A Benchmark of NonStop SQL Release 2 Demonstrating Near-Linear Speedup and Scaleup on Large Databases

Susanne Englert
Jim Gray
Terrye Kocher
Praful Shah

Technical Report 89.4
May 1989
Part Number 27469

A Benchmark of NonStop SQL Release 2 Demonstrating Near-Linear Speedup and Scaleup on Large Databases

Susanne Englert
Jim Gray
Terrye Kocher
Praful Shah

Technical Report 89.4
Tandem Part No. 27469
May 1989

A Benchmark of NonStop SQL Release 2 Demonstrating Near-Linear Speedup and Scaleup on Large Databases

Susanne Englert, Jim Gray, Terrye Kocher, Praful Shah

Tandem Computers, Inc.

May 1989

Tandem Technical Report 89.4

Abstract: NonStop SQL is an implementation of ANSI/ISO SQL on Tandem Computer systems. In its second release, NonStop SQL transparently and automatically implements parallelism within an SQL statement. This parallelism allows query execution speed to increase almost linearly as processors and discs are added to the system -- speedup. In addition, this parallelism can help jobs restricted to a fixed "batch window". When the job doubles in size, its elapsed processing time will not change if proportionately more equipment is available to process the job -- scaleup. This paper describes the parallelism features of NonStop SQL and an audited benchmark that demonstrates these speedup and scaleup claims.

Table of Contents

Introduction.. 1

The Uses of Parallelism: Speedup vs Scaleup 2

OLTP Scaleup -- Exploiting Inter-Transaction Parallelism 6

Batch Speedup and Scaleup -- Exploiting Intra-Transaction Parallelism 8

The NonStop SQL Release 2 Benchmark 12

Why Does NonStop SQL Get Near-Linear Speedups and Scaleups............. 20

Summary and Future Directions.. 22

Acknowledgments.. 22

References... 23

Introduction

NonStop SQL is an implementation of ANSI/ISO SQL on Tandem Computer systems. In its second release, NonStop SQL transparently and automatically implements parallelism within an SQL statement. The first release focused on functionality: NonStop SQL was the first SQL system to offer distributed data, distributed execution, and distributed transactions. It was integrated with Tandem's operating system, transaction monitor, and application generator. It provided fault-tolerant execution with node autonomy. As a consequence of this design, NonStop SQL has good performance for online transaction processing (OLTP) applications. It was benchmarked at over 200 DebitCredit transactions per second (tps). This performance is still unrivaled by other relational data management systems.

NonStop SQL's second release had three main goals:

- Improved compliance with the ANSI/ISO SQL standard and the emerging SQL2 standard.
- Improved operational utilities such as online reorganization, quick index build, etc.
- The ability to automatically and transparently exploit Tandem's parallel hardware and software architecture to execute relational queries in parallel.

This article focuses on this third issue, the use of parallelism to improve performance. Basically, Release 1 was good for OLTP, and Release 2 is good for batch and OLTP.

The motivation for this focus on parallelism is fairly simple. NonStop SQL is being increasingly used for large databases -- databases in the 100GB range. Occasionally, queries need to search the entire database. Without parallelism, such a query scanning at one megabyte per second would take about a day. With parallelism, this time could be reduced to less than an hour. Large databases require the use of parallelism in scanning data and in utilities such as data loading, dumping, reorganization, or index build.

A second motivation for the use of parallelism within SQL queries is to combine OLTP and batch processing needs. OLTP systems have many processors and discs to support many small transactions doing random IO. By automatically converting SQL statements to parallel execution, NonStop SQL can apply this OLTP hardware to a batch job running against the online database. This completes the performance requirements for a single Data Base of Record system by bringing query and batch jobs from the information center, working on stale data, into the online environment which has the current data.

The Uses of Parallelism: Speedup and Scaleup

Parallel processors can divide a big problem into many smaller ones to be solved in parallel. This divide-and-conquer approach can be applied in two ways:

Speedup: To solve a given problem faster by breaking it into many smaller problems.

Scaleup: To solve a larger problem in the same amount of time by applying proportionately more processing power to the problem.

For example, consider a manufacturing application which does Materials Resource Planning (MRP) as a batch job each night. If the MRP batch run takes thirty hours on a uni-processor, the customer will want a parallel processing system in which ten processors can solve the problem in three hours -- this is a speedup problem. If the problem doubles in size (because new products are added) the customer faces the scaleup problem: will double the processing and storage power still deliver the MRP answer within three hours?

The scaleup problem is also faced by Online Transaction Processing (OLTP) applications which grow. If a company grows to twice as many customers then its order-entry transaction processing system will be presented with twice as many transactions per second. Can the system be scaled up to meet this increased demand? Will double the terminal network, processors, and database deliver double the transaction throughput?

Speedup and scaleup are very similar problems, but they differ in interesting ways. Scaleup usually has economic benefits, speedup usually costs more -- trading time for money. It is possible for a system and application to have good scaleup properties, but not have any opportunities for speedup. So, although the concepts are closely related they are not identical.

When used for speedup, parallelism is rarely cheaper than a sequential solution; rather, parallelism trades time for money by buying more equipment to solve the problem in less time. At best, parallelism used for speedup gives an even or "linear" tradeoff between time and money: twice as much equipment (money) produces an answer in half as much time (speedup). In some cases, the hardware has already been bought for OLTP, so batch programs may be run at times of light load to get "free" speedup.

When used for scaleup, parallelism usually is the most economical solution. It allows the customer to buy equipment as needed. The system can grow by adding processing and storage modules as the demand grows. By contrast, a system which does not scale requires that the hardware be replaced with a more powerful system, rather than being incrementally expanded. The ability to scaleup a system has significant financial benefits over the more traditional design in which one replaces the hardware with new equipment in order to upgrade performance.

The challenge is to design a system which automatically gives linear speedups and scaleups. This is not always possible. Some problems do not decompose at all. To give the classic example, you can't make babies faster by employing more people [Brooks]. But, as shown by the Gamma System at the University of Wisconsin [Dewitt-2] and by the benchmark results described here for NonStop SQL, one can get linear speedups and scaleups on large problems for each of the relational operators: select, project, aggregate, join, insert, update, and delete.

Previous benchmarks demonstrated linear scaleups of NonStop SQL for Online Transaction Processing workloads [NonStop SQL 3]. OLTP workloads consist of many small jobs, and so suffer from startup and skew problems when each small transaction is further decomposed into parallel units of work. Consequently, OLTP workloads do not typically display linear speedups.

Speedup is a measure of how much faster a parallel system solves the problem. Formally, speedup is defined as:

$$\text{Speedup} = \frac{\text{OldElapsedTime}}{\text{NewElapsedTime}}$$

As more processors and discs are added, the new elapsed time for the job should be proportionally less than the old time. The ideal speedup curve looks like Figure 1.a which shows linearly increasing speedup as more processors and discs are added to the problem.

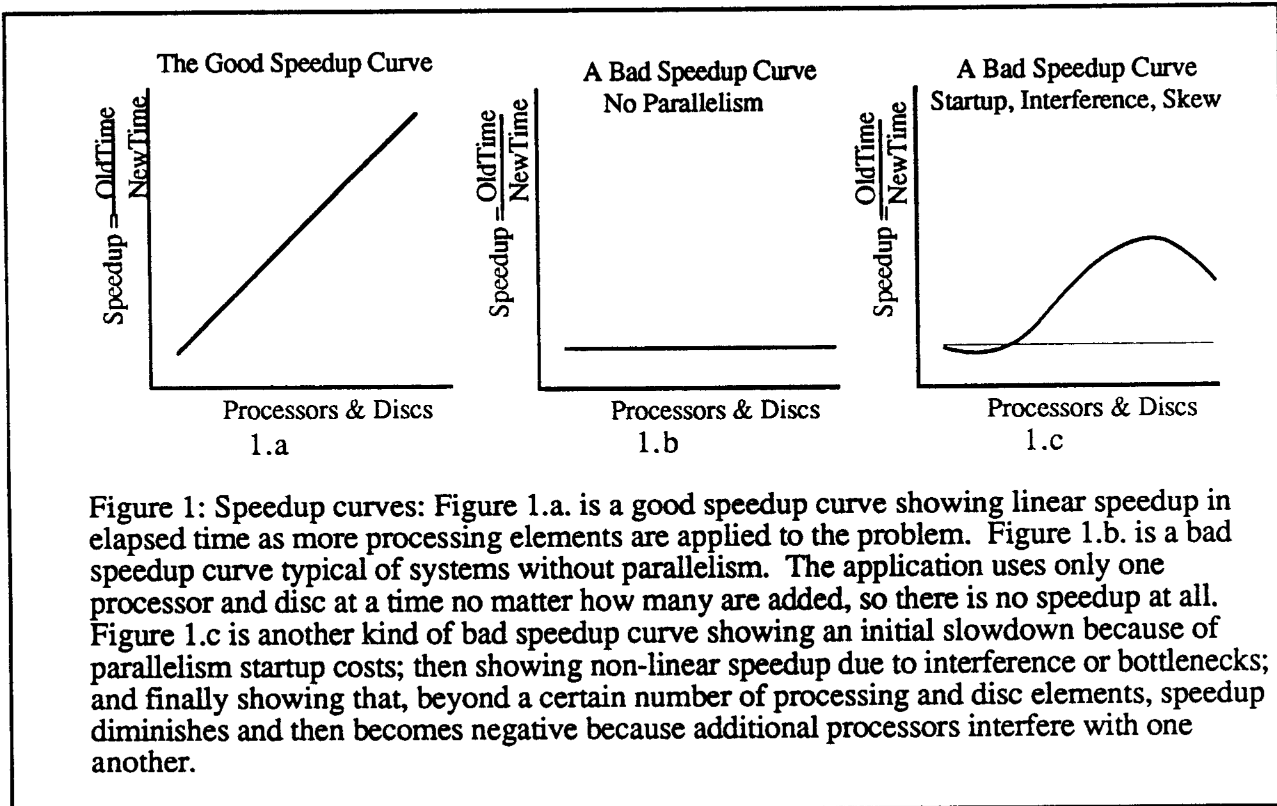

Figure 1: Speedup curves: Figure 1.a. is a good speedup curve showing linear speedup in elapsed time as more processing elements are applied to the problem. Figure 1.b. is a bad speedup curve typical of systems without parallelism. The application uses only one processor and disc at a time no matter how many are added, so there is no speedup at all. Figure 1.c is another kind of bad speedup curve showing an initial slowdown because of parallelism startup costs; then showing non-linear speedup due to interference or bottlenecks; and finally showing that, beyond a certain number of processing and disc elements, speedup diminishes and then becomes negative because additional processors interfere with one another.

Most systems do not display any speedup because they do not exploit parallelism (Figure 1.b). For example the first release of NonStop SQL did not automatically decompose relational operations into smaller jobs that could be independently executed on multiple processors and discs. Rather a single job used a single processor and a single disc at a time -- parallel execution was obtained by running many independent transactions in parallel. In essence, the parallelism was explicit in the application consisting of many small jobs. So Release 1 of NonStop SQL displayed a speedup of 1 for a single job no matter how many processors and discs were added. The major innovation of NonStop SQL Release 2 is to automatically detect opportunities for parallelism within single SQL statements and to automatically give "batch" applications linear speedups by executing the SQL statement in parallel on multiple processors and discs.

Even parallel systems do not typically have linear speedups because of startup, interference, and skew problems (Figure 1.c). Parallel processors take more time to start working on the job, much as a larger group of people takes longer to get to work on a shared project -- this is called the *startup* problem. Once the processors start working, they may interfere with one another or may queue behind some bottleneck. A typical example of this appears in shared-memory multi-processors where a six processor system may only have three times the power a uni-processor due to memory or software interference. If such a system grows to ten processors, the tenth processor

may well introduce more interference than it contributes as speedup. The net effect is a slowdown of the system beyond a certain number of processors -- this is called the *interference* problem. A third problem arises when the job is divided into such small units that the startup and processing variance is larger than the processing time for each tiny part -- this problem is called *skew*. As skew begins to dominate, there is no speedup advantage in further sub-dividing the job. These startup, interference and skew problems are inherent [Smith]; so the speedup curve in Figure 1.a must ultimately flatten out and then begin a downward trend.

Scaleup measures the ability of a parallel system to deal with a growing problem. Scaleup actually has two forms -- batch scaleup and OLTP scaleup. In both cases, scaleup postulates that the problem and system both double in size, but they differ in their goals in using parallelism:

batch scaleup goal: keep a large job's elapsed processing time constant .

OLTP scaleup goal: double the system's transaction throughput (transactions per second rate) while keeping response times the same.

Formally:

$$\text{BatchScaleup} = \frac{\text{NewElapsedTime}}{\text{OldElapsedTime}}$$

and

$$\text{OltpScaleup} = \frac{\text{NewThroughput}}{\text{OldThroughput}}$$

Good `BatchScaleup` numbers for an n-processor n-disc system are close to "1" while good `OltpScaleup` numbers for such a system are close to "n" .

The need for batch scaleup is familiar to customers with applications which must fit in a batch window, typically the daily eight-hour graveyard shift. As the problem grows, it must still be processed within that fixed time period. The ideal batch scaleup curve looks like Figure 2.a which shows constant processing time as the problem and system grow proportionately in size. Figure 2.b. is a more typical batch scaleup curve. As the problem size grows, the processing time grows.

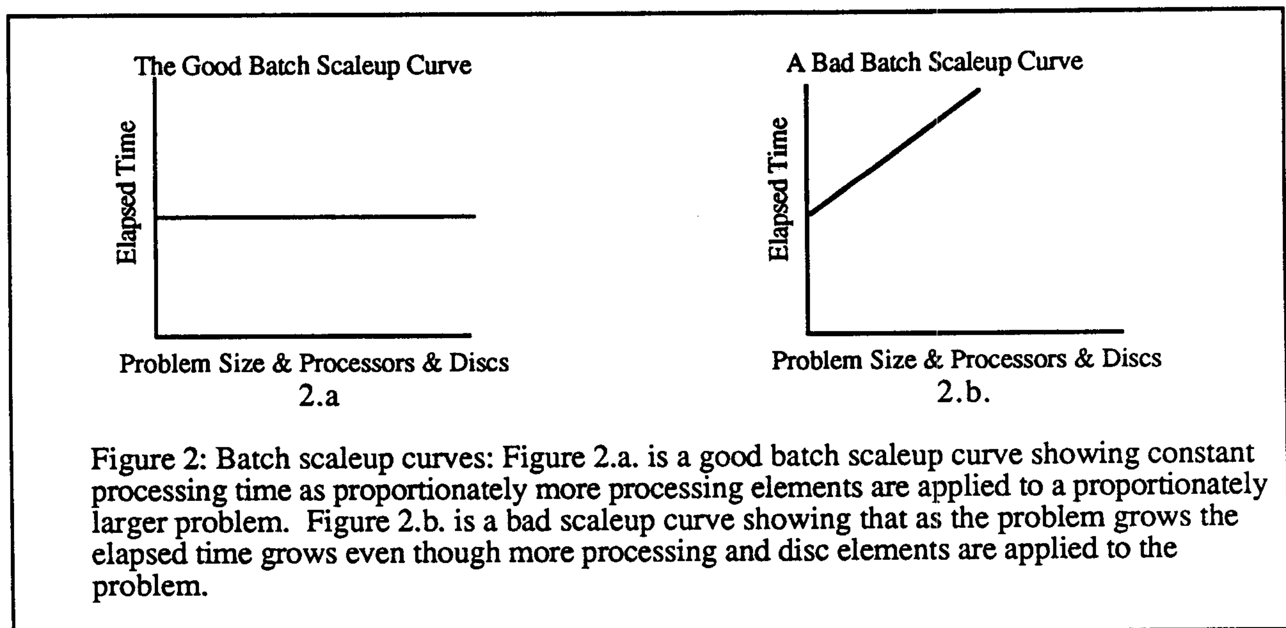

Figure 2: Batch scaleup curves: Figure 2.a. is a good batch scaleup curve showing constant processing time as proportionately more processing elements are applied to a proportionately larger problem. Figure 2.b. is a bad scaleup curve showing that as the problem grows the elapsed time grows even though more processing and disc elements are applied to the problem.

The good and bad OLTP scaleup curves are similar to the speedup curves -- except that the abscissa is labeled with throughput rather than speedup. Figure 3 gives an example of a good OLTP scaleup curve. The transaction throughput scales linearly as the number of terminals, processors, and discs is increased.

A system with good speedup on a large problem will probably have good batch scaleup on smaller problems. The real question is how large the problem must become before one hits some kind of system limit or bottleneck. Ideally, this limit is far beyond the economic barriers of the system. A system that scales to a terabyte database, or ten thousand transactions per second would be fine -- since such a large system could cost in excess of a hundred million dollars with today's technology. The goal is to hit an economic barrier before the system hits a software or hardware limit.

OLTP Scaleup -- Exploiting Inter-Transaction Parallelism

Release 1 of NonStop SQL was designed to be good at online transaction processing. OLTP workloads consist of many relatively simple transactions executing in parallel. This situation naturally lends itself to parallel processing; the parallelism is inherent in the application. Many independent messages arrive and can be processed independently. Each transaction operates on part of a distributed database, but the likelihood that two transactions will interfere with one another is quite small. So OLTP is a natural candidate for scaleup. One of the key design goals of NonStop SQL was to give near-linear scaleup for OLTP applications. To our knowledge, Tandem is the only vendor to accomplish this in a commercial product. The key techniques used were requester-server design, row locking, group commit, avoiding hotspots on sequential files, and avoiding hotspots in areas such as system catalogs, recovery log, and transaction scheduling [Gawlick] [Helland]. A more detailed description of the system architecture may be found in [NonStop SQL 1, 2].

To demonstrate the scaleup of NonStop SQL on OLTP problems, Tandem did a sizeable benchmark involving 4, 8, 16, and 32 processors and their attendant discs and network. The database, terminal network, and transaction load were proportionately scaled according to the rules of the DebitCredit benchmark [Anon]. The resulting scaleup curve showing throughput (transactions per second) vs system and problem size is shown in Figure 3 (adapted from [NonStop SQL 2]). We believe the benchmark could have scaled well beyond 32 processors, but that seemed an economic place to stop -- being a 10 million dollar machine configuration. Subsequently, NonStop SQL has demonstrated similar scalability on a wide variety of OLTP applications [Cassidy].

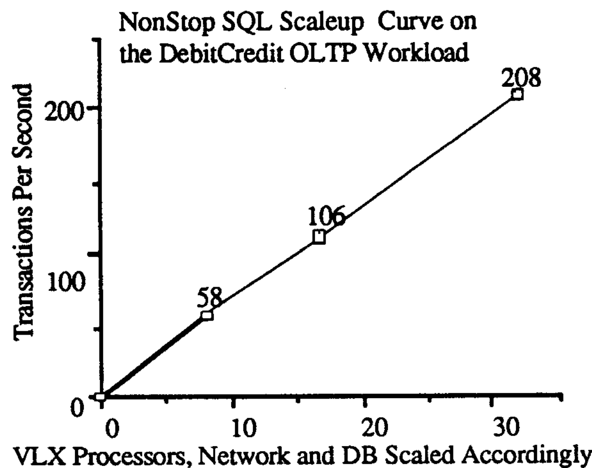

Figure 3: OLTP scaleup of NonStop SQL Release 1 on the DebitCredit OLTP workload. As processors grew from 4 to 32, the database grew from 3.2 million rows to 25.6 million rows and the terminal network grew from 320 terminals to 2560 terminals. A 26GB history table was also maintained. As the graph shows, the throughput of the system scales almost linearly with the workload. This shows linear OLTP scaleup of the system.

Release 2 of NonStop SQL has this same good scaleup for OLTP workloads. The only parallelism feature added for OLTP in Release 2 is parallel update of indices. Typically, a relational database will have five or ten indices on each relational table. These indices are maintained to allow fast or sorted access to the table when only a few of the row attributes are specified. For example the employee table is typically indexed by employee number, last name, first name, soundex name, department, location, job code, phone number, etc. Whenever a row is inserted, updated, or deleted these indices must be maintained. Maintenance of multiple indices can add to the response time of inserts, updates, and deletes. In Release 1 of NonStop SQL this index maintenance was done serially, one-index-at-at-time. In Release 2 of NonStop SQL, maintenance of indices is done in parallel when an insert, update, or delete is performed. Tests indicate that inserts now run in almost constant time, independent of how many indices are defined on the table -- an index maintenance speedup. So now there is no response time penalty for maintenance of many indices. Of course, the customer must still pay the cpu maintenance cost and the storage cost for the indices.

To summarize then, Release 1 of NonStop SQL demonstrated almost linear scaleup on OLTP workloads. This scaleup has been done beyond a ten million dollar system -- close to the economic barrier. There seem to be no inherent software or hardware limits to the OLTP scaleup. The key to the successful scaleup was to exploit the inter-transaction parallelism inherent in OLTP applications. Release 2 tries to automatically exploit parallelism within a single program -- intra-transaction parallelism

Batch Speedup and Scaleup -- Exploiting Intra-Transaction Parallelism

The key contribution of NonStop SQL's second release is to exploit Tandem's parallel architecture to give linear speedups and scaleups for "batch" SQL operations. SQL has a non-procedural set-oriented data manipulation language. SQL data manipulation operations are built up from the following basic operations:

- *select* all the set's rows satisfying a predicate
- *project* (remove) certain fields from all rows in a set
- *aggregate* all values in a set (e.g. compute sum or average)
- *join* the rows in two sets on some attribute to form a new set

Each of these operations produces a new set. This set may in turn be fed into other operations or the resulting set may be deleted, updated, or inserted into an existing table. This ability to arbitrarily compose relational operators is a key to the power of the relational model. If a system can give linear speedup and scaleup for each of these operators, then it should give linear speedup and scaleup for any combination of them.

A select requiring a complete scan of a table provides the simplest example of how one can get linear speedup on relational operators. Consider a 1GB table consisting of 10,000,000 rows each 100 bytes long. This table could be stored on a single disc accessed by a single processor or it could be partitioned across ten discs and processors: each having one tenth of the table (see Figure 4). In this case, if the SQL system scans all ten discs in parallel it gets a speedup of ten. If the database system gives location transparency, this partitioning is invisible to the application program which can run in any of the processors and access the partitioned table as a single logical table.

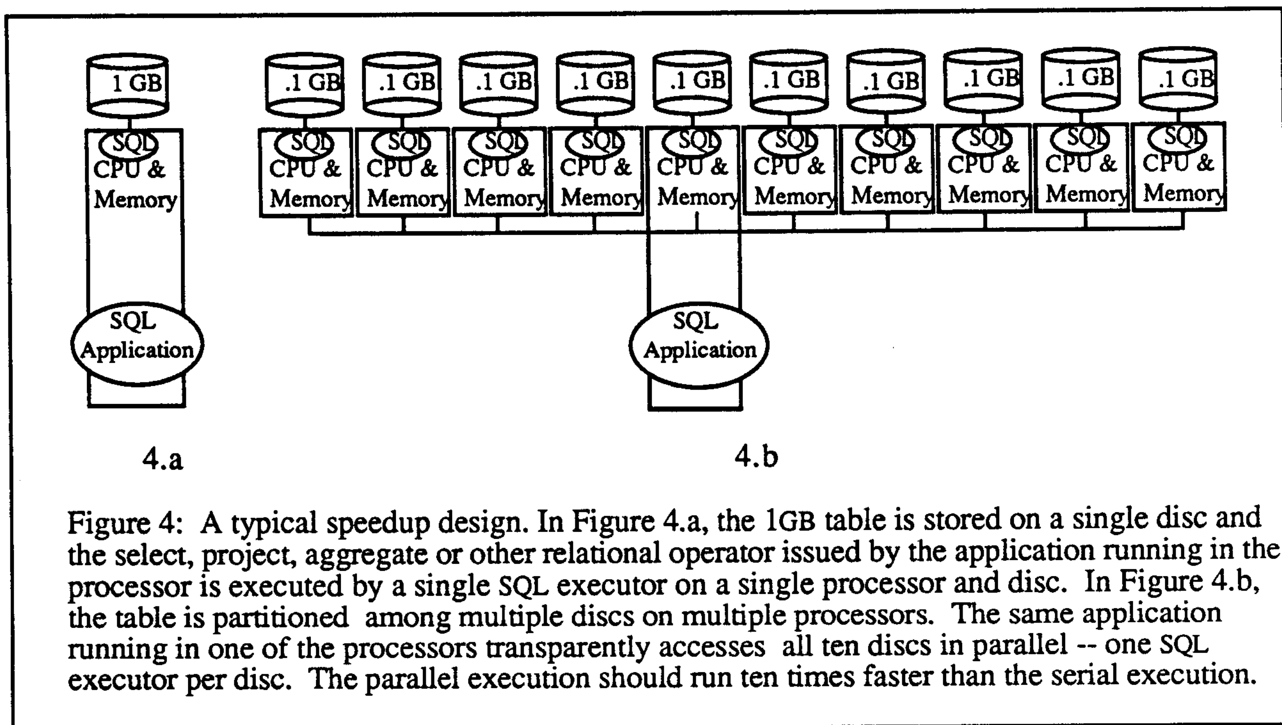

Figure 4: A typical speedup design. In Figure 4.a, the 1GB table is stored on a single disc and the select, project, aggregate or other relational operator issued by the application running in the processor is executed by a single SQL executor on a single processor and disc. In Figure 4.b, the table is partitioned among multiple discs on multiple processors. The same application running in one of the processors transparently accesses all ten discs in parallel -- one SQL executor per disc. The parallel execution should run ten times faster than the serial execution.

The corresponding linear scaleup for NonStop SQL comes when the database grows by a factor of ten from a 1GB table consisting of a 10,000,000 rows to a 10GB database consisting of 100,000,000 rows. As the table is scaled up, it is spread among ten discs and processors (see Figure 5). In this case, if the application's SQL statement scans all ten discs in parallel it can scan the larger 10GB database in the same time as it can scan the original 1GB database. This gives the system a batch scaleup (as in Figure 2.a.). Again, location transparency hides this partitioning from the application program -- the partitioned table looks like a single logical table.

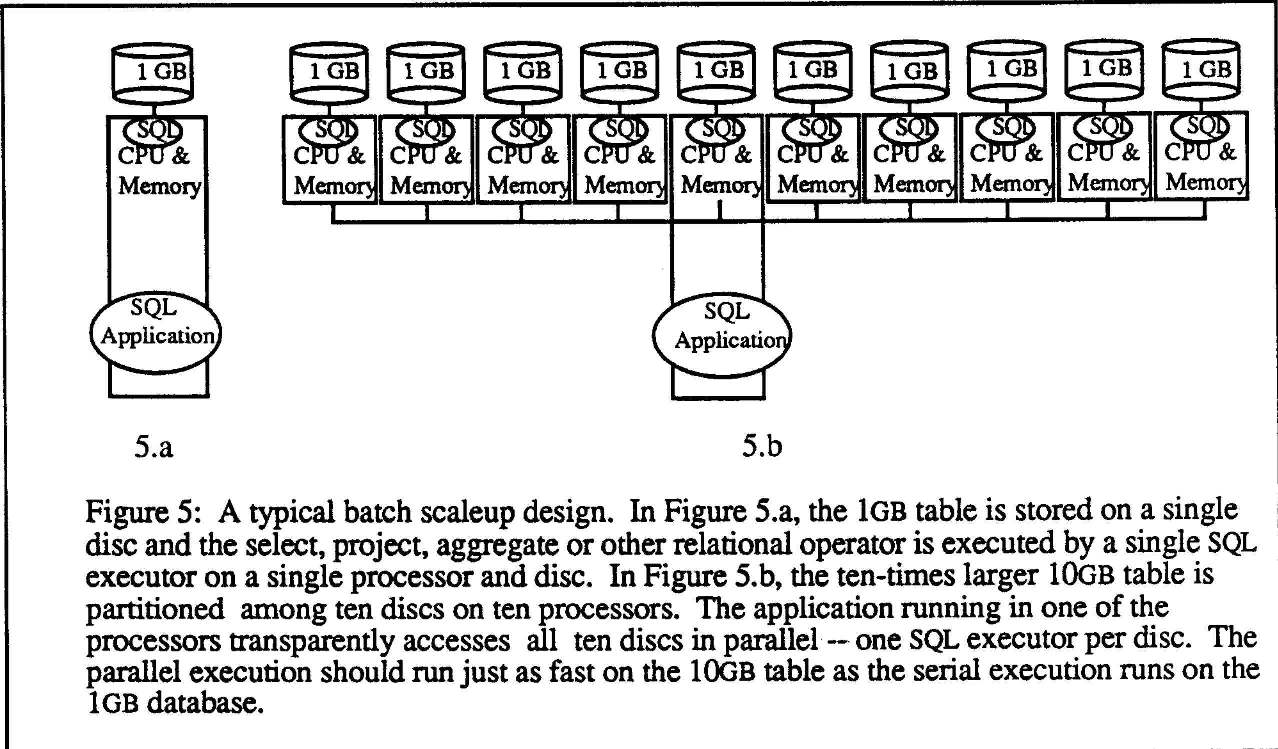

Figure 5: A typical batch scaleup design. In Figure 5.a, the 1GB table is stored on a single disc and the select, project, aggregate or other relational operator is executed by a single SQL executor on a single processor and disc. In Figure 5.b, the ten-times larger 10GB table is partitioned among ten discs on ten processors. The application running in one of the processors transparently accesses all ten discs in parallel -- one SQL executor per disc. The parallel execution should run just as fast on the 10GB table as the serial execution runs on the 1GB database.

From these two examples, it can be seen that the key to the parallelism of NonStop SQL depends on horizontally partitioning the data among multiple discs and processors, and on subcontracting relational operator execution to each processor when the query is invoked. This idea generalizes directly to project, aggregate, update, and delete operators, each of which can be subcontracted to the partitions of the data in this way.

A similar scheme applies to join operations. NonStop SQL has three parallel join strategies. The simplest and most common can be used if both tables being joined are already partitioned in exactly the same way, and if the join is on a prefix of the table's clustering keys. For example, if the outer table is the item master and the inner table is the item details of an invoice application, then it is quite likely that the physical database design will cause the item details to be clustered together on one disc and that the corresponding item master will be clustered on the same disc (i.e. the partitioning keys of the item master table will be the same as the partitioning keys of the item detail table). In this case, the join is processed by the individual processors and discs, and no inter-processor communication or interference occurs. The join benchmark used below is such a clustered join: the two tables are joined via their primary keys. A second join strategy comes into play if one of the tables is small, is joined on a key field, and if the other table is large and partitioned. In that case, each partition of the large table does a nested join in parallel with the small (inner) table. The third strategy applies if both tables are large, or if no useful key fields are involved in the join. In this case, the two tables are repartitioned among all local processors using a hash function. When the partitioning is complete, the join has been divided into many small joins

which can be done independently. This partitioning and the subsequent joining can be done in parallel. The individual partition joins are usually done as sort-merge joins. As shown in research prototypes [Dewitt-1&2], these techniques give both linear speedups and linear batch scaleups for join operations.

These ideas are similar in spirit to the designs of Teradata [Teradata], Gamma [Dewitt-1] [Schnieder], Bubba [Smith], and Prospect [Reuter]. The algorithms are a subset of those used in the University of Wisconsin Gamma Database machine. The novelty here is that the algorithms are implemented atop a conventional commercial multi-processor rather than on a specialized database machine like the Teradata or Gamma systems. In addition, NonStop SQL provides full transaction integrity for applications and data distributed in a local network or a wide area network. This means that the NonStop SQL system can also be used for OLTP, networking, and can run application programs. Database machines require that the user buy and maintain a separate general purpose system to run the computer network, transaction monitor, application programs, and operator interface. This two-system approach is inconvenient, and if the general-purpose system cannot provide speedup and scaleup as the application grows, then the database machine's speedup and scaleup benefits may be lost.

The discussion of parallel query execution did not touch on the many bottlenecks that may arise. If the answer set of a select, project, or join is directed to an application, then the speed of the system is limited by how fast the application can read and process the answer set. Since the Cobol, C, or Pascal application runs on a single processor, it will be the ultimate bottleneck. Typically, SQL filters out most of the rows using predicates, so this bottleneck is not reached until considerable parallelism has been achieved, but that bottleneck remains a barrier to transparent parallelism. If that barrier is a problem, the application must be partitioned into several parallel applications, each working on a partition of the answer set [Reuter]. To avoid this bottleneck internally, NonStop SQL horizontally partitions all intermediate answers among the processors and discs so that there are no bottlenecks within the processing of a SQL statement.

Also, little mention was made of the insert, update, and delete operators. The generalization of Figures 4 and 5 to update and delete are fairly obvious: update and delete operations can be subcontracted to the table partitions and executed in parallel. Insert is a more interesting case -- if the target of the insert is an entry sequenced table, which is common for intermediate tables and for query answer sets, the end of the sequential table is a hotspot and consequently a potential bottleneck. To avoid this bottleneck, a new kind of entry-sequenced table is introduced -- a horizontally partitioned entry-sequenced table. If requested, inserts to such a table are directed to the end of the local partition rather than to the global end-of-file. If the target file is partitioned to match the partitioning of the query executors this divides the hotspot among the executors and eliminates the bottleneck.

Sequential read and write performance is substantially improved in NonStop SQL Release 2. Sequential reading now benefits from the disc process detecting sequential access and doing asynchronous bulk read-ahead of data (up to 60KB transfers). Typically the disc process never waits for disc when doing reads -- by the time the disc process needs it, the data has been read from disc into memory. Similar logic applies to inserts, updates, and deletes. Sequential write operations are transparently buffered into 4kb blocks by the SQL executor and then sent to the disc process. This sequential block buffering typically reduces the number of messages for a sequential insert or cursor-update by a factor of twenty or more. It preserves update consistency by locking key ranges of the target table. Once the sequential insert or update data arrives at the disc process it generates a single log record to cover all the inserts or updates. This saves messages to the logging system. The sequential updates, inserts, and deletes are buffered in cache until the write-ahead-log protocol is satisfied, and until enough data has been accumulated to allow a single large transfer of multiple blocks to disc. As a consequence of this read-ahead and write-behind logic, application and disc execution on sequential reading and writing are completely overlapped, a traditional kind

of parallelism.

Parallel query execution can exploit all the processors in a cluster or a network. The database can be partitioned among nodes at London, New York, and Tokyo. In this case, NonStop SQL would automatically processes queries in parallel at all three sites. This benefit can also be a problem if the processors are supposed to be executing online transactions at the same time. NonStop SQL has two mechanisms to allow these parallel queries to run without disrupting the response of online transactions. First, the batch read queries can use BROWSE ACCESS locking which allows the query to access data without setting any interfering locks and without being stalled by the locks of others. Many batch reports can operate with this form of locking because they are doing ad hoc or statistical reporting and need only an approximate view of the database. Operations which need a consistent picture of the database can use either STABLE ACCESS locking or REPEATABLE ACCESS locking, but in this case OLTP transactions may be delayed by the batch operation.

Given all this parallelism, load control is a major issue for mixed batch and OLTP environments. NonStop SQL goes to considerable lengths to prevent a batch program from monopolizing the system at the expense of OLTP applications. First it uses prefetch and postwrite of large data transfers (up to 56KB) to minimize the number of disc IOs generated by sequential programs. In addition, these sequentially accessed pages are quickly aged in the buffer pool to prevent them from flooding the disc cache with pages which will never be referenced again.

To deal with processor scheduling, the Guardian operating system has a preemptive priority scheduler. By running batch applications at low priority, the user can assure that batch operations will be serviced only when no high-priority task is ready to execute. Traditionally there has been an anomaly in this design, called *the priority inversion problem* -- if a low-priority task asks a high priority server to perform an operation, then the request is executed at the server's high priority. This problem is especially bad in a network where a low-priority job in a lightly-loaded processor can create many requests in an already-busy processor. A classic example of the priority inversion problem can occur with the Tandem disc server (DP2) which generally runs at very high priority. DP2 has always processed its input queue in priority order -- but processes each individual request at high priority. With Release 2 of NonStop SQL, DP2 momentarily dispatches each request at the requester's priority. This allows other processes to perform their work and approximates uni-processor priority scheduling. The DP2 request is processed only when the requester's priority becomes the highest-priority ready task. Once the request is actually being processed, it is run at high priority since DP2 consumes high priority resources. On a long-running requests, the disc process "comes up for air" every few records to see if other requests are pending. If requests are pending, the disc process preempts processing of the current request. This simple strategy seems to allow NonStop SQL to mix online and batch transaction processing, using priority scheduling as the load control mechanism.

To summarize, NonStop SQL has many mechanisms detect and exploit parallelism. These include parallel query plans and algorithms, requestor-server structuring, sophisticated concurrency control, distributed transaction management, preemptive priority scheduling, read-ahead, write behind, and so on. The question is, does all this really provide linear speedups and scaleups? To establish that, we measured the actual system on some sample problems.

The SQL Release 2 Benchmark

SQL's first release benchmark demonstrated the OLTP scaleup of NonStop SQL and demonstrated higher transaction throughput than any other relational database system. The goal of the second release benchmark was to demonstrate the speedup and batch scaleup claims of the previous section on the basic relational operators. The benchmark workbook is available from Tandem Computers [NonStop SQL-4].

The benchmark database tables were all modeled on the Wisconsin Database schema [Bitton]. The rows are 200 bytes long, consisting of integer and character fields filled in with random values. The definition of a single table with n rows is given below.

```
CREATE TABLE f1( unique1    NUMERIC(8), -- unique random [0..n-1]
                 unique2    NUMERIC(8), -- primary key, unique random [0..n-1]
                 two        NUMERIC(8), -- random [0..1]
                 four       NUMERIC(4), -- random [0..4]
                 ten        NUMERIC(4), -- random [0..10]
                 twenty     NUMERIC(8), -- random [0..20]
                 onepct     NUMERIC(8), -- random [0..(n/100)-1]
                 tenpct     NUMERIC(8), -- random [0..(n/10)-1]
                 twenpct    NUMERIC(8), -- random [0..(n/5)-1]
                 fiftypct   NUMERIC(8), -- random [0..(n/2)-1]
                 hundpct    NUMERIC(8), -- random [0..n-1]
                 odd1pct    NUMERIC(8), -- random [1..(n/100)-1]
                 even1pct   NUMERIC(8), -- 2 x odd1pct
                 stringu1   CHAR(52),   -- random string
                 stringu2   CHAR(52),   -- random string
                 stringu3   CHAR(52),   -- random string
                 PRIMARY KEY unique2 );
```

In the actual definition, all fields were declared `NOT NULL SYSTEM DEFAULT` and the numerics were declared unsigned. These attributes were omitted above for the sake of brevity. All tables were transaction protected, were allocated with 4KB pages and large extents, and used row-granularity locking (these are defaults in NonStop SQL).

A data generator was constructed which builds the table partitions in parallel. Multi-gigabyte tables can be built in less than an hour. The generator uses a novel scheme for generating the random values [Englert].

The query set consisted of:

- Table Scan -- This query scans the entire table but never returns a row. It measures how quickly NonStop SQL can sequentially scan rows.
- Select 1% -- This query scans the entire table and inserts a random 1% of the rows in a target table. It measures the overhead of returning data to the application and inserting it in a target table.
- Average -- This query computes the average value of a field of the table to test the performance of aggregate operations.
- Update 1% -- This query scans the entire table and updates 1% of the rows at random. It measures the overhead of logging and locking updates.
- Join -- This query joins a table with a copy of itself via the primary key. The join includes a 1% select and a 50% project so that the target table is of manageable size (1% of the original table). The result of the join is inserted in a new table.

The queries were done in two contexts -- speedup and scaleup, and on two hardware platforms: the Tandem CLX processor -- an entry-level system, and the Tandem VLX processor -- a high-performance system. This gives a total of four curves for each query.

To give a rough estimate of the performance of these systems, each CLX/780 processor is rated at about 4tps (DebitCredit transactions per second), so the eight processor CLX system is rated at 30tps. The VLX processor is rated at 7tps so the sixteen processor VLX system is rated at over 100tps (see Figure 3).

The CLX configuration consisted of two, four, or eight processors, each with 16MB of memory and two mirrored pairs of data discs. The CLX discs each hold about 300MB of formatted data.

The VLX configuration consisted of two, four, eight, and sixteen processors, each with 16MB of memory. The disc configuration was similar to the CLX -- each processor had two mirrored pairs of data discs. The VLX discs each hold about 800MB of formatted data.

In all cases, the first cpu had an extra disc pair to store programs and the second processor had an extra disc to store the transaction log (audit trail). All disc caches were configured at 2MB.

To simplify the measurements, all processors and their discs were attached at all times, but for the two-processor cases only the first two processors and their discs were used, and so on for the four-processor and eight-processor tests. In all cases all discs were configured as mirrored volumes.

The speedup tests used a fixed size table. It was successively partitioned among 2, then 4, then 8, then 16 processors and discs. In each case the elapsed time for each query was measured and the resulting speedup curves plotted. For obvious reasons, the fixed size tables were called F2, F4, F8, and F16.

A fixed partition size was chosen for the scaleup tests. Then the table was grown from 2 partitions, to 4 partitions, to 8 partitions, to 16 partitions, each time doubling the size of the table. The growing size tables were called G2, G4, G8, and G16.

The row sizes were always 200 bytes, the number of rows per table is displayed in Table 1.

Table 1. Table Sizes		
Type	VLX	CLX
F Tables	8,000,000 rows	2,441,440 rows
G Tables	1,000,000 rows/partition	420,000 rows/partition

For example, table G16 on the VLX has 16 million rows evenly divided among sixteen mirrored discs attached to sixteen VLX processors -- the table itself has 3.2GB of data. The table F8 on the CLX consists of 2,441,400 rows partitioned among eight CLX processors and discs, each partition contains 305,180 rows.

These table sizes were chosen to allow all the tables, temporary results, and answers to fit on the disks at once. All the tables (F2,..., F16, G2,...,G16) were built, and then the experiments were run. These particular row counts were also chosen to allow the entire test suite to run within a day, so that all the tests could be audited in a reasonable time. Since there are five queries and fourteen tables (eight on VLX and six on CLX), there are seventy tests in all. So the average test had to run in a few minutes. Unfortunately, we were too optimistic, and could only audit fifty nine of the seventy tests.

The results of the insert queries were directed to an entry-sequenced table. This table was partitioned among the processors to allow parallel inserts. To reduce messages, the SQL executor buffers inserted rows into 4K blocks (this is called sequential block buffering of inserts). Writes to disc obeyed the write-ahead-log-protocol and were asynchronous. The inserts were typically transferred to the disc process in 4K blocks and to discs in 28K transfers.

The tests were run as a script fed to the interactive query interface (SQLCI). This interface optionally displays statistics on elapsed time, cpu time, rows accessed, messages sent, and so on. In addition Tandem's Measure™ performance monitor was run during the tests to measure cpu, process, message, file, and disc activity. The tests were audited by Codd & Date Consulting [Sawyer].

To simplify operations and resource control, the user must explicitly request NonStop SQL to consider parallel query plans, the default is to generate sequential plans just as in Release 1. So at the beginning of the run, the script contains the directive:

```
CONTROL EXECUTOR PARALLEL EXECUTION ON;
```

Thereafter, all compiled plans will use multiple executors if that is the fastest way to get the answer. Serial plans will still be used for queries such as single row selects or updates which get no parallel speedup or scaleup. Parallelism can be turned off by executing the directive:

```
CONTROL EXECUTOR PARALLEL EXECUTION OFF;
```

The graphs that follow are drawn from the table which appears in the auditor's report [Sawyer]:

Speedup Results

	VLX				CLX		
Partitions	2	4	8	16	2	4	8
Table Scan	1.00	1.99	3.86	7.21	1.00	1.95	3.70
Select & Insert	1.00	1.99	3.87	7.31	1.00	1.98	3.67
Average	1.00	2.02	3.86	7.51	1.00	1.97	3.86
Update	1.00	2.00	3.67	7.00	1.00	1.86	3.49
Join	-	-	-	-	1.00	1.99	3.82

Scaleup Results

	VLX				CLX		
Partitions	2	4	8	16	2	4	8
Table Scan	1.00	1.00	1.01	1.05	1.00	.99	1.02
Select & Insert	1.00	1.01	1.02	1.04	1.00	1.01	1.06
Average	1.00	1.00	1.03	1.05	1.00	1.01	1.02
Update	1.00	1.03	1.08	1.10	1.00	.98	1.04

Table 2: Speedup and scaleup ratios of all audited tests.

The graphs treat the two-processor case as a speedup of two over the one processor case, so all the speedup numbers are doubled. In addition, the CLX scaleup graphs are shown above the VLX graphs to indicate that a job runs about twice as fast on a VLX.

The first series of tests studied the speedup and batch scaleup of a table scan -- reading all the rows in a table and evaluating a false predicate against them. The actual query was:

```
SELECT  *
FROM    =table
WHERE   hundpct > ?tablesize
            FOR BROWSE ACCESS;
```

The notation "=table" is a logical table name as supported by NonStop SQL. This same query is run repeatedly, successively setting "=table" to F2, F4,...,F16, G2, G4,..., G16. The term "?tablesize" is a host-language variable set to the size of the table being scanned prior to each run. Since there is no index on hundpct, a full table scan is needed. Since hundpct is always less than tablesize, this predicate is always false and no rows are returned by the select. The query was run for tables F2, F4, F8, and F16 on the VLX and CLX to get the speedup curves in Figure 6.a. The query was run on tables G2, G4, G8, and G16 to get the scaleup curves in Figure 6.b. Since the CLX only has eight processors, F16 and G16 were not run on the CLX. The resulting graphs are:

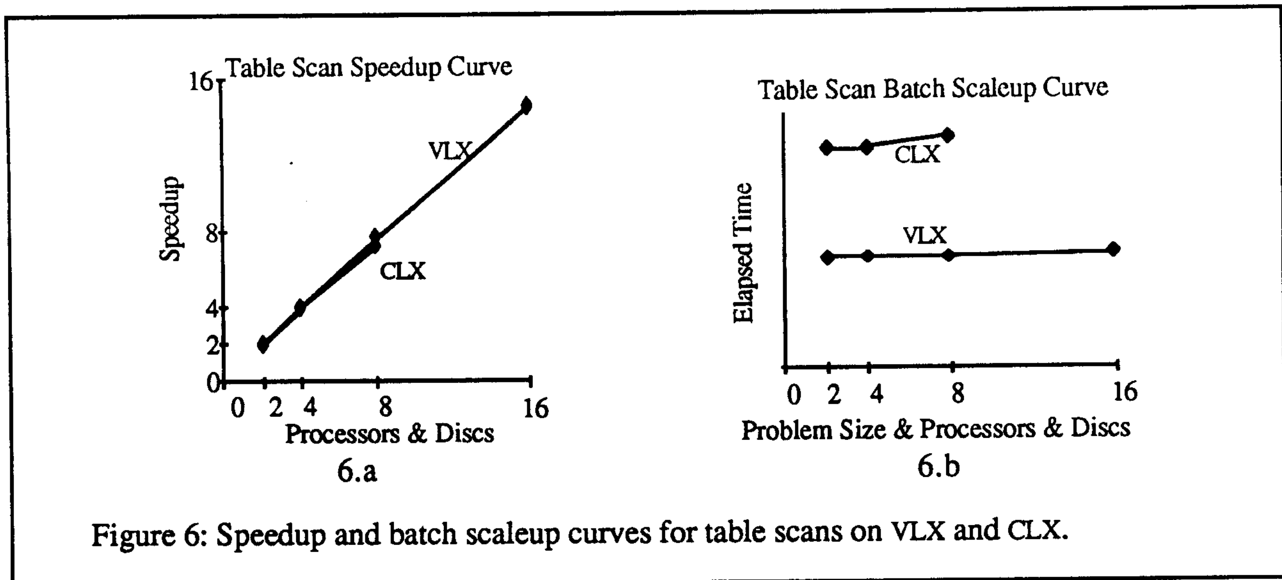

Figure 6: Speedup and batch scaleup curves for table scans on VLX and CLX.

Figure 6 shows a near-linear speedup and scaleup for both the CLX and VLX on table scans. The slight non-linearity of the CLX and VLX is caused by startup delays -- the job's elapsed time is only a few minutes at full speedup. When the job elapsed time is that short, then the startup time for the eight or sixteen SQL execution processes (a second or two per process) causes a slight slowdown of the overall job. Based on this observation, NonStop SQL will start the executors in parallel in the future, much as is done in Bubba [Smith]. On the positive side, the curves in Figure 6 do not display interference or skew problems.

Notice that the query above, and all the queries to follow have location transparency, the query does not say where the table is. The table could have been partitioned between Tokyo, New York, and London. The query also transparently requests parallelism: no special programming was required. If the data had partitions at those three cities, appropriate SQL executors would be created to scan the data at city -- if that were the fastest way to get an answer. This is transparent parallelism.

The next series of tests studied the speedup and batch scaleup of a 1% select query which inserts its output in an entry sequenced table. The result table was partitioned to exploit the parallel insert feature. The actual query was:

```
INSERT INTO =result
SELECT  *
FROM    =table
WHERE   hundpct < (?tablesize/100)
        FOR BROWSE ACCESS;
```

The hundpct column takes random values between 0 and the ?tablesize, so this query selects a random 1% subset of the table ("?tablesize" is set to the relevant table size in rows before the query is executed). The query was run for all the tables on the VLX and CLX to get the speedup and scaleup curves in Figure 7.

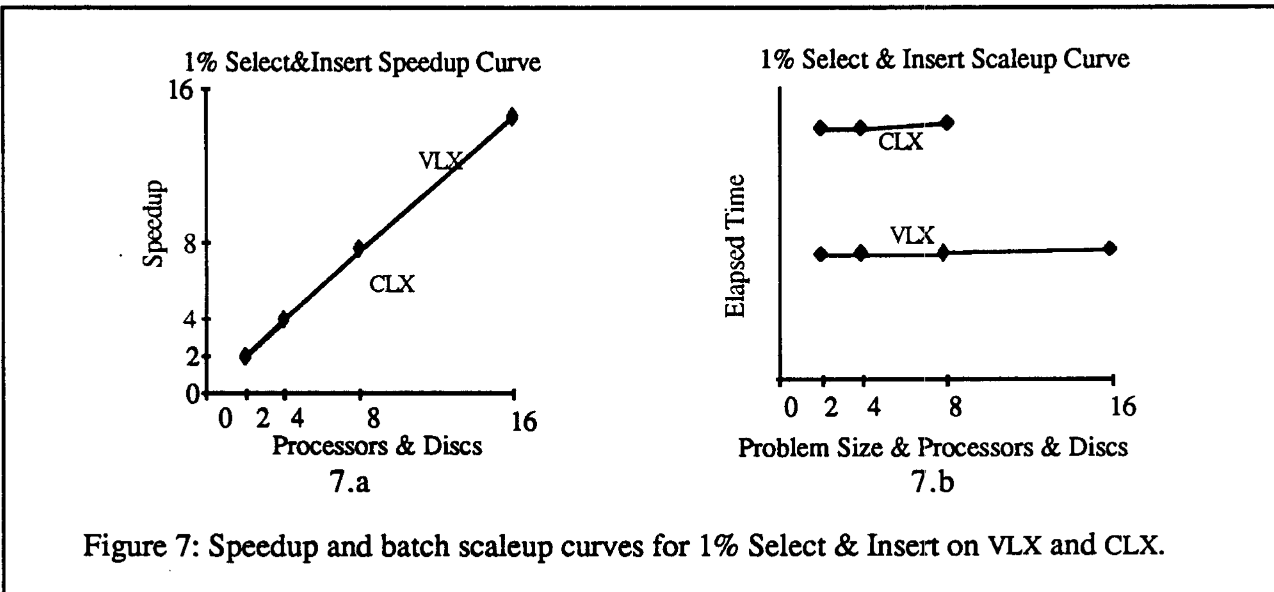

Figure 7: Speedup and batch scaleup curves for 1% Select & Insert on VLX and CLX.

As can be seen, the curves in Figure 7 show near-linear speedup and scaleup. They are virtually identical to Figure 6 -- except here the speedup curve of the CLX coincides with the VLX and the startup times cause less distortion of the scaleup curves because startup is a smaller fraction of elapsed time.

The plan chosen for this Select and Insert was to create an SQL executor at each partition/processor. Each executor asked the corresponding disc process to do a sequential scan of its partition of the table. The disc process returned 1% of the partition to the executor which in turn inserted those records into a local partition of the result table. These inserts were transparently sequential-block buffered by the SQL system so that groups of forty inserts were sent to the result table as a single message. The block of inserts generate a single log record when they arrive at the result disc process. When a large group of sequential inserts accumulates in the disc buffer pool, all these blocks are transfer to disc as a single long transfer.

The third tests studied the speedup and batch scaleup of an aggregate query, computing the average value of a numeric field of a table. The actual query was:

```
SELECT  AVG(onepct)
FROM    =table
        FOR BROWSE ACCESS;
```

The query was run for all the tables on the VLX and CLX to get the speedup and scaleup curves in Figure 8.

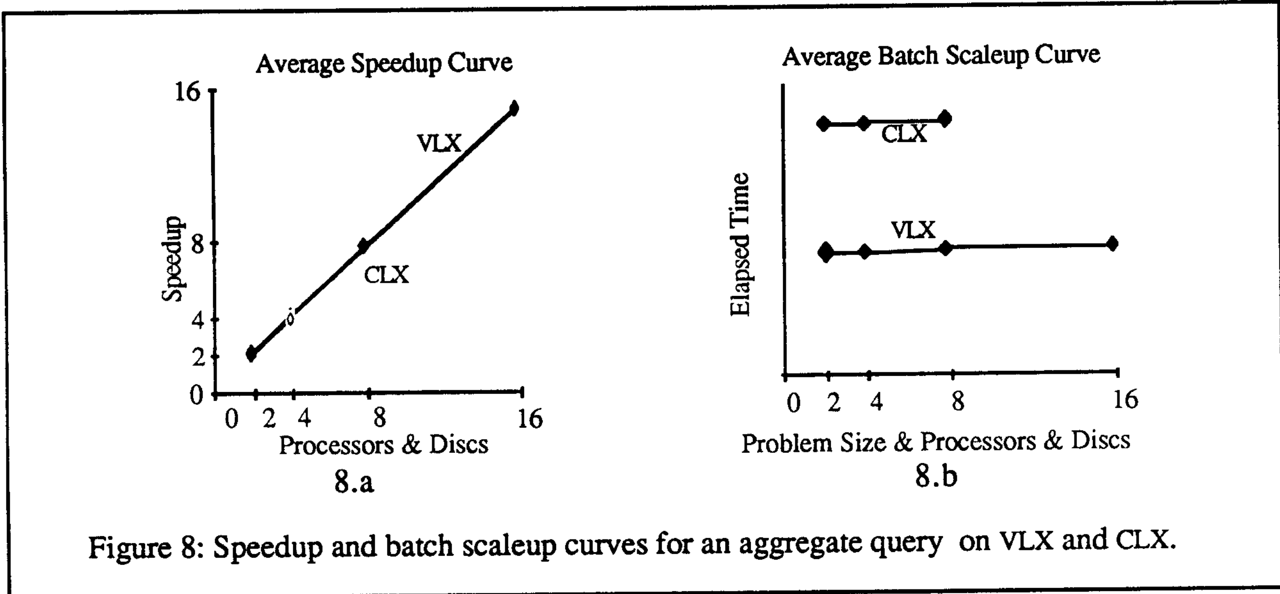

Figure 8: Speedup and batch scaleup curves for an aggregate query on VLX and CLX.

Again, these tests show near-linear speedup and scaleup. The executed plan had each processor compute the row sum and count of its partition. The, application process simply combined all these individual computations to compute the global average.

The fourth tests studied the speedup and batch scaleup of an update query, the query scanned the table in parallel and updated 1% of the rows all within one transaction. This query tests the ability for the transaction log to absorb the updates generated by 8 CLX processors and 16 VLX processors.The actual query was:

```
UPDATE  =table
SET     four = four + 117
WHERE   hundpct < (?tablesize / 100);
```

The query was run for all the tables on the VLX and CLX to get the speedup and scaleup curves in Figure 9.

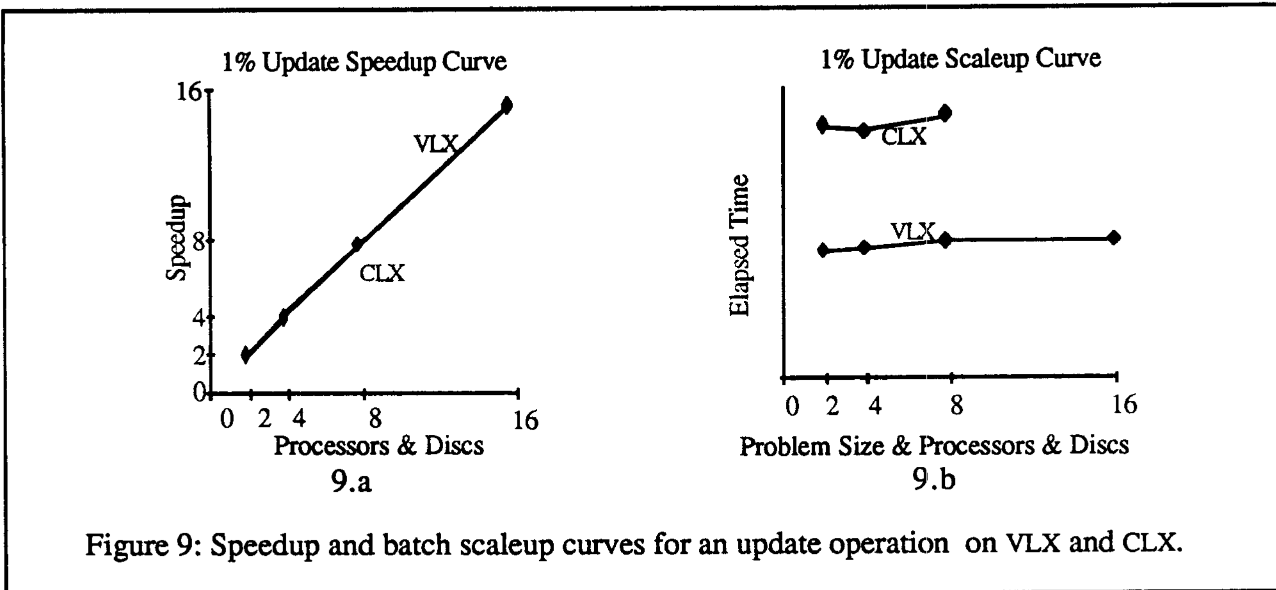

Figure 9: Speedup and batch scaleup curves for an update operation on VLX and CLX.

The curves show near-linear speedup and scaleup. The decline in the elapsed time for the 4-processor CLX case is within experimental error.

This query runs as a single transaction. It defaults to STABLE ACCESS (all rows are locked when read), but locks are immediately released if the row is not updated. This query generated about 16MB of log data on the F16 table. This did not saturate the log process and disc (in cpu 1). If this had been a 100% delete, the log process and disc would have to absorb about 4GB of audit trail. In that situation, the query would have bottlenecked on the log. Speedup and scaleup would be limited in that case.

In fact, this test shows the greatest non-linearity of any of the tests: a 12.5% deviation at sixteen processors. This indicates that the log processing (being done by cpu 1) is slowing the update work being performed by cpu 1. If this became a serious issue, then the database discs could be moved from cpu 1 so that it could be devoted to supporting the log activity.

The final tests studied the speedup and batch scaleup of a parallel join operation. In this test each of the tables F2, F4,.., G2,...,G16 were joined with copies of themselves. The join was via the primary key (unique2). In order to limit the size of the join answer, a selection expression was added which limited the qualifying tuples to 1% of the table. The result of the join is placed in a partitioned entry sequenced target table. The actual query was:

```
INSERT INTO =result
SELECT  one.unique1, one.unique2, one.two, one.four,
        one.ten, one.twenty, one.onepct, one.tenpct,
        one.twenpct, one.fiftypct, one.hundpct,
        one.odd1pct, one.even1pct, one.stringu1,
        two.stringu2, two.stringu3
FROM    =table1 one, =table2 two
WHERE   one.unique2 = two.unique2
   AND  one.hundpct <= (?tablesize/100 -1)
   AND  two.hundpct <= (?tablesize/100 -1)
        FOR BROWSE ACCESS;
```

The SQL plan chosen to execute this query in parallel is to scan the two tables (in parallel), filtering out the desired 1% of the rows. Each pair of partitions is joined in parallel and the result of that mini-join is inserted into the result table. The insert operation is similarly partitioned. This is a completely parallel operation and so near-linear speedup and scaleup are expected.

Our plan was to execute the above query for all tables and on both the CLX and VLX platforms. Unfortunately, the previous tests consumed our benchmark window. So we were only able to run and audit the CLX speedup tests.

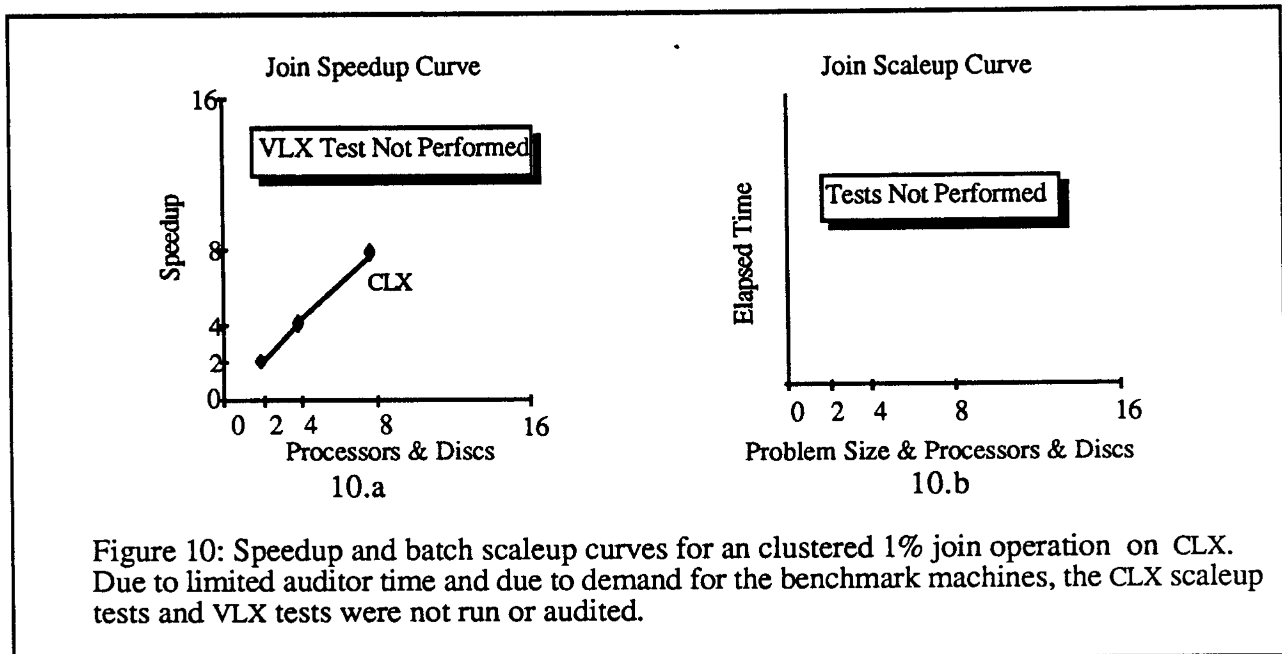

Figure 10: Speedup and batch scaleup curves for an clustered 1% join operation on CLX. Due to limited auditor time and due to demand for the benchmark machines, the CLX scaleup tests and VLX tests were not run or audited.

This one test is quite promising. We believe the other cases will be similarly linear and will be tested soon.

Why Does NonStop SQL Get Near-Linear Speedups and Scaleups?

These results contradict the folklore that multi-processors don't give linear speedup or scaleup. People have been building multi-processors for thirty years, and each one has had difficulty scaling past ten or twenty processors. This folklore ignores the distinction between conventional shared-memory multi-processors, shared-disc multi-processors, and Tandem's shared-nothing design [Bhide] [Stonebraker]. Figure 11 contrasts these three designs diagrammatically.

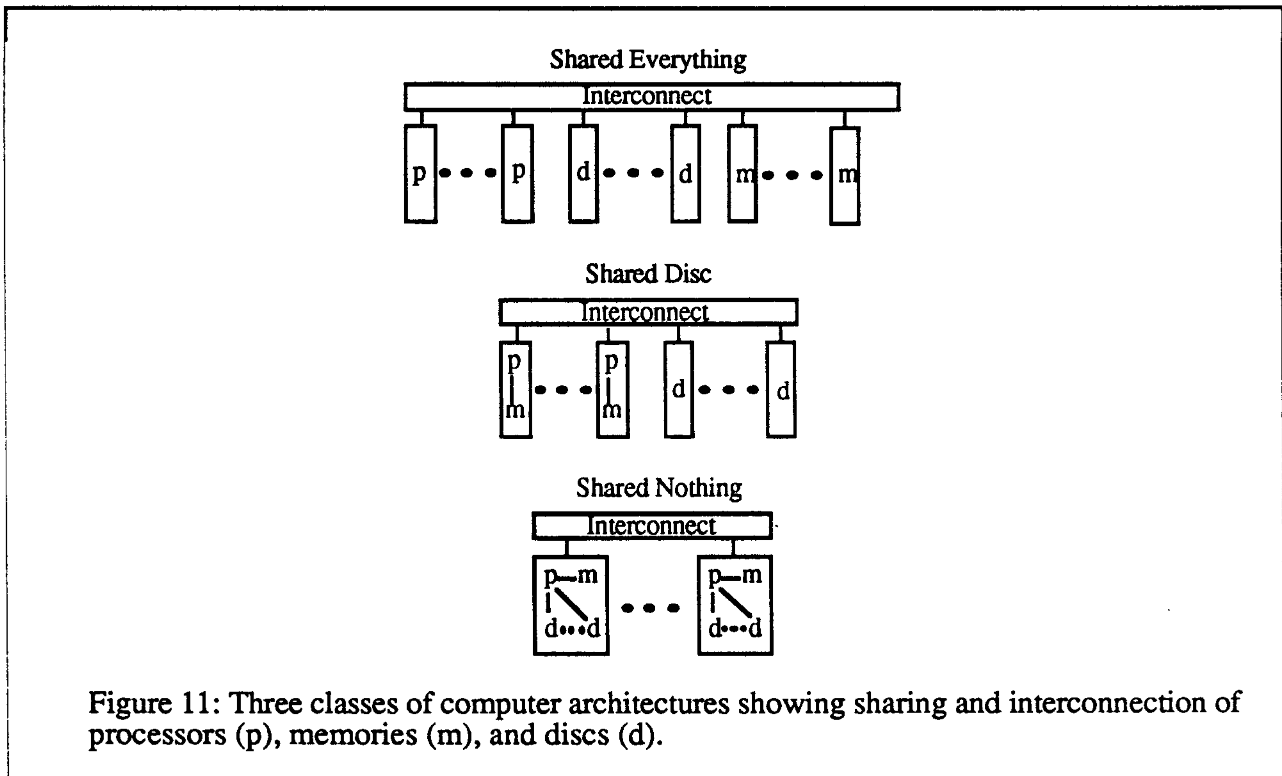

Figure 11: Three classes of computer architectures showing sharing and interconnection of processors (p), memories (m), and discs (d).

In a shared-everything design, all processors can access all memories and discs. This design is typified by the IBM 3090 which scales to six processors. A shared-everthing design has inherent speedup and scaleup problems because all traffic must pass over the interconnect. The interconnect becomes a bottleneck either from contention or from physical constraints like the speed of light which limits the interconnect size.

The first step in dealing with the interconnect bottleneck is to partition the memory among the processors (or groups of processors), but share the discs among all processors. Such a design is called a shared-disc architecture and is typified by the DEC VaxCluster. This design reduces many of the interconnect problems, but still has severe problems with disc cache interference.

A shared-nothing design completely partitions processors and discs, letting them communicate only via high-level (SQL level or application level) messages. This greatly reduces interconnect traffic and eliminates the cache invalidation problem. It achieves this reduction by moving programs to the data rather than data to the programs. Only answers come back to the programs. Typically a great deal of data is examined to produce one answer. The benchmark here gave some examples of this approach. The Tandem NonStop system typifies a shared-nothing design.

Previous research prototypes such as Gamma at University of Wisconsin [Dewitt-1], and Bubba at MCC [Smith] have demonstrated near-linear speedups. Teradata has made similar demonstrations

on special purpose hardware [Teradata]. All these systems are shared-nothing designs like Tandem's. So the results presented here can be viewed as an application of the ideas pioneered by these other groups to a commercially-available general-purpose shared-nothing machine.

We believe that a shared-nothing architecture is the key to getting such speedups and scaleups. Shared-disc and shared-memory systems have consistently displayed bottlenecks due to interference when tens or hundreds of processors are connected to solve a single problem. The results presented here certainly show that shared-nothing does work. There is a thirty year history of failures for the shared-disc and shared-memory designs. But that does not prove that such designs can never be made to give linear speedups and scaleups -- it just shows that it's much harder to make them scale.

Summary and Future Directions

The results presented here demonstrate linear speedup and scaleup of NonStop SQL on the basic relational operators. No skew or interference problems were observed in the benchmark. Minor startup problems were observed, but a solution to them is well understood. These speedups and scaleups were observed for both the Tandem CLX and the Tandem VLX. The scaleups were done to a 16 processor system with no visible bottlenecks. The results were audited by Codd and Date Consulting [Sawyer].

This second release of NonStop SQL is the first installment on intra-transaction parallelism. Work is currently progressing on making all aspects of the system have good speedup and scaleup properties, as well as being incremental and online. Data Definition operations such as adding, altering or dropping columns, adding assertions, adding referential integrity constraints, or adding indices can all benefit from parallelism. Additional join algorithms are being implemented and experiments are being done in structuring highly parallel applications [Reuter].

Acknowledgments

The parallel features of NonStop SQL were designed and implemented by Yung-Feng Kao (parallel partition inserts), Haleh Mahbod (SQL language), Mark Moore (parallel index maintenance), Carol Perarson (parallel sort), Mike Pong (parallel optimization), and Franco Putzolu and Amardeep Sodhi (parallel executor). The sequential read and write optimizations were designed and implemented by Andrea Borr (disc process) and Julia Lai and Harjit Sabharwal (file system). To their credit, the pre-alpha software we tested worked without problems.

The benchmark used the facilities of the Cupertino Benchmark Center. Steve Shugh was our liaison, Jeff Maturano and Phill Rose assembled our VLX and CLX. Ray Glastone helped with some of the performance reports.

Tom Sawer of Codd and Date was our Auditor.

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.

[Bhide] A. Bhide, "An Analysis of Three Transaction Processing Architectures", Proc. 14th VLDB, Sept. 1988, pp. 339-350

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

[Brooks] F. Brooks, *The Mythical Man Month -- Essays on Software Engineering*, Addison Wesley, 1982.

[Cassidy] J.Cassidy, T.Kocher, "NonStop SQL: The Single Database Solution -- State of California Department of Motor Vehicles Performance Benchmark", Tandem Computers Part no. 300102, April 1989.

[Dewitt -1] D. Dewitt, et. al., "Gamma - A High Performance Dataflow Database Machine",Proc. 12th VLDB, Sept. 1986, pp. 228-236.

[Dewitt -2] DeWitt, D., et. al., "A Performance Analysis of the Gamma Database Machine", Proceedings of the 1988 ACM SIGMOD Conference, June, 1988.

[Englert] S. Englert, Gray, J., " Generating Dense-Unique Random Numbers for Synthetic Database Loading", Tandem Technical Report (in preparation).

[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.

[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.

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

[NonStop SQL-1] Tandem Database Group, "NonStop SQL, A Distributed, High-Performance, High-Availability Implementation of SQL", Tandem Technical Report 88.2, Cupertino, CA, April 1988, revised May 1989

[NonStop SQL-2] *NonStop SQL Benchmark Workbook,* Part No. 84160, Tandem Computers Inc, Cupertino, CA, March 1987.

[NonStop SQL-3] Tandem Performance Group, "A Benchmark of NonStop SQL on the DebitCredit Transaction", SIGMOD 88, ACM, June 1988.

[NonStop SQL-4] *NonStop SQL Release 2 Benchmark Workbook,* Courier Order No. 103900, Tandem Computers Inc, Cupertino, CA, June 1989.

[Reuter] A. Reuter, et al, "Progress Report #5 of Prospect Project", Institute of Parallel and Distributed Super-Computers, Univ. Stuttgart, May 1989.

[Sawyer] T. Sawyer, "Auditor's Report on NonStop SQL Release 2 Benchmark", Codd and Date Consulting, San Jose, CA., June 5 1989.

[Schnieder] D. Schnieder, D. Dewitt, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment", Proc. 1989 ACM SIGMOD, May 1989, pp 110-121.

[Smith] M. Smith, et al., "An Experiment on Response Time Scalability", Proc. Sixth Int. Workshop on Database Machines, June 1989

[Stonebraker] M. Stonebraker, "The Case for Shared-Nothing", Database Engineering, V.9.1, Jan. 1986.

[Teradata] "The Genesis of a Database Computer: A Conversation with Jack Shemer and Phil Neches of Teradata Corporation", IEEE Computer, Nov. 1984. or *DBC/1012 Database Computer System Manual, Release 1.3*, C10-0001-01, Teradata Corp., Los Angeles, Feb. 1985.

Distributed by

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