NonStop SQL, A Distributed, High-Performance, High-Availability Implementation of SQL

The Tandem Database Group

April 1987
revised July 1988

Abstract: NonStop SQL is an implementation of ANSI SQL on Tandem Computer Systems. It provides distributed data and distributed execution. It can run on small computers and has been benchmarked at over 200 transactions per second on a large system. Hence, it is useable in both the information center and in production environments. NonStop SQL provides high-availability through a combination of NonStop device support and transaction mechanisms. The combination of SQL semantics and a message-based distributed operating system gives a surprising result: the message savings of a relational interface pay for the extra semantics of the SQL language when compared to record-at-a-time interfaces.

This paper presents the system's design rational, and contrasts it to previous research prototypes and to other SQL implementations.

TABLE OF CONTENTS

INTRODUCTION

NonStop SQL is an implementation of ANSI SQL [ANSI]. In addition to the ease-of-use implicit in SQL, NonStop SQL is a high-performance, distributed SQL which 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 required to support hundreds of transactions per second running against hundreds of gigabytes of database.

Prior SQL implementations are marketed as information center tools or as productivity tools. Their ease-of-use is accompanied by a significant performance penalty. These vendors typically offer a second, non-SQL, system for production applications. Tandem rejected this "dual database" strategy as being too expensive to support, and too expensive and cumbersome for customers to use. A major goal was to produce a system that could be used on large and small systems and in the information center as well as for production on-line transaction processing applications.

NonStop SQL had several other design goals:

1) To be integrated with the Tandem networking and transaction processing system.
2) To provide NonStop access to data -- in case of a fault, only the affected transactions are aborted and restarted: data is never unavailable.
3) To support modular hardware growth, and as a consequence support tens of processors executing hundreds of transactions per second.
4) To allow data and execution to be distributed in a local and long-haul network.

These goals are related. Tandem's existing support for networking and transactions gave a good basis for distributed data and execution. NonStop operation is Tandem's hallmark. The challenge was to integrate the SQL language with this preexisting Tandem system architecture.

Just as importantly, some goals were excluded from the first release. There was little attempt to exploit the parallel architecture of the Tandem system to get parallelism within a transaction in the style of Teradata [Teradata]; rather, parallelism is exploited by having multiple independent transactions executing at once. The implementation did not focus on solving the heterogeneous database problem. In addition, beyond an interactive SQL interface and a report writer, not much work was devoted to user interfaces like QBE or a fourth-generation language. Rather, work focused on the SQL engine and features to help application programmers build systems in conventional ways.

Now that NonStop SQL is available, we are seriously considering projects in each of these neglected areas. The NonStop SQL design provides an excellent base for a highly parallel SQL implementation. In addition, SQL is a natural base for data sharing among heterogeneous systems, because most systems support SQL. It is also an excellent base for end-user and fourth generation languages.

AN OVERVIEW OF THE TANDEM SYSTEM

Tandem builds a single-fault tolerant, distributed system for on-line transaction processing. The system can be grown in small increments by adding processors and disks to a cluster, and clusters to a network.

Hardware Architecture

The Tandem hardware architecture consists of conventional processors each with up to 16MB of main memory and a 5MB/sec burst multiplexor io channel. The processors do not share memory. Dual 20MB/sec local networks can connect up to 16 processors. A fiber-optic local network allows four-plexed 1MB/sec interconnect of up to 224 processors (see Figure 1). A variety of long -haul public network protocols can be used to connect up to 4000 processors.

Communication and disk device controllers are dual ported and attach to two processors so that there are dual paths to the storage and communication media.

Disk modules come in two styles -- low-cost-per-actuator and low-cost-per-megabyte. These modules are packaged 2, 4, 6, or 8 to a cabinet. Typically, each disk is duplexed so that media and electronic failures of a disk will not interrupt service.

Operating System and Network

Processes are the software analog of hardware modules. They are the unit of functionality, failure, and growth. Messages are used to communicate among processes. Shared memory communication is avoided because it gives poor fault containment and because it limits the ability of processes to reside anywhere in the network. The operating system kernel provides processes, process pairs, and a reliable datagram service among nodes in the cluster [Bartlett]. A privileged layer of software uses these datagrams to provide a session-oriented message system among processes in the cluster, and transparently extends the message system to a long haul net based on leased lines, X.25, SNA, and other protocols.

Above the message system, everything looks like a process. A device is a process, a file is a process and a running program is a process. An application OPENs a process by name. Then the application operates on the object with procedure calls in conventional Cobol style. The underlying message system turns these into remote procedure calls. Typical operations are READ and WRITE, but different object types support a variety of other operations.

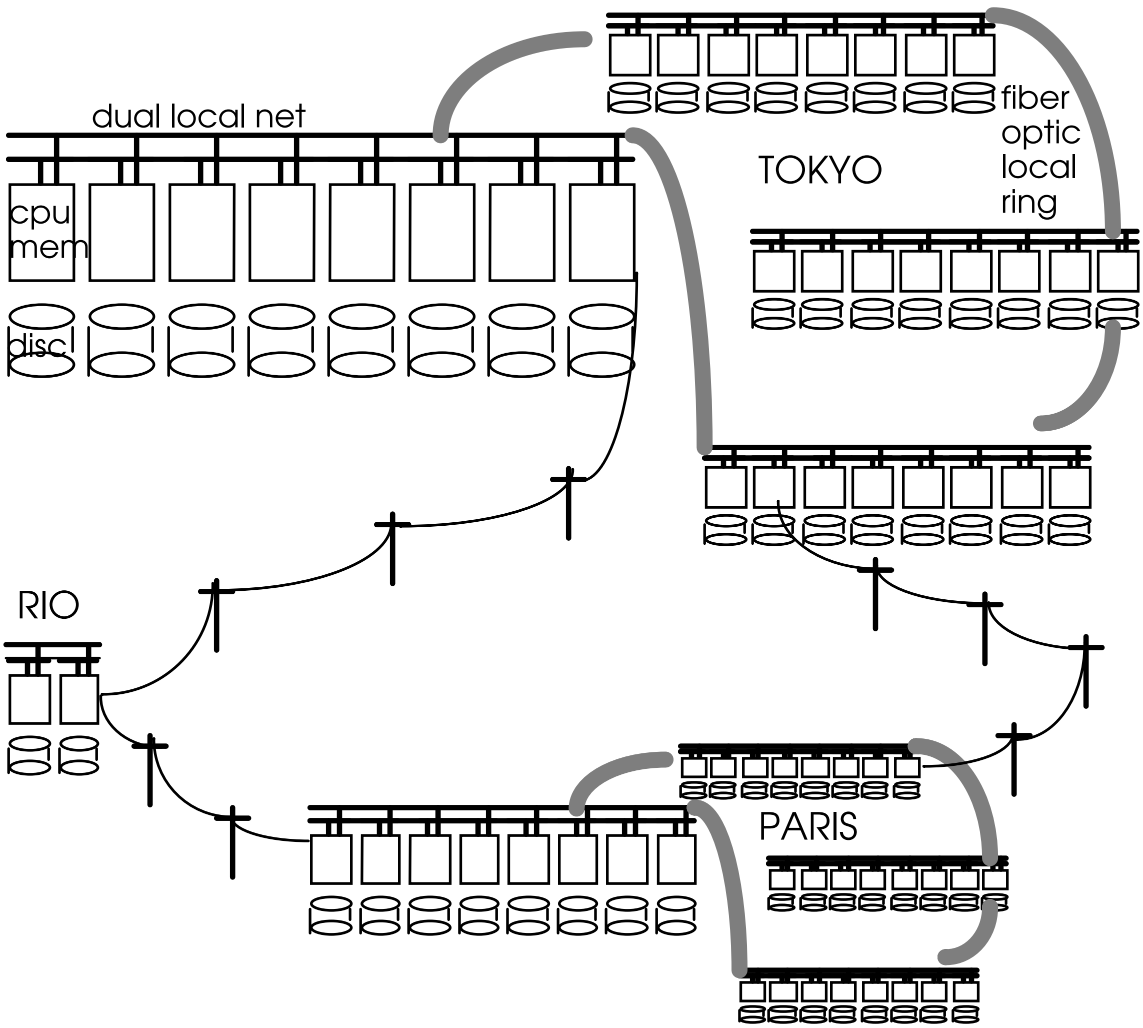

Figure 1. A schematic of a distributed Tandem system showing three sites. Two of the sites are large clusters of 32 processors and associated disks linked by a high speed local network. One site is a small two-processor system. The sites are linked via a public or leased network.

Objects are named by "site.process.directory.object". Security is checked at open (and purge, rename, secure,...) by the message system to see that the requester has access to the site, and by the process to see that the requester has access to the object. A conventional access-control list scheme is used to control security [Safeguard]. It optionally provides logging of accesses or access violations.

Data Management

The majority of applications built on Tandem systems are programmed in Cobol. Most of the data management tools, generically called Encompass, are built to support that development style. Encompass is built around the concept of an application dictionary which holds the definitions of all data structures used by programs, files, reports, and display screens [Pathmaker].

File creation may be driven off this dictionary.

A relational query product, a database browser, and an application generator are all built atop this dictionary.

The underlying file system, called Enscribe, is of particular interest here because NonStop SQL co-exists with it and has a similar design. Enscribe supports unstructured (Unix like) files which are used to store programs and text. It also supports three kinds of structured files: key-sequenced (B-tree), relative (direct access), and entry-sequenced (insert only at end). Any structured file may have multiple secondary indices (B-trees).

Enscribe files and indices may be partitioned among disks of the network based on key ranges. This horizontal partitioning is transparent to the application. The division of labor in file management is instructive (see Figure 2). Each partition of a file has a label describing the file. When the file system (client) opens the file, the disk process (server) returns this descriptor. Based on information in this descriptor, the file system then opens all related partitions of the file, and all indices on the file and their partitions. When a read or write request is presented to the file system, it uses the record key to decide which disk process can service the request. If the request involves reading via a secondary index, the file system first sends a read to the disk process managing the appropriate index and, based on the index record, sends a request to the appropriate base-file disk process. Similarly, the file system is responsible for issuing inserts and deletes on alternate indices when records are inserted, updated or deleted in the base table. The disk process is responsible for authorization, media management, locking, logging, management of file structures (B-trees, end of file, etc), and management of a main memory cache of recently used disk pages.

In addition, the disk processes provide fault tolerance by executing as NonStop process pairs which tolerate single faults of media, paths, and processors. The disk process supports a DO/UNDO/REDO protocol for transaction protection.

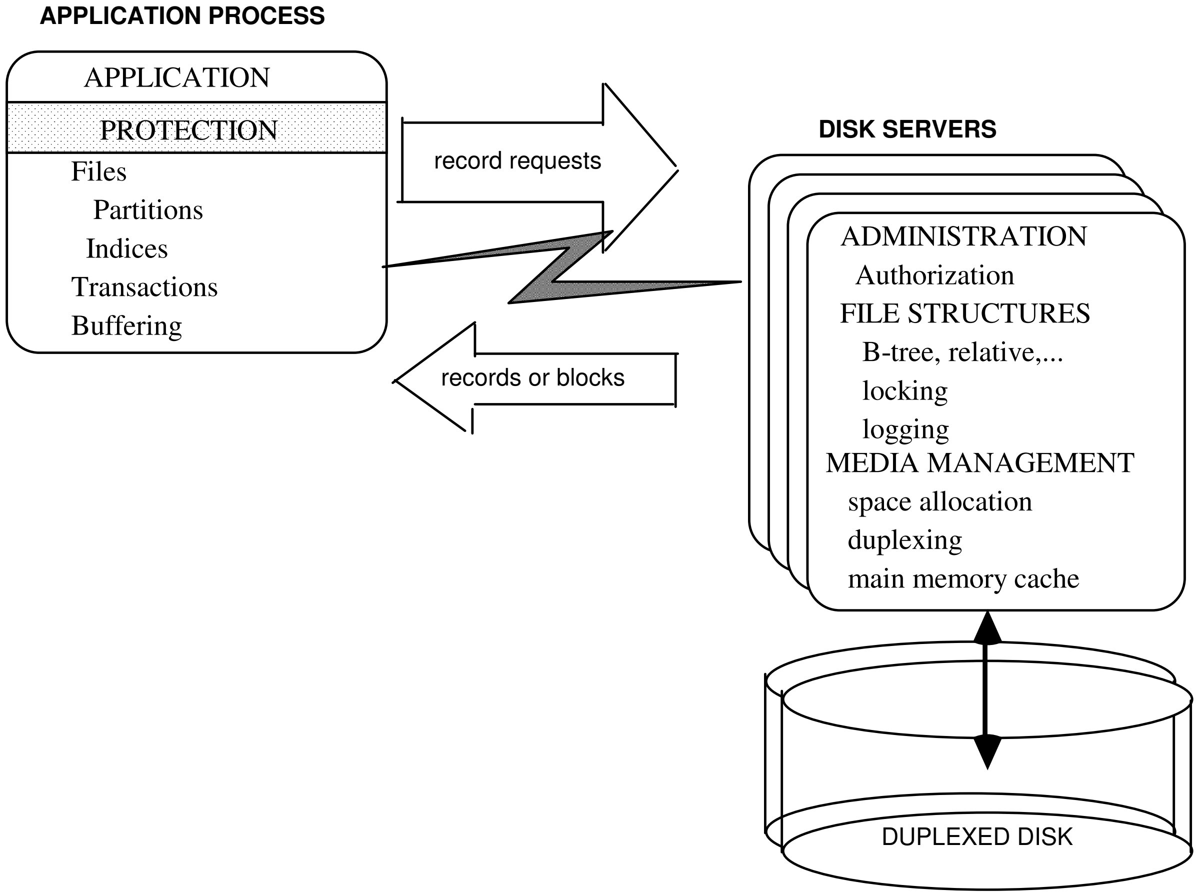

Figure 2. The division of labor in Enscribe between the application process, the file system, and the disk processes. The file system runs as a protected subsystem of the application process. Disk processes run in a processor connected to the disk they manage. In general, the file system communicates with many disk processes and the corresponding disks. This figure can be compared to Figure 6 which shows the division of labor in the NonStop SQL system.

Transaction Management

Files may be designated as audited (transaction protected), either when they are created or at a later time. Updates to audited files are automatically protected by locks belonging to the transaction and by undo and redo log records.

An application program can issue BeginTransaction, which assigns a unique transaction identifier. This identifier is automatically included in all messages sent by the application and by its servers. All updates to audited files generate log records and locks tagged with this transaction identifier. Eventually, the application calls EndTransaction, which commits the transaction if possible, or it calls AbortTransaction, which undoes the audited updates of the transaction [Borr].

The underlying mechanism provides transaction back out, distributed transactions with a commit protocol implemented with a non-blocking, grouped, two-phase, and presumed-abort protocols [Mohan], [Helland]. A single transaction log (audit trail) is maintained at each site. This audit trail provides undo, redo, and media recovery for both old (Enscribe) and new (SQL) data.

The Tandem system tolerates any single fault without interrupting service. If there is a double fault or an extended power outage, the transaction recovery system recovers data by aborting all uncommitted transactions and redoing recently committed transactions. In case of double media failure, the transaction manager supports media recovery from an archive copy of the data and the transaction log by redoing recently committed transactions. Archive dumps of the database can be captured while the database is being updated -- that is, media recovery can work from a fuzzy dump.

Why SQL?

Perhaps the most controversial decision of the NonStop SQL project was to abandon compatibility with Tandem's existing data base products and adopt an SQL interface instead. After all, Encompass was the first commercial distributed database system. It has many strong features and a loyal following.

In retrospect, the choice of SQL seems less courageous since SQL has become the standard data management interface. At the time, the rationale for adopting SQL was that the Encompass dictionary is passive and proprietary. Encompass provided a record-at-a-time interface for programmers and little data independence. Like most such systems, it was built on top of the file and security system, rather than being integrated with it. Customers were asking for an integrated and active dictionary -- one which had no "back-doors" and one which assured consistency between the dictionary and applications. In addition, customers were asking for support of views and integrity constraints. SQL provides views and a standard data definition and manipulation language. In addition, NonStop SQL provides an active, distributed, and integrated dictionary. Tandem is gradually evolving its Encompass application development environment to a dictionary based on SQL.

After settling on SQL, the build-vs-buy decision had to be made. Several software houses were willing to port their SQL system to Tandem hardware. This alternative offered a low-cost and low-risk solution. It also offered low-benefit. Tandem wanted SQL to be integrated, fault-tolerant, high-performance, and distributed. So, a second courageous decision was made: to start from scratch. Fortunately, several Tandem developers and managers had experience on other SQL implementations. This considerably reduced the risk of a new implementation.

NonStop SQL LANGUAGE FEATURES

The NonStop SQL language is based on SQL as documented in the System R papers [Astrahan], the SQL/DS manuals [SQL/DS], the DB2 manuals [DB2], and the ANSI SQL definition [ANSI]. Extensions and variations were added to support distributed data, high-performance, operational interfaces, and integration with the Tandem system. When development began, the ANSI SQL standard [ANSI] did not exist. Fortunately, only minor changes were required to achieve close compliance with the ANSI standard.

Naming

The first chore was to decide how naming, directories, and security should work. Standard SQL naming has the form "user.table". This is inadequate for a distributed system with local autonomy. In such a system, user names are qualified by site name and table names need to be more complex so that names may be created without consulting any other site [Lindsay].

NonStop SQL names objects like any other objects in the Tandem system. That naming convention is "site.process.directory.object". These names are used to name tables, indices, views, and programs. Naming of columns follows the ANSI SQL conventions. Integrity constraints are named by ANSI SQL identifiers so that diagnostic messages can explain which constraint is violated.

Considerable care was taken to make catalogs and naming automatic. The Tandem default naming works for SQL objects. Programs, tables, views, and all other system objects are named in the same way. The distinction between logical and physical schema is almost invisible (automatic) because table names and their corresponding file names are the same. The goal was that most SQL examples from Date [Date] should work without change when entered from a terminal.

Having one naming convention for the whole Tandem system simplifies learning and operating the system.

Logical Names for Location Independence

System administrators and application designers need the ability to bind a program to new tables without altering the source code. In production systems, a program is typically created and tested in a test environment and then moved to a production environment. In a distributed system a program may be duplicated at many different sites. Running a report against many instances of a generic table is another common situation. With most SQL systems, each of these situations require editing the program and changing the table names to reference the desired tables.

Some SQL systems offer synonyms, a user-specific name mapping of the form

```
  CREATE SYNONYM programmer.emp FOR test.emp
```

so that the name programmer.emp will map to test.emp. When the program is moved to production, the synonym is changed to

```
  DROP SYNONYM programer.emp
  CREATE SYNONYM programer.emp FOR production.emp
```

As an alternative, NonStop SQL offers logical names, called "defines", which allow users to rebind a program's table names at SQL compile or run time without altering the source program. *Defines* are similar in function to OS/360 DD cards, TSO ALLOCATEs, and CMS FILEDEFs. They associate a logical name with a physical name and its attributes. Defines are used for many other purposes within the Tandem system. For example, they may carry label tape attributes. The define statement is supported by all command interpreters. In addition, a process may programmatically interrogate and alter its defines. This programmatic interface differentiates defines from their counterparts in other systems.

Logical names are used as follows by NonStop SQL. Consider the example:

```
  SELECT  dept
  FROM    emp
  WHERE   empno = :empno
```

If this is developed in a test environment, then 'emp' will be bound to a particular table. When the program is moved to a production environment, it should run with the 'emp' bound to the production 'emp' table. In NonStop SQL this is done by coding:

```
  SELECT  dept
  FROM    =emp       -- the "=" implies a logical name
  WHERE   empno = :empno
```

The statement is bound to a particular table or view by issuing a define like:

```
  DEFINE  =emp,  FILE site.process.test.emp
```

After the program has been tested, it may be moved to production by simply changing the define to:

```
  DEFINE  =emp,  FILE site.process.production.emp
```

When a compiled SQL program is invoked with a different define for "=emp", the new define overrides the compile-time define and causes temporary recompilation of that statement. Process creation propagates defines so that the child process has the same naming environment (context) as its parent.

The contrast between defines and synonyms can be summarized by:

- Both provide an aliasing mechanism for data definition and data manipulation.

- Defines are supported as a standard part of the Tandem system naming mechanism and so provide a general tool that works for all files, tables, devices, and processes. Synonyms are unique to the SQL environment.

- Defines are a process attribute specified explicitly, while synonyms are a user attribute and are stored in the catalogs.

- Defines require special syntax (an "=" preceding the name) while synonyms fit within the SQL naming scheme.

Defines were supported because they are part of the Tandem naming system. Synonyms will probably also be supported someday.

Dictionary and Catalogs

The descriptive information representing the logical schema is kept in an SQL database called the dictionary. 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. This design rule, called local autonomy, implies that the dictionary be partitioned into catalogs -- each catalog acting as a local dictionary for tables at that site. The transaction mechanism is used to protect updates to catalogs and so maintain the consistency among catalogs at different sites. A simplified diagram of the catalogs is shown in Figure 3.

A site can have a single catalog, a catalog per project or application, or a catalog per individual. It all depends on the organization's management. The

```
   CREATE CATALOG <directory>
```

command creates the catalog tables described by Figure 3 in the designated directory. The command also registers the catalog in a site directory so that all catalogs may easily be found.

When an SQL object is created, it is registered in the dictionary. Creation commands can explicitly specify a catalog; otherwise, the default catalog of the executing process is used. For example, the table creation command:

```
  CREATE TABLE site1.disk1.dir.emp (    emp_no INTEGER,
                                        dep_no INTEGER,
                                     PRIMARY KEY emp_no )
        PARTITION  site2.disk2.dir.emp FIRST KEY 10000,
        CATALOG site2.disk3.dir4;
```

creates a table partitioned across two sites of the network. Local autonomy requires that the definition be replicated in catalogs at each participating site. So in the example above, the second partition is registered in a catalog at site2. Executing this create statement makes entries in the process' default local catalog and also in the explicit remote catalog named site2.disk3.dir4. Compiled versions of the table descriptions are stored in the file labels as part of the file manager's disk directory. All necessary 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 SQL compile time. This point will be amplified later. A program using this table from site3 will be registered in a catalog at site3. The program's usage of the table will be recorded in the USAGES table at site1 and site3. The catalog tables shown in Figure 3 have the following semantics

- TABLES has the name, owner, and security of each table or view.
- VIEWS has the view definition stored as a character string.
- COLUMNS has a description of each column of each table or view.
- BASETABS has information that applies to tables but not to views.
- CONSTRAINTS has the text of integrity constraints attached to tables.
- FILES describes the physical files that support tables and indices.
- PARTITIONS gives name and catalog of partitions of any table in this catalog.
- INDEXES describes an index file, its key and whether it is unique.
- KEYS describes the permutation of the index columns to the base table columns.
- PROGRAMS describes programs which contain SQL. It tells when each program was last compiled, and whether the compilation has since been invalidated by changes to the database schema.
- COMMENTS stores user provided descriptive text about objects.
- USAGES is a concordance which shows who uses what. It is used to generate where-used reports and to direct program invalidation when a relevant table or view definition changes, and to relate indices and views to their base tables.

An additional table not shown in the figure gives the current catalog's version number.

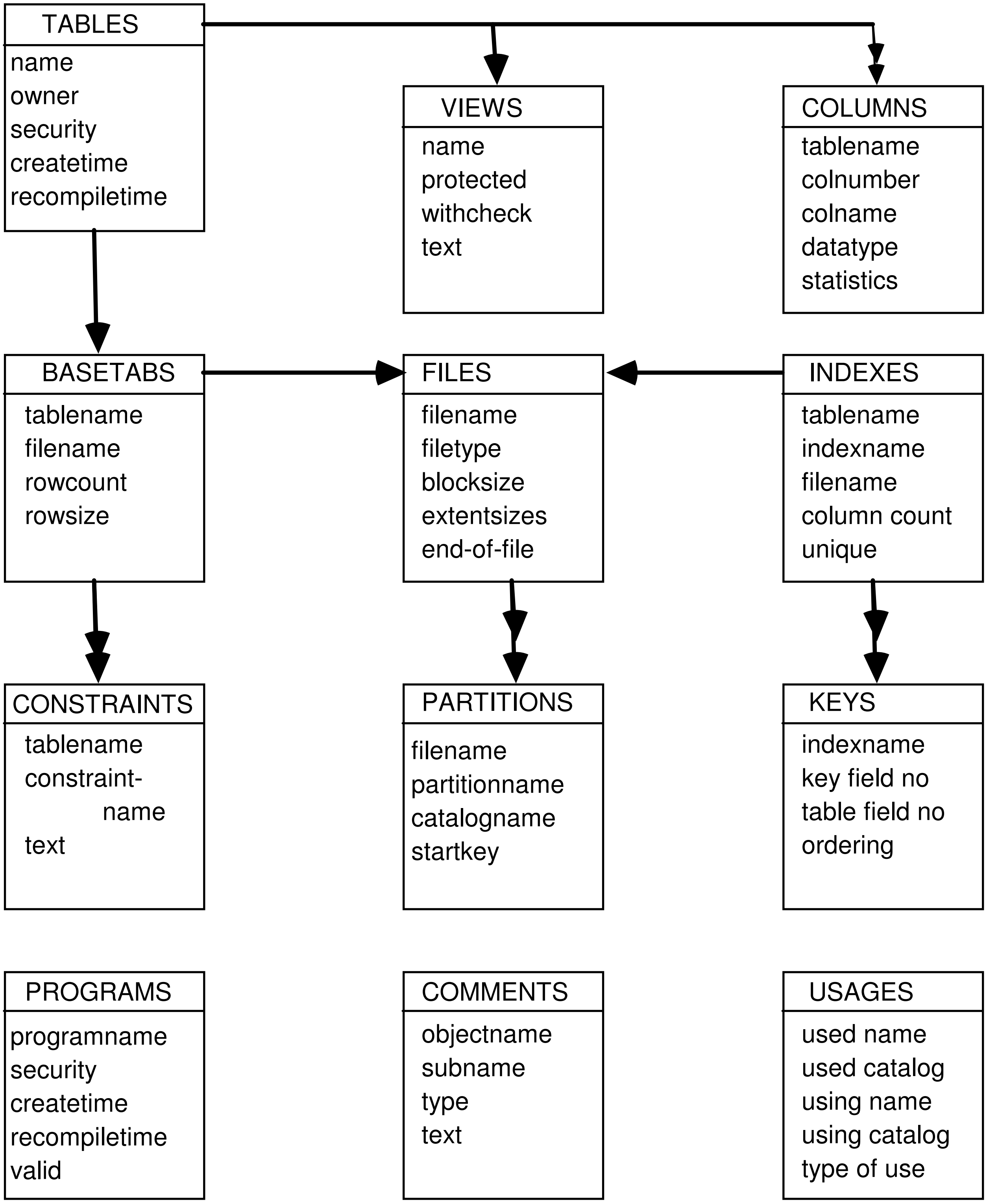

Figure 3. A schematic of the catalog tables.

Unifying Logical and Physical DDL

The SQL language integrates data definition and data manipulation in one language. This is a nice simplification when compared to the DBTG separation of data definition from data manipulation [DBTG]. NonStop SQL continues this simplification by merging the definition of logical schema (tables) with the definition of physical schema (files). It hides the logical-physical distinction from the user, but the underlying implementation makes a clear distinction between logical and physical attributes. To give a specific example, the definition of the EMP table created in the previous section combined logical design information, primary keys, columns,..., with some physical design information, in that case partitioning of the table into two physical files. All this information is stored in the dictionary. In addition, the table content is represented by a physical file named EMP partitioned across two nodes.

In NonStop SQL, when a table is created the user can specify its physical attributes. Unspecified attributes are defaulted. To be compatible with ANSI SQL, all physical attributes have defaults. For example the default organization is indexed, and the default block size is 4k bytes. The naive user can be completely unaware of the logical-physical distinction. In addition, this distinction is only visible at data definition time. Data manipulation statements are completely insulated from the physical attributes of files; they operate on logical tables.

Identifying table names with file names eliminates the concept of DBSPACE [DB2] or individual DATABASEs in Ingres [Ingres] or Informix [Informix].

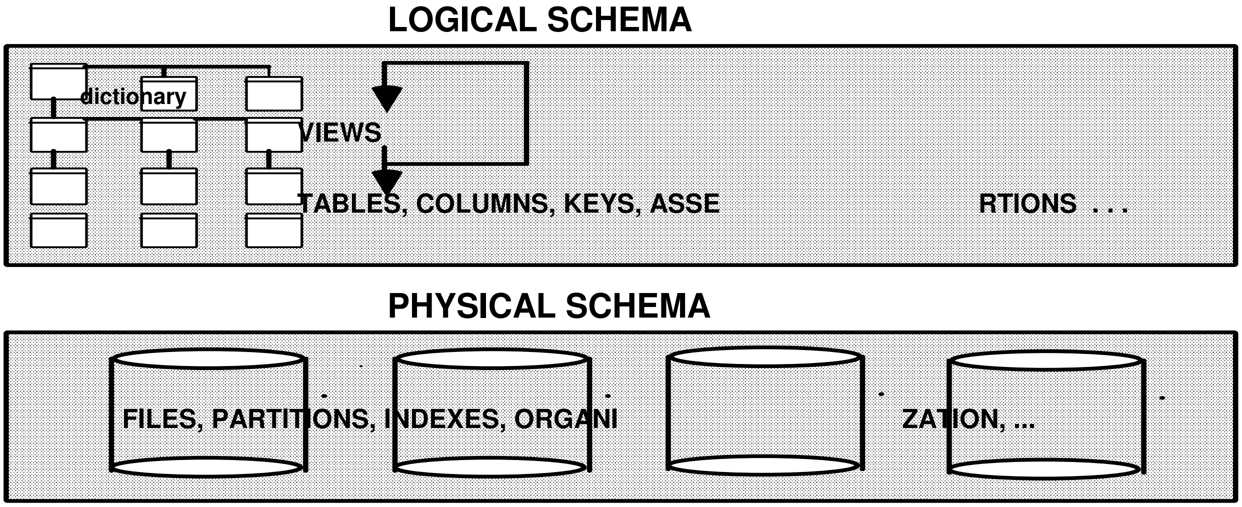
LOGICAL SCHEMA
dictionary
VIEWS
TABLES, COLUMNS, KEYS, ASSE RTIONS . . .
PHYSICAL SCHEMA
FILES, PARTITIONS, INDEXES, ORGANI ZATION, ...

Figure 4. NonStop SQL implements a two schema architecture. The logical

schema is described by the dictionary. The physical schema is described by the dictionary and the file labels.

Logical Table Attributes

NonStop SQL implements all the logical attributes of Level 2 ANSI SQL DDL [ANSI] with the exception of the `user` data type, and supports the time and interval types introduced by SQL2. The initial omission of nulls was based on Date's compelling arguments against nulls and for defaults [Date]; but, now that nulls are part of the standard, NonStop SQL will support them in release 2. Defaults are implemented as specified in ANSI SQL. Each type has a default value which the user can override.

Users may add integrity constraints to a table. Constraints are named, single variable queries. The constraint is validated against the table when the constraint is first defined. Thereafter any insert or update which violates the constraint will be rejected. Constraints are enforced by the disk process (file server). This removes many integrity checks from the application program. 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> [ANSI].

The ANSI SQL numeric types of INTEGER, NUMERIC, and DECIMAL up to 64 bits are supported. Fixed and variable length character strings are supported.

Records cannot span blocks and so are limited to 4K bytes in the initial release. Indices involving multiple fields, ascending and descending, may be defined on tables.

Comments on all objects are supported. Referential integrity will be implemented in a later release.

Physical Table Attributes

Tables and indices can be horizontally partitioned in the network. Each partition resides on a particular disk volume. The partitioning criteria are user-specified low-key values. New partitions can be added to the end of a table. Although partitioning tables in a network is much discussed, NonStop SQL is the first SQL system to offer partitioning of tables across multiple sites of a network with full transaction recovery.

For performance, various file organizations are supported. The standard file organization is a B-tree based on the table's primary key. Sophisticated users can specify a relative file (directly addressable records), or an entry sequence file (insert only at the end, records cannot grow once inserted). If the user does not specify an organization or key when the table is created, then the default is a key sequenced table with the GMT julian timestamp of each insert used as the system generated key record key. System generated keys for relative, entry sequenced, and default files appear in the table as a column named SYSKEY.

By default, tables are covered by transaction protection (logging and locking), but this is a user option. Each table has a default lock granularity which can be set or changed. The default is record granularity, but the designer can request generic locking on a key range. As mentioned later, these defaults can be overridden by a program or statement.

Views

As the R* designers pointed out, implementing views in a distributed system with local autonomy is not straightforward. The ideas of "shorthand view" and "protection view" were borrowed from the R* project [Williams]. On closer inspection, only the names were borrowed. In NonStop SQL, any view which is a simple projection and selection of a single table may be declared to be a protection view. A protection view inherits the organization, indices and partitioning of its underlying table. In addition, protection views are updateable. To simplify authorization, only the table owner can define (and own) a protection view. The accessor is authorized to the view.

Protection views are implemented by the disc servers and consequently really do provide protection -- one can grant access to the view without granting access to the underlying tables. This differs from other implementations which place a security mechanism on top of the operating system protection mechanism or even those like R* which implemented security in the client node's data manager. Node autonomy requires that security be implemented at the node which owns the data. There are no "back-doors" to this protection mechanism. The disc server knows about tables and views and there is no "lower-level" access to the "underlying" files. This is one of the benefits of integrating SQL with the operating system.

By contrast, a shorthand view is simply a macro definition. When the view name is invoked, the view definition is substituted and the authorization and query optimization is as though the macro body had been entered directly. So 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 via projection, selection, joins and aggregates; but, they do not provide protection and are not updateable. They can be used to denormalize the database or subset it in convenient ways.

Data Manipulation

NonStop SQL supports Level 2 ANSI DML with the exception of partial backout of failed set operations. The data manipulation interface was extended in the areas of naming, concurrency control, and dynamic SQL. As already mentioned, naming was extended to work in the Tandem style of network naming and logical table names. INSERT, SELECT, UPDATE, DELETE, subqueries, aggregates, correlation, group by, functions, and cursors, are provided as in the standard. Outer join and time arithmetic are supported as specified by SQL2 [ANSI-2]. Dynamic SQL is supported in the style of SQL2, SQL/DS, or DB2 with some innovations to handle description of input and output variables.. The major DML innovations were in the areas of locking and consistency.

As in most other SQL implementations, NonStop SQL requires cursors used for update to be declared `FOR UPDATE ON <field list>`. This allows SQL to avoid the Halloween Problem (see page 35). In addition it tells the SQL executor to get an exclusive lock and avoid the most common form of deadlock -- two processes read and then try to update a hotspot. Unfortunately, this is not part of the SQL standard.

All update operations on transactional files automatically acquire exclusive locks held to end of transaction (degree 1 consistency is automatic). For selections, the programmer is given the option of dirty data (`BROWSE ACCESS`), cursor stability (`STABLE ACCESS`), repeatable reads (`REPEATABLE ACCESS`). These correspond to degree 1, 2, and 3 consistency [Gray]. In addition, the statement can specify the desired lock granularity and mode. These options are an extension to the SELECT statement syntax. Cursor stability is the default.

Rather than clutter SQL syntax with Tandem extensions, the `CONTROL TABLE` verb was added. This verb allows the programmer/user to modify some of the defaults associated with all tables, with a particular table, or with a particular statement. For example, to change the timeout to one second on all requests, the user may execute:

```
     CONTROL TABLE * TIMEOUT 1 SECOND;
```

When encountered by the compiler, this statement affects all subsequent statements until another `CONTROL TABLE` is encountered. When executed dynamically, this statement overrides dynamic compile-time settings and timeout values for all subsequent dynamic SQL statements.

In the first release, the `CONTROL TABLE` command allows a program to specify the lock granularity, lock protocol, and style of lock waits. Later releases will have additional options.

This design is more general than the SQL2 standard which associates control with the transaction or program rather than with the statement. Finer granularity control on a statement-by-statement basis or table basis is essential for tuning high-performance applications.

Degree 3 consistency was sacrificed in one case to avoid hotspots. When inserting at the end of an entry sequenced or relative file, NonStop SQL locks the inserted record but does not lock the end-of-file. This gives only degree 2 consistency [Gray], since insert order may differ from commit order unless a table lock is acquired. Locking the end of the table (file) for the duration of a transaction is equivalent to a table lock for sequential files -- a well known bottleneck in high-performance systems. IMS/FastPath solves this problem by deferring sequential inserts until transaction commit [Gawlick]. The FastPath design has more complex semantics and also sacrifices degree 3 consistency since the records do not appear in the file when they are inserted.

Transaction Management and Concurrency Control

Tandem's Encompass data management system provides a transaction mechanism that includes transaction rollback and distributed transactions. Nonstop SQL is integrated with this transaction manager [Borr]. 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. Having one transaction log per site eases system management when compared with the non-integrated log-per-subsystem of other designs.

When a table is created, it is audited by default. Updates to audited tables are automatically protected by locks belonging to the transaction and by undo and redo log records. The user may declare a table to be non-audited by altering the audit attribute.

Any program locking or updating an audited table must be part of a transaction (must do a `BEGIN WORK` or be servicing a request from some process that did a `BEGIN WORK`). As explained earlier, the operating system message kernel manages the propagation of transaction identifiers as part of the requester-server (remote procedure call) flow within the transaction.

The locking features include file, generic, and record 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 sixty seconds.

The CONTROL TABLE verb allows the user to specify the handling of lock waits. The user may request "bounce" locks which never wait for a busy resource; rather, they return an exception. In addition, NonStop SQL supports read-through locks (BROWSE ACCESS), which allow reading dirty data (uncommitted updates). The CONTROL TABLE command allows a program to alter these attributes on a table-by-table or statement-by-statement basis.

Non-audited objects complicated the design, since non-transactional locking is explicit and has complex semantics. In particular, the state of locks and cursors after a COMMIT WORK verb is difficult to explain. NonStop SQL defined some rules which seem consistent. A FREE RESOURCES verb was introduced to free locks and cursors on non-audited tables. The COMMIT WORK and ROLLBACK WORK verbs implicitly issue a FREE RESOURCES verb. For simplicity, design experts recommend against using non-audited tables for anything but temporary results.

Non-audited tables are useful for batch applications using the old-master new-master recovery protocol. A table may be made temporarily un-audited during a batch run. Non-audited tables are used as scratch files for sort and the query evaluator. In addition, to reduce log activity and the consequent large log files, index creation and table load run non-audited and then alter the table to be audited.

Local Autonomy

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

Data definition operations pose more difficult problems. If a table is partitioned among several nodes of a network, then dropping the table requires work at each of those nodes -- both updates to the catalogs and deletion of the files. 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 then the DDL operation is denied. This violates local autonomy. There are no attractive alternatives here. NonStop SQL takes the following approach. The local owner can DISCONNECT a table to break all linkages of the table with unavailable sites. This is a unilateral operation.

Thereafter, the user can operate on the table without complaint. At a later time the user can run a utility which attempts to CONNECT the pieces of a disconnected table back together.

Conversational Interface

Almost all SQL systems come with a program which executes any SQL statement entered from a terminal. This is an easy program to write since most SQL implementations provide dynamic SQL: a facility to PREPARE, DESCRIBE and EXECUTE any SQL statement (much like EVAL in LISP).

Systems differ in the bells and whistles they provide with this basic facility. The NonStop SQL conversational interface (SQLCI) includes extensive documentation via on-line help text, online diagnostic messages, and an iterative report writer.

The NonStop SQL manuals are written so that they may be accessed by the conversational interface. The user can ask for help on any topic in the manual or for explanation of an error message. For example, HELP CREATE TABLE will display the syntax, semantics, and examples from the manual. This imbedded documentation is very convenient.

The report writer features included headings, footings, breaks, data display formats for numbers and dates, windowing, margins, folding, and if-then-else controls on data display. Report writer commands are separated from the SQL commands. The answer set is defined with SQL and a report defined with the report writer syntax, then the report is displayed. The report definition is iterative; the report format can be altered and then the report regenerated.

Dynamic SQL can be exercised from the conversational interface. Parameters can be included in SQL statements. Statements can be PREPARED and EXECUTED. Parameters can be varied after the prepare and before the execution of the prepared (compiled) statement.

The user can define transaction boundaries with BEGIN WORK and COMMIT WORK or ROLLBACK WORK. If no transaction is specified, the conversational interface starts and ends one for each SQL statement.

In addition, the conversational interface can execute program scripts, gives complete statistics on the cost (cpu, records examined, records affected, messages, message bytes, lock waits, and IO) of queries, produces detailed diagnostic messages in case of error, EXPLAINs the query execution plan chosen by the SQL compiler, and has interfaces to the following utilities:

- Convert to and from "old" Enscribe files.

- Duplicate and copy sets of objects
- Display table definitions
- Describe sets of files and tables
- Update the statistics for a table
- Load and unload data for interchange or reorganization
- Verify and correct dictionary consistency
- Generate "where-used" reports on programs, views, and tables
- Text editor

Last but not least, the interface is always listening to the terminal so the user can interrupt the query execution at any time and cancel the transaction. Surprisingly, this is not typical for SQL conversational interfaces. It certainly caused some trouble. A dedicated process was introduced just to listen to the terminal.

Programmatic SQL

NonStop SQL had four goals: ease-of-use, distribution, high availability, and high-performance. Since the SQL language is non-procedural, the key to high performance is good compilation of the SQL statements and efficient communication of data between programs written in Cobol85, Pascal, C, or Tal and the SQL database system.

As is standard with SQL systems [ANSI-2], SQL statements are embedded in the host language, bracketed by `EXEC SQL` and `END-EXEC` keywords. An SQL preprocessor scans the program text and produces a host language program (Cobol) with the SQL statements replaced by calls to the SQL executor. In addition, the preprocessor produces a file containing the SQL statements. The Cobol85 program is compiled by the Cobol85 compiler to produce an object program. The SQL compiler transforms SQL statements to a set of execution plans, one for each SQL statement in the source program (see Figure 5A).

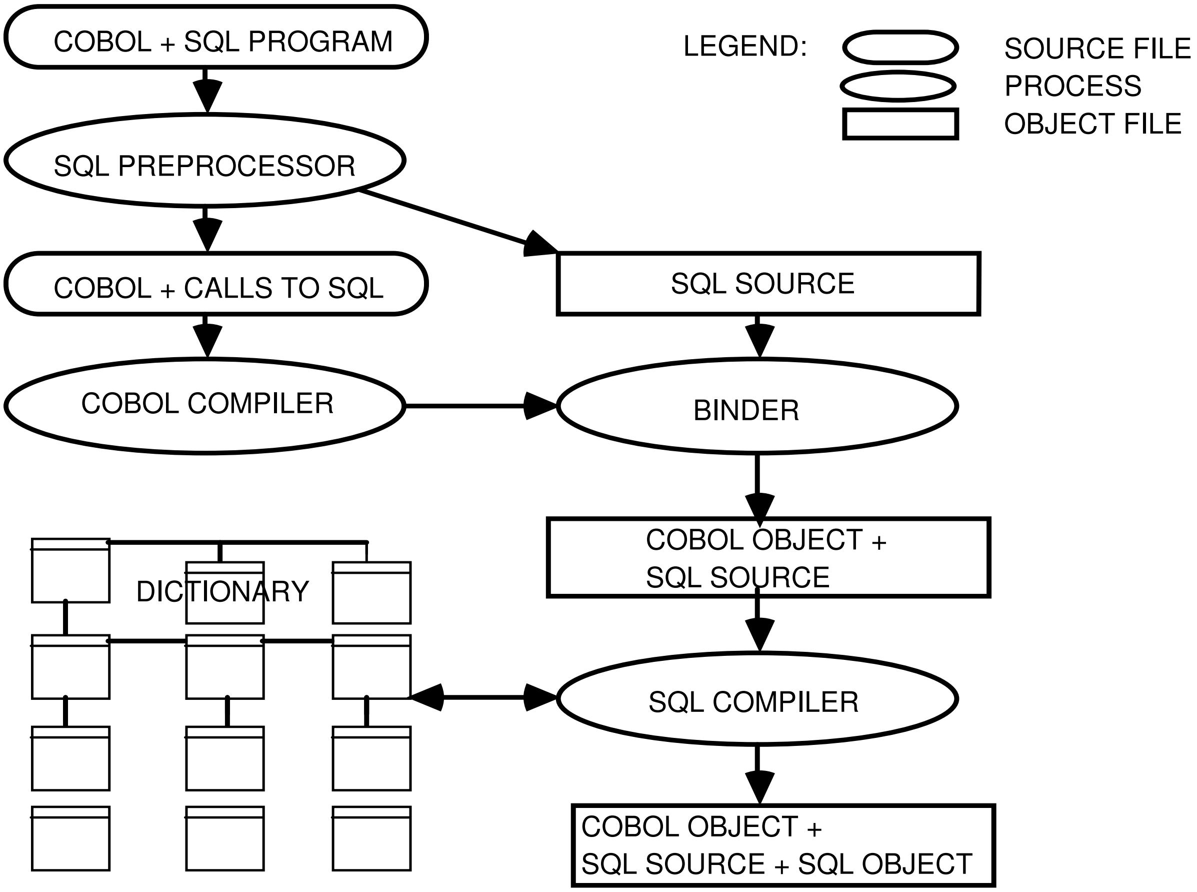

Figure 5A: Compilation of a Cobol85 plus SQL program. Cobol85 and SQL source is separated by SQL preprocessor which produces a Cobol85 source program and a object file containing the SQL source. The Cobol85 compiler feeds the object program to the binder. The binder combines this with the SQL source sections. The SQL compiler reads these source sections and the SQL dictionary to produce an SQL object program which is included in the "whole" object file and registered in a catalog. This object file is executable by the hardware and SQL executor.

Integrating SQL With Host Language Compilers

The C, Pascal, and TAL compilers were modified to directly understand NonStop SQL and produce SQL source sections in the object file they produce. They run as co-routines with the SQL compiler. This integrated design eliminates the preprocessor step and the consequent multiple listings and files. This integration gives rise to figure 5B which can be compared to Figure 5A.

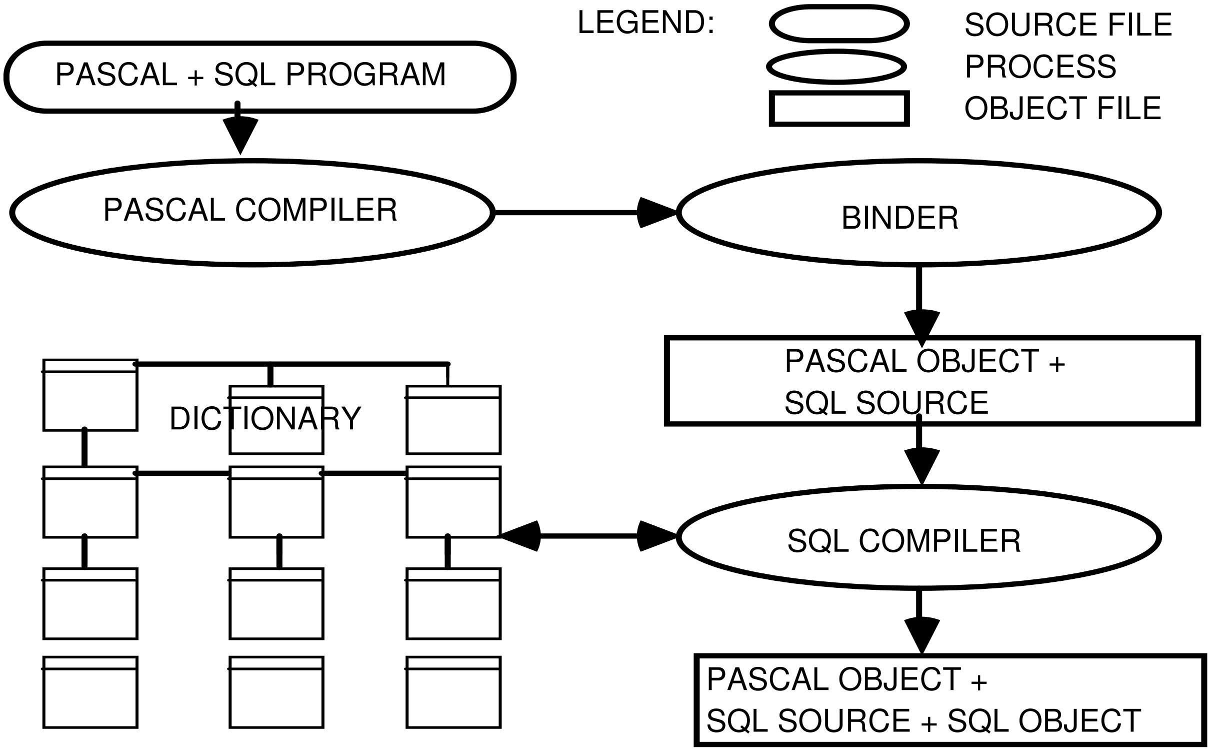

Figure 5B. The Pascal, C, and Tal compilers directly support the NonStop SQL language without need for a preprocessor. The consequent data flow is shown here in contrast to figure 5A.

Host Language Features

NonStop SQL's programmatic interface has many features to ease programming, including comprehensive diagnostics imbedded 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, and generation of trace information so that application programmers can track errors back to source language statements.

In addition, it supports separate compilation. That is, a cursor may be defined in one compilation and used in another. Program "B" can use cursor "C" of program "A" by referring to cursor "A.C".

Integrating SQL Programs With Object Programs

Tandem's implementation of program compilation is similar to the original System R implementation [Astrahan]. An innovation lies in the binding of the SQL source and object programs with the Cobol85 (or other host language) object modules. The resulting object program is a single object which can be moved, copied, secured, archived, or purged without having to manipulate one or more separate "access modules". By contrast, most other SQL systems store the SQL program in the catalogs separate from the object program and unavailable to user commands other than `DROP PROGRAM`.

The Tandem binder (aka link editor) was modified to support SQL source program sections and SQL object sections and the dependency 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 various compilations to produce an executable object program file. The SQL preprocessor produces an SQL source section for each program unit which the Cobol85 compiler tells the binder to include as a section of the Cobol85 object program. The user can bind several object programs together and present the result to the SQL compiler. The binder automatically includes the SQL source sections used by the object programs into a new object file. The SQL compiler takes any object program as input, extracts all the SQL source sections from the object program, and chooses a plan for each statement. The combined plans are stored in an SQL object section of the object program (see Figure 5A).

If an SQL statement needs to be recompiled, the SQL compiler reads it from the SQL source section of the executing object program file.

This design allows programs to be archived and moved without accessing the SQL source. It greatly simplifies the management of SQL programs. The recurring theme here is that close integration of SQL with standard system tools has considerable benefits in simplicity and functionality.

Static and Dynamic Compilation

An SQL program must be compiled and registered in a catalog before it may be run. This step is called static compilation and is part of the macro step of SQL + Cobol85 compilation.

If the program is moved or renamed, the new program will not be registered in a catalog and so it must be statically SQL compiled before it may be used.

Once compiled, an SQL program will be automatically recompiled when the database changes or the plans become invalid. Adding or dropping an index to a table, updating statistics on a table, adding columns to a table or changing some attributes of the table are examples of changes which can invalidate a program.

If a program is invoked with logical names that override the compile time logical names, it will be recompiled. In addition, if a needed access path is inaccessible because the network or media is down, then the program will be recompiled to work with the available data.

As the name suggests, automatic recompilation is transparent to the application program. It happens when the invalid statement is first invoked. A statement may become invalid during the program execution. If so, the statement is recompiled on the fly as part of the user's transaction and then executed with the new plan.

Based on user experience, this design was changed as follows:

Disable Automatic Compilation: If an index is unavailable, the SQL executor will recompile the statement and pick a plan not using the index. There may be no reasonable plan in this case (sequential scan of a terabyte is not reasonable). Users can *disable* automatic recompilation of a program. In this case the program will get an error if the plan is invalid.

Compilation on Demand: Some programs are generic, they run against a different database on each invocation. Defines are used to remap the table names on each invocation. Tandem's performance monitor which analyzes a performance measurement gives a classic example of this -- each measurement is a different set of tables. When such a program is invoked, the default is to recompile all its SQL statements. Since the performance monitor has about one thousand SQL statements, it has a lengthy and expensive startup. If most invocations of the program involve only a small fraction of the SQL statements, much of this SQL compilation effort is wasted. So there is an option to compile invalid SQL statements *on demand* (when used), rather than recompiling all of them at process initiation.

Run Time Statistics and Error Reporting

The NonStop SQL programmatic interface has two control blocks which feed information back to the application program. Detailed diagnostics are provided in the SQL Communications Area (SQLCA) in case of error. Any SQL error is reported by an error code along with up to seven diagnostic messages. A routine is provided to format these messages in a specified national language and display them to a diagnostic stream or to a buffer.

In addition, the SQL executor returns detailed statistics in the SQL statistics area (SQLSA). These include a table-by-table count of records used, records examined, disk reads, messages, message bytes, waits, time-outs, escalations, and busy times. Designers can either instrument their programs by reading these counters or they can use the operating system's measurement facility [Measure] to read these SQL counters along with other system counters.

IMPLEMENTATION

The NonStop SQL system and its utilities are about .7 million lines of code implemented by up to 25 developers over 3 years. This was the sixth SQL implementation for one developer. Five others had done it at least once before. Many of the developers had implemented other relational systems. We tried not to repeat our earlier mistakes.

Implementing SQL itself took only a few people; most of the effort went into integration with the rest of the system, and into utilities. The quality assurance, performance assurance, documentation, marketing, education, and beta test efforts involved many more people in the product.

The system is integrated with the operating system. When the system is up, NonStop SQL is up. One does not bring up an SQL database; it is just there. This contrasts with most other designs. In addition, because the operating system and SQL authorization are integrated, there is no "logon" to SQL; one logs on to the system. Once a user logs on to the system, the entire network provides a single system image.

Users can access SQL databases from a conversational interface or they can write Cobol85 plus SQL programs to access the data. These programs can be installed as servers in a Pathway transaction processing system to provide a high-performance application [Pathway].

Figure 6 shows the structure of an executing application program. The program has a code segment and a data segment (2^{32} bytes). The program's SQL statements invoke the SQL executor, a set of library routines which run in the application's environment.

The executor has its own data space. The executor manages query execution by managing logical names, collecting records from various tables via the file system, combining them, and returning the results to host language variables in the user program. It calls the file system with single-variable requests. The executor sends data definition statements to a separate process (not shown in Figure 6) which manages the catalogs. The catalog manager is implemented as a separate process for authorization reasons -- only the catalog manager process can write catalog tables.

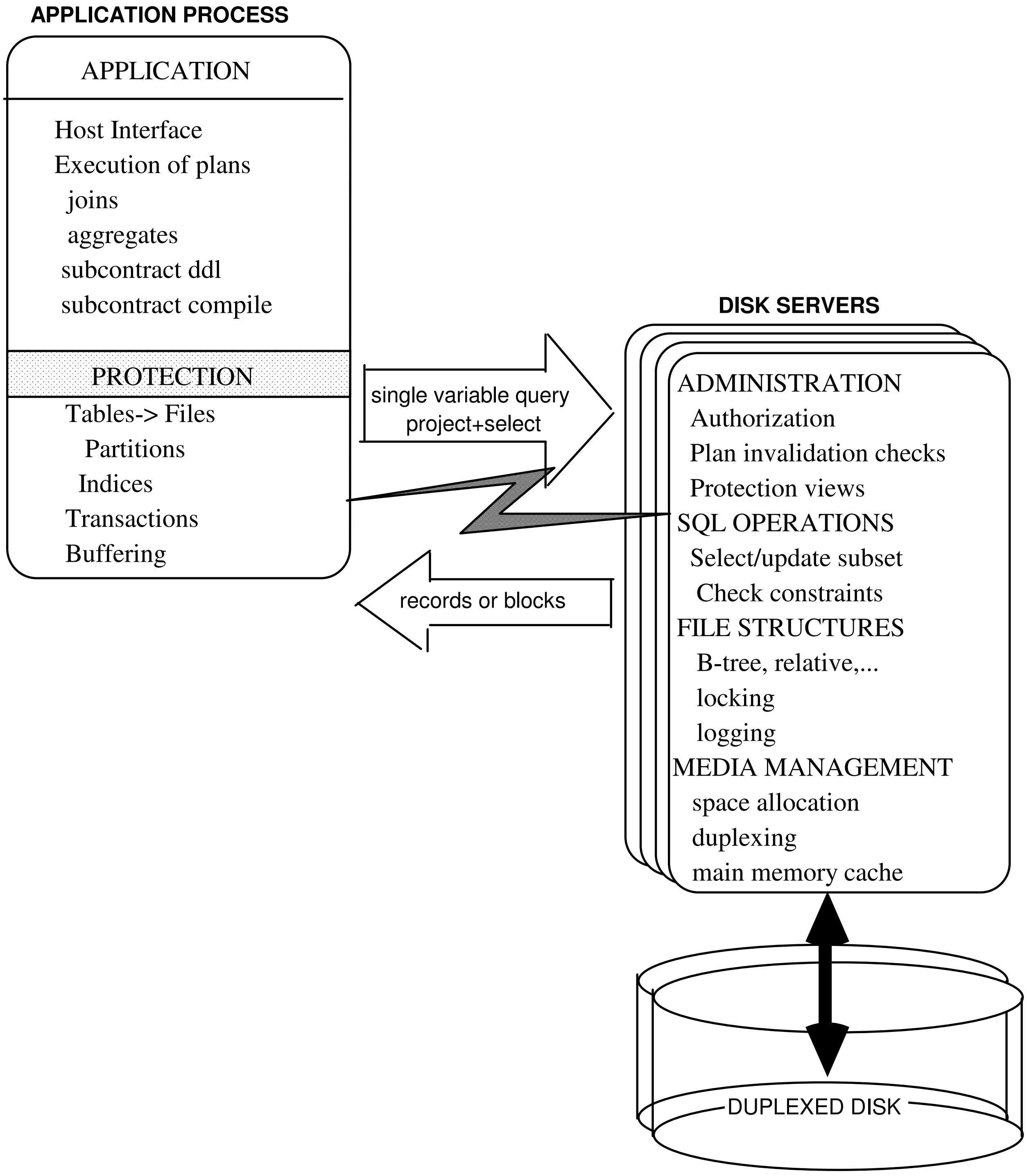

Figure 6a. The division of labor within a compiled and executing Cobol plus SQL program. The application calls the SQL executor which handles the SQL plans and subcontracts single variable queries to the files system and disc process. The file system manages the physical schema. The disc process manages subsets and integrity constraints on table fragments (partitions). See also figure 6b.

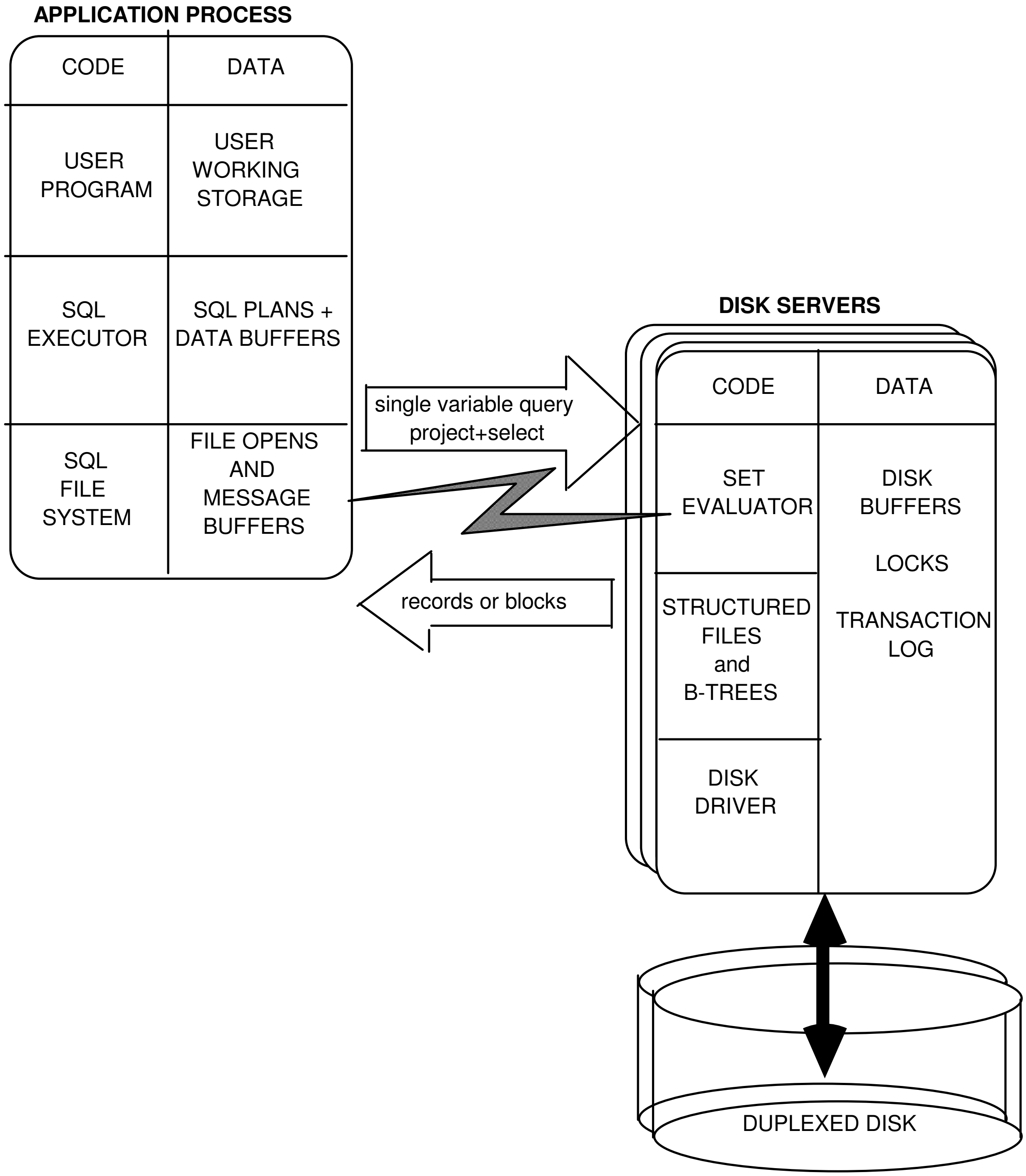

Figure 6b. The structure of a compiled and executing program. The application calls the SQL executor which in turn calls the file system. The file system sends single-variable query requests to disk processes. The disk process does projections and selections on tables and protection views to produce a record subset. This subset is returned to the file system and executor or is updated or deleted by the disk process.

The file system manages the physical schema. It handles opens of files and indices, partitioning, and sends requests to appropriate disk processes, and buffers the replies. When a table is updated, the file system manages the updates to the base file and to all the secondary indices on the file. When a record is retrieved via an index, the file system first looks in the index and then in the base file. This is because the index may well be on a disk separate from the base table, so the disk process cannot do index maintenance in general. If a retrieval can be entirely satisfied by the index, the base table is not accessed -- simple selects, Palermo's semi-joins [Palermo], and minimum values are typical uses of this feature.

Each disk volume is managed by a set of disk processes. These processes have a common request queue and a shared buffer pool which they coordinate via semaphores. The disk processes implement file fragments. They manage disk space, access paths, locks, log records, and a main memory buffer pool of recently used blocks. A disk process operates on the single-variable query, scanning the database (usually via the primary index) to find records which satisfy the selection expression. Once the records are found, the disk process either operates on the records (update expression or delete), or the projected records are returned to the file system. If the request is a very long one (more than 10 ios), the disk process returns to the file system asking it to continue the request by reissuing it. This prevents the processes from being monopolized by a particular requester if other work is queued.

The file system is one protection domain (it is privileged). Each disk process is a separate protection domain (a process). Each disk process authorizes the application process to the table when the file system sends the file open request. An open to a protection view is authorized by the disk process, which checks that the requester has the needed authority to the view.

The SQL Compiler

The NonStop SQL compiler, which includes an optimizer, uses the standard tricks of a SQL query compiler. It picks an SQL statement execution plan which minimizes a cost function, combining io cost, cpu cost, and message cost. Single-variable queries are subcontracted to the disk processes storing the data partitions, but otherwise the entire query is executed by the application program process. In particular, all joins are computed by moving the projected and selected fragments to the SQL executor in the application process. The compiler has two join strategies (nested and merge). It sorts a table if no appropriate access path is available for a join.

NonStop SQL's approach to distributed query execution is controversial. Other systems, notably R* [Williams], devote considerable effort to optimizing the distributed execution of join queries. We believe that these optimizations make little sense in a local network where communication is fast and cheap, and that joins in a long-haul network are infeasible for tables of credible size. Consequently, not much effort was devoted to this problem. When the high priority features have been implemented, more effort will be devoted to the transcontinental join problem.

The compiler composes shorthand views and treats protection views just like ordinary tables. It handles logical names (does name binding), and resolves partially qualified names within the process' working directory.

In the style of DB2, the compiler generates a set of control blocks which are interpreted by the SQL executor, rather than generating machine language code. It was felt that the control block interface was more maintainable and that the only critical path of the SQL executor is expression evaluation, which is highly optimized.

As explained earlier, the `CONTROL TABLE` statement allows the sophisticated user to give the SQL compiler hints concerning lock granularity, and lock protocol. The verb applies to both static and dynamic SQL. With time, many other hints will likely be added to the `CONTROL TABLE` directive to allow sophisticated users to tune their SQL statements.

The compiler invokes the parallel sorter, FastSort, using multiple processors if the user has provided defines requesting a parallel sort [FastSort]. FastSort gets considerable speedup on large files (over 1MB) by using multiple processors and disks.

The unique features of the compiler are detailed below.

Subcontracting Single-Variable Queries to Disk Processes

The NonStop SQL disk process will execute any selection, update, or delete involving a single-variable query -- one involving only column names of a single base table, relational operators, and literals. For example, only a single message is needed to execute the command:

```
    UPDATE =account
    SET balance = balance * 1.07
    WHERE balance > 0;
```

which pays 7% interest to all positive accounts. The use of a single message to do set updates, deletes and, selections is the key optimization of NonStop SQL. As will be seen in the performance section, the consequent reduction of message traffic and message volume gives NonStop SQL a performance advantage over record-at-a-time database interfaces.

Sequential Block Buffering or Portals

Data specified by a single-variable query may be returned to the SQL executor in a block of records rather than a record-at-a-time -- this is the concept of portals described by Rowe [Rowe]. Two forms of blocking are supported, "physical" in which a physical disk page is returned, and "virtual" in which a projection and selection of the data is returned. Clearly, if the single-variable query is very selective, virtual blocking will have much reduced message cost and consequently be very attractive. In addition, sequential inserts and updates may be buffered in the client and sent to the disc server as a group. This requires locking the target key range in advance, and also requires checking integrity constraints in the client so that there will be no logical errors when the buffered insert is actually sent to the server.

Much of the performance advantage of NonStop SQL over its Tandem predecessor, Enform, derives from intelligent use of buffering. Enform had only physical buffering of reads. Physical buffering reduces both cpu time and elapsed time by a factor of three when compared to the record-at-a-time interface. Virtual buffering gives NonStop SQL an additional factor of 3 over physical buffering on many Wisconsin benchmark queries [Bitton]. Initially, the compiler's use of buffering was conservative because it implied coarse granularity (file) locking. But once the benefits of buffering were quantified, a form of block locking was implemented so that almost all sequential queries can use buffering without locking the entire table.

Compilation and Local Autonomy

Data unavailability can invalidate a compiled plan. For example, if the index for a table is unavailable because its disk or communication line is down, then a plan using that index will be invalid. Local autonomy requires that the SQL statement be recompiled and executed to use the available data and access paths. The NonStop SQL compiler implements this requirement by using the partial information in the local catalog.

Invalidating Compiled Statements

When the database design changes or the table statistics change, compiled plans become invalid. NonStop SQL's approach to invalidation differs from other systems in two ways. First, NonStop SQL invalidates plans whenever there is a chance that the change may affect the plan. For example, most SQL systems do not automatically recompile when an index is added. This could confuse a user who added an index and noticed that it did not help performance at all. NonStop SQL will automatically recompile the program in this case and in any other case where the plan may change.

NonStop SQL also differs in the way it invalidates programs. Most SQL systems invalidate SQL compilations by clearing the valid bit of the plan in the catalog. In such systems, whenever a program executes a transaction, it first reads the plan's valid bit from the catalog to assure that it is still valid, and then proceeds. If the bit is off, the program automatically re-SQL-compiles itself.

NonStop SQL rejected this approach because the valid bit can become a bottleneck if every transaction must access it, and because local autonomy implies that the database may change while some program is inaccessible. So NonStop SQL maintains the program valid bit only as a hint of what programs need recompilation.

NonStop SQL adopted a timestamp scheme to invalidate compiled statements. Associated with every table and view is a timestamp called the "redefinition" timestamp. This timestamp is bumped every time the table definition changes. Sample changes are add/drop index, update statistics, and add partition. These timestamps are stored in the catalog and in each file label.

When a statement is compiled, the redefinition times of all tables and protection views used by the plan are saved as part of the compiled object program. When the program executes, it sends an OPEN message containing the redefinition time to each fragment of the table. The disk process checks the program's authorization to the table and rejects the OPEN if the requester is not authorized. In addition, the disk process checks the redefinition time of the fragment against the redefinition time presented by the requester. If they do not match, the requester has an invalid plan and must recompile.

These OPENs remain valid for the life of the process. There is no additional check by the program when it runs subsequent transactions. If the definition changes while the table is open, then the disk process invalidates any opens on that table. The next time a request arrives for that table, the disk process informs the opener that its plan is invalid and is automatically recompiled (see Figure 7).

When the SQL executor senses an invalid statement, it launches an automatic recompilation of the statement, and then executes the resulting plan. This in turn causes a new OPEN of the table with a more recent table definition time.

The catalogs and the program file descriptor each carry a summary bit which tells whether the program's stored SQL compilation is valid. This bit is a hint for static SQL recompilation of programs and for initial SQL recompilation of the program when the process first starts execution.

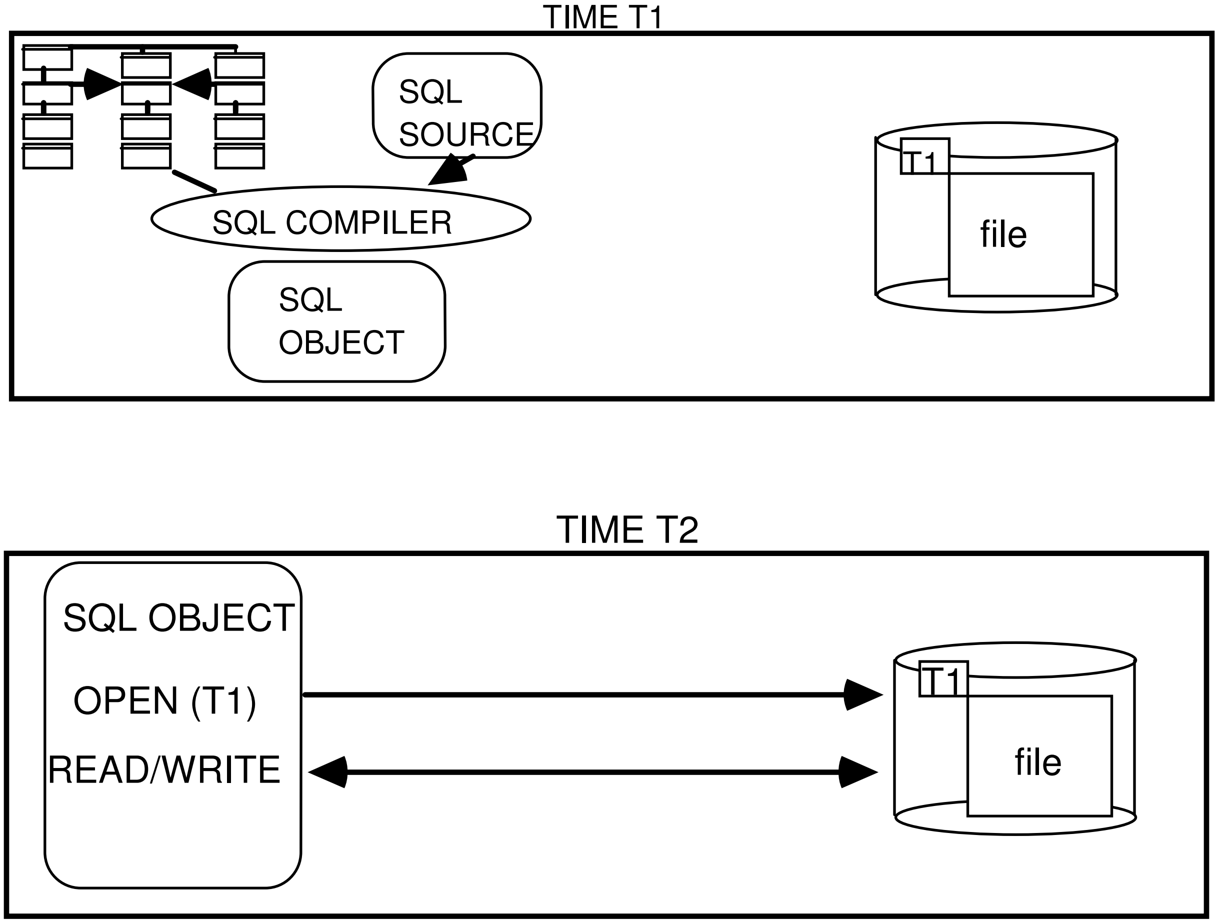

Figure 7. Plan validation and invalidation in NonStop SQL is done with timestamps. The plan records the redefinition timestamp of each table. When the plan opens the table it presents this timestamp which is matched with the timestamp in the file label. If the timestamps don't match, the plan is invalid. If the table changes, the open is invalidated. This design reduces catalog access when compared to other implementations.

Table Opens vs Cursor Opens

The Tandem system is designed for on-line transaction processing. In such systems, early binding is the watchword. Programs are compiled, installed and the system is brought up. Thereafter, the system might run for several months without change or interruption of service.

The ground rule in such systems is to do all the checking at startup so that there are no extra instructions in the "normal" path for the next few months. As a consequence, the NonStop SQL executor OPENs tables when they are first referenced by the application, and keeps the tables OPEN until the execution plan is invalid and a new OPEN with a new redefinition time is needed. Subsequent references to the table by any statement of the process will share this single OPEN. In particular, if many cursors are defined on the table, they will all use the same OPEN. The OPEN serves three purposes: it covers the redefinition/invalidation issue; it authenticates the requester; and it provides a virtual circuit between the SQL requester and the SQL server.

A table with ten partitions, and with three indices, each with five partitions will require twenty five opens (10+3*5). If most of these indices or partitions are not used by the program, the open cost may be a substantial wasted overhead and may cause slow startup. As a consequence, opens are done on demand. One partition of the table is opened. Thereafter, table partitions and index partitions are opened on demand. If an index becomes unavailable, it is reopened on demand the next time is is accessed.

When a transaction commits, all its locks are released and all its cursors are invalidated. But the OPENs persist to support the next transaction.

NonStop Operation

As the name suggests, NonStop SQL provides NonStop operation. Not only is the application program protected by transaction locking and logging. In addition, all device drivers and system processes run in NonStop mode so that they tolerate any single hardware failure and many software failures without disrupting service.

The disk processes maintain mirrored disks so that a disk failure will not disrupt service. One disk can be repaired and revived while the mirror disk is operating.

If a process or processor fails, all transactions involved in that process or processor are aborted (undone), but unrelated transactions are unaffected by the failure. Ownership of all devices (disks and communications lines) is automatically switched to other processors.

The network hides link failures as long as an alternate path is available.

If some partitions of a file are inaccessible, all available partitions are still accessible. Partitions are reopened on demand when they become available.

PERFORMANCE

Single-variable Query Processing Performance

Pushing single-variable queries (ones involving only literals, host variables, and record fields) out to the disk processes is a major optimization. For simple expressions, the disc server can process a record in about 1000 instructions. So on the Wisconsin Benchmark a VLX processor can scan the database at 2,600 records per second, or 520KB/sec. At this rate the VLX is CPU bound (it is a 3 mip processor). If the records are larger, say 400 bytes rather than 200 bytes, then the system becomes IO bound at about 700KB/sec. This is very competitive. For updates, a VLX can update a column at 600 records per second. Deletes run at a similar speed (all these numbers assume the updates are transaction protected).

This sequential performance is based on the automatic use of read-ahead and write-behind. The disc process will switch to sequential prefetch and postwrite on demand or if it detects three sequential accesses in a row by a client. In sequential mode, the disc process does two things: it moves data to and from disc in large transfers (typically 30KB to 60KB), and it ages used sequential pages quickly to prevent sequential flooding of the cache. Similar "bulk-IO" logic is included in the cache flush logic needed for quick system restart from disc. Main memory databases can be written to disc with a few large transfers rather than many small ones. The use of bulk io reduces the number of IOs, cpu consumption, and controller consumption. It allows batch sequential and random online transactions to coexist [Borr-1].

When sequential scanning is moved to the client, CPU demand rises by a factor of three and the throughput drops by the same factor of three due to message traffic. If sequential buffering is not used, another factor of three is typically lost due to the increased message traffic.

Performance on the DebitCredit Benchmark

We 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 to match the performance of the Cobol record-at-a-time interface to within 25% when executing the DebitCredit transaction, a simple on-line transaction processing application which updates three tables by key and inserts a record in a fourth table [Anon]. But when the application was coded and measured, the SQL application used less cpu time and the same number of ios as the record-at-a-time interface. This is surprising. No one would believe that CICS plus DB2 would have a shorter pathlength than CICS accessing the native CICS data management routines; yet that is the analogous statement.

Certainly it costs more to execute SQL statements -- they have quite a bit more semantics. On the other hand, each statement does more, so there is a possibility that fewer statements can be used. That is the key to understanding the surprising result that the DebitCredit transaction is faster on SQL than on Enscribe. To see this in detail, consider the following two programs. Ignoring errors, the database aspects of the code for the two applications are:

COBOL RECORD AT A TIME **COBOL + SQL**

```
READ account WITH LOCK           UPDATE account
   KEY IS act-number.            SET balance = balance + delta
ADD delta TO account.balance.    WHERE number = :act-number;
REWRITE account.

READ teller WITH LOCK            UPDATE teller
   KEY IS teller-num.            SET balance = balance + delta
ADD delta TO teller.balance.     WHERE number = :teller-num;
REWRITE teller.

READ branch WITH LOCK            UPDATE branch
   KEY IS branch-number.         SET balance = balance + delta
ADD delta TO branch.balance.     WHERE number=:branch-number;
REWRITE branch.
                                  INSERT INTO HISTORY VALUES (
MOVE timestamp TO history.timestamp.              :timestamp,
MOVE act-number TO history.act-number.            :act-number,
MOVE teller-number TO history.teller-number.      :teller-num,
MOVE delta TO history.delta.                      :delta);
WRITE history.
```

The record-at-a-time interface has seven calls -- 3 reads, 3 writes, and an insert. An IMS or DBTG interface would have seven similar calls. But SQL is able to do the application in four calls -- 3 updates and an insert. This economy translates into a three message savings on a base of seven, a 40% savings in messages! Such savings make SQL a wonderful interface for message based systems.

As a consequence of these savings, the SQL cpu cost is slightly smaller than the record-at-a-time cost (about 4% less over all and about 10% reduction in the database path-length). This savings is achieved while still providing the added functionality of SQL.

Tandem's smallest system, the EXT 10 was benchmarked at 4TPS. The benchmark was then scaled up to 4, 8, 16 and 32 VLX cpus to demonstrate modular growth of the system and transparent data distribution between the EXT 10 and the VLX complex. A plot of the TPS rating of the VLX system as processors were added is shown in Figure 8.

The software was benchmarked at 208 DebitCredit transactions per second on a 32 VLX processor cluster. At this high rate, each transaction does one physical read and one mirrored write on average; all other ios are buffered across many transactions. This configuration showed no bottlenecks and so could have been grown much larger. The price/performance of NonStop SQL is competitive with the very best transaction processing systems.

There are still many opportunities to improve performance of NonStop SQL in later releases. In particular, SQL lends itself to the IMS FastPath techniques of Field Calls and optimistic locking hidden under the SQL language interface [Gawlick].

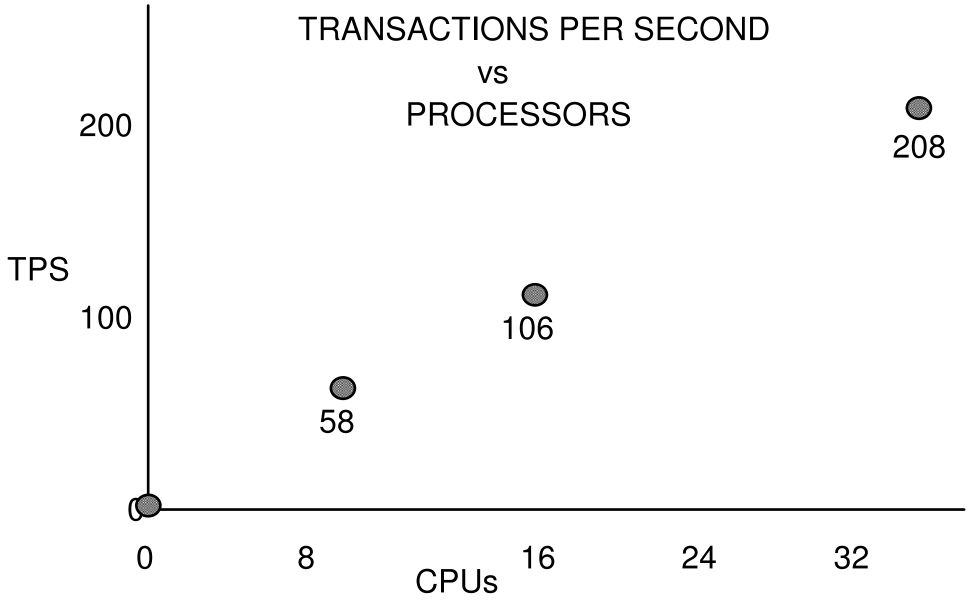

Figure 8. A chart showing the TPS rating of NonStop SQL as VLX processors and disks are added in increments of 8 processors. The third configuration is shown in Figure 1. Not shown in that figure is a 12 processor system simulating the 25,000 terminal network which is submitting the transactions to the measured system.

This performance and price/performance came as a surprise to us. It hinges on the unexpected synergy between message-based systems and distributed databases which allow modular growth, and the use of SQL, which saves lots of messages and so improves performance and functionality at the same time. In fairness, Ted Codd had predicted that relational systems would make distributed databases possible [Codd]. Codd was right.

Ad Hoc Query Performance

NonStop SQL was also benchmarked on the Wisconsin benchmark to test its performance on ad hoc queries [Bitton]. Considerable performance data is available for other relational systems running on processors ranging from a M68000 to a IBM 4381-P13 and a Teradata machine. It is difficult to compare these data to NonStop SQL's performance since the Tandem hardware and software are quite different from the other systems.

The only direct comparison possible is between NonStop SQL and Enform, the Tandem relational query product which is part of Encompass. As a median, NonStop SQL performs twice as well as its predecessor on the same hardware. This speedup comes from better plans in some cases, but most of the speedup comes from the distributed query execution implicit in subcontracting single-variable queries to the disk process.

Indirect comparisons suggest that NonStop SQL is about as fast as the fastest centralized relational systems when run on comparable hardware.

Performance Observations

The performance assurance of NonStop SQL exposed some language features which have inherently bad performance and which require language extension to correct. These are discussed in the following sections.

The Halloween Problem

Several SQL implementations have a famous performance bug called the Halloween problem bug. The Halloween problem was discovered one trick-or-treat eve, hence the name. It applies to queries of the form

```
UPDATE  payroll
SET salary = salary +1;
```

If this query uses an access path ascending on salary, then it will try to give all employees an infinite raise. First it will increment all the zero salary people giving them a salary of 1. Then it will give everyone with a salary of 1 a raise, and so on.

As a consequence, many SQL systems refuse to use an index on a field mentioned in the update clause. This gives very poor performance on common queries. Consider the query:

```
UPDATE employee
SET    department = 100
WHERE  department = 55;
```

the idea of not using the index on department number is a poor decision. It implies scanning the entire table. This is not acceptable for tables of credible size. In addition, this is not an instance of the Halloween problem.

NonStop SQL can use an index for an update statement if:

The value clauses (right hand side of assignments) do not involve fields of the index or,

The where clause fully specifies the index key as equal to literals or host language variables.

These are special cases, but they cover some very common situations. The example above qualifies twice since it satisfies both of these criteria for an index on department.

In addition, when a cursor is defined for update, NonStop SQL insists that the clause `FOR UPDATE ON` <field list> be included. This tells the compiler what fields might be involved in the Halloween problem, and consequently which indices to avoid. The SQL executor refuses to update undeclared fields via such a cursor.

Group support

SQL treats each field of a record as an individual entity. So the Cobol group

```
03 DATE.
     04 MM  PIC(99).
     04 DD    PIC(99).
     04 YY    PIC(99).
```

cannot be called DATE in SQL. Rather each field must be individually named by the program and moved by SQL. Typical applications have 100 such fields per record on average, and 1000 fields per record is not unusual.

An `INSERT` or `FETCH` statement must specify all 1000 values in the correct order for such a table. This is error-prone and inconvenient.

As defined in the ANSI standard, the SQL executor must move each field to and from individual host language variables. So, if the programmer wants to insert or fetch a record with 1000 fields, the field move logic is exercised 1000 times for each record select or insert. No matter how well that logic is optimized, it will have a difficult time competing with the record-at-a-time interface which treats the entire record as a string of bytes and moves it with a single instruction.

The solution to this is to provide group support in the SQL language.

A related issue is comparing groups of fields. In most SQL systems if one wants to do a lexical compare of several fields, the syntax is:

$$(a = :a \text{ AND } b = :b \text{ AND } c \geq :c) \text{ OR } (a= :a \text{ AND } b \geq :b) \text{ OR } a > :a$$

by introducing tuple variables one gets the syntax:

$$(a, b, c) \geq (:a, :b, :c).$$

This construct is surprisingly common; fortunately, this new syntax is now a standard part of SQL2.

Parameters at Compile time

If the SQL optimizer is presented with literals in a selection statement, it can generally make a reasonable estimate of the selectivity of the predicate. If, on the other hand, it is presented with host variables, it assumes that a fraction of the domain will be selected. The fraction is inversely proportional to table size. This wild guess may lead to a poor plan. For example, consider the query:

```
SELECT customer
FROM   accounts
WHERE  account_number BETWEEN :min AND :max;
```

Suppose that there are a million accounts. The optimizer will guess that this query will select 333,333 records. If, on the other hand, it sees the query:

```
SELECT customer
FROM   accounts
WHERE  account_number BETWEEN 50000 AND 60000;
```

then the optimizer will guess that 10,000 records will be selected. This disparity may result in bad query execution plans. To our knowledge, all relational systems suffer from this illness. This problem is serious for SQL programs; almost all SQL statements will involve host-language variables rather than literals. It is troublesome that SQL optimizers are unable to guess set cardinalities in this case.

There are three proposals (not implemented) to deal with this problem. The first is to let the user give the compiler hints.

The second idea is to execute the program in "training mode" with sample parameters, compile the statements on the fly, and save the plans used for the presented literals as the plan to be used in production.

A third scheme is to set a threshold for guesses. If the compiler cannot find a cheap plan when host language variables are present, it would simply not compile at static compilation time. At run time, the statement would be compiled dynamically on first invocation with the literals bound in for the host variables. This is similar to the first scheme but is more automatic.

Update Statistics

The SQL compiler picks execution plans based on estimated table sizes, record sizes, index selectivity, and for each field, the number of unique values, and the minimum and maximum values. All these indicators are collected by a utility SQL command called `UPDATE STATISTICS`.

For the 20GB database used in the DebitCredit benchmark, the first implementation of this utility would have taken about a day to collect all the statistics on the various tables. Since NonStop SQL is intended for databases far in excess of 20GB, something had to be done. `UPDATE STATISTICS` now samples the database and estimates the statistics. Now its maximum running time is a few minutes.

A complete scan is done for small tables (<4MB). For a 20GB table, a 64,000 bit hash table is allocated for each field; and, 32,000 records are examined. The values of each record's fields are hashed into its hash table; and, the min and max value of each field is maintained. After all these records have been examined, the minimum value, maximum value, and number of unique values are estimated. This algorithm is in the spirit of Sample Counting described by [Astrahan-2].

SUMMARY

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

- The first distributed SQL -- it offers distributed data, distributed execution, and distributed transactions.
- A data management system that runs on small (30K$) computers and on large (30M$) computers and on many sizes in between.
- A NonStop SQL -- it tolerates any single fault without interrupting service.
- The first high-performance SQL -- it has been benchmarked at over 200TPS with no bottlenecks in sight.
- An SQL with a cost per transaction comparable to the ugliest record-at-a-time high-performance data management systems.

As a consequence, it is useable in both the information center and in production environments.

NonStop SQL disposes of the myth that relational systems are inherently slow, much as Fortran disposed of the myth that assembly language was required for efficient code.

The combination of SQL semantics and a message-based distributed operating system gives a surprising result: 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 were designed together. The disc process directly executes single-variable SQL queries. The Tandem naming mechanism was extended to support default catalogs; the authorization mechanism was extended to cover tables, views and other SQL objects; object programs were extended to include SQL sections; and the measurement facility was extended to measure SQL counters. In addition, the implementation integrated SQL objects and verbs with the Tandem transaction manager, network and NonStop mechanisms.

This integration comes at some cost: it is not portable to other machines. On the other hand it has considerable benefits in usability, simplicity and performance.

REFERENCES

[Anon] Anon et al., "A Measure of Transaction Processing Power", Datamation, V. 31.7, April 1985, pp. 112-118.

[ANSI] "Database Language SQL", American National Standard X3.135-1986.

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

[Astrahan] M. Astrahan et al., "System R: a Relational Approach to Database Management", ACM TODS 1.2, June 1986.

[Astrahan-2] M. Astrahan et al., "Counting Unique Values of an Attribute Without Sorting", IBM RJ 4960, December 1985

[Bartlett] J. Bartlett, "A NonStop Kernel", Proc 8th ACM SOSP, Dec. 1981.

[Bitton] D. Bitton, et al., "Benchmarking Database Systems: A Systematic Approach", Proc. 9th VLDB, Nov 1983.

[Borr] A. Borr, "Transaction Monitoring in Encompass", VLDB, Sept. 1981.

[Borr-1] A. Borr, "High Performance SQL Through low-level System Integration", SIGMOD 88, ACM, June 1988.

[Codd] "Relational Database: A Practical Foundation for Productivity", CACM 25.2, Feb. 1982.

[Date] *An Introduction to Database Systems*, Volume 1, Addison Wesley, April 1986.

[DB2] *IBM Database 2 General Information Manual*, IBM Form No. GC 26-4073-2, Armonk, NY, Feb. 1986.

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

[Gawlick] D. Gawlick, "Processing Hot Spots in High Performance Systems", Proc. IEEE Compcon, Feb. 1985.

[Gray] J. Gray, et al. "Granularity of Locks and Degrees of Consistency in a Shared Database", *Modeling in Database Management Systems*, G.M. Nijssen ed., Jan 1976.

[Helland] P. Helland, et. al. "Group Commit Timers and High Volume Transaction Systems", to appear in this volume, also Tandem Technical Report 88.3, Cupertino, CA, May 1988.

[Informix] *Informix SQL Relational Database System, Users Guide*, Part No. 200-41-1015-8, Relational Database Systems Inc., Menlo Park, CA, June 1986.

[Ingres] *The INGRES Papers: The Anatomy of a Relational Database Management System*, M. Stonebraker ed., Addison Wesley, May 1985.

[Lindsay] B. Lindsay, "Object Naming and Catalog Management for a Distributed Database Management System", 2nd Int. Conf. on Distributed Computer Systems, IEEE, April 1981.

[Mohan] C. Mohan et al., "Transaction Management in the R* Distributed Database Management System", ACM TODS, V11.4, Dec. 1986.

[Measure} *Measure User's Guide*, Part No. 82440, Tandem Computers Inc, Cupertino, CA, Dec. 1986.

[NonStop SQL] *Introduction to NonStop SQL*, Part No. 82317, Tandem Computers Inc, Cupertino, CA, March 1987.

[NonStop SQL-1] *NonStop SQL Benchmark Workbook,* Part No. 84160, Tandem Computers Inc, Cupertino, CA, March 1987.

[NonStop SQL-2] Group, T.P., "A Benchmark of NonStop SQL on the DebitCredit Transaction", SIGMOD 88, ACM, June 1988.

[Pathway] *Introduction to Pathway*, Part No. 82339, Tandem Computers Inc, Cupertino, CA, June 1985.

[Pathmaker] *Introduction to Pathmaker*, Part No. 84070, Tandem Computers Inc, Cupertino, CA, March 87.

[Palermo] F. Palermo, "A Database Search Problem", *Information Systems: COINS IV*, J. Tou ed., Plenum, 1974.

[Rowe] L. Rowe, "Database Portals: A New Application Programming Interface" VLDB, Aug 1984.

[SafeGuard] *SafeGuard User's Manual*, Part No. 82539, Tandem Computers Inc, Cupertino, CA, Feb. 1987.

[SQL/DS] *SQL/Data System Concepts and Facilities,* IBM Form No GH24-5013, Armonk, NY, Feb. 1982.

[Teradata] "The Genesis of a Database Computer: A Conversation with Jack Shemer and Phil Neches of Teradata Corporation", IEEE Computer, Nov. 1984.

[Williams] R. Williams et al., "R*: An Overview of the Architecture", IBM Research Report RJ3325, San Jose, CA, Dec 1981.