is a single request that can update or delete multiple records. Single-variable queries and SET operations can be contracted (assigned) to a disk process. This is important to optimizing performance.

The disk process scans a table to find records that satisfy the selection expression, either returning qualifying projected records to the file system or performing the update or delete. To prevent one request from monopolizing it, the disk process returns control to the file system after ten I/Os; the file system then continues a request by reissuing it.

Tandem NonStop SQL is integrated with GUARDIAN 90.

NonStop SQL developers expected the benefits of remote execution of single-variable queries for set operations but did not anticipate the benefits for single-record operations. The original goal was, when executing the debit-credit transaction[1], to match (within 25%), the performance of the COBOL record-at-a-time interface. Surprisingly, when the application was measured, it used less CPU time and the same number of I/Os as the record-at-a-time interface (*NonStop SQL Benchmark Workbook*).

Though it can cost more to execute an SQL statement than to make an I/O request to a file manager, an SQL statement has more varied semantics. Because each SQL statement accomplishes more, there is a strong possibility that fewer statements will be executed.

Integrated Architecture

NonStop SQL is integrated with the operating system; when the system is up, NonStop SQL is up. One does not bring up or allocate an SQL database; it is simply there. This contrasts with most other SQL designs. In addition, because the operating system and SQL authorization are integrated, there is no "logon" to SQL; when the user logs on to the system, he is automatically logged on to SQL. The entire network provides a single-system database.

Naming and Security. Having *site.process. directory.object* as one system-wide naming convention simplifies learning and operation. These names are used for tables, indexes, views, and programs. Naming of columns follows the ANSI SQL conventions. Integrity constraints are named and numbered so that diagnostic messages can explain which constraint is violated. For similar reasons, NonStop SQL adopted GUARDIAN 90 security.

[1] A simple OLTP application that updates three tables by key and inserts a record in a fourth table (Anon., et al., 1985).

Transaction Management. Tandem's ENCOMPASS data management system provides a mechanism that includes transaction rollback (TMF) and distributed transactions (Borr, 1981). NonStop SQL integrates with this transaction manager, so that a single transaction log (audit trail) is maintained at each site. This log provides undo, redo, and media recovery for old (ENSCRIBE) and new (SQL) data. One transaction can contain both ENSCRIBE and SQL calls, and is recorded in a single log per site. A single transaction management facility for all data access greatly simplifies system management.

Distributed Database

A distributed database has many aspects. The Gartner Group (Braude, 1987) and Date (Date, 1987) have published criteria for measuring the degree to which a product can be considered distributed. The most important factors are location transparency (data that can be accessed from anywhere in a network), local autonomy, data distribution, catalogs, and processing. NonStop SQL satisfies all these criteria.

Local Autonomy. Local autonomy requires that NonStop SQL provide access to local data even if part of it is unavailable and the site is isolated from the rest of the network. For compiled SQL plans, this means that the SQL compiler must automatically and transparently pick a new plan if a chosen access path (i.e., index) becomes unavailable.

Data definition operations are more difficult. Dropping a partitioned table requires work—both updates to the catalogs and deletion of the files—at each node in which a partition resides. Changing table attributes has similar requirements. In general, NonStop SQL requires that all nodes related to a table participate in the DDL operation. If any relevant node, catalog, or disk process is unavailable, the DDL operation is denied. In a distributed system, local autonomy is at odds with data integrity. For DDL operations, Tandem elected in favor of integrity over local autonomy.

Data. A physical file, which may be horizontally partitioned across multiple disks and nodes, represents the table content. Indexes on a table may also be partitioned across multiple disks and nodes that can differ from the location of the table's partitions. In addition, NonStop SQL provides a unified view of the database; it eliminates the concept of DBSPACE (*IBM Database 2 General Information Manual*, 1986) or separate, autonomous databases offered by other vendors.

Catalogs. A traditional SQL system consists of a catalog (a set of tables) that includes information describing the tables and other database objects. The catalog is restricted to a single system. The database, which is defined by all objects registered in that catalog, is distinct from any other database represented by other catalogs on the same or remote systems.

The NonStop SQL dictionary, on the other hand, is the union of all such catalogs in a network; the dictionary is distributed and there are no barriers to accessing SQL data throughout a Tandem network. To preserve local autonomy, objects must be registered in a local catalog, and to ensure the dictionary's integrity, TMF protects all catalog tables.

Because there can be many catalogs, the file label of each table, view, or program includes the name of the catalog for that object. Information about a table is replicated at every site having a fragment of the table, so that the local parts of the table can be accessed even if the site is disconnected from the network.

Processing. Currently NonStop SQL provides distributed processing by contracting single-variable queries to remote disk processes. The current join strategy is to return the results of each single-variable query (after selection and projection) to the SQL executor running as part of the application process.

Production Features

NonStop SQL production features include defines, locking, constraints, object files and language support, host language features, CONTROL TABLE, and utilities.

Defines. System administrators and application designers need to be able to bind a program to new tables without altering the source code. This is necessary in production systems where a program is tested in one environment and moved to a production environment, in distributed systems where programs are duplicated at different sites, and in situations where a report runs against many instances of a generic table. IBM JCL and COBOL FD statements solved this problem in 1964, but most SQL implementations reintroduce the problem. NonStop SQL, however, offers logical names, called *defines*, which allow users to rebind a program's table names at SQL compile time or run time without altering the source program.

Locking. Major NonStop SQL DML innovations are in the areas of locking and consistency. The locking features include table, set, and row granularities; automatic escalation to coarser granularity; implicit or explicit shared and exclusive lock modes; three degrees of consistency (selectable on a per-statement basis); and a LOCK TABLE verb. Deadlock detection is via timeout. The default timeout is 60 seconds. Most defaults can be overridden per statement or using CONTROL TABLE.

All update operations on transactional (audited) files automatically acquire exclusive locks held to end-of-transaction (degree 1 consistency is automatic). The programmer has the option of accessing dirty data (BROWSE ACCESS), cursor stability (STABLE ACCESS), or repeatable reads (REPEATABLE ACCESS). These correspond to degree 1, 2, and 3 consistency (Gray, 1976).

Constraints. Users may add integrity constraints (named, single-variable queries) to a table. When the constraint is first defined, it is validated against the table. Thereafter, any insert or update operation that violates the constraint will be rejected. The disk process (file server) enforces the constraints, which removes many integrity checks from the application program. Updates through protection views obey these constraints.

Since constraints are named and may be added or dropped at any time, NonStop SQL's implementation is slightly more general than the ANSI SQL definition of CHECK CONSTRAINT (*Database Language SQL*, 1986).

Object Files and Language Support. Tandem's implementation of program compilation is similar to the original System R implementation (Astrahan, 1986). NonStop SQL, however, introduces the binding of SQL source and object programs with the T16 object modules. The resulting object program is a single object that can be moved, copied, archived, or purged without having to manipulate one or more separate "access modules." In contrast, most other SQL systems store the SQL program in the catalogs, requiring special handling and catalog access at run time.

The Tandem BINDER program was modified to support SQL source program and object sections and, also, to support relationships between the object program and its SQL sections. The binder combines code sections, data sections, symbol table sections (for the symbolic debugger), and other types of sections from separate compilations to produce a single executable object-program file.

Because the SQL compiler reads SQL statements from the SQL source section of the object-program file, programs can be archived and moved without accessing the SQL source. This greatly simplifies the management of SQL programs. In this area, as with many others, the close integration of NonStop SQL with standard system tools has considerable benefits in simplicity and functionality.

Host Language Features. NonStop SQL's programmatic interface has many features to ease programming, including:

- Comprehensive diagnostics embedded in output listings.
- Ability to invoke data declarations of tables from the catalogs.
- Support for WHENEVER (exception handling).
- Support for multiple levels of copy libraries.
- Generation of tracing information so that application programmers can trace errors back to source-language statements.

Because NonStop SQL supports separate compilation, a cursor may be defined in one compilation and used (e.g., OPENed, FETCHed) in another separately compiled program. C, Pascal, and COBOL85 support are currently available, and TAL support is under development. Integrating C and Pascal support directly into the compilers eliminates the extra listings and work files resulting from preprocessing.

CONTROL TABLE. A production-oriented, OLTP-capable RDBMS had to provide user controls not included in the SQL standards. Rather than change standard SQL syntax, Tandem added the CONTROL TABLE verb to allow specification of "lock-wait" duration, "table-versus-record" locks, and "bounce" locks (i.e., never wait for a busy resource). NonStop SQL also allows specification of consistency on the basis of either a table or an SQL statement.

Tandem's BINDER program was modified to support SQL source programs.

This design contrasts with other SQL systems that associate control with the transaction or program rather than the statement. Finer granularity control on a statement-by-statement basis or table basis is essential for tuning high-performance applications.

Utilities. A distributed database poses new challenges to managing data. Typical nondistributed SQL systems have taken the view that the database is a single physical object to be backed up and restored. In Tandem's view, this is insufficient. The NonStop SQL utilities (BACKUP/RESTORE, TMF, LOAD, COPY, CONVERT, DUP) must manage a logical view of the database: individual tables, partitions, and so on. NonStop SQL developers are still discovering new opportunities to improve these capabilities (e.g., adding, dropping, and splitting partitions).

Special Facilities

EXPLAIN. To assist with application design and tuning, the EXPLAIN facility produces a report showing the access plan chosen for an SQL statement issued from SQLCI or for all SQL statements in an SQL program. The report documents the use of indexes, sorting, and join operations.

HELP and ERROR Text. Documentation for NonStop SQL is available in the NonStop SQL manuals and, also, on-line via the SQLCI facility. The documentation that appears using SQLCI is identical to the information appearing in the manuals. The on-line documentation is easy to maintain because it is derived from the same source as the manuals.

Where-Used Reports. Many customers have requested a facility to generate reports or queries indicating which programs use (or depend on) particular objects (e.g., tables, views). The NonStop SQL DISPLAY command provides that facility.

Conclusion

In the last few years, virtually every commercial computer vendor has built or bought an SQL system. NonStop SQL is unique in that it offers:

- Distributed data, distributed execution, and distributed transactions with full location transparency.
- A data management system that runs on small and large computers.
- Toleration of any single fault without interrupting service.
- The first high-performance SQL benchmarked at over 200 transactions per second (tps) with no bottlenecks in sight.
- A cost per transaction comparable to non-SQL, record-at-a-time, high-performance data management systems.

NonStop SQL can be used for both decision support and OLTP, and disposes of the myth that relational systems are inherently slow. The combination of SQL semantics and a message-based, distributed operating system revealed that the message savings of a high-level interface pay for the extra semantics of the SQL language when compared to record-at-a-time interfaces.

NonStop SQL is the first SQL system to be integrated with an operating system. The SQL executor and file system work together so that the disk process directly executes single-variable SQL queries. Authorization covers SQL objects, programs include SQL sections, and the measurement facility measures SQL events. Because of these characteristics, NonStop SQL has considerable benefits in usability, simplicity, and performance.

References

Anon., et al. 1985. A Measure of Transaction Processing Power. *Datamation.* Vol. 31.7, pp. 112-118.

Astrahan, M., et al. 1986. System R: A Relational Approach to Database Management. *ACM Transactions on Data Base Systems.* Vol. 1.2.

Borr, A. J. 1981. Transaction Monitoring in ENCOMPASS: Reliable Distributed Transaction Processing. In *Proceedings of the Seventh International Conference on Very Large Data Bases*, September. Republished as Tandem TR 81.2.

Braude, M. 1986. The Evolving Software Industry. In *Second Annual Software Management Strategies (SMS) Conference.*

Braude, M. 1987. Rules for Distributed DBMS. *Software Management Strategies.* Gartner Group, Inc. T-150-314.1, February.

Codd, T. 1982. Relational Database: A Practical Foundation for Productivity. *Communications of the ACM.* Vol. 25.2.

Database Language SQL. 1986. American National Standards Institute. ANSI X3.135-1986.

Database Language SQL 2 (ANSI working draft). 1986. American National Standards Institute. ANSI X3H2 87-8.

Date, C. 1987. Twelve Rules for a Distributed Data Base. *ComputerWorld.* June 8.

NonStop SQL Benchmark Workbook. Part no. 84160. Tandem Computers Incorporated.

Overview of SQL/Extension-1. American National Standards Institute. ANSI X3H2-86-14, February 1986.

Palermo, F. 1974. A Database Search Problem. *Information Systems: COINS IV.* Plenum.

Query Management Facility: General Information. IBM Manual GC26-4071. International Business Machines.

Tandem Database Group. 1987. *NonStop SQL: A Distributed, High-Performance, High-Availability Implementation of SQL.* Tandem Computers Incorporated. Part no. 83061. Also Tandem Computers TR87.4.

The Genesis of a Database Computer: A Conversation with Jack Shemer and Phil Neches of Teradata Corporation. 1984. *IEEE Computer.* November.

Acknowledgments

Many people deserve credit for this paper and the introduction of the NonStop SQL product. I would like especially to acknowledge the entire NonStop SQL development team for the April 1987 Tandem Database Group paper (Tandem Database Group, 1987), from which most of this material has been derived.

Howard Cohen is currently the manager of NonStop SQL development. He has worked on NonStop SQL since 1984, starting as project technical leader. Before coming to Tandem, he worked for a major semiconductor manufacturer responsible for defining and implementing an information center, and for two timesharing vendors as a software developer and a manager of database development. Howard holds a B.S. in Mathematics from Harvey Mudd College and an M.S. in Computer Science from Stanford University.

NonStop SQL Optimizer: Basic Concepts

Tandem's NonStop SQL offers high performance as well as the ease of use and high functionality associated with the SQL database language. The NonStop SQL optimizer, a component of the NonStop SQL compiler, plays an important role in the high performance of NonStop SQL by automatically selecting the most efficient access plan for retrieving data from the database.

In a traditional database management system (DBMS), such as IMS or ENSCRIBE, there is a lack of data independence between the database and the application. This has two implications. First, the application must be aware of the underlying physical structures of the database. If these physical structures change, the application must be modified to reflect the new physical structures of the database.

Second, the application has the responsibility of selecting the access plan. The application must explicitly choose an index to access a file. If multiple files are to be accessed, the application must specify the order in which each file is to be accessed and identify the index that will access each file. Such specifications can become very complex.

By contrast, NonStop SQL has data independence and automatic access-plan selection (Tandem Database Group, 1987). The NonStop SQL optimizer generates an efficient access plan to evaluate a given query. This frees the application programmer to concentrate on designing query results rather than specifying the means to achieve those results.

This is the first of two articles describing the NonStop SQL optimizer. This article reviews the basic concepts important for understanding the process of selecting an access plan. They include the characteristics of tables and indexes, the concept of selectivities, and the various methods of performing joins. They are discussed in the following order:

- Table and index.
- Sequential block buffering (SBB).
- Selectivity.
- Join evaluation.

This article assumes readers are familiar with the SQL language and relational terms such as selection and projection (Date, 1986).

The second article, "NonStop SQL Optimizer: Query Optimization and User Influence," describes the heuristics used by the optimizer in performing automatic access-plan selection. The second article also discusses ways in which the user can influence the optimizer in choosing an access plan.

Table and Index

In NonStop SQL, data items are logically stored in a *table* (Date, 1986), which is made up of user-defined fields called *columns*. Each SQL table is implemented as a physical file that can be key-sequenced, relative, or entry-sequenced (Tandem Database Group, 1987).

Each physical file has a unique *primary key*. If the file is key-sequenced, the user or a default called SYSKEY defines the primary key to the file. If the file is relative or entry-sequenced, the key is the record number and has the name SYSKEY. The physical file is sometimes called the *primary index*. Additional indexes may also be defined on one or more columns of the table; the columns need not be contiguous. Each additional index is implemented as a separate key-sequenced file.

To access records in a table, one can read the underlying file sequentially or supply a primary key value. If an index is available, one can read the index sequentially or supply the key value for the index, obtain the primary key value from the index record, and use this key value to read the underlying file of the table.

An index is an efficient way to access data only if the number of records to be retrieved is small. For example, consider a table named INVENTORY with the columns ITEM_NO, ITEM_NAME, ITEM_DESCRIPTION, RETAIL_PRICE, and PRODUCER. INVENTORY is implemented as a key-sequenced file. The total length of each record is 400 bytes and the block size is 4000 bytes. The file contains 100,000 items. ITEM_NO is the primary key column. An index on the column PRODUCER is also available. The index column PRODUCER and the primary key field, ITEM_NO, total 24 bytes. Suppose the user wants to find the RETAIL_PRICE information on items with an ITEM_NO between 20 and 2000 with the query:

```
SELECT ITEM_NAME, RETAIL_PRICE,
   PRODUCER
FROM INVENTORY
WHERE ITEM_NO BETWEEN 20 AND 2000
```

If 1000 items fall within this range and if NonStop SQL uses the primary key, only 1000 records need to be retrieved because there is a begin and end key (i.e., ITEM_NO BETWEEN 20 AND 2000). The number of physical I/Os required is about 100 (10 records per block). On the other hand, if NonStop SQL uses the index, the entire index must be read because there are no begin-key and end-key values for the index column PRODUCER. Because the ITEM_NAME column is not part of the index, the table must be read for each record in the index. As a result, the number of physical I/Os would be far greater than 100.

As a second example, suppose the user wants information on the 1000 items produced by DEL MONTE:

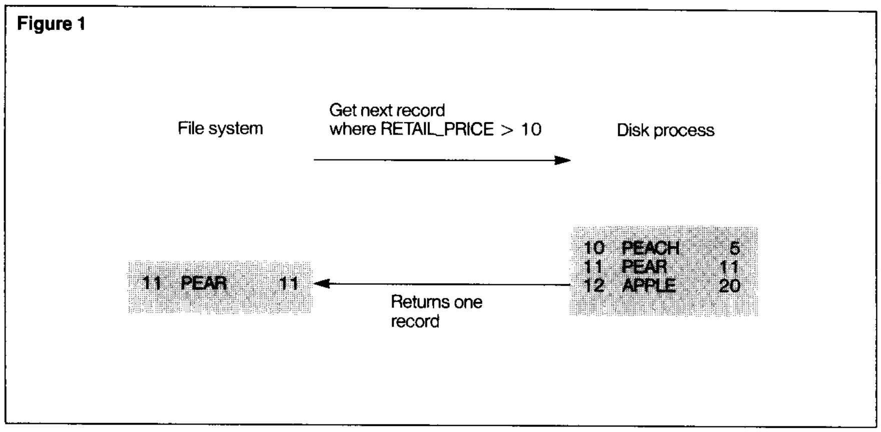
```
SELECT ITEM_NAME, RETAIL_PRICE
FROM INVENTORY
WHERE PRODUCER = "DEL MONTE"
```

If the index is not used, the whole table must be read because there are no restrictive begin- and end-key values for the primary key ITEM_NO. SQL must perform about 10,000 physical I/Os (100,000 items at 10 records per block) to read through the table. However, if SQL uses the index on PRODUCER, the number of physical I/Os required is about 1000 because the begin-key and end-key values (i.e., DEL MONTE) of the index have been specified in the query.

Another situation in which using an index is more efficient than reading the table directly is when all the information can be obtained from the index file. For example, the user wants the information on PRODUCERs and ITEM_NOs:

```
SELECT ITEM_NO, PRODUCER
FROM INVENTORY
```

If the index is not used, the number of physical I/Os is around 10,000 (see previous examples). If the index on PRODUCER is used, the whole index must be read. This requires only about 600 I/Os (each 4000-byte index block can contain 4000/24 ≅ 160 index records, and 100,000 records require about 600 pages). However, because all the requested columns (PRODUCER and ITEM_NO) can be found in the index, there is no need to read the table.



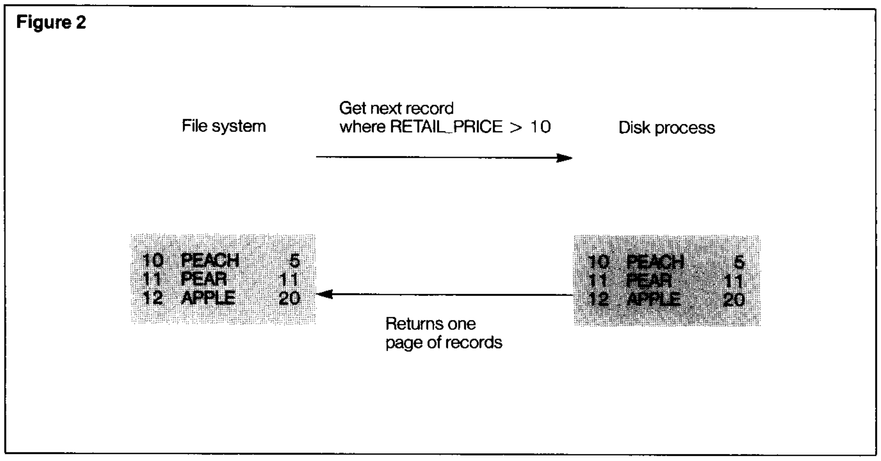


Figure 1.
The record-at-a-time interface returns one record per message.

Figure 2.
The physical sequential block buffering (SBB) returns one page of records per message.

A common misconception holds that if the user specifies some columns of an index, NonStop SQL will use the index. This is not necessarily true. For example, assume that the first two columns of an index are PRODUCER and ITEM_NAME, respectively. Specifying a predicate on ITEM_NAME alone would not make the index very useful. In the current implementation, the general rule is that the prefix (PRODUCER, in this example) of a key must be specified before it is efficient to use the index to retrieve records.

Sequential Block Buffering

In Tandem systems, the file system is a collection of system library routines that run in the process environment of the application process. Through file system-procedure invocations, the application process sends requests to the disk process.

In ENSCRIBE, each request for a record from the application process causes a record to be returned from the disk process unless the application process requests the sequential block buffering (SBB) feature. In this case, the disk process returns a copy of a physical block of records to the file system. When the application process requests the next record, the file system returns the next record from the copy of the physical block of records. Therefore, SBB reduces the amount of requests (messages) between the file system and the disk process by the file's physical blocking factor (i.e., the number of records per block).

In addition to supporting the record-at-a-time interface, NonStop SQL supports physical and virtual SBB. Physical SBB was described in the previous paragraph. In virtual SBB, the disk process does the selection and projection of data. (For more information, refer to the accompanying article, "High-Performance SQL through Low-Level System Integration.") The first objective of virtual SBB is to further reduce the number of messages between the file system and the disk process. The second objective of virtual SBB is to reduce the amount of data transfer between the file system and the disk process. Virtual SBB is unique to NonStop SQL.

The following example illustrates the differences among these interfaces. Consider the query:

```
SELECT ITEM_NAME, RETAIL_PRICE
FROM INVENTORY
WHERE RETAIL_PRICE > 10
   FOR REPEATABLE ACCESS
```

The table INVENTORY contains 100 records. Each record contains the columns ITEM_NAME, RETAIL_PRICE, ITEM_ON_HAND, and COMMENTS. There are 90 items with a RETAIL_PRICE greater than 10. The sizes of the columns are 20, 4, 4, and 400 bytes, respectively—a total of 428 bytes. There is no index on the table. The query is evaluated by sequentially reading the INVENTORY table.

In the record-at-a-time interface (Figure 1), the disk process returns each complete record that satisfies the predicate (RETAIL_PRICE > 10) to the file system. To evaluate this query, the file system will send 90 messages to the disk process. The disk process transfers 38,520 bytes of data (90 records of 428 bytes each) to the file system.

If physical SBB (Figure 2) is used to evaluate the query, the disk process returns a physical block of records to the file system. The file system then examines each record in the returned block and tests the record against the predicate. After all the records in the block have been processed, the file system asks for another block. The amount of data transferred from the disk process to the file system is the same as in the record-at-a-time case. However, the file system only needs to send two messages (assuming 4-Kbyte data blocks) to the disk process.

With the virtual SBB (Figure 3) interface, the disk process returns in a block only the requested columns from records that satisfy the predicate. In the previous example, only the columns ITEM_NAME and RETAIL_PRICE are returned to the file system. Therefore, the disk process returns 2160 bytes of data (90 records at 24 bytes per record) to the file system. Because the answer to the query can be contained in one 4-Kbyte page, the file system needs to send only one message to the disk process.

Figure 3

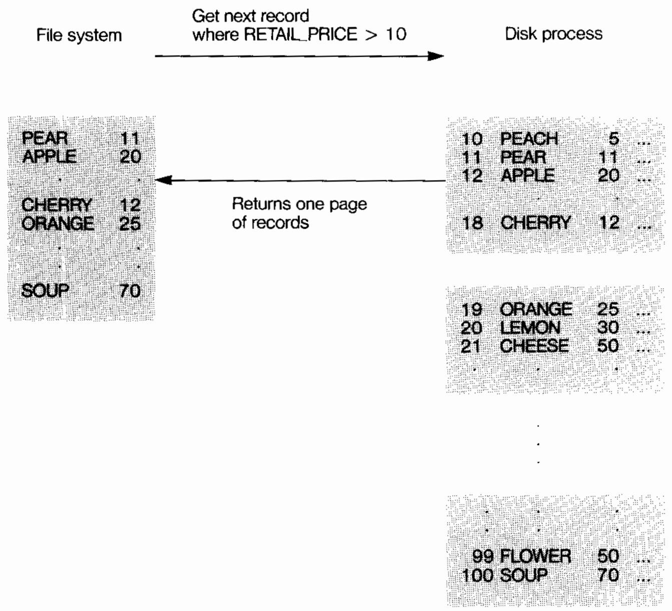


Figure 3.

Virtual sequential block buffering (SBB) returns one page of records (with selection and projection) per message. Data may be from more than one physical data page.

Selectivity

Selectivity, defined as the fraction of records that satisfy a condition, is a concept central to selecting an access plan in NonStop SQL and other relational systems (Selinger, 1979). There are three types of selectivity: predicate, table, and index.

Predicate Selectivity

A predicate is a condition that a record must satisfy in order to be returned to the application. For example:

ITEM_NO > 10

is the predicate in the query:

```
SELECT ITEM_NAME, RETAIL_PRICE
FROM INVENTORY
WHERE ITEM_NO > 10
```

The selectivity of a predicate is the fraction of records in a table that satisfy the predicate. For example, assume there are 100 items in the INVENTORY table. The selectivity of the predicate

ITEM_NO > 10

is 0.9, or 90%, if 90 out of 100 records will satisfy the condition specified by the predicate.

Table Selectivity

Table selectivity is the fraction of records that satisfy all the predicates of a query. Using the previous example where 90 out of 100 items in the table satisfy the search condition, the table selectivity is also 0.9, or 90%. Generally, because more than one predicate may be specified with a query, the table selectivity is not equal to predicate selectivity. For example:

```
SELECT ITEM_NAME, RETAIL_PRICE
FROM INVENTORY
WHERE ITEM_NO > 10 AND
   ITEM_NAME = "PINEAPPLE"
```

In this query, there are two predicates. Each predicate has its own selectivity. Assume that the predicate:

ITEM_NAME = "PINEAPPLE"

has a selectivity of 0.01, or 1%. The table selectivity is the product of the individual predicate selectivities:

$0.9 \times 0.01 = 0.009$

In general, if *n* predicates are all connected by AND operators, and if the predicates are independent of one another (i.e., the values in different columns are independent of one another), the table selectivity of the predicates is estimated as the product of the *n* individual predicates. If two predicates are connected by the OR operator, the composite selectivity is estimated as:

sum of the individual predicate selectivities –
product of individual predicate selectivities

Index Selectivity

Index selectivity is the fraction of index records that must be examined in evaluating a query. Consider the query:

```
SELECT ITEM_NAME, RETAIL_PRICE
FROM INVENTORY
WHERE ITEM_NO = 20
```

Suppose there is a unique index with a key on ITEM_NO in the INVENTORY table. If this index is chosen to evaluate the query, only one record must be examined because the index is unique. If there are 100 items, the index selectivity for the index is 1%, or 0.01.

Computation of Selectivity

NonStop SQL estimates selectivities based on statistics obtained prior to the compilation of queries. Statistics on a column can be collected with the UPDATE STATISTICS command (*NonStop SQL Programming Reference Manual*). Furthermore, the data is assumed to be uniformly distributed within the range specified by the statistics. The statistics used by NonStop SQL to compute selectivities are:

- The second-high and second-low values of a column.
- The number of unique values of the column.

To avoid the extreme values that may be very different from the rest of the values, NonStop SQL does not use the first-high and first-low values of a column.

The selectivity of a predicate involving a column with a numeric attribute is computed as a linear extrapolation of the values within the range of values specified by the second-high and -low values. For example, the RETAIL_PRICE column of the INVENTORY table has a second-high value of 99 and a second-low value of 2. The selectivity of the predicate:

RETAIL_PRICE > 10

is

$$\frac{\text{second-high value} - \text{supplied value}}{\text{second-high value} - \text{second-low value}}$$

which equals

$$\frac{99 - 10}{99 - 2} \cong 0.91$$

For predicates of the form "column = value," the selectivity of the predicate is:

$$\frac{1}{\text{unique values of column}}$$

Selinger (1979) gives further examples of predicates and their selectivities.

If statistics are not available for a column or if the value specified in a predicate is a host variable (as in a COBOL program), NonStop SQL assumes an arbitrarily chosen selectivity for the predicate if the predicate is not of the form "column = value." The selectivity of a predicate involving host variables cannot be computed because the value of the host variable is not known at compile time. For example, the selectivity of the predicate:

RETAIL_PRICE > :*host_variable*

is chosen to be 0.33. However, the selectivity for:

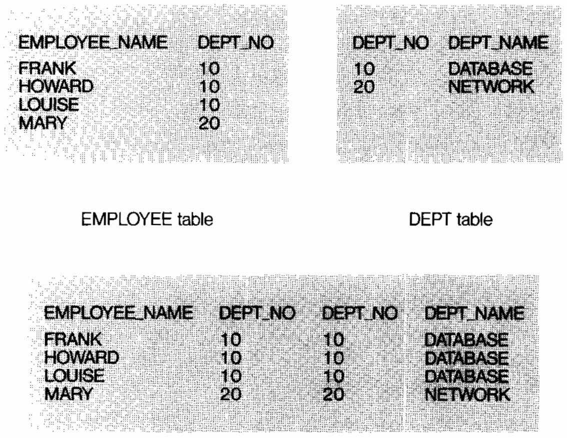
RETAIL_PRICE = :*host_variable*

can be reasonably computed, since the computation does not depend on the supplied value. The selectivity of a predicate of the form "column = value" is:

$$\frac{1}{\text{number of unique values of the column}}$$

If the default selectivity differs very much from the actual selectivity, NonStop SQL may choose an inefficient access plan for the query. Therefore, it is strongly recommended that users periodically collect statistics with the UPDATE STATISTICS command. The accompanying article, "NonStop SQL Optimizer: Query Optimization and User Influence," has more information on the UPDATE STATISTICS command.

Figure 4



EMPLOYEE_NAME	DEPT_NO
FRANK	10
HOWARD	10
LOUISE	10
MARY	20

EMPLOYEE table

DEPT_NO	DEPT_NAME
10	DATABASE
20	NETWORK

DEPT table

EMPLOYEE_NAME	DEPT_NO	DEPT_NO	DEPT_NAME
FRANK	10	10	DATABASE
HOWARD	10	10	DATABASE
LOUISE	10	10	DATABASE
MARY	20	20	NETWORK

Figure 4.

EMPLOYEE table joined with DEPT table and EMPLOYEE. DEPT_NO = DEPT.DEPT_NO.

Join Evaluation

Users may "join" two tables to form a new, wider table with more columns. When tables are joined, each new record is formed by concatenating two records, one from each of the original tables. The paired records must have the same value in the joining column. For example, the query:

```
SELECT EMPLOYEE_NAME, DEPT_NAME
FROM EMPLOYEE, DEPT
WHERE EMPLOYEE.DEPT_NO =
   DEPT.DEPT_NO
```

joins the tables together on the column DEPT_NO (Figure 4). The predicate EMPLOYEE.DEPT_NO = DEPT.DEPT_NO is called a join predicate. Note that this query also requests a projection of the columns EMPLOYEE_NAME and DEPT_NAME. The join in this example is known as "equi-join." If the joining criterion is a comparison operator other than equality, the join is known as "theta join" (Date, 1986). NonStop SQL supports both types of joins. Two tables may be joined even if there are no joining predicates. In this case, concatenating every record in one table with every record in the other table creates the new table.

The most popular methods of implementing the join operation are the nested loop and sort merge methods (Selinger, 1979). NonStop SQL implements these two methods.

Nested Loop

The nested loop algorithm retrieves records one at a time from a table called the *outer table* and compares them with the records in a second table, called the *inner table*. The algorithm retrieves the records from the inner table that satisfy the join predicate and concatenates them with the corresponding records from the outer table.

Sort-Merge-Join

The sort-merge-join algorithm requires that the joining columns of the outer and inner tables be in ascending or descending order. If the join column of a table is not in the required order, the table is sorted on the join column into a temporary table. A record is retrieved from the outer table, another record is retrieved from the inner table, and the values of the join columns for the two records are compared.

If the values are the same, the records are concatenated, projected, and returned to the user, and the position of this inner record is remembered. The next inner record is retrieved and the process is repeated until the join-column values of the inner and outer table records are different. The next outer record is then retrieved; if the join-column value is the same as before, the inner table is positioned to the "remembered" position and the process is repeated.

If the join-column value of the inner record is less than that of the outer record, the next inner record is retrieved until the value of the inner record is greater than or equal to that of the outer record. If the join-column value of the inner record is greater than that of the outer record, the next outer record is retrieved until the outer record has a value greater than or equal to that of the inner record.

This process is repeated until all the records from the outer table have been examined.

Conclusion

The basic concepts of access-plan selection discussed in this article are important to understanding the process of access-plan selection used by the NonStop SQL optimizer. Different indexes provide different degrees of efficiency in accessing a table. Selectivities (predicate, table, and index) are used to evaluate the efficiency of an index. Sequential block buffering can be used to increase the efficiency of an index. Finally, different join methods are used to improve the efficiency of evaluating join queries.

References

Date, C. 1986. *An Introduction to Database Systems.* Addison Wesley.

NonStop SQL Programming Reference Manual. Part no. 82318. Tandem Computers Incorporated.

Selinger, P., et al. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the International Conference on Management of Data.* Association for Computing Machinery (ACM).

Tandem Database Group. 1987. *NonStop SQL, a Distributed High-Performance, High-Availability Implementation of SQL.* Tandem Technical Report 87.4. Tandem Computers Incorporated.

Acknowledgments
I would like to thank the members of the Tandem Database Group for contributing ideas on optimization and reviewing this paper.

Mike Pong is currently a member of the SQL Compiler Group in Transaction Networks Division. Before joining the SQL Compiler Group, Mike designed and implemented the autorollback feature of DP1 TMF.

NonStop SQL Optimizer: Query Optimization and User Influence

The NonStop SQL optimizer plays an important role in the high-performance operation of NonStop SQL. For each SQL query, the optimizer generates an access plan that efficiently retrieves the requested data from the database. By automatically selecting an access plan, the optimizer frees the application programmer to concentrate on designing query results, thus improving programmer productivity.

This is the second of two articles describing the NonStop SQL optimizer. The first article, "NonStop SQL Optimizer: Basic Concepts," briefly outlines the advantages of the NonStop SQL optimizer over the mechanisms of traditional database management systems. It then reviews the basic concepts important for understanding the optimizer. Readers who have not had extensive experience with query optimization are urged to read "Basic Concepts" before reading this article.

This article describes the heuristics used by the optimizer in performing automatic access-plan selection. It also discusses ways in which the user can provide access information (such as creating additional indexes) that will influence the optimizer to select a more efficient access plan.

The Goal of Query Optimization

The goal of the NonStop SQL optimizer is to select the most efficient access plan to evaluate a query. For a query that references a single table, an access plan consists of directions to access the table using a specified index and begin/end keys. For a query that references multiple tables, an access plan also specifies the order in which the tables should be accessed. NonStop SQL defines the most efficient access plan as the one that takes the least time to complete the evaluation of a query.

A number of parameters affect the execution time of a query, including the number of physical I/Os to be performed, the number of instructions to be executed, the number of sorts to be performed, and the amount of data to be transferred between processes. In determining the most efficient access plan, the optimizer performs the query modifications and determines:

- The indexes that should be considered.
- The type of sequential block buffering (SBB) that should be used.
- The cost of using each index.
- The order in which tables should be accessed (in a query that references multiple tables).
- The time when a subquery should be evaluated.
- The most efficient access plan when some indexes are not available.

Query Modification

Query modification refers to the modification of a query into another form that is logically equivalent to the original query. The modified query either exposes information hidden in the original syntax or reduces the complexity of the query. This section describes the query modifications performed by the NonStop SQL optimizer.

LIKE Processing

NonStop SQL supports the LIKE predicate of the SQL language. The LIKE predicate allows the user search for records that match a pattern. For example, the query:

```
SELECT NAME, PHONE_NUMBER
FROM PHONE_BOOK
WHERE NAME LIKE "CH%"
```

will retrieve all NAMEs that start with the string "CH" (e.g., CHARLES, CHRIS, and so on). NonStop SQL transforms the above query into:

```
SELECT NAME, PHONE_NUMBER FROM
   PHONE_BOOK
WHERE NAME LIKE "CH%"
   AND NAME > = "CH"
   AND NAME < "CI"
```

With this modification, NonStop SQL can take advantage of an index on NAME (if one exists) and use values in the predicates:

```
NAME > = "CH" and NAME < "CI"
```

as the begin and end keys to the index. Thus, only records that are alphabetically equal to or after "CH" and before "CI" will be retrieved.

Remove Sort Requests

A sort is logically required when a query specifies that:

- The result should be presented in a certain order (using the ORDER BY clause).
- Duplicates should be removed (via the DISTINCT key word).
- The result should be grouped (via the GROUP BY clause) on certain columns.

Because sorting is an expensive operation, NonStop SQL tries to minimize the number of sorts that must be performed for a query.

When a query contains both an ORDER BY and a SELECT DISTINCT request, it might be possible to use one sort to satisfy both requests if the ORDER BY list is a subset of the DISTINCT list. Consider the query:

```
SELECT DISTINCT ITEM_NAME,
   RETAIL_PRICE,
   RETAIL_PRICE * ITEM_ON_HAND
FROM INVENTORY
ORDER BY ITEM_NAME, 3
```

Assume that INVENTORY is an entry-sequenced table with no index. The query can be evaluated with a single sort if the sorting with the no duplicate option is on columns ITEM_NAME, RETAIL_PRICE × ITEM_ON_HAND, and RETAIL_PRICE. The result will be presented as specified in the SELECT list, but the sort column order will be altered.

In NonStop SQL, the formation of groups requires that the grouping columns must be in ascending or descending order. If they are not already in one of those orders, they must be sorted before the grouping operation can be performed. If a query contains both an ORDER BY and a GROUP BY request, a sort due to the ORDER BY request may be eliminated if the ORDER BY list is a "prefix" of the GROUP BY list. For example:

```
SELECT ITEM_NAME, RETAIL_PRICE,
   COUNT (*)
FROM INVENTORY
GROUP BY ITEM_NAME, RETAIL_PRICE
ORDER BY ITEM_NAME
```

is equivalent to:

```
SELECT ITEM_NAME, RETAIL_PRICE,
   COUNT (*) FROM INVENTORY
GROUP BY ITEM_NAME, RETAIL_PRICE
```

If a query contains both a DISTINCT function and a GROUP BY request, and if the DISTINCT column is not already in the GROUP BY list, the sort due to the DISTINCT function can be avoided by adding the DISTINCT column to the list of ordering columns when performing the sort. This technique works because the SQL executor can detect a change in value in the DISTINCT column if it is in sorted order. For example, the query:

```
SELECT PRODUCER, COUNT (DISTINCT
   CATEGORY) FROM INVENTORY
GROUP BY PRODUCER
```

asks for a list of producers and a count of the different categories of items produced by the producer. The sort due to the COUNT DISTINCT request can be avoided if, in sorting for the GROUP BY request, the sort columns are PRODUCER and CATEGORY instead of PRODUCER only. Consider this sorted list (ITEM_NAME has been added for clarity only):

PRODUCER	CATEGORY	ITEM_NAME
DEL MONTE	FRUIT	PINEAPPLE
DEL MONTE	FRUIT	PEACH
DEL MONTE	VEGETABLE	BEANS

When performing the grouping on PRODUCER, the SQL executor remembers the last value for CATEGORY and increments the count for CATEGORY only if the new value for CATEGORY is different from the old one (as in the third record).

Finally, NonStop SQL also avoids unnecessary sorts by removing the ORDER BY clauses if no column is present in the SELECT list. The ORDER BY clause is also removed if it is in a subquery.

Determining Useful Indexes

In some queries, the most efficient access plan can be determined without doing much computation. In others, an index that seems to have no use may actually play an important role in the query optimization. This section presents examples of these two cases.

The Halloween Problem

Consider the following query:

```
UPDATE INVENTORY SET RETAIL_PRICE =
   RETAIL_PRICE * 1.1
WHERE RETAIL_PRICE >20
```

The query requests that the price of all items in the INVENTORY table be increased by 10%. Assume there is a non-unique index on RETAIL_PRICE and the index contains the following records before the update:

RETAIL_PRICE
10
40

Suppose the index on RETAIL_PRICE is the chosen access plan in a query requesting records that satisfy the predicate:

RETAIL_PRICE >20

The system finds the record with a retail price of 40 and updates it to 44. When the system looks for the next record that satisfies the predicate, it finds the same record but with a value of 44 for RETAIL_PRICE. This goes on forever. This phenomenon is known as the "Halloween Problem."[1]

[1] This problem is supposed to have been discovered on Halloween; hence the name.

Many database management systems avoid the Halloween problem by ignoring the index on the column being updated (RETAIL_PRICE in the previous example) and choosing another index as the access path. Often, this results in an inefficient access plan. However, it is perfectly correct to choose the index on RETAIL_PRICE for the following query, even though RETAIL_PRICE is being updated:

```
UPDATE INVENTORY SET RETAIL_PRICE
   = 200
WHERE RETAIL_PRICE BETWEEN 300
   AND 400
```

If there is no other index for the INVENTORY table and the index on RETAIL_PRICE is not going to be used, the whole table must be read. If the table is large, using the index is much more efficient.

NonStop SQL will consider using the index on a column being updated if either one of the following conditions is satisfied:

- Predicates have specified all key columns in the index with the "equal" binary operator. For example:

```
UPDATE INVENTORY
SET RETAIL_PRICE = RETAIL_PRICE * 1.1
WHERE RETAIL_PRICE = 20
```

- No column is referenced on the right-hand side of the SET clause, and the index selectivity of the index is less than 20%. For example:

```
UPDATE INVENTORY
SET RETAIL_PRICE = 20
WHERE RETAIL_PRICE >80
```

The less-than-20% restriction for index selectivity limits the number of records updated more than once.

MIN and MAX Functions

The processing of the MIN or MAX function usually requires reading the entire table. However, if an index exists on the column that is an argument of the MIN or MAX function, reading the first or the last record will yield the MIN or MAX value. For example:

```
SELECT MIN(RETAIL_PRICE)
FROM INVENTORY
```

Assume that RETAIL_PRICE is the first key field of an index. In this case, other indexes need not be considered.

Deciding If SBB Should Be Used

One goal of the NonStop SQL optimizer is to minimize the number of messages and the amount of data transferred between the file system and the disk process. Using sequential block buffering (SBB) is one way to achieve this goal. However, the optimizer does not always choose to use it, since SBB involves a certain amount of overhead. For instance, if only one or two records must be retrieved, SBB will not be used.

If SBB is to be used, NonStop SQL must decide the type of SBB (physical or virtual) to use. Remember that in virtual SBB, the disk process does projections and selections before returning a virtual block to the file system. Therefore, physical SBB is used when the disk process can only do a minimal amount of filtering (selection and projection). For example, the query:

```
SELECT *
FROM INVENTORY
```

asks for the whole INVENTORY table. Virtual SBB would provide no savings in this case. In general, NonStop SQL uses physical SBB if both the following conditions are satisfied:

- More than two-thirds of a record must be retrieved or examined. (The value two-thirds is arbitrarily chosen.)
- Most records examined will satisfy all the predicates. (The difference between the table and index selectivities is very small.)

Assume that the primary key of the INVENTORY table is the column PRODUCER. The query:

```
SELECT *
FROM INVENTORY
WHERE PRODUCER < > "DEL_MONTE"
```

requests all columns. The index selectivity and table selectivity are the same. Physical SBB will be chosen to evaluate the query.

Physical SBB has one restriction. If the user specifies FOR CURSOR STABILITY or FOR REPEATABLE READ access in a query, physical SBB cannot be used because the disk process does not support the notion of "block level" locking. However, if the FOR BROWSE access is specified or the user has indicated to the SQL compiler that the table is to be locked via the CONTROL TABLE TABLELOCK command, NonStop SQL does consider using physical SBB.

Virtual SBB is used when the disk process can perform substantial filtering. In general, virtual SBB is used when one of the following conditions is satisfied:

- Less than two-thirds of a record must be retrieved or examined.
- Most records examined will not satisfy all the predicates. (The difference between the table and index selectivities is large.)

Consider the following query:

```
SELECT *
FROM INVENTORY
WHERE PRODUCER > "DEL_MONTE"
   AND RETAIL_PRICE BETWEEN 1 AND 2
```

Again, the primary key of the INVENTORY table is the column PRODUCER. Further assume that the selectivity of the predicate:

```
PRODUCER > "DEL MONTE"
```

is 80% (0.8) and the selectivity of the other predicate:

```
RETAIL_PRICE BETWEEN 1 AND 2
```

is 20% (0.2). The index selectivity is 0.8 and the table selectivity is 0.16 (0.8 × 0.2). Virtual SBB is used even though all columns are to be retrieved because only a small fraction of the records examined will satisfy all the predicates.

Cost Associated with an Access Plan

In determining the most efficient access plan, the NonStop SQL optimizer assigns a numeric cost to each index it considers. If the query references multiple tables, the optimizer also considers the different combinations in which the tables can be joined. Each of these combinations is also assigned a numeric cost. In the final phase of access-plan selection, the optimizer chooses the plan with the minimum numeric cost.

What Is Cost?

In NonStop SQL, cost is an estimate of the amount of time the system takes to complete evaluation of a query. It is an estimate because there are many variables that the optimizer is not aware of at compile time or that NonStop SQL cannot control. For example: the type of CPU can change at run time, the load of the system may not be accurate by the time that the query is executed, or the elapse time for the completion of a query varies depending on the type of output device. For all these reasons, cost in NonStop SQL does not carry a unit of time.

Though cost is an estimate and not an exact measure, it is very useful for comparing the relative efficiency of different access plans for a given query. Because the cost measurement is an estimator and cannot use exact units of time, NonStop SQL expresses cost in the "equivalent" number of physical I/Os that must be issued to complete the query.

Cost has many components, only one of which is the number of physical I/Os. However, to facilitate the computation of cost, all other components of cost are expressed in number of physical I/Os. For example, assume it takes 2000 instructions to evaluate a predicate in a 2-MIPS CPU. The time to evaluate the predicate is 1 msec. If a physical I/O takes 30 msec, the cost to evaluate the predicate is equivalent to 1/30 physical I/O.

In other database management systems (DBMS), the cost formula is much more simple. For example, the cost includes only the physical I/O cost.

The Cost of Accessing a Single Table

In NonStop SQL, the cost of using an index to access a table in a query that references only one table is:

Cost(index)
= Cost(physical I/O)
+ Cost(record overhead)
+ Cost(evaluating predicates)
+ Cost(transfer)
+ Cost(message)
+ Cost(sub-query)
+ Cost(sort)

The resolution of cost is one physical I/O. Therefore, if the cost of a component is less than one physical I/O, the cost for the component will be truncated to 0.

Physical I/O Cost. *Cost(physical I/O)* is the estimated number of physical I/Os that must be performed to retrieve all the records that satisfy the predicates of the query. This includes all physical I/Os to retrieve the requested data. Consider the query:

```
SELECT ITEM_NUMBER, ITEM_NAME,
   RETAIL_PRICE
FROM INVENTORY
WHERE RETAIL_PRICE >100
```

Assume there is an index on RETAIL_PRICE of the table INVENTORY. INVENTORY contains 10,000 records, and each record is 100 bytes. Assuming a page size of 4 Kbytes, INVENTORY has approximately 250 pages. ITEM_NUMBER is the primary-key column and is 4 bytes, and RETAIL_PRICE is also 4 bytes. Therefore, the index record has a size of 10 bytes (key tag + 4 + primary-key size), and the index has about 25 blocks. Finally, assume that 100 records, or 1% of the records, will satisfy the predicate.

If the query is to be evaluated using the primary key, 250 pages must be read from disk, and the Cost(physical I/O) would be 250. If the query is to be evaluated with the index, 102 pages must be read (two index pages + one data page for each qualifying index record), and the Cost(physical I/O) would be 102.

Record Overhead. *Cost(record overhead)* is the CPU time, expressed in terms of physical I/Os, associated with handling records. This includes the cost of setting up various control blocks and is dependent on the number of records examined. In NonStop SQL:

Cost(record overhead)
= overhead per record
× number of records to examine

For example, assume that a processor can perform 2 million instructions per second and its disks can perform 30 I/Os per second. If 2000 instructions are required before a record can be examined, the overhead per record would be approximately 0.03 I/O (2000 instructions would take 1 ms, which is approximately the time to perform 0.03 physical I/O). If 10,000 records must be examined, Cost(record overhead) would be 300.

Cost Per Predicate. *Cost(evaluating predicates)* is the average CPU time, expressed in terms of physical I/Os, spent in evaluating predicates. It is dependent on the number of records examined and the number of predicates that cannot be used as begin and/or end keys. In NonStop SQL:

Cost(evaluating predicates)
= number of predicates that cannot be used as a begin/end key
× number of records to be examined
× overhead in evaluating one predicate

In NonStop SQL, the same code performs the predicate evaluation even if the disk process, file system, and the SQL executor evaluate the predicate. The "overhead in evaluating one predicate" is a weighing factor computed in a fashion similar to the "overhead per record" in the previous section.

Message Cost. *Cost(message)* is the CPU time, expressed in terms of physical I/Os, spent in sending messages between the file system and the disk process. This measurement is dependent on the type of SBB being used. (The savings achieved by using SBB was discussed in the preceding article, "NonStop SQL Optimizer: Basic Concepts.") NonStop SQL computes Cost(message) as:

Cost(message)
= cost per message
× number of messages

The cost per message is a weighing factor computed in a fashion similar to overhead per record in the "Record Overhead" section.

Transfer Cost. *Cost(transfer)* is the estimated elapsed time, expressed in terms of physical I/Os, for transferring data from the disk process (possibly remote) to the file system. In general, transfer cost is negligible for local transfers; it becomes substantial with remote transfers. Cost(transfer) is computed as:

Cost(transfer)
= transfer rate
× amount of data to be transferred

For example, assume that 4000 bytes are to be transferred from a remote node to the local node, and that the two nodes are connected by one 4-Kbit-per-second communication line. Using the typical disk-transfer rate of 30 I/Os per second, Cost(transfer) is 240.

Subquery Cost. *Cost(subquery)* is the estimated cost of executing a subquery and is computed as *Cost(index')*, where *index'* is the index chosen to execute the subquery. The evaluation of subqueries will be discussed in the section "Subquery Processing."

Sort Cost. *Cost(sort)* is the estimated cost of sorting records in a particular order. (The sort would be initiated by an ORDER BY, DISTINCT, or GROUP BY request or by the use of the sort-merge join.) NonStop SQL supports two types of sort: in-memory and external. An in-memory sort is very efficient for a small number of records (less than 400). Tandem FASTSORT is used when more than 400 records are to be sorted (Tsukerman, 1986). Again, Cost(sort) is the estimated time to sort the specified records expressed in terms of equivalent physical I/Os.

The Effects of Indexes and Predicates on Costs

Because the complete record is not stored in an index, the cost of using an index is different from the cost of scanning the table. Predicates also play an important role in determining the cost associated with an index because some predicates can be used as a begin key or end key for one index but not for other indexes. This section describes the cost formulae when different indexes and predicates are available. NonStop SQL considers six different situations when computing the cost of using an index.

Case 1. The primary key file is available, and predicates of the form "column = value" specify all the key columns.

For example, assume the columns LAST_NAME and FIRST_NAME are the primary key columns of a local table PHONE_BOOK and the following query is specified:

```
SELECT LAST_NAME, FIRST_NAME,
   LOCATION
FROM PHONE_BOOK
WHERE LAST_NAME, FIRST_NAME =
   "DAVIS", "JOHN"
```

Because all key values have been specified and the primary key is unique, a simple key position and read will produce the desired result. If the root of the file is assumed to be in cache continuously, the following is true:

Cost(physical I/O) = index levels - 1

Cost(index) equals Cost(physical I/O) because all other costs are much smaller than 1 and will be truncated to 0.

Case 2. An index is available, and predicates of the form "column = value" specify all the key columns.

For example, assume the column PHONE_NUMBER is the key column of a unique index on PHONE_BOOK and the following query is specified:

```
SELECT LAST_NAME, FIRST_NAME,
   LOCATION
FROM PHONE_BOOK
WHERE PHONE_NUMBER = "725-6000"
```

Because all key values have been specified, a simple keyed read on the index followed by a keyed read on the base table will produce the desired result. Again, assuming that the root blocks of the files are always in cache:

Cost(physical I/O)
= index levels of index - 1
+ index levels of primary file - 1

Again, Cost(index) equals Cost(physical I/O) because all other costs are much smaller than 1 and would be truncated to 0.

Case 3. The primary key file is available; the predicates of the form "column = value" has not specified all the key columns.

Since only some of the key columns have been specified by predicates, possibly more than one record will satisfy the search conditions. First, the index selectivity is computed, and if no key value can be used as a positioning key, the whole index must be read. If some predicates can be used as positioning keys, the index selectivity is computed as the composite selectivity of these predicates. Once the index selectivity has been determined, the number of blocks that must be read can be computed and Cost(physical I/O) is:

index selectivity
= number of non-empty blocks in the
× primary key file

The number of records that must be examined is:

index selectivity
× number of records in the primary key file

Since more than one record may be examined, Cost(record overhead) might be significant and is computed as:

Cost(record overhead)
= number of records examined
× overhead per record

For example, assume that LAST_NAME, FIRST_NAME are the primary key columns of a local table, PHONE_BOOK. Consider the following query:

```
SELECT LAST_NAME, PHONE_NUMBER
FROM PHONE_BOOK
WHERE LAST_NAME >"DAVIS"
```

Since the prefix (LAST_NAME) of the primary key is specified in a predicate, the predicate can be used as a positioning key. However, not all the key columns (LAST_NAME, FIRST_NAME) have been specified. In this example, the predicate selectivity of:

LAST_NAME > "DAVIS"

is also the index selectivity. Further assume the following:

Predicate selectivity = 10%
Number of non-empty pages in index = 100
Number of records in the index = 10,000
Overhead per record = 0.025 I/O

Therefore, Cost(physical I/O) is 10 (0.1 × 100). The number of records that must be examined is 1000 (0.1 × 10,000) and Cost(record overhead) is 25 (0.025 × 1000). The other components of the cost are negligible in the example and are truncated to 0. Cost(index) is 35. In this example, the cost of the plan is dominated not by I/O but by the per-record cost.

Case 4. An index is available, and the predicates of the form "column = value" have not specified all the key columns.

This case is similar to the previous one, except that a physical I/O is incurred for each qualifying record in the index. For example, assume that PHONE_NUMBER is the key column for the index and the following query is specified:

```
SELECT LAST_NAME, LOCATION,
   PHONE_NUMBER
FROM PHONE_BOOK
WHERE PHONE_NUMBER > "725-6000"
```

Using the same assumptions as in the previous case, the cost of reading the index, Cost(physical I/O), is 10 (0.1 × 100). The number of index records that satisfies the predicate is 1000 (10,000 × 0.1). For each of these records, another read must be made to the table to obtain other requested data (e.g., LOCATION). If the table is much larger than 1000 blocks, each of these reads to the table will result in a physical I/O. Therefore:

Cost(table physical I/O) ≅ 1010
Cost(record overhead) ≅ 25 (0.025 × 1000)

The other costs are insignificant compared to the ones just computed. Cost(index) is approximately 1035.

Case 5. An index is available, the predicates of the form "column = value" have specified all the key columns, and all the requested columns can be found in the index.

This case has the same cost formula as Case 1 because all the requested columns can be found in the index.

Case 6. An index is available, the predicates of the form "column = value" have not specified all the key columns, and all the requested columns can be found in the index.

Because all the requested columns can be found in the index, the index behaves as if it were the primary key file in the cost computations (no extra read is required on the base table for each qualifying index record). Thus, the cost computation is identical to Case 3. For example, assume that PHONE_NUMBER is the key column of the index and LAST_NAME, FIRST_NAME are the primary key columns of the table PHONE_BOOK. The following query can be satisfied by the index alone:

```
SELECT LAST_NAME, FIRST_NAME,
   PHONE_NUMBER
FROM PHONE_BOOK
WHERE PHONE_NUMBER >"725-6000"
```

Choosing among Access Plans That Have the Same Cost

Because the estimated cost associated with an access plan is only approximate, the costs associated with several access paths may be very close to one another. NonStop SQL currently defines costs as "very close" if they are within 10% of one another. When this occurs, NonStop SQL uses the following heuristics, listed in order of preference, in selecting between two access paths with very close costs:

- A local index (as opposed to a remote index).
- An index in which predicates of the form "column = value" have specified all the key columns.
- An index with a lower selectivity.
- An index with a lower estimated cost.

The object of the heuristics is to choose a local index that has the least number of qualifying records that must be examined.

Determining Join Order

Selecting an access plan for queries involving a join of two or more tables is an extension of the process of selecting access plans for single-table queries. In addition to determining the cost for accessing a table before the join, the optimizer evaluates the different ways to join the tables.

NonStop SQL supports two methods of join evaluation. Therefore, the number of combinations of joining two tables with no alternate index is four. With three tables, this number increases to 12. In general, the number of different ways to join tables increases exponentially as the number of tables increases. To reduce the number of possibilities the optimizer has to examine, certain heuristics are used.

When two tables T1 and T2 are joined, a composite table (T1 join T2) is formed. This notation will be used in the discussion of joins. When considering the different ways of joining tables, NonStop SQL considers only two-way joins that involve either two tables or one table and a composite table. This reduces the number of combinations that must be examined and also simplifies evaluation. For example, if tables T1, T2, T3, and T4 are to be joined, the following combination will not be considered:

((T1 join T2) join (T3 join T4))

However, the following combination will be considered:

(((T1 join T2) join T3) join T4)

For each two-way join, two join methods are considered: nested join and merge join. If their joining columns do not have the same order, the optimizer considers sorting either or both the outer and inner table before performing the merge join.

The number of join combinations is further reduced by considering only combinations in which a join predicate relating the inner table and the outer composite table exists. This means that, given a composite table C, a join of C with a table A will be considered if either:

- There is a join predicate relating A with the tables in C but not with any table not in C.
- There is no other table to join with C that would satisfy the previous condition.

For example, consider the following query:

```
SELECT EMP_NAME, DEPT_NAME,
   SALARY, JOB_TITLE
FROM EMPLOYEE, DEPT, JOB
WHERE JOB_TITLE = "MANAGER"
   AND DEPT.DEPT_NUM
   = EMPLOYEE.DEPT_NUM
   AND JOB.JOB_TYPE
   = EMPLOYEE.JOB_TYPE
```

The EMPLOYEE table contains the columns EMP_NAME, DEPT_NUM, JOB_TYPE, and SALARY. The DEPT table contains the columns DEPT_NUM, DEPT_NAME, and LOCATION. The JOB table contains the columns JOB_TYPE and JOB_TITLE. The following join combinations will not be considered:

((DEPT join JOB) join EMPLOYEE)

or

((JOB join DEPT) join EMPLOYEE)

The number of combinations is also reduced by discarding more expensive combinations that give the same ordering of the resultant records. For example, assume the EMPLOYEE table has EMP_NAME as the primary key and DEPT_NUM as the key column of an index. The query:

```
SELECT EMP_NAME, DEPT_NAME,
   SALARY
FROM EMPLOYEE, DEPT
WHERE EMPLOYEE.DEPT_NUM
   = DEPT.DEPT_NUM
```

asks for employee and department information. The information is retrieved by joining the EMPLOYEE and DEPT tables. There are several ways to join the tables:

Choice A:	(EMPLOYEE with primary key file join DEPT)
Choice B:	(EMPLOYEE with index join DEPT)
Choice C:	(DEPT join EMPLOYEE with primary key file)
Choice D:	(DEPT join EMPLOYEE with index)

Depending on the access plan used for the outer table, the order in which the records are presented is different. For example, the records will be presented in EMP_NAME order for choice A and DEPT order for choices B, C, and D. If another table, JOB, is to be joined, NonStop SQL will only consider joining JOB with the composite from choice A or the composite from the least expensive of choices B, C, and D. This reduces the number of combinations to be joined with JOB from four to two. In general, NonStop SQL will discard all but the least expensive of the combinations for a given order.

Join Cost. In estimating the cost of performing a join, NonStop SQL computes the cost of accessing each of the tables involved in the join. The cost of accessing each table is computed in the same way as in single-table queries, except that predicates must be identified as associated with a particular table. This is necessary because some join predicates cannot be evaluated until a qualifying record for an outer table has been retrieved. Consider the query:

```
SELECT EMP_NAME, DEPT_NAME,
   SALARY
FROM EMPLOYEE, DEPT
WHERE EMPLOYEE.DEPT_NUM =
   DEPT.DEPT_NUM
   AND DEPT_NUM <100
```

If the DEPT table is the outer table of the join, the predicate:

```
DEPT_NUM <100
```

can be used to scan DEPT. Thus, it is involved in the computation of the cost of accessing DEPT. However, the predicate:

```
EMPLOYEE.DEPT_NUM =
   DEPT.DEPT_NUM
```

cannot be used in the cost computation for DEPT because no record has been retrieved for EMPLOYEE yet. However, it could be used in the computation of the cost of accessing EMPLOYEE.

Once the cost of accessing each table in the join has been determined, the cost of the join can be determined. For a nested join of two tables, the cost is:

Cost(A join B)
= cost(A) + n
× cost(B)

where n is the number of records that satisfy the non-join predicates on table A (i.e., n is the number of times the inner loop must be performed). For example, assume that DEPT is the outer table of the join, EMPLOYEE is the inner table of the join, and 1000 employees are in departments with department number less than 100. If the cost of accessing DEPT is 10 and the cost of accessing EMPLOYEE is 20, Cost(DEPT join EMPLOYEE) is 10 + 1000 × 20, or 20,010.

For a merge join, the cost is:

Cost(A join B)
= Cost(sortA if needed)
+ Cost(sortB if needed)
+ Cost(accessing A or sorted A)
+ Cost(accessing B or sorted B)

Subquery Processing

NonStop SQL supports the construct of nested queries or subqueries. A subquery is a query that appears in a predicate. For example, the query:

```
SELECT ITEM_NAME, RETAIL_PRICE
FROM INVENTORY
WHERE RETAIL_PRICE >SELECT
   RETAIL_PRICE FROM INVENTORY
WHERE ITEM_NAME = "PINEAPPLE"
```

asks for information on items that cost more than pineapples. The SELECT that appears on the right-hand side of the predicate is a subquery. The other SELECT is sometimes called the outer SELECT. In the previous example, the subquery will be evaluated to determine the price of pineapples. This price is then substituted in the predicate. For example, if the price of pineapples is 20, the query will be evaluated as if:

```
SELECT ITEM_NAME, RETAIL_PRICE
FROM INVENTORY
WHERE RETAIL_PRICE >20
```

has been specified. Since the subquery is only evaluated once, the cost of evaluating the original query is the sum of the cost of evaluating the individual SELECTs. Theoretically, there is no limit to the depth of nestings. The practical limit is the amount of compile-time and runtime resources (e.g., stack space and extended segment space).

In the previous example, the subquery is independent of the outer SELECT because it can be evaluated without any knowledge of the result of the outer SELECT. This independence allows the subquery to be evaluated only once.

A subquery is dependent on the outer query if the subquery references values from the outer query. For example:

```
SELECT ITEM_NAME, RETAIL_PRICE
FROM INVENTORY OUTER
WHERE RETAIL_PRICE >SELECT
   AVG(RETAIL_PRICE) FROM INVENTORY
WHERE PRODUCER = OUTER.PRODUCER
```

selects information on items that cost more than the average price of the items produced by the same producer. The subquery in this example is dependent on the outer SELECT because it references the PRODUCER column of a record retrieved for the outer SELECT. This correlation forces the evaluation of the subquery for every record retrieved from the outer SELECT. The overall query is more expensive to evaluate because of the repeated evaluation of the subquery. If the INVENTORY table contains 100 records, the cost of evaluating the query will be:

Cost(outer SELECT)
+ 100
× Cost(inner SELECT)

Choosing an Access Plan When Some Indexes Are Not Available

In a distributed system where some or all information is replicated (e.g., an index is a special case of replication), it is useful to be able to get to the required data if some system resource (e.g., an index) is not available. For example, assume the table PHONE_BOOK is in the volume $PHONE, and it has an index on PHONE_NUMBER that is stored in the volume $NUMBER. Consider the following query:

```
SELECT LAST_NAME, FIRST_NAME,
   PHONE_NUMBER FROM PHONE_BOOK
WHERE PHONE_NUMBER = "725-6000"
```

Suppose further that NonStop SQL has chosen to use the index on $NUMBER to retrieve the requested data. If the volume $NUMBER is not available at run time, many DBMS would return an error to the application. NonStop SQL will try to find an alternate path to the data.

At static compile time, the SQL compiler requires that all information on the table be available so that the most efficient access plan can be selected. This is a reasonable requirement since statically compiled SQL objects will be executed repeatedly. If any information is not available at static compile time, the compiler sets a flag indicating that the query must be recompiled at run time. A valid SQL object is still produced because all information may be available for other queries in the same program.

At run time, the SQL executor recompiles the query when it encounters the flag indicating that such a recompile is necessary. The SQL executor instructs the SQL compiler to ignore unavailable information during this compile. If all information is now available, the most efficient access plan can be selected. If some information is still not available, the SQL compiler tries to select the most efficient access plan based on the information that is available.

If, after this recompilation, the chosen index is still not available (e.g., the communication line to a node is down, or the volume containing the chosen index is down after the recompile), the SQL executor tries once more. This time it instructs the SQL compiler to choose the primary key file of the table as the access path. This allows the application to access the data even when a system resource is unavailable.

User Influence on Access-Plan Selection

NonStop SQL automatically selects an efficient access plan. However, NonStop SQL can achieve even greater efficiency if the user provides more information about the table being queried or offers a greater choice of access plans. For example, the database administrator can create indexes on fields that are frequently mentioned in queries. This section describes operations a user can perform to influence the choice of access plans by NonStop SQL.

Creating Additional Indexes

If an application contains many queries that reference a column in a table, an index on the field would improve the performance of some of the queries. For example, consider the query:

```
SELECT ITEM_NAME, RETAIL_PRICE
FROM INVENTORY
WHERE RETAIL_PRICE = 100
```

If there is no index on RETAIL_PRICE, NonStop SQL has to scan the table and evaluate the predicate:

```
RETAIL_PRICE = 100
```

against each of the records in the table. An index on RETAIL_PRICE would improve the performance of the query dramatically. On the other hand, the same index might not help the following query:

```
SELECT ITEM_NAME, RETAIL_PRICE
FROM INVENTORY
WHERE RETAIL_PRICE >100
```

The reason is that ITEM_NAME is not part of the index. Thus, for every index record that satisfies the predicate, a physical I/O must be incurred before the column ITEM_NAME can be retrieved from the table. However, if the query only selects columns that are included in the index, the index on RETAIL_PRICE will help. For example, if ITEM_NO is the primary key column:

```
SELECT ITEM_NO, RETAIL_PRICE
FROM INVENTORY
WHERE RETAIL_PRICE >100
```

Index will also help the performance of a query that requires the result to be presented in a certain order or grouped according to certain columns.

Thus, additional indexes will help the performance of some but not all queries. Users should use the EXPLAIN facility to determine if the extra index will be used by NonStop SQL. (See the *NonStop SQL Programming Reference Manual* for further details on this facility.) Furthermore, while adding indexes may help the performance of some select queries, queries that update the index columns will incur the overhead of having one more index to update.

Update Statistics

NonStop SQL provides an UPDATE STATISTICS utility to collect and save statistics on columns and tables. The SQL compiler uses these statistics to determine the selectivities of predicates, indexes, and tables. Since selectivities directly influence the cost of access plans, it is important that statistics on a table are close to the real values; this increases the likelihood that NonStop SQL will choose an efficient access plan.

A NonStop SQL installation should follow several simple rules in using the UPDATE STATISTICS facility:

- Do not run UPDATE STATISTICS when a table is empty. Run UPDATE STATISTICS only after a table has been loaded with data.
- If the performance of many queries deteriorates, do not run UPDATE STATISTICS before consulting EXPLAIN. If the performance degradation is due to the fragmentation of blocks in a table, running UPDATE STATISTICS and recompiling the queries might not help. It might be better to first reorganize (reload) the table.
- It is usually a good idea to determine the effect of UPDATE STATISTICS on production queries. This can be accomplished by bracketing UPDATE STATISTICS and EXPLAIN on the queries in a transaction.

Specifying Table Lock with the Control Table Directive

NonStop SQL may choose a different access path if it knows that a table lock could be used. In NonStop SQL, a user can specify the use of table locks with either of two commands:

- LOCK TABLE *name*.
- CONTROL TABLE *name* TABLELOCK ON.

LOCK TABLE is an SQL statement. CONTROL TABLE is an SQL compile-time directive. Currently, each SQL statement is compiled independently. Therefore, the SQL compiler has no idea that a LOCK TABLE statement has been encountered prior to the current statement; it might even be in another COBOL program. Because the CONTROL TABLE command is a compile-time directive, the SQL compiler is aware that the user wants to use a table lock on queries that reference the specified table.

Because it eliminates the overhead of lock maintenance, requesting a table lock on a table improves the performance of queries that touch many records, and it is more likely that real SBB will be selected. However, concurrency will be reduced. Thus, an application might want to request table locks with the CONTROL TABLE directive for batch-type queries.

Using Joins Instead of Subqueries

In SQL, certain queries can be formulated in different ways and yet result in the same set of records. For example, if DEPT_NO is unique, the query to retrieve the names of employees in DEVELOPMENT can be expressed as either:

```
SELECT EMPLOYEE.NAME
FROM EMPLOYEE
WHERE EMPLOYEE.DEPT IN
   (SELECT DEPT.DEPT_NO FROM DEPT
   WHERE DEPT.NAME =
   "DEVELOPMENT")
ORDER BY EMPLOYEE.NAME
```

or:

```
SELECT EMPLOYEE.NAME
FROM EMPLOYEE, DEPT
WHERE EMPLOYEE.DEPT_NO
   = DEPT.DEPT_NO
   AND DEPT.NAME = "DEVELOPMENT"
ORDER BY EMPLOYEE.NAME
```

The first formulation of the query uses a subquery; the second uses a join. Although both formulations produce the same result, their performances are likely to be very different. The second formulation will always match or outperform the first one because, in the first formulation, the user has dictated how the query is to be performed (i.e., perform the subquery first and then perform the main query). In the second formulation, NonStop SQL has the flexibility to determine the order of the join and is therefore able to choose the most efficient way to execute the query.

Using Multivalue Predicates

The current ANSI SQL language does not allow a user to specify a concatenation of columns in the specification of a predicate (ANSI, 1986). To solve the problems arising from this limitation, NonStop SQL provides a multivalue predicate construct. The examples in this section illustrate these problems and show how NonStop SQL addresses them.

Assume that the table EMPLOYEE has an index that consists of the columns LAST_NAME, FIRST_NAME. The problem is to list all employee records with names that come after "DAVIS JOHN." A common but incorrect formulation of the query is:

```
SELECT *
FROM EMPLOYEE
WHERE LAST_NAME >"DAVIS"
   AND FIRST_NAME >"JOHN"
```

The problem with this query is that employees such as "DAVIS STEVEN" will not be listed, since LAST_NAME must be after "DAVIS." The correct formulation using SQL syntax is:

```
SELECT *
FROM EMPLOYEE
WHERE (LAST_NAME = "DAVIS" AND
   FIRST_NAME >"JOHN")
   OR LAST_NAME >"DAVIS"
```

However, such a formulation would cause NonStop SQL to read the entire table instead of using the index on LAST_NAME to evaluate the query. One way to increase the chance that the SQL optimizer will use the index is to formulate the query as follows:

```
SELECT *
FROM EMPLOYEE
WHERE LAST_NAME > = "DAVIS"
   AND ((LAST_NAME = "DAVIS"
   AND FIRST_NAME >"JOHN")
   OR LAST_NAME >"DAVIS")
```

This looks more and more like telling NonStop SQL how to evaluate the query instead of describing the problem. This is, of course, contrary to the spirit of relational DBMS. Furthermore, if the index contains more than two key columns, the formulation becomes more complex.

Using context-free servers is another common application in which this problem appears. For example, suppose a server processes only a batch of employees per request from the requester. The server would like to position to a record following one that has a key value supplied by the requester. If the keys supplied are the prefix of a multikey index, the previously described problem will appear. Also, unnecessary records are scanned many times. For example:

```
SELECT *
FROM EMPLOYEE
WHERE LAST_NAME > = :last_name
   AND ((LAST_NAME = :last_name
   AND FIRST_NAME >:first_name)
   OR LAST_NAME >:last_name)
```

would rescan all employees with FIRST_NAME before *:first_name*.

NonStop SQL solves this problem with the multivalue predicate construct. This feature allows the user to specify composite keys as a group. For example, it is easier and more logical to write the previous query as:

```
SELECT *
FROM EMPLOYEE
WHERE LAST_NAME, FIRST_NAME
   >"DAVIS", "JOHN"
```

The NonStop SQL optimizer treats this multivalue predicate like any other predicate. The columns in the multivalue predicate are used as keys if a multicolumn index exists on, say, LAST_NAME, FIRST_NAME.

Conclusion

The NonStop SQL optimizer selects the most efficient access plan for a given query by examining the different ways to access tables. The access plan consists of the index to be used, the type of SBB to be used, the order in which tables are to be accessed (in a query that references multiple tables), when a subquery should be evaluated, and what to do when an index is not available.

The user can improve the performance of evaluating a query by providing the optimizer with additional information prior to SQL compile time. By creating additional indexes on frequently used fields, keeping up-to-date statistics on columns and tables, specifying table locks, using joins instead of subqueries, and using multivalue predicates, the user can help the optimizer select the best methods to access a given body of data, and thus maximize the high performance of NonStop SQL.

References

ANSI. 1986. Database Language SQL. American National Standards X3.135-1986.

NonStop SQL Programming Reference Manual. Part no. 82318. Tandem Computers Incorporated.

Tsukerman, A., et al. 1986. *FASTSORT: An External Sort Using Parallel Processing.* Tandem Technical Report 86.3. Tandem Computers Incorporated. Reprinted in the *Tandem Systems Review.* Vol. 2, No. 3. Tandem Computers Incorporated. Part no. 83938.

Acknowledgments
I would like to thank the members of the Tandem Database Group for contributing ideas on optimization and reviewing this paper.

Mike Pong wrote this article, as well as the preceding article, "NonStop SQL Optimizer: Basic Concepts."

NonStop SQL Reliability

Tandem's NonStop SQL is the database management system (DBMS) foundation for Tandem customers' on-line transaction processing (OLTP) applications. NonStop SQL is also a complex software subsystem that achieves high performance and user productivity by pushing complicated decision making about optimal data access into the SQL system, away from user concern. While the payoff is great, this level of sophistication raises important reliability issues.

This article describes the methods Tandem is using to ensure that NonStop SQL is reliable and remains so throughout its life. These methods have been very effective in identifying and correcting defects long before product release. The result is an SQL system that has been well received by Tandem customers and is recognized in the industry as the first relational DBMS to meet OLTP demands for performance and reliability.

Overall Approach

Tandem's NonStop SQL Quality Assurance (QA) team, within the Database Software Development Group, performed the work described in this article. The QA team achieves its objectives by participating in all phases of the software life cycle, beginning with product requirements and continuing after product release. Ultimately, however, a product must prove its acceptability by successfully running through a series of stringent tests that demonstrate that the functional, performance, and reliability goals of the product have been achieved.

User Expectations for Reliability

Users expect Tandem to deliver reliable software, though most users do not commit critical applications to new software until they gain experience and become comfortable with it. As the product becomes integrated into their production environment, most users expect the software's reliability to improve over time. In fact, they gradually come to trust features that have worked steadily and tend to assume that once a function has become stable, it will remain so forever.

Unfortunately, it is a property of the software development process that modifications in one part of the system may lead to unexpected side effects in some other, seemingly unrelated, part. Subtle interactions of new code with existing stable code can sometimes destabilize trusted features. To ensure that a user's expectations for reliability are not betrayed, Tandem develops and maintains extensive regression test libraries.

Figure 1

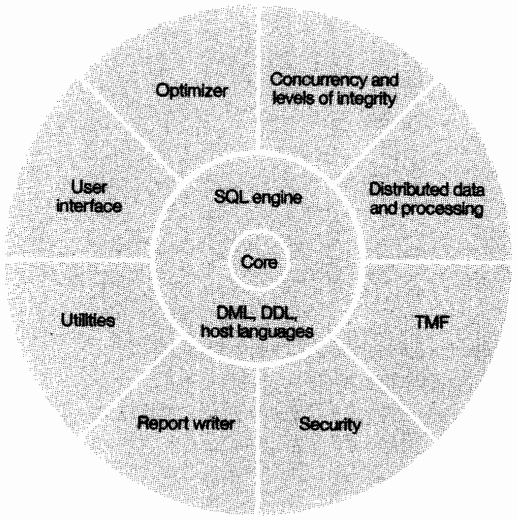


Figure 1.
Layering of test units. Core tests (e.g., table creation, database population, rudimentary data retrieval, and basic embedded SQL processing) demonstrate correct functioning of the simple DDL and DML statements needed to create an environment for the SQL engine tests. SQL engine tests focus on SQL as a language, independent of the Tandem operating environment. The outermost layer ties the SQL language to the full Tandem NonStop SQL environment.

A *regression test library* is a systematically developed collection of permanent tests and tools designed to assess the quality of a specific product. It is run with every release of the software and is, therefore, a very powerful tool for locating parts of the system in which reliability has regressed.

Determining Functional Reliability

Determining whether or not NonStop SQL behaves according to its documented semantics is the first step toward ensuring its overall reliability.

Layering of the Test Units

The NonStop SQL regression test library is a collection of test-unit modules arranged in a hierarchical manner. The layering follows a natural ordering in which the functionality tested by one layer solidifies the foundation for the tests above it. QA considers the layering approach to be superior to separating all functions into independent, specialized test units that focus solely on a single function. Layering permits a gradual movement toward more sophisticated combinations of features, and it also maintains good fault-isolation characteristics.

Figure 1 illustrates the layering of the test units. QA applies tests from the core outward to find fundamental problems quickly and get them repaired before moving on to more complex processing in outer layers.

Special-Purpose Databases

One of the challenges for the QA developers was to construct a set of databases capable of providing the full spectrum of attributes present in real customer databases, while being storage-efficient and transportable, and having properties that could be understood easily by QA and product development teams. The simplistic solution of copying an existing customer database was ruled out early in the project. Previous experience with real customer databases and their applications has shown that for testing and analysis purposes, they tend to be bulky and lopsided. They invariably exercise certain limited paths in the product over and over again, leaving other parts entirely untested.

It also became clear early in the project that building one general-purpose database, which would satisfy all the testing requirements, would not be feasible. Therefore, five special-purpose databases were developed, each designed to meet the needs of one area of testing.

The *order-entry database* is used primarily by the core test units and the SQL engine test units. The tables represent customers, employees, inventory, orders, and other data items typically used in a company. It is easy to understand the meaning of the tables and their relationships by reading the databases or the test queries. This database is ideal for the initial shakedown of the product, since defects produce results that are easily understood and indisputable. The other databases tend to be more esoteric and non-intuitive, requiring greater time and effort to analyze problems.

The *select database* is structurally more complex than the order-entry database; all data types are represented, indexes are variously ascending, descending, contiguous, noncontiguous, and so on. This database was designed primarily with the semantics of the SQL SELECT statement in mind, though it is used for the other Data Manipulation Language (DML) statements as well.

With the SELECT statement, a user can express very complex retrieval requests involving many tables, perhaps using nested subqueries, and specifying operations such as grouping, ordering, and aggregation. The select database contains very carefully contrived row values so that complicated SELECT statements can be written that will demonstrate correctness by successfully retrieving data. While the select database is very small compared to a customer database, the relationships among the tables and their data are more complex than those in a typical application database. The result is that QA has tested SELECT statements whose complexity ranges from trivial to much more complex than most users will probably need to write.

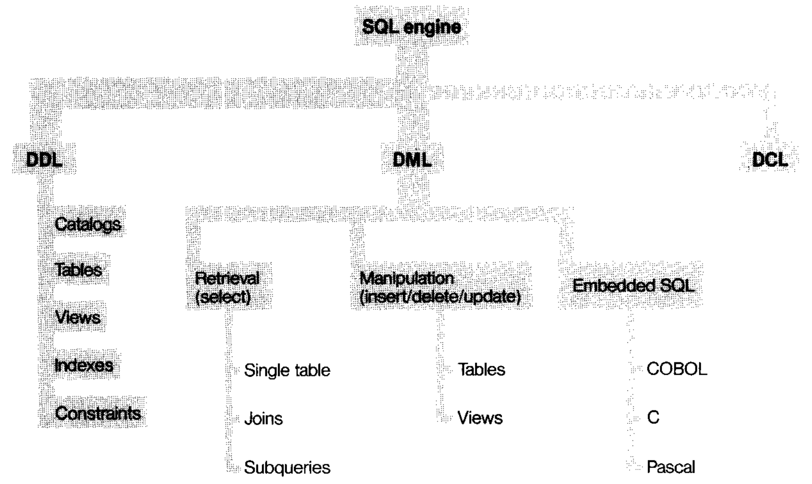
The *optimizer database* departs significantly in structure and philosophy from the order entry and select databases. Occupying disk space approaching 80 Mbytes and containing tables made up of 10 to 100 columns, it dwarfs the other databases. The data in these tables is not intended to be understood on a row-by-row basis like the others, and is in fact mechanically generated. Instead, it is the gross properties of the optimizer database that make it useful. Typical properties considered in its design are ranges and distributions of data values, column selectivities, uniqueness, and block sizes. These properties are exploited by carefully written tests that elicit certain behavior on the part of the optimizer. The optimizer database also supports UPDATE STATISTICS, EXPLAIN, and normalizer testing.

The *report writer database* is designed for the SQL report writer. It contains text and numeric data values in tables big enough to generate reports typical of real applications. Row and column sizes, data values, and column data types were chosen to demonstrate report writer functions, including totaling, titles, breaks, and folding. Because the tables in this database have no relationship to each other, operations such as joins are not expected, and there are no indexes or partitions; this database is only suited for report writer testing.

Figure 2.

Hierarchy of SQL engine tests. SQL engine tests include DDL and DML tests. DDL tests create and alter a wide range of objects, from system catalogs to objects with complex view/index/ partition dependencies. DML tests are divided into data retrieval (reading), data manipulation (writing), and embedded SQL processing for all host languages.

Figure 2



The *convert database* contains both ENSCRIBE files and SQL tables. Its purpose is to facilitate the conversion from ENSCRIBE objects, described using ENSCRIBE Data Definition Language (DDL), to equivalent objects defined using NonStop SQL DDL, and vice versa. Since not all elements of ENSCRIBE DDL have equivalent representations in NonStop SQL DDL (i.e., occurs, subfields), one of the key goals was to design a database that could expose cases where conversion algorithms produced either intuitively bad conversions or possibly even mangled SQL tables. Because one-to-one conversion is often impossible, particular care was taken to make sure the convert database provided well-balanced support for error-handling tests as well as positive-function tests.

Validating the SQL Engine (DML/DDL)

The first hurdle NonStop SQL faces in testing is the validation of the NonStop SQL engine—the nucleus of the product that implements NonStop SQL DML and DDL statements. These test units view SQL in its purest form, as a language independent of a particular system and operating environment. They are arranged in a hierarchy (Figure 2) beginning with the creation of the simplest, most fundamental objects in an SQL system using DDL and extending through complex manipulation of SQL objects using DML statements. These tests make extensive use of the order-entry and select databases.

Distributed Testing

Distributed testing extends the SQL engine-software test units into the Tandem network environment. All DDL and DML functions that work in a single system must be demonstrated to be logically equivalent with objects on remote systems. These SQL tests access and manipulate combinations of remote catalogs, network-partitioned and indexed tables, and network views.

Negative testing is especially important for distributed SQL applications because network-line failures may prevent access to a portion of the database. The distributed tests carefully set up scenarios that cause network-failure events and then analyze error handling and fault recovery.

One particularly interesting type of fault recovery is known as "local autonomy." Some network and disk failures may disconnect part of the database but leave access to enough of it to allow DML operations to continue. In fact, when a strategy defined in an already compiled query specifies a part of the database removed by a disruption, NonStop SQL may be able to use alternative access strategies to perform the query.

Local-autonomy test units construct scenarios in which partitions or indexes are made unavailable; this is done by disabling volumes or network lines or by removing file labels while leaving catalog entries for the index or partition intact. Run-time error recovery is significant, since it involves one or possibly two auto-recompilations. The tests verify that the result is a transparent recovery that returns the desired data despite the disruption.

Levels of Integrity and Concurrency

NonStop SQL provides levels of integrity to ensure database consistency with concurrent processing. Progressively higher levels of integrity provide higher degrees of isolation from other users but also reduce concurrency by locking larger ranges of data.

When a user requests a certain level of integrity, NonStop SQL must lock, at a minimum, the specific set of rows in the database that satisfies the request. Under certain conditions, NonStop SQL may actually lock more than the minimum required set, usually for esoteric implementation reasons or because a larger range may be judged to be more efficient to process. The price, of course, is potentially reduced concurrency because other users have to wait on superfluous locks.

Naturally, improving locking schemes is an ongoing design objective for NonStop SQL development. However, modifications to the programs that share this function (optimizer and disk process) can be deceptively subtle, because changes alter concurrency but do not alter the meaning of SQL statements. Furthermore, inadvertently introduced concurrency degradation may not be obvious, especially under moderate loads. There is a danger that performance could drop unexpectedly under heavy production loads.

Because this code is so complex, it is possible for well-intentioned tuning in one place to have severe effects in another place. Consequently, level-of-integrity tests have two equally important goals:

- Make sure levels of integrity work.
- Detect concurrency setbacks if they do occur.

QA's approach meticulously diagnoses the behavior of NonStop SQL using lock-step parallel processing combined with knowledge of how the locking algorithms should be working. Locks are taken by a "locker" process and systematically tested by a synchronized "challenger" process. This method has been very successful at rapidly pinpointing minute changes in locking behavior.

Figure 3

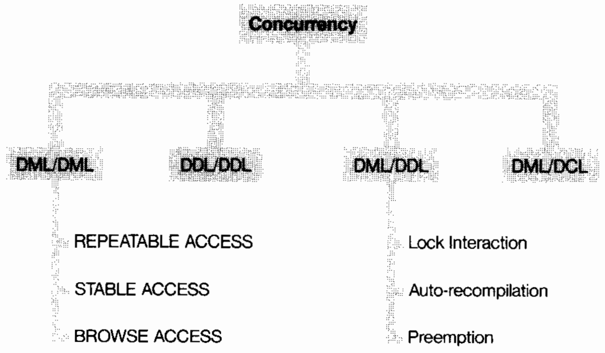


Figure 3.

Hierarchy of concurrency tests. Concurrency tests are divided so that competing elements of the SQL language demonstrate their concurrency interactions. DML/DML tests concentrate on levels of integrity, DML/DDL tests focus on the effects that DDL statements have on DML operations, and DDL/DDL and DML/DCL tests confirm that the correct locking interactions are taking place.

Concurrency Test Units

The most prevalent concurrent interactions in an OLTP environment occur between programs that each contain DML statements—for example, servers accessing the same set of tables. Extensive tests have been developed to cover interactions of all combinations of DML statement types with all combinations of levels of integrity (Figure 3). These tests use the lockstep synchronization approach, pitting DML statements against each other and using various mixes of lock granularity, lock duration, exclusion mode, and ownership.

Though most concurrent interactions are DML/DML interactions, several other very important forms must be considered. Some arise from the fact that NonStop SQL object management is built on the principle of an active dictionary. NonStop SQL guarantees that the user has a correct, well-formed definition of an object at all times. This means that DDL operations can change the characteristics of objects while they are being used. This carries with it a whole new set of concurrency issues, namely, how DDL operations interact with other simultaneously occurring DML and DDL operations.

DML/DDL interactions have locking issues similar to their DML/DML counterparts but also have several intriguing side effects. Naturally, DDL statements lock the object being operated on, so DML statements (or other DDL statements) must deal with handling locks in the usual way. For example, lock granularity and protocols for lock acquisition and queuing are tested just as in the DML/DML cases. However, this alone does not constitute sufficient DML/DDL testing because a DDL statement could cause auto-recompilation or preemption.

A host-language program containing NonStop SQL DML statements will have its SQL statements transparently recompiled if an SQL object that the program uses has been altered by a concurrent DDL operation. The automatic recompilation may apply to every SQL statement in the program or to only one or two statements, depending on precisely when the DDL operation occurs. Similarly, in certain cases, an active DML statement may be preempted by certain DDL operations rather than forcing the DDL statement to wait. Tests have been carefully constructed to provoke all of these situations and to ensure correct locking and error-handling behavior.

Concurrency is a typical example of where SQL is uniformly tested, from the most common to the most extreme cases. Though many of these cases appear to fall outside the mainstream of concurrency functionality, QA is committed to protecting the user from major system failure or data corruption, even if it stems from a rare concurrency event.

Determining Performance Reliability

In most database management systems prior to SQL, the application programmer made all decisions about how data was retrieved and manipulated in the database. This included fundamental access strategies such as uses of indexes, file-positioning modes, lock granularity, types of buffering, choices of file ordering for joins—in short, nearly all operations an application needed to perform the desired function. The application programmer had total control of the methods used to access the database, which meant he or she had direct influence over the performance of retrievals and manipulations.

NonStop SQL dramatically changes this situation by elevating programmer control above direct file-system interface calls. Data retrieval and manipulation occur exclusively in SQL language statements embedded within the host-application code. The resulting productivity gain is tremendous, since application programmers simply specify what they want to have happen, rather than going through elaborate and error-prone programming steps to say how the result is to be achieved. The methods for retrieving data are now relegated to a sophisticated component within NonStop SQL called the optimizer.

While the productivity benefits are clearly welcomed by SQL users, the loss of direct control may make some users uneasy. These users must now trust the optimizer to choose the optimal access strategy for their queries. They are concerned that the optimizer may make fundamentally bad decisions that will lead to unnecessary performance degradations in their application. SQL QA has developed a strategy for minimizing the risk of such events.

Optimizer Testing Strategy

Benchmarking probably comes to mind immediately as an attractive method for optimizer testing. Rather than using benchmarking, the SQL QA team directly examines the access plans chosen by the optimizer. The main advantages of this approach are a significantly greater diagnostic capability over benchmarks and independence from other parts of SQL.

This has important strategic implications because SQL QA must find optimizer problems rapidly during the early part of the release when other performance-critical parts of the system, such as the disk process, may not yet be reliable. This strategy insulates SQL QA from those dependencies, so reliability assessment can begin right away.

Overview of Optimizer Tests

The key to successful optimizer testing is the design of the database. Data-value distributions, placement of indexes, and precisely generated selectivities must be carefully implemented to allow test queries to demonstrate convincingly that the best access plan was chosen over many possible candidate plans. The test database must also be designed so that its properties accurately represent those of production databases.

The optimizer database developed by SQL QA has been carefully constructed to embody the attributes of full-sized, distributed production databases, while possessing well-understood properties required for deterministic testing. (For a definition of selectivities and other concepts related to the function of the optimizer, refer to the accompanying article, "NonStop SQL Optimizer: Basic Concepts.")

The optimizer tests are divided into two broad classes. The first deals with access-path selection; the second with join-strategy algorithms.

For access-path selection testing, queries are designed so that predicates have predetermined selectivities using a carefully chosen set of indexes. In these tests, one of the many indexes will be superior to the others for the given predicates. Many variables, such as physical-I/O cost, record overhead, cost of sorts, subquery-processing costs, and transfer costs, are considered in the design of the tests to make sure the cost-formula assumptions are correct. Knowing the formulas, the test designer attempts to construct scenarios in which the optimizer could erroneously make less than optimal trade-offs.

Test automation plays a key role in achieving NonStop SQL reliability.

The join-strategy algorithm tests use a similar philosophy but also consider which table order and join method are optimal. These tests use specially designed multiple-table join predicates, single-table predicates with known index selectivities, and ordering clauses to challenge the optimizer. The cost formulas for joins provide the variables that the test designer manipulates in the test. These include the relative costs of performing a merge join or nested-loop join, sorting requirements needed to support merge join and satisfy ordering clauses, and the cost of joining the tables in a particular order.

Distributed optimizer tests add another dimension to access-plan selection and join-algorithm selection by considering the effects of network access. The optimizer must take into account factors such as the cost of choosing a remote index versus a local index, the possibility of increased attractiveness of real or virtual sequential block buffering, and the cost of joining remote tables.

In both access-plan selection testing and join-algorithm selection testing, correctness is determined by using a special QA tool interfaced to the SQL compiler. The tool displays the final plan chosen by the optimizer as well as the alternate plans considered and discarded. Armed with this information, SQL QA and Development locate and eliminate bad choices, thereby preventing poor optimizer decisions from reaching production applications.

Optimizer Reliability Assessment—Future Plans

NonStop SQL QA is continuing to enlarge its collection of tools and techniques for assessing the reliability of the optimizer. Several new methods are planned that will further improve the testing process.

For example, the SQL development team is building a QA tool interface for inhibiting normal optimizer access-plan selection that determines the least costly access plan. This technique will allow QA to check the relative cost of many rejected alternatives by comparing run times and actual I/O costs with estimates. The QA tool will automatically spot serious discrepancies between the approximations made by the optimizer and the real cost of an operation, especially when a sub-optimal plan was chosen as the best one.

The NonStop SQL QA team will also be developing a benchmark to provide end-to-end checking of the optimizer, executor, file system, and disk process. The benchmark will be run early in the release cycle, typically several months before full-scale benchmarking. Its purpose is to quickly identify release-to-release regressions in performance. The benchmark will have little diagnostic capability, but it will provide early warnings of performance problems. It will also help focus the other optimizer tests that do have excellent fault-isolation characteristics.

The Effect of Test Automation on SQL Reliability

Running all of the NonStop SQL tests through one complete pass requires five days, 24 hours per day on a dedicated four-processor TXP™ system. In a typical major release, four to six passes of testing, analysis, and repair are required before all significant problems have been shaken out. Obviously, thorough regression testing of SQL does not come cheaply. Test run time and consumption of human and machine resources are significant.

Test automation plays a key role in achieving Tandem's SQL reliability and cost-effectiveness objectives. Labor-intensive test units, no matter how good they are, fall victim to human error and fatigue when run over and over again. Moreover, a regression test library that cannot guarantee absolute repeatability will let problems slip through and erode everyone's confidence in the testing process. This is potentially acute for SQL because, in the worst case, testing could take months. Without automation, it would be impossible to have confidence in the completeness and correctness of the crucial final cycles of testing.

QA realized this danger at the beginning of the SQL project. Consequently, a goal going into the project was to achieve 100% automation by investing in automation tools at the start. These tools confine labor intensity to tool and test development and analysis of SQL failures, and exclude it entirely from test running.

ALIEN (Automated Library Environment), the NonStop SQL frame manager, and COVER are three of the tools that have given SQL QA concrete yet economical ways of increasing confidence in the reliability of SQL. With a product as large and complex as SQL, these tools provide an invaluable means for systematically managing, implementing, and measuring the testing process.

Automated Library Environment

The first tool developed for SQL was ALIEN, a general-purpose productivity tool for regression-test library management. ALIEN provides a standardized, modular structure for supporting fully automated test units. By managing test selection, setup, execution, logging, error recovery, results reporting, and cleanup, ALIEN removes the QA developer from the testing loop. The tester selects any combination of tests through an easy-to-use keyword interface, leaves the tests running unattended, and returns later to examine the test-results report. The tests are self-evaluating, so human intervention is required only to analyze the causes of failures.

SQL Script Processing

Another key productivity tool developed for the SQL project is called the SQL frame manager. It is essentially a customized script processor that runs under ALIEN and divides a test unit into subtests known as "frames."

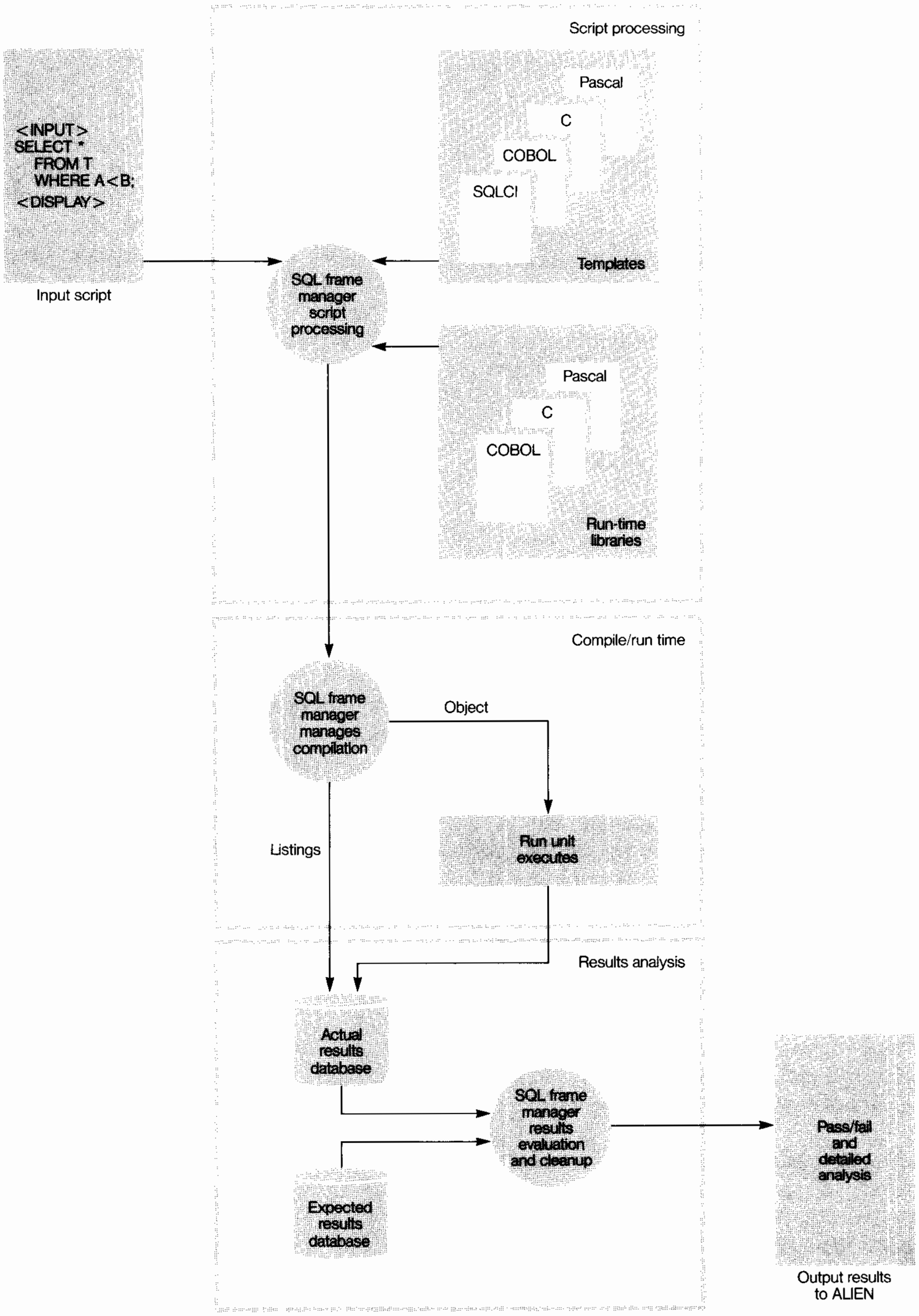
The payoff from the SQL frame manager is best illustrated by the following observation. Suppose an SQL user is given an SQL query to try out. The user would almost certainly run it on the SQL conversational interface (SQLCI) rather than embedding the statement in a host language, since embedded SQL requires an order-of-magnitude greater effort. The SQL frame manager eliminates this difference by allowing a QA developer to write an embedded SQL test as easily as an SQLCI test.

Figure 4.

SQL frame manager. The SQL frame manager accepts an input script containing SQL statements and SQL frame manager commands, merges program shells (templates) and predefined run-time libraries with the SQL statements to form programs, handles all necessary compilation steps including error recovery, and finally executes the program.

Test results can include compiler listings, utility output, and SQL statement results. The SQL frame manager automatically compares test results with a database of known correct results to determine whether the tests pass or fail.

Figure 4



The SQL frame manager does this by making all scripts look like SQLCI input, except that legal input is extended to allow embedded SQL statements such as cursor operations. If the user tells the SQL frame manager to test using SQLCI, the script text is passed directly to SQLCI for processing. If the user selects a particular language (COBOL, C, or Pascal), the SQL frame manager processes the statements into the correct form for the target language, embeds them into a generic "template" program, provides error and I/O routines, and then manages the preprocessor, language compilation, SQL compilation, and execution steps as needed (Figure 4). The process that makes embedded SQL more difficult to use than SQLCI has been made transparent by the SQL frame manager.

The transparency concept is extended even further into the complex domain of concurrency testing, again improving productivity dramatically. A single SQL frame manager script can be written that describes the actions of two completely separate host-language processes. The script for each process is written in the usual way but is divided down the middle of the page using the vertical bar character. The script text to the left of the line belongs to one process; the text on the right belongs to the other. The SQL frame manager decomposes the script into two actual host-language programs and then runs them. The SQL frame manager also supplies semaphore primitives to allow the test developer to fully synchronize the two processes.

Figure 5 shows a concurrency-script segment involving process A (left-hand text) and process B (right-hand text). SQL frame manager statements appear inside angle brackets. The SQL frame manager translates most commands into host language code. For example, <BEGIN WORK> generates a call to BEGINTRANSACTION. Synchronization primitives <WAIT> and <SIGNAL-*n*> allow the processes to operate in lockstep. When process A executes a <WAIT> statement, it will wait at that point in the program until process B executes a <SIGNAL-A> statement.

Figure 5

```
<COMMENT>  Show that if process A obtains locks using an update
<COMMENT>  statement with repeatable access, and process B
<COMMENT>  attempts to select an updated row, then process B
<COMMENT>  will timeout waiting for the locks to be released.


<COMMENT>        PROCESS A                              PROCESS B

<WAIT>                                   | <SIGNAL-A>
<BEGIN-WORK>                             | <WAIT>
                                         |
UPDATE table01                           |
    set col_1 = 100                      |
    where col_2 between 40 and 60        |
    repeatable access                    |
                                         |
<SIGNAL-B>                               |
<WAIT>                                   | <BEGIN-WORK>
                                         |
                                         | SELECT col_1
                                         |     into :user_host_var
                                         |     from table01
                                         |     where col_2 = 50
                                         |     stable access
                                         |
                                         | <SIGNAL-A>
                                         |
<COMMIT-WORK>                            | <COMMIT-WORK>
```

Figure 5.
SQL frame manager concurrency script segment. The script is decomposed into two embedded SQL host-language programs. Text on the left belongs to process A; on the right, to process B. This format makes the test objective self-evident, which would not be true using two decoupled host-language programs.

The SQL frame manager simplifies embedded SQL programming by hiding an enormous amount of irrelevant detail from the script developer. Examples include transparent error handling and reporting, the striking absence of all host-language code except for host-variable references, and simultaneous management of two separate compilation and execution streams.

With the SQL frame manager, a complex concurrency test can be written that very clearly expresses the desired interaction between the two programs. Once again, the SQL frame manager has transformed the trouble-prone test scenario of setting up and controlling two embedded SQL processes into a task as simple as writing an SQLCI OBEY file.

Besides simplifying embedded SQL processing, the SQL frame manager supports a robust testing environment for the scripts. The SQL frame manager provides easy ways to modularize large tests into test cases, initialize the test environment, capture the results of test runs, and automatically compare expected versus actual results. It also couples directly into the ALIEN system, so that the results of SQL frame manager script runs are handled without any interface programming between the two tools.

All of the reliability benefits the SQL frame manager provides stem from the basic fact that it significantly simplifies the task of developing SQL tests. For example, it eliminates the incentive to rely too heavily on SQLCI, thereby removing a bias that could cause embedded SQL to be under-tested. Also, hiding the mundane details of the embedded SQL environment allows more tests to be written in the same amount of time. Similarly, isolating the code under test from test-support activities such as test setup and evaluation leads to cleaner, more focused tests. The SQL frame manager has made SQL tests cheaper to develop and test coverage more robust, especially in the embedded SQL and concurrency areas.

The COVER Program Path Analyzer

COVER is a path-analyzer tool that measures which statements in a program have been executed as a result of running a test. COVER indicates what percentage of a program has been traversed and exactly which statements and procedures have not been executed. While many COVER users are interested in overall coverage percentages, SQL QA concentrated on using it to discover untested parts of SQL.

From the start of the project, the strategy was to apply COVER halfway through the SQL test-development cycle to determine whether areas of SQL were unaccounted for in the formal test plan and test-specifications documents. The objective was to avoid a surprise test deficiency near the time when the first customer shipment occurred. Fortunately, no oversights in the plan were discovered. However, if there had been an oversight, there would have been sufficient time to react and correct the exposure.

Conclusion

This article has described some of the methods used to ensure the reliability of NonStop SQL. Only a small fraction of Tandem's total quality assurance effort is mentioned here. Other Tandem QA organizations, including Low-Level Database QA, File System and Disk Management QA, and Systems Integration QA, use their own techniques to uncover SQL-related defects. Alpha and beta site testing, careful development-cycle phase reviews, unit testing, and performance assurance testing also make significant contributions to total product quality. Tandem's ongoing commitment of significant resources and technology to NonStop SQL is a commitment to maintaining high quality and customer satisfaction.

Reference

Tandem Performance Group. 1988. Tandem's NonStop SQL Benchmark. *Tandem Systems Review.* Vol. 4, No. 1. Tandem Computers Incorporated. Part no. 11078.

Acknowledgments

I would like to thank the members of the NonStop SQL Quality Assurance team, Steve Flournoy, Phil Koza, Bruce Maigatter, and Joan Zimmerman, for their significant contributions to the content of this article.

Claude Fenner joined Tandem in 1980 and is currently the manager of the NonStop SQL High Level Quality Assurance team. He has participated in a number of projects, including TMF, PDT, ENCOMPASS, and tool development. Prior to joining Tandem, he worked for several software houses.

NonStop SQL Data Dictionary

The NonStop SQL data dictionary embodies the qualities of NonStop SQL as a whole: availability, reliability, flexibility, and ease of use. It is active and closely integrated with the Tandem™ operating system. Because the dictionary is distributed, users can always access available data even when other nodes in the network are not available. It also supports the concept of location transparency in that the user observes the same function whether the dictionary comprises one catalog or many.

This article assumes knowledge of distributed data and distributed database management systems (DBMS). However, a brief overview of the issues surrounding a data dictionary is provided prior to discussing the NonStop SQL data dictionary, the objects it describes, and the SQL catalog manager that maintains it. Some standard dictionary terms are defined at the end of the article.

What Is a Data Dictionary?

A data dictionary is the repository for recording, storing, and processing information about the system objects that contain or manipulate data. It is actually a database that describes the database managed by the DBMS. It is not an application dictionary, which describes information about the objects needed to develop applications (e.g, interprocess message definitions and COBOL structure definitions). Nor is a data dictionary a media catalog describing information pertaining to data-storage devices. The set of all such dictionaries, taken together, is sometimes referred to as a *repository*.

Data Dictionary Functions

A data dictionary has five basic functions.

Object Description. All objects in the database should be described in the data dictionary. Most databases have logical and physical objects, corresponding to a logical schema and a physical schema.

Logical objects provide users with a perspective of the database that is devoid of any physical information about database characteristics. Tables and views are two examples.

On the other hand, physical objects are described by the physical attributes of a database. They represent the physical organization of the database. Examples are files, partitions, indexes, and replicas. Physical attributes of these objects include location on disk, file organization, and storage allocation.

Relationships among Objects. Most objects in a database are related in one way or another. Semantic relationships between objects should be kept in the data dictionary, so that the same relationship can be enforced by the system under different applications. For example, a typical relationship between the department record and employee record is that each and every department name in the employee record must also exist in the department name of the department record. The enforcement of all such relationships provides what is called referential integrity.

Access Paths. Access paths are used to move data from an external storage device to an application. A dictionary contains information about all available access paths of any particular object—for example, the physical structure and location of the data or the number and types of indexes.

Statistics on the Database. Statistics on the database are very useful in query optimization. Statistics that can help the query optimizer choose a better execution plan should be stored and maintained in the dictionary. Some important statistics are the file size, the distribution of data values in a column of the table, the number of rows in a file, and the number of unique values in a column.

Protection of the Database. Database protection includes database security and database consistency. The data dictionary may include the information about the users' authorization to access the database and the integrity constraints of the database.

The NonStop SQL Data Dictionary

The NonStop SQL data dictionary contains descriptions of all NonStop SQL objects as well as information about their use. It is distributed, active, and integrated with Tandem system software.

Distributed

A distributed dictionary can provide better availability of dictionary definitions. If objects are described in a fragment of the dictionary at the local site, then work need not be interrupted when the system is disconnected from the network.

An object's data definition is stored in the system where the data resides. To obtain the definition of a remote object, the application must access the remote file. This approach eliminates redundant copies of a data definition and facilitates creation and alteration of objects.

Each object or partition of an object is only stored once. However, if two related objects are described in two different dictionaries, then the relationship between those objects is stored in both dictionaries.

Active

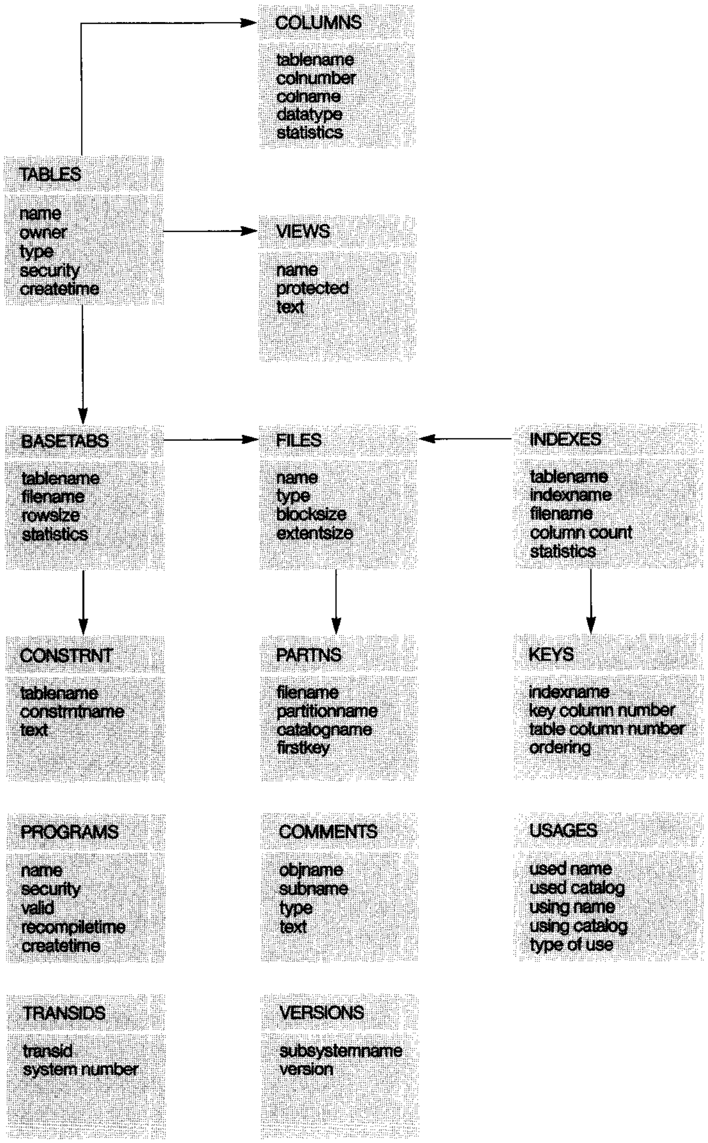
The NonStop SQL dictionary always correctly describes the objects in the database. When an object such as a table is altered, programs that were compiled using an old description of the object are invalidated. The next time the programs are executed, they are automatically recompiled to use the current definition of the altered object.

An active data dictionary contains descriptions that exactly reflect the way the system treats the object *at all times*. Conversely, a passive data dictionary does not reflect changes to system objects in the data dictionary. When this happens, the data dictionary no longer reflects the state of the object.

The term, "active data dictionary," is somewhat misleading. The data dictionary itself is not active—it is only a collection of data on disk. What makes the dictionary active is that all system software that accesses the database has been coded to reference and, when necessary, alter the data dictionary whenever an object is manipulated. If any interfaces alter the structure of the database and bypass the dictionary, the dictionary cannot be said to be active.

Figure 1.
Catalog structure.

Figure 1



BASETABS	– Describes attributes of tables.
COLUMNS	– Describes the columns of tables and views.
COMMENTS	– Keeps comments on columns, constraints, indexes, tables, and views.
CONSTRNT	– Describes constraints defined on tables.
FILES	– Describes attributes of files that contain tables and indexes.
INDEXES	– Describes indexes defined on tables.
KEYS	– Describes key columns on indexes.
PARTNS	– Describes partitions of tables and indexes.
PROGRAMS	– Describes SQL object program files.
TABLES	– Describes tables and views.
TRANSIDS	– Keeps TMF transaction IDs for current DDL operations on the catalog.
USAGES	– Describes dependencies between SQL objects.
VERSIONS	– Keeps version information about the catalog.
VIEWS	– Describes attributes of views.

Integrated with Tandem System Software

The NonStop SQL dictionary is fully integrated with Tandem's GUARDIAN 90™ operating system, SQL compiler, Transaction Monitoring Facility (TMF), file system, and disk process. This integration offers many benefits. For example, if the dictionary security were not integrated with the operating system's security, a hostile programmer could subvert the DBMS's security mechanism by issuing operating-system calls to open, read, and write data files managed by the DBMS, thereby corrupting the database.

NonStop SQL Data Dictionary Structure

The NonStop SQL data dictionary has two components: a compile-time dictionary comprising a set of catalog tables, and a run-time dictionary comprising a set of disk-file labels. Taken together, all the SQL file labels and SQL catalog tables form the SQL data dictionary.

Catalogs

A catalog is an application database made up of SQL tables; the function of the application is to describe SQL objects. Together, these SQL tables can be used to describe any SQL object in the system. The data in the SQL tables is stored in text form and can be queried with standard SQL statements. The catalog structure is shown in Figure 1.

Each SQL object must be described in a catalog at that system. Therefore, each system using NonStop SQL must have at least one catalog. Partitioned objects (tables, indexes, and views) have one complete catalog entry per partition. A partition must be described in a catalog residing at the same node as the partition. This restriction is made so that NonStop SQL can provide local autonomy for data access.

A catalog has the same name as the subvolume in which it resides. All the catalog tables that make up a catalog reside in the same subvolume. Each subvolume can have only one NonStop SQL catalog, but any number of catalogs can be created on a system. It might be useful to store logically separate databases in different catalogs for ease of use or for security purposes. However, storing objects in different catalogs provides no functional benefit. The DBMS always functions as if all database objects were stored in a single catalog.

Disk-File Labels

Each disk volume on a system has a disk directory. For each NonStop SQL object on the volume, there is one disk-file label in the directory that contains the name of the object, the name of the catalog in which the object resides, the security information associated with the object, and other information about the object. The file labels are not in text form.

The disk-file labels contain all the information needed by the low-level run-time components of NonStop SQL, the file system, and the disk process. This information allows NonStop SQL to open and operate on data without accessing the catalog. Since the catalogs are not accessed at run time, they should not become a performance bottleneck.

The Catalogs Table

Each system in the network has a single table, called the Catalogs table, that describes all the catalogs in the system. This table is modified only when a catalog is created or destroyed. By restricting write authority to this table, the user can prevent catalogs from being created and dropped. By further restricting write authority to the catalogs that exist on the system, the user can prevent new SQL objects from being created. This ability to limit the number of people who can create database objects is called resource authority.

System Catalog

Each system in a network has a special catalog called the system catalog. In the system catalog, the DBMS records its own database objects such as the Catalogs table and the Programs entry for the SQL conversational interface (SQLCI) command interpreter.

Local Autonomy and Catalog Consistency

NonStop SQL supports local autonomy for data access. The NonStop SQL compiler can compile SQL statements against local data using only the information present in local catalogs. The file labels contain all the information the NonStop SQL system needs at run time. Thus, the local catalogs and file labels provide enough of the dictionary to permit access to local data.

For certain Data Definition Language (DDL) operations, the DBMS designer must choose between local autonomy and data-dictionary consistency. For example, suppose a system administrator wants to remove an object from the system, but that object (such as an index on a remote table) is reflected in a catalog at another system that is unavailable. In this case, NonStop SQL will not allow the object to be dropped using a standard DDL command. Thus, where DDL operations are concerned, NonStop SQL opts for the integrity of the data dictionary over local autonomy.

NonStop SQL Naming

NonStop SQL users may refer to database objects using either logical names or physical names. A logical name is also known as a *define* name. It can have up to 30 characters, and the first character must be an equal sign. Logical names can be used wherever a physical name is expected by the DBMS.

Physical names conform to the Tandem convention for GUARDIAN 90 file names: "system.volume.subvolume.object". An example is \DC.$A.B.PARTS. The sole exception is a catalog name, which stops at the subvolume.

When using define names, the user must establish an operating context that maps all the logical names to their corresponding physical names. If the logical-to-physical mapping in effect at run time differs from the mapping in effect when a program was compiled, the section of the program referring to that logical name will be dynamically recompiled.

NonStop SQL Objects

NonStop SQL objects are database entities that can be created, manipulated, or dropped by means of SQL commands. All NonStop SQL objects are described in the NonStop SQL dictionary. The basic NonStop SQL objects are catalogs, files, tables, indexes, partitions, views, and SQL object programs.

Catalogs

The system catalog is created automatically when SQL is first installed on the system. Users can create other catalogs to suit their own policies. A sample catalog can be created as follows:

```
CREATE CATALOG \DC.$A.CAT1
```

Files

A disk file is the physical storage for data in the database. Like ENSCRIBE, NonStop SQL supports three types of file structures:

- In an entry-sequenced file, each new record is stored at the end of the file in chronological sequence and the primary key is a system-generated record address.

- In a key-sequenced file, each new record is stored in the sequence of the primary key value, using B-trees.
- In a relative file, each new record is stored at the relative record location specified by its primary key, which is either a user-defined or system-defined relative record number.

Base Tables

A Base table is the logical representation of data stored in a physical disk file. It defines data in columns and specifies a primary key. In addition, the CREATE TABLE statement defines physical file attributes, such as block size, file organization, and so on. The creation of a Base table implicitly creates a physical disk file with the same name. For example, an Orders table in a key-sequenced file is created as follows:

```
CREATE TABLE \DC.$A.B.ORDERS
   (LOCATION    CHAR(10)
   , PARTNO     INTEGER
   , UNITCOST   INTEGER
   , QUANTITY   INTEGER
   , KEY (LOCATION, PARTNO)
   )
   ORGANIZATION KEY SEQUENCED
   CATALOG \DC.$A.CAT1
```

Figure 2 shows sample data in the table \DC.$A.B.ORDERS.

Indexes

An index is an alternate access path to data in a table and is stored in a key-sequenced file. The creation of an index implicitly creates a physical disk file with the same name. For example, for the Orders table created above, one can create an index called ORDERS0 that will use the PARTNO column to provide fast, indexed access, as follows:

```
CREATE INDEX          \DC.$A.B.ORDERS0
ON                    \DC.$A.B.ORDERS
( PARTNO )
CATALOG \DC.$A.CAT1
```

Figure 3 shows sample data in index \DC.$A.B.ORDERS0.

Figure 2

LOCATION	PARTNO	UNITCOST	QUANTITY
DC	1000	10	100
DC	1001	10	100
LA	1002	10	130
SF	2000	40	400
SJ	1000	40	600

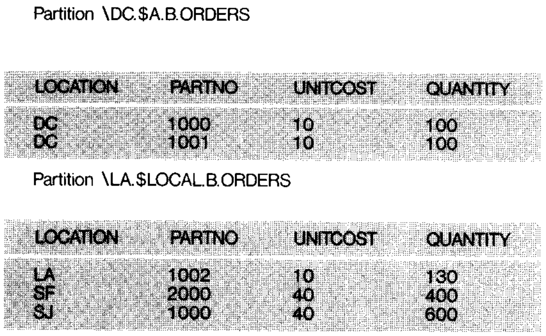
Figure 3

KEYTAG	PARTNO	LOCATION	PARTNO
1	1000	DC	1000
1	1000	SJ	1000
1	1001	DC	1001
1	1002	LA	1002
1	2000	SF	2000

Figure 2.
Sample data in table \DC.$A.B.ORDERS.

Figure 3.
Sample data in index \DC.$A.B.ORDERS0. The KEYTAG column is provided for future extendability. This permits storing multiple indexes in a single file.

Figure 4



Partition \DC.$A.B.ORDERS

LOCATION	PARTNO	UNITCOST	QUANTITY
DC	1000	10	100
DC	1001	10	100

Partition \LA.$LOCAL.B.ORDERS

LOCATION	PARTNO	UNITCOST	QUANTITY
LA	1002	10	130
SF	2000	40	400
SJ	1000	40	600

Figure 4.
Sample data in partitions.

Partitions

A partition is a portion of a table or index that resides in a particular disk volume. It is based on the concept of horizontal fragmentation. (A relation is horizontally fragmented if the rows in the relation are grouped into separate files. A relation is vertically fragmented if the columns of the relation are grouped into separate files.) Tables, indexes, and protection views can be partitioned.

An entry in a catalog for each partition of a table or index fully describes the table or index. Each partition contains relationships indicating the names of all the other partitions of the same object. This is different from the mechanism for storing relationships between different objects, in which only the relation between the primary partitions of those objects is stored. The following table creation command creates a table partitioned across two sites of the network.

```
CREATE TABLE \DC.$A.B.ORDERS
  ( LOCATION     CHAR(10)
  , PARTNO       INTEGER
  , UNITCOST     INTEGER
  , QUANTITY     INTEGER
  , KEY (LOCATION, PARTNO)
  )
  CATALOG \DC.$A.CAT1
PARTITION (\ LA.$LOCAL.B.ORDERS
  CATALOG \LA.$LOCAL.CAT2
  FIRST KEY ("LA", 0) )
```

The logical table definition is stored in catalog \DC.$A.CAT1. The second partition definition is stored in catalog \LA.$LOCAL.CAT2. When the table is referenced at SQL compile time, either catalog can provide the necessary information.

The physical table definition is stored in the disk label of \DC.$A.B.ORDERS. The second partition definition is stored in the disk label of \LA.$LOCAL.B.ORDERS. Figure 4 shows sample data in partitions.

Views

A view is a logical definition of a relation but has no physical existence. The data presented by a view is, instead, derived from a Base table. NonStop SQL supports two types of views: protection views and shorthand views. There is no performance penalty for using either type of view.

Protection Views. A protection view has protection attributes. It can be derived from a single table by taking either a projection of the columns of the table, a selection of the rows of the table, or both. A protection view provides a form of field-level security because the view can be secured, updated, and read.

For example, for the Orders table, one can create a protection view to ensure that all local users can access the PARTNO, QUANTITY, and LOCATION of orders issued from San Jose, as follows:

```
CREATE VIEW \DC.$A.B.LORDERS
(PARTNO, QUANTITY, LOC)
AS SELECT PARTNO, QUANTITY, LOCATION
FROM \DC.$A.B.ORDERS
WHERE LOCATION = "SJ"
FOR PROTECTION
SECURE "AAAA"
```

Figure 5 shows sample data in protection view \DC.$A.B.LORDERS derived from the table \DC.$A.B.ORDERS.

Each protection view has a separate disk-file label that contains the compiled form of the view definition to be used by the disk process at run time. The logical description of the view is recorded in the catalog. This is used by the SQL compiler or catalog manager when the view is referenced in an SQL statement. It can also be used for reporting purposes. For partitioned tables, the protection view is partitioned like the table. The view definition is replicated in every catalog that describes the partition and in every disk-file label of the partition.

Figure 5

PARTNO	QUANTITY	LOC
1000	600	SJ

Figure 5.
Sample data in protection view \DC.$A.B.ORDERS derived from the table \DC.$A.B.ORDERS.

Shorthand Views. A shorthand view can be derived from one or more tables or views by joining tables or views, taking projections of the columns, taking selections of the rows, or a combination of these methods. A shorthand view can be read but not updated or secured. Moreover, the user's security is tested against the security of each table and protection view that the shorthand view comprises.

For example, a shorthand view can be created to retrieve a list of part numbers, the names for suppliers of each part, and the quantity on hand, as follows:

```
CREATE VIEW \DC.$A.B.GETPARTS
   (PNUM,SNAME,QTY)
AS SELECT PARTNO, SUPPNAME, QUANTITY
FROM \DC.$A.B.ORDERS,
   \SJ.$A.B.PARTSUPP
WHERE \DC.$A.B.ORDERS.PARTNO =
   \SJ.$A.B.PARTSUPP.PARTNO
```

Figure 6 shows sample data in the shorthand view \DC.$A.B.GETPARTS, derived from the table \DC.$A.B.ORDERS and the table \SJ.$A.B.PARTSUPP.

Figure 6

Sample data in table \DC.$A.B.ORDERS

LOCATION	PARTNO	UNITCOST	QUANTITY
DC	1000	10	100
DC	1001	10	100
LA	1002	10	130
SF	2000	40	400
SJ	1000	40	600

Sample data in table \SJ.$A.B.PARTSUPP

PARTNO	SUPPNAME
1000	GE
1001	GM
1002	GE
2000	GM

Sample data in shorthand view \DC.$A.B.GETPARTS

PNUM	SNAME	QTY
1000	GE	100
1000	GE	600
1001	GM	100
1002	GE	130
2000	GM	400

Figure 6.
The sample data in shorthand view \DC.$A.B.GETPARTS is derived from table \DC.$A.B.ORDERS and table \SJ.$A.B.PARTSUPP.

Each shorthand view has a separate disk-file label that contains the name of the catalog describing the view. The logical description of the view is recorded in the catalog that is used by the SQL compiler or catalog manager when the view is referenced in an SQL statement.

It should be restated that shorthand views can derive data from protection views. Thus, a secure view of two tables can effectively be realized by creating protection views with the desired security on the underlying tables and then joining them together with the shorthand view. This approach was taken in preference to allowing shorthand views to be secured directly, because in a distributed system it is important for security to be tested where the data, not the user, resides. Protection views provide this mechanism; shorthand views do not.

SQL Object Programs

An SQL object program is an object file containing executable machine-language instructions produced from a host-language source program with embedded SQL statements. It also contains an SQL plan corresponding to each SQL statement that was embedded in the program along with the text of that statement (for dynamic recompilation). All valid SQL object programs are described in an SQL catalog. When a program containing SQL statements is compiled, the program is registered in a catalog, and relationships are recorded that indicate which SQL objects are used by the program. Users can invoke a where-used utility that traverses these relationships and reports which SQL objects depend on other SQL objects in the system.

User-Defined Constraints

User-defined constraints are as much attributes as objects. Constraints are conditions associated with a table that must be satisfied before rows can be inserted or updated. These conditions help to maintain data integrity. A constraint is an expression that combines the values found in a row of a table with any number of comparison operators and literals. A constraint cannot refer to other rows or tables.

For example, for the Orders table, one can create an integrity constraint to ensure that the ordered quantity must be a positive integer, as follows:

```
CREATE CONSTRAINT
   QUANTITY_CONSTRNT
   ON ORDERS
   CHECK QUANTITY > = 0
```

After a constraint is created, it is added to the Constrnt table of the catalog and into the file label. In the previous example, the Constrnt table will contain the text string "QUANTITY > = 0", which is used by the catalog manager when the constraint is referenced in an SQL DDL statement. The file label will contain the compiled form of the constraint, which can be executed by the disk process at run time. If the Orders table has partitions, the constraint will be replicated in every catalog that describes the partitions and every disk-file label of the partitions.

NonStop SQL Catalog Manager

The NonStop SQL catalog manager is the focal point for all updates to the SQL data dictionary. The catalog manager is the only process licensed to update the data dictionary. All dictionary updates are routed through it. In particular, the catalog manager executes all the SQL DDL commands. The catalog manager coordinates catalog-table updates with disk-label updates to preserve consistency between catalog tables and disk labels.

Catalog Manager Architecture

The catalog manager consists of the main program, the parser, the binder, the normalizer, and the execution routines. The main program interfaces with other SQL components and passes commands to the other catalog-manager components. The parser is responsible for parsing SQL DDL statements. The binder is responsible for name resolution. The normalizer is responsible for parse-tree transformation for the execution routines. The execution routines are responsible for performing operations such as CREATE, DROP, and ALTER that are specified in the parse tree.

Dictionary Services Provided by the Catalog Manager

SQL Initialization Service. The SQL initialization service is supported by a special interface to the SQLCI. As part of the initialization process, the SQLCI invokes the catalog manager to create the system catalog.

Execution of DDL Commands. The execution of NonStop SQL DDL commands is supported by an interface to the SQL executor. The SQL executor executes compiled SQL commands or statements against the database. However, when the executor encounters a DDL command, the executor sends the request to the catalog manager. This results in updates to the appropriate catalog tables and to the corresponding disk-file labels.

In addition to supporting the creation, alteration, and dropping of system objects, the catalog manager supports the collection and storing of statistics on the distribution of data in a table. Statistics are refreshed when the user issues the UPDATE STATISTICS command.

DDL Utility Service. The NonStop SQL DDL utility service is supported by an interface to the SQL utilities. For example, when restoring an SQL object, the BACKUP/RESTORE utility program invokes the catalog manager to create the object.

File System Service. The NonStop SQL file system service is supported with the interface to the ENSCRIBE procedures RENAME, SECURE, and PURGE. For example, when purging an SQL program, the PURGE procedure invokes the catalog manager to purge the disk-file label of the SQL program and to delete the corresponding catalog entries. This is necessary for the data dictionary to be active.

Object-Program Maintenance. The NonStop SQL object-program maintenance service is supported with the interface to the SQL compiler. For example, when compiling a host-language source program with embedded SQL statements, the SQL compiler invokes the catalog manager to register the SQL object program and the relationships between the program and all SQL objects referenced by the program. This is necessary for the data dictionary to be active.

Database Consistency and Security

The NonStop SQL data dictionary depends on the Transaction Monitoring Facility (TMF) to ensure consistency between the catalog tables and the file labels. TMF protects SQL tables and file labels from damage due to system or media failures. TMF transactions are used in the catalog manager to coordinate all activity affecting the SQL catalog and file labels.

NonStop SQL is integrated with the Tandem GUARDIAN 90 operating system. GUARDIAN 90 protection mechanisms also apply to SQL objects. The security information of SQL objects is stored in the disk-file label and in the SQL catalog. The catalog manager enforces certain security policies between SQL objects to facilitate the SQL operations. For example, the catalog manager insists that a user who is authorized to access an SQL table must also have the same authority on all associated indexes.

All security checks are performed by the disk process. Furthermore, the disk process checks for the caller's licensed bit for certain operations. For example, only licensed processes can create (or change) disk-file labels of SQL objects or update entries stored in the SQL catalog.

Conclusion

NonStop SQL provides a distributed DBMS and a distributed database. A distributed data dictionary is used to describe this database. It comprises a compile-time dictionary and a run-time dictionary. The compile-time dictionary allows the user, the SQL compiler, and the utilities to easily access the definition of all SQL objects, while the run-time dictionary allows the file system and the disk process to efficiently access all required definitions on SQL objects.

The data dictionary is designed to be active and integrated with Tandem system software to satisfy the customer's requirements on function, security, integrity, performance, availability, and ease of use.

Dictionary Terminology

Distributed Database

A distributed database is a single database whose objects reside on more than one system in a network of systems. It functions in all respects as if the entire database resided at a single site. Moreover, a distributed database can, in theory, allow for the creation of relations that span multiple machines. Two techniques for distributing relations across systems are partitioned (or fragmented) data and replicated data.

Relations: Partitioned and Replicated

When a relation is *partitioned* across many sites, each site contains a subset of the relation. There are two types of partitions: horizontal and vertical. A relation is horizontally partitioned if the rows in the relation are grouped into separate files. A relation is vertically partitioned if the columns of the relation are grouped into separate files.

When a relation is *replicated* at many sites, each site contains a copy of the relation. Data is replicated either to obtain higher availability of the data in a network or to improve performance.

Location Transparency

Using a distributed DBMS, a distributed database should behave exactly as an ordinary centralized database does. Security issues aside, any stream of database commands should provide identical results when entered at any system in the network.

Information Resource Dictionary System

The American National Standards Institute (ANSI) is working on an industry-wide dictionary standard called the Information Resource Dictionary System (IRDS). This standard would allow people to create objects of arbitrary types to be included in a dictionary. The structure of this dictionary is predefined. Because this dictionary does not come with definitions of specific object types, the IRDS dictionary is not a data dictionary, but rather a dictionary that could be used to describe a data dictionary or any other kind of dictionary.

References

Cardenas, A. 1984 (2nd Edition). *Data Base Management Systems*. Allyn & Bacon.

Ceri, S., and Pelagatti, G. 1984. *Distributed Databases: Principles & Systems*. McGraw-Hill.

Braude, M. 1987. Rules for Distributed DBMS. *Software Management Strategies*. The Gartner Group, Inc. T-150-314.1.

Anderton, M., and Gray, J. 1985. *Distributed Computer Systems—Four Case Studies*. TR 85.5. Tandem Computers Incorporated.

NonStop SQL Conversational Interface Reference Manual. Part no. 82319. Tandem Computers Incorporated.

Tandem Database Group. 1987. *NonStop SQL: A Distributed, High-Performance, High-Availability Implementation of SQL*. Tandem Computers Incorporated.

Wiederhold, G. 1984. Databases. *Computer*. Vol. 17, No. 10.

Rob Holbrook is the software product manager responsible for NonStop SQL. He has six years of experience as a software designer in Tandem's Database Group. He worked for three years on the development of NonStop SQL. He received a B.A. in Economics and M.S.E.E. in Computer Engineering from Stanford University.

Don-Min Tsou is one of the original designers of NonStop SQL. He joined Tandem Software Development in May 1983. Prior to this, he participated in the development of several widely used software products, including an easy-to-use query language. He received a B.S. in Electrical Engineering from National Taiwan University, and a Ph.D. in Computer Science from Pennsylvania State University.

Technical Paper: High-Performance SQL through Low-Level System Integration

One of Tandem's major goals when implementing NonStop SQL was to provide not only the high functionality and ease of use associated with SQL, but also a performance rating high enough to make SQL an efficient choice in a production environment. NonStop SQL achieves high performance through an implementation that integrates SQL record access with the pre-existing disk I/O and transaction management subsystems, DP2 and TMF (Transaction Monitoring Facility). This low-level system integration reduces message traffic and CPU consumption by putting SQL optimizations at the lowest levels of the system. Examples of these optimizations are message traffic reduction 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. The result is an SQL system that matches the performance of the ENSCRIBE database management system, while inheriting such pre-existing architecturally derived features as high availability, transaction-based data integrity, and distribution of both data and execution.

Tandem's Approach to a High-Performance SQL

Many other vendors have implemented SQL as an "add-on layer" to the existing system and, as a result, provide minimal integration with the pre-existing architecture. This approach keeps development costs low and has a minimal impact on underlying system software but results in poorer performance. The "add-on layer" approach introduces one or more of the following:

- An SQL-specific transaction management system with a proprietary audit trail (log).
- A disk cache (buffer pool) management mechanism that operates as a layer above the native file system.
- An SQL-specific concurrency control mechanism that operates as a layer above the pre-existing concurrency control mechanism.
- New access method logic, specific to SQL tables, that operates as a layer above the native file system.

When these SQL system-support mechanisms operate as a layer above (not integrated with) the pre-existing database management system (DBMS) mechanisms, they cannot perform as efficiently as the native file system or DBMS. There are two reasons for this. First, multiple layers increase the path length. Second, low-level optimizations that improve the performance of the old DBMS do not necessarily apply to the SQL layer.

By contrast, Tandem's integrative approach does not re-implement any of these DBMS system-support mechanisms. Instead, SQL-specific logic has been introduced into the corresponding subsystems supporting ENSCRIBE, the pre-existing DBMS. Integration with Tandem's networking and distributed transaction management subsystems allows NonStop SQL to inherit pre-existing facilities for high availability, fault tolerance, and distribution. In addition, the inherited distributed architecture will allow progressively fuller exploitation of parallelism to improve performance in the future.

As compared with the layered approach, pushing NonStop SQL support logic to the lowest levels of the system produces performance gains by reducing low-level path lengths. It further 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, which are also in part describable as low-level, path-length savings.

Compared to the ENSCRIBE record-at-a-time interface, NonStop SQL significantly reduces message traffic by introducing a field-level interface to the low-level disk I/O system and by delegating to the disk process (low-level disk-file server) such SQL functions as field projection, predicate evaluation, and set-oriented retrievals, updates, and deletes. In addition, delegating an update via "update expression" (e.g., SET ACCOUNT.BALANCE = ACCOUNT.BALANCE - DEBIT) to the disk process eliminates the extra message that would otherwise be needed by 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 (*NonStop SQL Benchmark Workbook*, 1987).

Overview of Tandem Architecture

The Tandem NonStop™ architecture consists of up to 16 loosely coupled processors interconnected by dual high-speed buses to form a single system or node (Katzman, 1978). Nodes can be connected into clusters by fiber-optic links or into "long-haul" networks via X.25, SNA, or other protocols. The goals of the architecture are fault tolerance, high availability, and modularity.

Hardware and software redundancy maintain I/O device availability despite single module failure. Hardware redundancy provides alternate physical paths to I/O devices, and software redundancy provides fault-tolerant, device-controlling "process pairs." The "primary" process and its hot-standby "backup" process run in two processors physically connected to the device (Bartlett, 1981). A transaction mechanism coordinates the atomic commitment of updates by multiple processes in the network (Borr, 1981).

A message-based operating system manages system resources and provides communication between processes executing in the same or different processors. The message system makes the distribution of hardware components transparent (Bartlett, 1981). I/O processes are system-level processes that manage I/O devices; the disk process is the I/O process that manages disk volumes (optionally replicated on "mirrored" physical drives for fault tolerance).

Components of the Disk Process

The disk process is actually a group of cooperating processes that share a message-input queue. The process group acts as the I/O server for files resident on the volume it manages. These files include code files and virtual memory swap files as well as NonStop 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 that support the file structures common to ENSCRIBE and NonStop SQL:

- Key-sequenced (B-tree).
- Relative (direct access).
- Entry-sequenced (insert only at end of file).

The cache management component of the disk process manages a main memory buffer pool that stages data to and from disk, using a least-recently-used algorithm that obeys "write-ahead-log" protocol (Gray, 1978). The cache provides transaction-protected database read and write services while minimizing 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 uses 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 NonStop SQL and ENSCRIBE; it locks at the file, record, or "generic" (key prefix) level for volume-resident SQL or ENSCRIBE data.

Transaction-management code and audit-generation code permeate 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 (Borr, 1981). The dual roles of TMF and the backup disk process in maintaining high device availability, fault tolerance, transaction consistency, and robustness to crash have been described in other literature (Borr, 1984).

NonStop SQL and ENSCRIBE share the same TMF audit trail (log), which resides on the audit-trail volume. A standard disk process manages the audit-trail volume. The audit-trail writing component of the audit-trail volume's disk process is highly optimized for long or "bulk" sequential I/Os using "group commit" (Gawlick, 1985) and audit piggy-backing to maintain a high transaction commit rate with a minimal number of I/Os.

The disk process is a group of processes sharing a message-input queue.

Figure 1

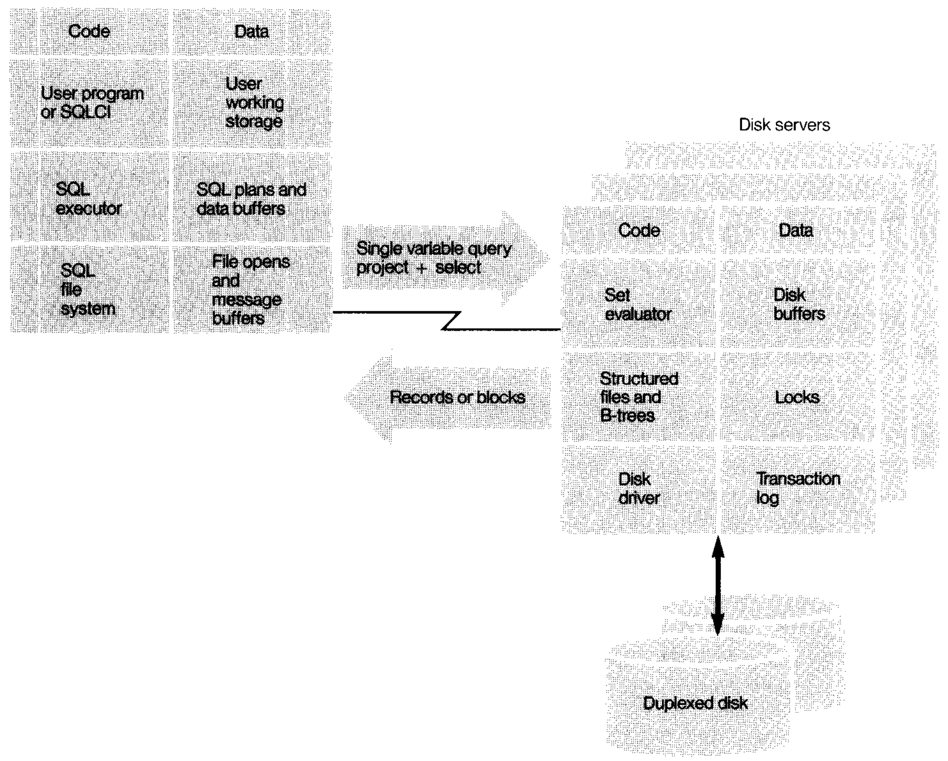


Figure 1.

The structure of a compiled and executing program. The application calls the SQL executor, which calls the file system. The file system sends single-variable query requests to the disk process, which does projections and selections on tables and protection views to produce a record subset. This subset is returned to the file system and executor or is updated or deleted by the disk process.

Rationale for Division of Labor between File System and Disk Process

The file system is a set of system library routines that have their own data segment but run in the process environment of the application 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 ENSCRIBE, the application program invokes the file system explicitly, calling such routines as OPEN, READ, WRITE, and LOCKRECORD to perform key navigation and record-oriented I/O.

In NonStop SQL, the application program's SQL statements invoke the SQL executor, a set of library routines that 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 compiled query (see Figure 1).

The distributed character of the Tandem architecture mandates division of labor between the file system and the disk process. Typically, database files in a Tandem application are spread across multiple disk volumes, which are 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.

This architecture makes the file system the natural locale for the logic (transparent to the caller) that manages access to the appropriate partition based on record key, manages access to the base-file record via a secondary key, or maintains secondary indices consistent with the update or delete of a base-file record.

For example, to implement a request to read via a secondary index, the file system first sends to the disk process that manages the index's volume a read request for the appropriate index record. After extracting the base-file record key from the index record, it 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 that manages that partition. These file system functions are common to both NonStop SQL and ENSCRIBE, although separate file system procedures perform them for the two systems.

The following sections describe the nature of the file-system disk-process (FS-DP) interface for ENSCRIBE and the reasons for designing 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 the record's 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., the number of records per block). After an FS-DP message returns a block to the file system, multiple record-at-a-time ENSCRIBE READ requests cause the file system to de-block its local block copy; then a message requesting the next block is sent to the disk process.

However, SBB under ENSCRIBE has limited utility because it locks at the file level only; no other locking is effective when SBB is in use. Because of this limitation, the user must have an OPEN-exclusion mode that 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 (*Database Language SQL 2*, 1986). User-specified predicates define selection criteria, update expressions, and integrity constraints. The field and set orientation of the user interface extend down to a field-oriented and set-oriented FS-DP interface, requiring 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), the disk process can evaluate this "single-variable query" for each record in a key range and use the query as a filter limiting the set of records processed or returned in the reply to the FS-DP message.

When an update expression specifies a new value for a field by using an expression that involves only literals and fields of the record at hand (e.g., SET ACCOUNT.BALANCE = ACCOUNT.BALANCE * 1.07), 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 (e.g., CHECK ACCOUNT.BALANCE > = 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.

NonStop SQL Statement Execution Reduced to Single-Variable Queries

Though a general SQL predicate can be multivariable (i.e., involve joins or expressions using fields of more than one table), the executor's file system invocations, mandated by the compiled query-execution plan, are expressed in terms of a single table, with optional access using 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 so that a single-variable query can be attached to the request message sent by the file system to the disk process, message traffic over the FS-DP interface can be reduced by filtering the data at its source. Because SQL selection and projection logic is pushed as low as possible in the system, the 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 NonStop SQL FS-DP interface, which has a set-oriented option, subcontracts selection and projection to the disk process wherever feasible. The disk process may be requested to operate on (i.e., to retrieve, update, or delete) a set of records that span a specified primary key range (may include all) and, optionally, satisfy 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.

Sequential Block Buffering

Using disk process selection and projection, the ENSCRIBE concept of sequential block buffering has been extended for NonStop SQL from "real" (RSBB) physical disk-block copies to "virtual" (VSBB) blocks. In VSBB, data is returned through the set-oriented FS-DP read interface after projected fields have been extracted from key-range-satisfying records that have optionally been subjected to a filtering predicate. This is similar to the concept of portals described by Stonebraker (Stonebraker and Rowe, 1984).

The locking restriction under ENSCRIBE (file locking only), which limited the usefulness of SBB, has been removed for NonStop 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, and VSBB gives NonStop SQL an additional factor of three over RSBB on many benchmark queries (Tandem Database Group, 1987). The performance gains of VSBB can be attributed 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 that involve selection and projection and return data by VSBB.

Message types:

GET∧FIRST∧VSBB
GET∧NEXT∧VSBB

Table EMP has the following fields:

EMPNO (primary key), NAME, HIRE_DATE, SALARY, ...

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

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 that 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 resend the predicate or the projection. These latter were saved in the subset control block created by the disk process at GET∧FIRST time.

Example 2: Real Sequential Block Buffering

The following statement, which involves no selection or projection, maps into a series of set-oriented read requests that return data using RSBB.

Message types:

GET∧FIRST∧RSBB
GET∧NEXT∧RSBB

```
SELECT * FROM EMP;
```

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.

Message types:

UPDATE∧SUBSET∧FIRST
UPDATE∧SUBSET∧NEXT

Table ACCOUNT has the following fields:

ACCTNO (primary key), BALANCE, ...

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

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 resend the predicate or the update expression. These latter expressions were saved in the subset control block 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/Os. Where possible, the disk process reads into cache buffers sequential strings of physical blocks (currently limited to 4 Kbytes maximum each) using "bulk" I/Os (currently limited to 28 Kbytes maximum). Of course, where physical clustering of key-sequenced data blocks has been broken due to B-tree splits and collapses, some bulk I/Os 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. With asynchronous pre-fetch, CPU-bound processing using data from the cache can occur in parallel with disk I/Os.

The disk process also uses bulk I/O 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, 1978). The disk process then writes the string to disk using the minimal number of bulk I/Os.

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 resulting "field-compressed" audit records are generally smaller than 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 that 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 because 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, 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. The size of the audit-trail data on disk and all audit-containing messages throughout the system is reduced as well.

Opportunities for Future Performance Enhancements for SQL

This paper has described the performance gains achieved by integrating NonStop SQL with pre-existing, low-level system mechanisms. These gains point the way to improving SQL performance in other areas, including:

- The FS-DP sequential-write interface.
- The constructs UPDATE WHERE CURRENT and DELETE WHERE CURRENT.
- A fuller exploitation of the Tandem system's parallel architecture.

The FS-DP Sequential Write Interface

Changing the FS-DP sequential write interface could result in performance gains similar to those achieved by using sequential block buffering for reads. Currently, 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.

However, to avoid a late-detected, duplicate-key condition, the disk process would have to keep an empty, sequential, target-key range locked by prior agreement with the file system. With this interface, the disk process could 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-audit if written to disk would then be written out using bulk I/O.

Update and Delete WHERE CURRENT Constructs

The performance gains achieved by using set-oriented update- and delete-request messages suggest that similar improvements may be made for the constructs UPDATE WHERE CURRENT and DELETE WHERE CURRENT. Currently, these constructs require one message per updated or deleted record. If the updates (deletes) were to occur in a buffer local to the file system and the buffer full of updates (deletes) was sent to the disk process in one message, substantial message traffic savings in the FS-DP interface could be realized.

Exploiting Tandem's Parallel Architecture

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 (Tandem Database Group, 1987). The overlap of I/O and CPU-bound processing inherent in asynchronous pre-fetch and write-behind is also a form of parallelism.

Furthermore, a current user option 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, 1986). Future opportunities for using intra-query parallelism include distributed query optimization, parallel executor-process structure, and no-wait disk process "message-sends" in the file system.

Tandem's continuing commitment to the implementation of NonStop SQL ensures that these performance-enhancement opportunities will be fully explored in the future.

Conclusion

By pushing SQL-specific logic to the lowest levels of the operating system, Tandem has obtained an SQL system that 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 compensate for the increased path length at higher levels needed to support the high functionality and ease of use of the SQL language.

In addition, system integration allows NonStop SQL to inherit from the pre-existing system the facilities that 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.

References

Bartlett, J.F. 1981. A NonStop Kernel. In *Proceedings of Eighth Symposium on Operating System Principles.* Association for Computing Machinery (ACM).

Borr, A.J. 1981. Transaction Monitoring in ENCOMPASS: Reliable Distributed Transaction Processing. In *Proceedings of the Seventh International Conference on Very Large Data Bases.* September. Republished as Tandem TR 81.2. Tandem Computers Incorporated.

Borr, A.J. 1984. Robustness to Crash in a Distributed Database: A Non Shared-Memory Multi-Processor Approach. In *Proceedings of the Tenth International Conference on Very Large Data Bases.* September. Republished in *Tandem Systems Review.* Vol. 1, No. 2. Tandem Computers Incorporated. Part no. 83935.

Database Language SQL 2 (ANSI Working Draft). 1986. ANSI X3H2 87-8.

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

Gray, J.N. 1978. *Notes on Data Base Operating Systems.* IBM Research Report: RJ 2188. International Business Machines Corporation.

Katzman, J.A. 1978. A Fault-Tolerant Computing System. In *Proceedings of the Eleventh Hawaii International Conference on System Sciences.*

NonStop SQL Benchmark Workbook. Part no. 84160. Tandem Computers Incorporated.

Pong, M. 1988. Access Plan Selection in NonStop SQL. *Tandem Systems Review.* Vol. 4, No. 1. Tandem Computers Incorporated. Part no. 400103.

Stonebraker, M., and Rowe, L. 1984. Database Portals: A New Application Program Interface. In *Proceedings of the Tenth International Conference on Very Large Data Bases.* September.

Tandem Database Group. 1987. *NonStop SQL, A Distributed, High-Performance, High-Availability Implementation of SQL.* Tandem Technical Report 87.4. Tandem Computers Incorporated.

Tsukerman, A., et al. 1986. *FastSort: An External Sort Using Parallel Processing.* Tandem Technical Report 86.3. Tandem Computers Incorporated.

Acknowledgments

While many members of the Tandem Database Group contributed ideas to the design of the low-level architecture for NonStop SQL, I would particularly like to acknowledge the contributions of Franco Putzolu.

Andrea Borr is currently working on disk process DP2 support for NonStop SQL. She previously worked on the design and development of the ENCOMPASS products ENFORM, TMF, and DP2. Before joining Tandem, Andrea spent 8½ years as a software developer and field analyst for two other mainframe vendors. Andrea holds a bachelor's degree in Mathematics from the University of Chicago and a master's degree in Computer Science from the University of Wisconsin. She was also a doctoral candidate at Stanford University.

Debugging TACL Code

The Tandem Advanced Command Language (TACL) is the standard interface to the GUARDIAN 90™ operating system. In addition to providing full command interpreter facilities, TACL can be used as a programming language.

Writing TACL routines involves a readjustment in perspective for programmer-analysts who are used to working with a traditional language such as COBOL. TACL is interpretive and is focused toward providing a high-level language for command processing. Functions such as system initialization, system monitoring, and job control are excellent uses for TACL.

In an error situation, TACL, like any other programming language, interprets code as far as it can before producing an error message. The TACL built-in debugger visibly demonstrates how TACL interprets code. It allows step-by-step execution, shows control flow, and permits examination and modification of variables. Because the debugger displays each step, it is especially useful when learning how to work with conditionals and arithmetic computations.

This article describes how to debug both high-level TACL code and #DELTA code. It is intended for programmer-analysts who are interested in writing TACL code and are familiar with the concept of a TACL macro and TACL constructs (e.g., #PUSH and #FRAME). An understanding of variable "invocation" and "expansion" is also helpful, but an understanding of #REQUESTER and #SERVER constructs is not necessary.

Terminology

For the purposes of this article, the term "TACL commands" refers to high-level TACL constructs such as #PUSH, #FRAME, and #SET. #DELTA constructs are referred to as "#DELTA commands."

In addition, a distinction is made between TACL *user* commands and *debug* commands. The TACL user commands include all nondebugging TACL commands, including STATUS, WHO, and RUN. These are described more fully in the *TACL Reference Manual*. Debug commands are used strictly for debugging.

Interactive TACL Debugger

TACL is a very powerful language; the fact that it has a good debugging environment provides added strength. This section describes how to use the interactive TACL debugger and includes syntax and examples of use.

Enabling the TACL Debugger

The TACL debugger can be enabled interactively from the TACL prompt, or by adding a line of code to a macro or routine. The syntax is as follows.

- At a TACL prompt[1], before invoking the macro or routine:

10>BREAK *variable*

where *variable* is a macro name or the name of a routine that has already been loaded.

- From inside a macro or routine:

```
?SECTION name MACRO
#SET #TRACE -1
      .
      .
      .
```

When the debugger is enabled, TACL waits for an instruction before it does its first expansion. (This is similar to the INSPECT debugging facility's RUND operation.) At this point, the user can set breakpoints and resume execution or step through the code.

Table 1.
Debug command syntax.

Command	Syntax	Description
BREAK	B[REAK] [*variable*]	Set a breakpoint on the specified variable or variable level. If *variable* is omitted, all breakpoints are listed.
CLEAR	C[LEAR] *variable*	Clear the breakpoint for the specified variable or variable level.
	C[LEAR]*	Clear all breakpoints.
DISPLAY	D[ISPLAY] *variable*	Display the contents of a specified variable or variable level.
MODIFY	M[ODIFY] *variable*	Enter new contents for the specified variable or variable level. To use, type M *variable* at the debug prompt. It will ask for the new contents. Press carriage return after each line. When done, type CTRL/Y.
RESUME	R[ESUME]	Stop debug mode and continue execution of code.
STEP	ST[EP]	Perform the next expansion and return to debug prompt. Press RETURN to continue stepping.

TACL Debugger Commands

The six TACL debugger commands are shown in Table 1[2].

Anything other than a debug command (e.g., user commands, such as STATUS *, TERM) will be passed through to the command interpreter (TACL). Note, however, that since the debugger is part of the TACL process running on the user's terminal, commands that could impact the routine being debugged (e.g., #UNFRAME) are not recommended.

Debug commands must reference declared variables. The TACL debugger displays each line before it is evaluated; therefore, a declaration (#PUSH) will be in effect when the debugger is displaying a line after the #PUSH.

A breakpoint will stop execution when the referenced variable is invoked as a function. A breakpoint is not effective when the variable is used as an argument to a function—for example, the code #SET*x* will not cause a breakpoint on variable *x*.

[1] The TACL prompt is a "greater than" sign (>). The number appearing to the left of the prompt is the count of the command in the sequence of commands the user has entered (e.g., a number 1 to the left of the prompt indicates the first command typed in, a 2 indicates the second command, etc.). The double equal signs (==) introduce TACL comments.

[2] In Table 1, brackets indicate an optional portion of the command. In all other places, brackets are part of the command syntax.

Debugging a TACL Macro

There are many ways of writing a TACL macro to list suspended PATHWAY terminals. Figure 1 is a sample macro showing one very simple way of doing this. This macro, however, has an error which can be located by using the debugger.

TACL debugging allows an inside view of how TACL is interpreting code.

When the INFO macro is invoked, it runs but does not display any data. The following appears:

```
5>RUN INFO
Suspended terminal(s):
6>
```

A manual run of PATHCOM shows two terminals in suspended state. This means that the TACL macro is not working correctly and a debug session is needed. The debug facility is enabled by adding the line #SET #TRACE -1 after the #SETMANY command. INFO is then invoked again:

```
9>RUN INFO
PATHCOM /OUTV rslt/ $trpm; status term *; &
exit³
-TRACE-
-10-
```

At the first prompt, set a breakpoint on *state* and resume execution:

```
-10-B state
-11-R
```

Continue to the first invocation of *state*:

```
Suspended terminal(s):
[#IF [#MATCH SUSPENDED [state]
                              ^
-BREAK-
-12-
```

The contents of variables can be displayed at this point:

```
-12-D state
-13-D termname
 line
-14-
```

Figure 1

```
NOTE: This example can be adapted for any PATHWAY environment by changing "$TRPM"
to an appropriate PATHMON name.

?TACL MACRO
#FRAME
#PUSH rslt termname state line                  == declare variables
#SETMANY rslt termname state line,              == initialize variables

PATHCOM /OUTV rslt/ $trpm; status term *; exit
                                                == run PATHCOM, store
                                                == output in rslt

SINK [#EXTRACT rslt][#EXTRACT rslt]             == throw out 2 header lines
#EXTRACTV rslt line                             == store first data line of
                                                == rslt into line

#OUTPUT Suspended terminal(s):
[#LOOP |WHILE| NOT [#EMPTYV rslt]               == loop until all rslt
|DO|                                            == lines have been read
   #SETMANY termname state, line                == get 1st 2 columns of line
   [#IF [#MATCH SUSPENDED [state]]              == see if 2nd = SUSPENDED
   |THEN|                                       == if so, print the line
      #OUTPUTV line
   ] == end of IF                               == put next rslt line into
   #EXTRACTV rslt line                          == line
   ] == end of LOOP
   #UNFRAME
```

Figure 1.

*Sample TACL macro. The macro is stored in a file called INFO. After doing a PATHCOM STATUS TERM * to investigate terminals running under a PATHMON named $TRPM, it stores the results into a variable called* rslt *and checks for suspended status. Finally it prints status information for each suspended terminal.*

[3] For the purpose of formatting this article, some of the lines of code were split. Normally, these examples would appear on one line.

Nothing is displayed for *state*, indicating null contents. The #SETMANY statement, which should have put the first two columns from *line* into *termname* and *state*, is not working as intended. Instead, it is putting the actual word "line" into *termname*. Brackets ([ ]) must be placed around *line* in order to get the correct results.

After making this change, invoke INFO again:

```
7>INFO
-TRACE-
-248-
```

Enter the RESUME command to begin normal execution:

```
-248-R
Suspended terminal(s):
FAX1 SUSPENDED 1121 FAX-TCP1 $FAX0
MSC1 SUSPENDED 1121 FAX-TCP1 $FAX0
8>
```

The macro now works correctly.

Interactive #DELTA Debugger

#DELTA is a programmable text manipulation facility that can be considered a special sublayer of TACL. #DELTA functions are characterized by one- or two-character command sequences and can be called by TACL routines and macros when special string editing needs to be done.

#DELTA functions must be operating correctly before the routine is used by the TACL code. TACL receives the result of a #DELTA function; the intermediate steps within a #DELTA function are not visible to the TACL debugger.

Interactive #DELTA can verify #DELTA results by allowing a user to step through test data with #DELTA commands.

Interactive #DELTA is enabled as follows:

```
8>#DELTA
#DELTA 9>
```

A text string can be passed to interactive #DELTA by appending it to the #DELTA command. However, it can be very useful to set up TACL variables with values that can then be used by multiple #DELTA tests. Variables for use by #DELTA are set up as follows:

```
10>#PUSH inp == declare an input variable
11>#PUSH outp == declare an output
   variable
12>#APPEND inp TEST DATA
13>#OUTPUTV inp == display variable's
   contents
TEST DATA
14>#DELTA
#DELTA 15>
```

Interactive #DELTA Commands

Two interactive #DELTA commands are used in conjunction with #DELTA command streams to view intermediate #DELTA results. (See Table 2.)

Table 2.
Interactive #DELTA commands.

Command	Syntax	Description
View	v	View the contents of the buffer. The pointer is represented by a period (.). Note: The view command allows additional range specification. For more information, please refer to the *TACL User's Guide.*
Display pointer position	.=	Display the character position of the pointer as an integer.

Note: Enter CTRL/Y to see the results of the view and Display pointer commands. Enter CTRL/Y twice to exit interactive #DELTA.

Debugging #DELTA Code

Figure 2 shows the INFO macro from Figure 1. The macro has been modified so that #DELTA is used to look for the suspended lines.

Before using the macro, the #DELTA portion can be tested using interactive #DELTA.

First, declare two variables for use by the #DELTA function:

```
10>#PUSH susp
11>#PUSH rslt
```

Next, place appropriate test data into *rslt*:

```
12>#APPEND rslt 177T RUNNING
M6530-TCP1 $TB1
13>#APPEND rslt FAX1 SUSPENDED
1121 FAX-TCP1 $FAX0
14>#APPEND rslt MSC1 SUSPENDED
 1121 FAX-TCP1 $FAX0
```

Display the contents of *rslt* to verify that the test data is correct:

```
16>#OUTPUTV rslt
177T RUNNING M6530-TCP1 $TB1.#D
FAX1 SUSPENDED 1121 FAX-TCP1 $FAX0
MSC1 SUSPENDED 1121 FAX-TCP1 $FAX0
```

Run interactive #DELTA, using the #DELTA command:

```
17>#DELTA
#DELTA 18>
```

Clear the text buffer, using the H and K #DELTA commands. (These commands are described more fully in the *TACL User's Guide*.)

```
#DELTA 18>HK
```

Display the contents of the buffer:

```
#DELTA 19>V
#DELTA 20>CTRL/Y
(.)
```

The buffer contains only the buffer pointer.

Bring the contents of *rslt* into the buffer using #DELTA's G command. The B and J commands will reset the pointer to the beginning of the buffer. Examine the contents with the V and CTRL/Y debug commands:

```
#DELTA 21>Grslt$
#DELTA 22>BJ

#DELTA 23>V
#DELTA 24><control>-Y
(.)177T RUNNING M6530-TCP1 $TB1.#D
```

Figure 2

```
?TACL MACRO
#FRAME

#PUSH rslt susp                                == declare variables
#SETMANY rslt susp,                            == initialize to 0

[#DEF lines DELTA |BODY|
    HK                                         == clear buffer
    Grslt$                                     == get PATHCOM result
    BJ                                         == start at beginning
    <:S SUSPENDED $; 0L :Xsusp$ 1L>            == loop through buffer
                                               == looking for
                                               == "SUSPENDED", then
                                               == store each suspended
    HK                                         == line into susp
                                               == clear buffer when
                                               == done
    ]

PATHCOM /OUTV rslt/ $trpm; status term *; exit
SINK [#EXTRACT rslt][#EXTRACT rslt]

SINK [#DELTA /COMMANDS lines/]                 == invoke function
#OUTPUT Suspended terminal(s):
#OUTPUTV susp                                  == display contents of
                                               == susp

#UNFRAME
```

Figure 2.
Modified INFO macro.

After one iteration of the search command, display the contents again:

```
#DELTA 25>S SUSPENDED$
#DELTA 26>V
#DELTA 27>CTRL/Y
FAX1 SUSPENDED(.) 1121 FAX-TCP1 $FAX0
```

The buffer is now pointing to the first occurrence of SUSPENDED.

For the macro to operate correctly, the entire line must be returned; use the 0L command to move the buffer pointer to the beginning of the line. Again, V and CTRL/Y are used to display the buffer.

```
#DELTA 28>0L
#DELTA 29>V
#DELTA 30>CTRL/Y
(.)FAX1 SUSPENDED 1121 FAX-TCP1$FAX0
```

The pointer was moved successfully. Next, check the code used to store the result.

```
#DELTA 31 >Xsusp$
```

Finally, exit interactive #DELTA in order to check the contents of *susp*:

```
#DELTA 37 >HK
#DELTA 38 >CTRL/Y
#DELTA 39 >CTRL/Y
39 >#OUTPUTV susp
FAX1 SUSPENDED 1121 FAX-TCP1 $FAX0
40 >
```

The #DELTA code worked correctly. It can now be incorporated into the TACL routine. (Interactive #DELTA could also be used to test the full iterative statement enclosed by < >.)

The #DELTA programmer must be aware not only of buffer contents, but also of what is happening with the result (expansion) of his #DELTA function. When a #DELTA function finishes, it will return the contents of the buffer as its expansion—potentially a multiple-line result. All functions (e.g., #OUTPUT) expect single-line arguments unless enclosed in square brackets. If the #DELTA result buffer is not needed, either an HK can be done within the #DELTA function, or a SINK can be done on its expansion. Results can be displayed or discarded as follows:

[#OUTPUT [#DELTA /COMMANDS lines/] to display results

[SINK [#DELTA /COMMANDS lines/] to discard results

Conclusion

Both the interactive TACL debugger and the interactive #DELTA functionality are straightforward and well integrated into the TACL development environment, and do not require much extra work or knowledge. They provide considerably more information than would be available from iterations of writing TACL code, running it, analyzing the output results and error messages, and rewriting the code. Both mechanisms greatly enhance TACL programming productivity and can also assist programmer-analysts in learning new TACL capabilities.

References

TACL User's Guide. Part no. 82420. Tandem Computers Incorporated.

TACL Reference Manual. Part no. 82421. Tandem Computers Incorporated.

TACL Programmer's Guide. Part no. 84111. Tandem Computers Incorporated.

Acknowledgments

The author would like to thank Roland Finlay, Dick Mahoney, and Alex Bentley for providing information about TACL.

Linda Gary Palmer is a senior account analyst in the Seattle District. Before joining Tandem in 1983, she worked in operating system development for another computer vendor. Linda has a degree in Information and Computer Science from the University of California, Irvine.

TANDEM PUBLICATIONS ORDER FORM

The *Tandem Systems Review* and the Tandem Application Monograph Series are combined in one free subscription. Use this form to subscribe, change a subscription, and order back copies.

For requests *within the U.S.*, send this form to:

Tandem Computers Incorporated
Tandem Systems Review
18922 Forge Drive, LOC 216-05
Cupertino, CA 95014

For requests *outside the U.S.*, send this form to your local Tandem sales office.

Check the appropriate box(es):

□ New subscription (# of copies desired ________)
□ Subscription change (# of copies desired ________)
□ Request for back copies. (Shipment subject to availability.)

Print your current address here:

____
COMPANY NAME

____
ADDRESS

____

____

____

____
ATTENTION

____
PHONE NUMBER (U.S.)

If your address has changed, print the old one here:

____
COMPANY NAME

____
ADDRESS

____

____

____

____
ATTENTION

____
PHONE NUMBER (U.S.)

To order back copies, write the number of copies next to the title(s) below.

NUMBER OF COPIES

Tandem Journal

________ Part No. 83930, Vol. 1, No. 1, Fall 1983

________ Part No. 83931, Vol. 2, No. 1, Winter 1984

________ Part No. 83932, Vol. 2, No. 2, Spring 1984

________ Part No. 83933, Vol. 2, No. 3, Summer 1984

Tandem Systems Review

________ Part No. 83937, Vol. 2, No. 2, June 1986

________ Part No. 83938, Vol. 2, No. 3, December 1986

________ Part No. 83939, Vol. 3, No. 1, March 1987

________ Part No. 83940, Vol. 3, No. 2, August 1987

________ Part No. 11078, Vol. 4, No. 1, February 1988

________ Part No. 13693, Vol. 4, No. 2, July 1988

Tandem Application Monograph Series

________ Part No. 83900, *Developing TMF-Protected Application Software,* March 1983, AM-005

________ Part No. 83901, *Designing a Tandem/Word Processor Interface,* March 1983, AM-006

________ Part No. 83902, *Integrating Corporate Information Systems: The Intelligent-Network Strategy,* March 1983, AM-007

________ Part No. 83903, *Application Data Base Design in a Tandem Environment,* August 1983

________ Part No. 83904, *Capacity Planning for Tandem Computer Systems,* October 1984

________ Part No. 83905, *Sociable Systems: A Look at the Tandem Corporate Network,* May 1985

________ Part No. 83906, *Transaction Processing on the Tandem NonStop Computer: Requestor/Server Structures,* January 1982, SEDS-001

________ Part No. 83907, *Designing a Network-Based Transaction-Processing System,* April 1982, SEDS-002

________ Part No. 83909, *A Multi-Function Network for Business Automation,* May 1982, SEDS-004

TANDEM EMPLOYEES: PLEASE ORDER YOUR COPIES THROUGH YOUR MARKETING LITERATURE COORDINATOR.

TANDEMCOMPUTERS

MARC BRANDIFINO
DEPT 2449
LOC NUM 50-00
NEW YORK NY DISTRICT