NCR 3700 — The Next-Generation Industrial Database Computer

Andrew Witkowski, Felipe Cariño, and Pekka Kostamaa

NCR/Teradata Advanced Concepts Laboratory (TACL)

100 N. Sepulveda Blvd. El Segundo, CA 90245

Abstract

The ***NCR 3700*** is our next-generation general purpose computer. Its design benefited from 10 years of Teradata DBC/1012 industrial experience, with which we hold a 70% market share in commercial VLDB applications. The NCR 3700 will be sold as a large parallel Unix system, and as a database computer. The NCR 3700 can run third party merchant databases, as well as the Teradata relational DBMS. Readers should keep in mind that all the design decisions assume VLDB (multi-terabyte) size applications. The NCR 3700 and the DBC/1012 have a close symbiotic relationship between the hardware and the software. The operating system and hardware contain features and extensions used by the database to exploit parallelism. This paper describes the NCR 3700 architecture, hardware capabilities, the *BYNET*™ interconnection network, the Unix/NS operating system, and the Teradata relational database. We describe our notion of a *Trusted Parallel Application (TPA)* that runs under our extended Unix operating system, *Unix/NS*. The most important TPA is the *Teradata relational database*. Real-world VLDB systems have difficult system administration problems. We conclude the paper by describing one of these problems, namely, the loading of massive amounts of data. The NCR 3700 computer and database software were designed to efficiently run new multi-terabyte level VLDB applications well into the 21st century.

1 The NCR 3700 Hardware Overview

The NCR 3700 exploits recent technology advancements in hardware and software. The software design utilizes new developments in parallel operating systems that exploit the NCR 3700 architecture, BYNET interconnection network, RAID [11] disk array storage and multi-processing at the board-level. The NCR 3700 primary design goal was to linearly scale-up CPU and I/O utilization as the systems grows. The NCR 3700 was designed to service the massively-parallel UNIX and Teradata's very-large database market. The largest DBC/1012 [8] database application is a mission critical 2-Terabyte database application; DBC/1012 case studies are documented in [4]. The NCR 3700 will be sold as a database computer [2, 5, 7, 8] as well as a massively-parallel UNIX system. Each processor module runs Unix/NS, an enhanced UNIX™ SVR4 operating system. This paper concentrates and emphasizes the NCR 3700 as a relational database machine for VLDB applications.

The NCR 3700 (Figure 1) basic building blocks are *Processor Module Assemblies (PMAs)*. The NCR 3700 has a single processor type, the PMA. The PMA is a processor board with four tightly-coupled processor units. The first PMA generation uses a quad 50MHz Intel 80486 CPU, and 512MB of memory. Each CPU has an on-chip 8KB cache and a second level 256KB cache. The channel and network attachments are through the Micro Channel Adapter (MCA) bus allowing us to connect to the IBM Channel as well as a variety of network hardware. Every PMA is logically composed of six boards: (1) a 4 CPU processor board, (2) SCSI host adapter board, (3) BYNET interface controller with a dedicated SPARC processor, (4) memory board, (5) power board, and (6) a board reserved for future use. The PMA boards share a memory bus, called the JDBus, which is capable of 200 MB/sec peak performance.

 Proceeding of the 19th VLDB Conference, Dublin, Ireland, 1993.

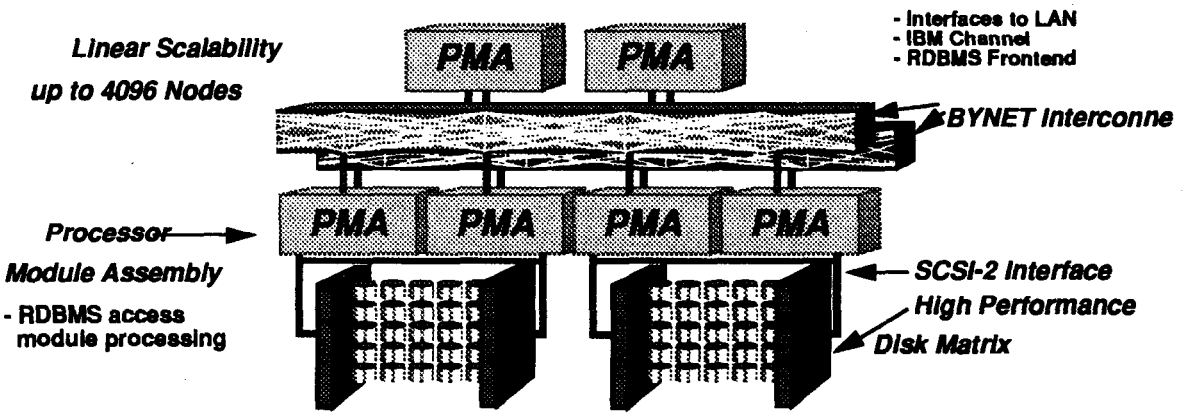

Figure 1: NCR 3700 Architecture

The disk arrays are connected to the PMA via a standard SCSI-2 interface. The disk array has dual access paths, including dual controllers. The data is stored on the disk array using a modified RAID logic, which can survive a single disk failure. The disk array can be configured with hot spares that are used to reconstruct the data of a failed drive. The initial disk array matrix is built using 1.6GB drives configured 5 deep and 6 wide giving 48GB of unformatted capacity. Future disk arrays will have more disks and support 99GB of unformatted capacity.

The NCR 3700 is configured as a series of *cliques*. A clique is a collection of PMAs and RAID disk arrays that are connected by the same bus controller, and thus can communicate directly. The simplest fault-tolerant clique configuration has two PMAs connected to a RAID disk array. In this configuration, when a PMA fails there is still access to the data via the second PMA. Determining the optimal clique configuration mix (ratio of PMAs to disk arrays) at this time, is a black art that involves an application specific complex trade-off between fault-tolerance, price and performance. The external requesting clients to the NCR 3700 are connected through various local/wide area networks and protocols such as: Ethernet, FDDI, Token Ring, TCP/IP, ISO/OSI, XNS and X.25. Mainframe clients are connected and served via streaming channels connected to the PMAs.

The DBC/1012 uses the *YNET*[16] interconnection network. The YNET is a broadcast based, 6MB/sec bandwidth interconnection network, with a hardware-based sorting mechanism, and global synchronization. The YNET design assumption was that the database software will broadcast messages to processors owning the data, but in practice a significant amount of messaging turn out be point-to-point. The YNET bandwidth is not scalable to the maximum 1000 processor board configuration. Based on these experiences, a new interconnection network was designed, the BYNET™ [10]. The BYNET preserved and enhanced YNET features, and addressed the scalability limitation.

The *BYNET*™ is a scalable, Banyan topology, multistage interconnection network that supports broadcast, multicast and monocast messages and is internally comprised of 8x8 crossbar switch nodes. Each PMA has two optical fiber connections to the BYNET. Each BYNET connection has a full-duplexed 10MB/sec bandwidth; hence two BYNETs per PMA provide 20MB/sec throughput. The BYNET is architected to interconnect up to 4096 NCR 3700 processor modules; thus the maximum monocast throughput is 40 GB per second per BYNET, minus contention. Both simulation and hardware analysis of a 64 processor module configuration exchanging 4KB messages at random indicate that the sustainable throughput is about 2/3 of the maximum. The *BYNET monocast messages* are two-way communication links established between two PMA processors. The requesting PMA processor communicates on a high capacity forward channel, while the responding processor can reply using a lower capacity backchannel.

The *BYNET broadcast* and *multicast* messages are two-way communication trees established between the sender PMA and the receiving PMAs. The BYNET supports both blocking and non-blocking protocols. The sender sends the data down the tree on the high capacity forward channels. The receivers respond on the low capacity backchannels. The responses are combined at the sender using one of the predefined combining functions. The combining functions are implemented in BYNET hardware/firmware and thus execute quickly. Selection of the combining function occurs when the communication tree is established. For broadcast or multicast the combining function selects the lowest response. Successful receivers respond with 1 and unsuccessful with 0 (unsuccessful here means that the software detected a failure, for example lack of memory resources to receive the message). Thus the sender knows the status of the broadcast. It then sends, on the forward channel, a commit or abort broadcast signal. There are many possible PMA failures. Two cases deal with PMA failure before query execution, and failure while a query is executing. When a PMA fails, the optical connection is broken, the BYNET reconfigures the PMA out of the network. If the PMA was running a transaction a recovery process is initiated, and if possible, the processes are migrated to the another PMA within the clique.

The BYNET provides a message sorting function used by the Teradata database for sorting data returned by individual senders. During the BYNET sort operation participating senders prepare messages consisting of two parts: the sorting key and the actual data. BYNET constructs a merging tree that spans all senders and whose root is the processor that initiated the sort, and passes up the tree messages in a sequence sorted by the key. Unlike YNET, however, the sorting function is implemented in software which gives us a greater flexibility in selecting the length of the sorting key (YNET limited the key to 512 bytes) and the sorting order. Software sorting does not impact the application executing on a PMA since BYNET on each PMA has a dedicated SUN SPARC processor. The BYNET also provides global synchronization facilities between PMAs, such as counting semaphores, and using monocast messages. Readers are referred to [3,4,5] for a more in-depth description and comparisons of the BYNET, DBC/1012 and NCR 3700.

2 Unix/NS - OS for Distributed Memory DBMS

The NCR 3700 Operating System, called here ***Unix/NS***, is based on the Unix SVR4 OS. It contains significant extensions for massively parallel systems, in particular Distributed Memory DBMSs. The extensions include the concepts of virtual processor and virtual disk, message and global synchronization system, segment system, and globally distributed objects. When compared to other parallel UNIX operating systems like Mach [13] or Chorus [14], Unix/NS has a more powerful communication and message addressing paradigm, and richer process-group management and global synchronization mechanism.

2.1 *Virtual Processors*

Virtual Processor (Vproc) and *Virtual Disk (Vdisk)* are abstractions that give an application, like the Teradata database, the illusion of having dedicated processor and disk devices. In Unix/NS a Vproc is an addressable collection of processes that can share resources such as memory segments, mailboxes, monitors, and files. Vprocs use a Vproc ID to identify resources acquired and processes performing Vproc operations. The Vproc ID is used to migrate processes, and thus sharing resources between Vprocs is not allowed. Virtual processors are typed, which allows us to group Vprocs which serve a common purpose; for example: Vprocs that manage the DBMS disks, Vprocs that parse queries, or Vprocs the interface with networks. Vproc type grouping allows *group-wide operations* such as broadcasting to Vprocs of type A and collecting performance data from Vproc of type A. A *Trusted Parallel Application (TPA)* is a collection of Vprocs that perform operations on an application. TPAs provide an isolation mechanism for executing more than one parallel application on the same machine. The Teradata database is a (special) TPA, but other parallel TPAs can also execute on the NCR 3700. Unix/NS provides the mechanisms to give every TPA Vproc the illusion that it operates on a dedicated parallel machine. Global operations like

such a failure took the entire logical disk (i.e., all disks connected to a processor) off-line. Vprocs can reduce the scope of a failure to the granularity of a single physical disk.

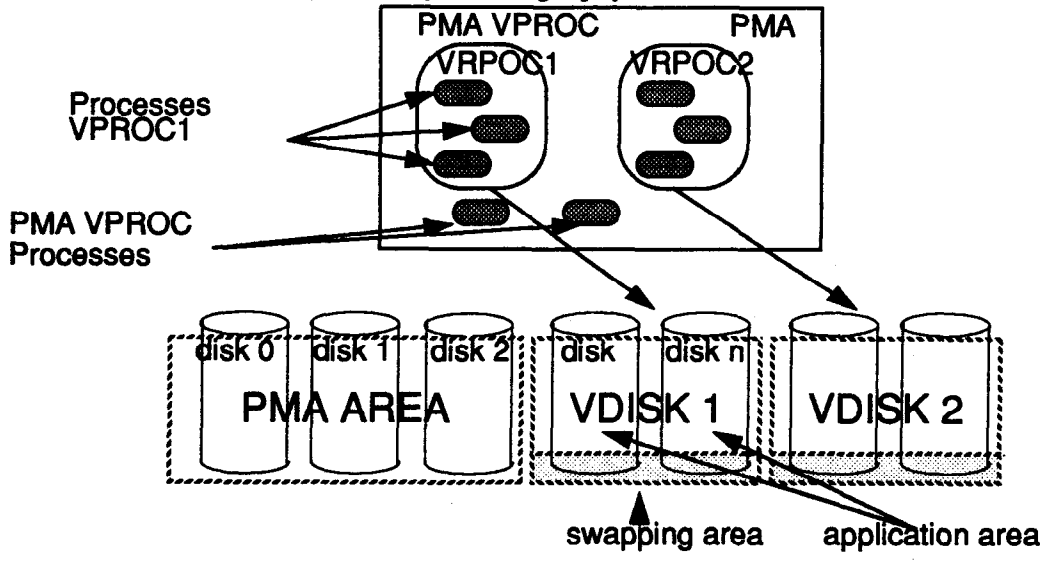

Figure 2: Vproc and Vdisk Diagram

Unix/NS provides a facility for adding PMAs and disk arrays to the system. Reconfiguration is a two step/ process: add hardware and repopulate system. New hardware is added and new disks containing the PMA Area are populated with OS software. The system administrator updates Unix/NS configuration files indicating the number and type of new Vprocs, the cliques where they are to execute and new LAN and channel addresses for the outside connections. Unix/NS then creates the Vprocs and empty Vdisks and replicates permanent GDOs on the new PMAs. The second step is TPA specific. The Teradata database TPA redistributes its data evenly among all Vdisks in the configuration.

2.3 *Global Distributed Objects*

Preventing a Vproc from sharing resources may lead to their inefficient resource utilization and replication. Unix/NS provides *Globally Distributed Objects (GDOs)* to eliminate such replications. A GDO is a memory object that is shared by all Vprocs. A GDO can be transient, in which case it does not survive restarts and must be initialized when the system comes back up, or it can be permanent in which case all changes to it are preserved across system restarts. Transient GDOs are used for structures which can change at each system restart, such as the system configuration tables. Unix/NS guarantees that each GDO starts at the same virtual address on all Vprocs, so that when a Vproc is migrated, all references to a GDO remain valid. Unix/NS provides a locking based, concurrency control mechanism for reading and writing of GDOs. The mechanism includes deadlock prevention, such that the offending reader or writer is notified if its operation would have caused a deadlock. It is left to the application to handle this. Unix/NS guarantees atomic updates of GDOs, so that the copies on all PMAs are always consistent in the presence of failures.

2.4 *Memory System*

Unix/NS provides memory management services, the *segment system,* for sharing memory between processes of a single Vproc, and for providing a segmented view of the vdisk space. This exists above and beyond Unix memory services. Two salient features are: (1) built-in concurrency control, and (2) mirroring of critical segments. Each allocated memory segment has a locking structure. Read locks, write locks and a version of intentional write locks are supported. Teradata database uses these segment system locking

The table is logically ordered within the AMP by the rowids which serve as the clustered index. The index is organized as a three-level B+ tree structure, where keys of the tree are the rowids. Usage of rowids as a key precludes usage of the index in range qualification. The organization of a non-clustered index, called here the *secondary index*, is the same as that of a data table. We refer to it as the *index table*. A row in the index table consists of the rowid, the index columns, and a column containing rowids of the corresponding rows in the data table. Rowid is constructed as before. Index columns are simply the columns on which the index hash has been declared. Within an AMP the index table is logically ordered by rowid again using a three-level B+ tree structure. There are three types of secondary indexes.

The *Unique Secondary Index (USI)* and the *Hashed Non-Unique Secondary Index (HNUSI)* are hash partitioned indexes where the index columns serve as the partitioning key. Locating a row using a USI value *V* involves at most two AMPs: one that owns the index row which it is found by applying the hash function to *V*, and one that owns the data row. The index row will contain the rowid of the data row, and thus the address of the AMP owning the row. Using HNUSI may involve more than two AMPs since the index row may point to more than one data row, each of which may reside on a different AMP. HNUSI is intended for non-unique columns where the average number of synonyms is small in comparison to the number of AMPs in the system.

The *Non-Unique Secondary Index (NUSI)* is a non-partitioned index. For each AMP it contains the rowids of the data rows owned by that AMP. Notice that in order to locate a set of data rows using NUSI value *X* all AMPs have to be involved. This is because NUSI is a non-partitioned index, and it is not know in advance which AMPs own rows whose non-partitioning columns have value *X*. If a table is declared to be a fallback table, all its secondary indexes are also protected by the fallback mechanism. Observe that the mechanism for fallback maintenance for the tables and indexes is the same since indexes have the same structure as the data tables. Observe that USI and HNUSI offer a very desirable scalability property of the index lookup load. Provided the same arrival rate of USI or HNUSI lookup operations, increasing the number of AMPs decreases linearly the index lookup load per AMP. NUSI does not have this property.

3.2 *Query Planning*

A query is parsed and optimized in the virtual IFP by the parser. The parser will first verify whether the query exists in the database buffer. The database buffer retains 300 of the most recently parsed and optimized queries. The query has to pass several qualifications to be able to use the cached steps. These include exact match of the query text and equivalent host types from where the query originated. If a query can re-use cached steps, the rest of the parsing and optimization is bypassed. The input to the parser is the user query, which contains one or more SQL statements. The parser translates this query into steps, which are executed by the AMPs. When optimizing a query that involves joining several tables, the optimizer uses a strategy known as the *Greedy Algorithm* with a one-table look-ahead. The optimizer does check for certain special cases, to avoid producing plans where the Greedy Algorithm generates sub-optimal join plans. All join plans are composed of binary joins.

The Greedy Algorithm starts the join with the two tables that are cheapest to join. With one-table look-ahead, the optimizer finds from all the possible tables the three tables that are cheapest to join. These three tables are joined first. The Greedy Algorithm is used to limit the number of combinations that the optimizer has to evaluate, since the number of possible join orders grows as a factorial of the number of tables in the query. As an example, there are 362,880 (= 9!) possible orders of 9 tables. The greedy algorithm with one table look-ahead limits these to 9x8x7 = 504 choices. Once the first three tables in the join order are picked, the next three are picked from 6x5x4 = 120 choices, and the last three are picked from 3x2x1 = 6 choices. The total number of combinations the Greedy Algorithm looks at is, therefore, 630. Note that the look-ahead join might or might not select the first join.

To estimate the cost of joining two tables, the optimizer uses statistics about the tables which can be explicitly collected by the user. If statistics were not collected on the table, the optimizer will estimate table demographics by random sampling of the table data blocks. A message is sent to a random AMP which counts the number of data blocks, then a random block is select. For this random block, the number of rows in the block are counted and returns an approximated row count to the optimizer. The optimizer caches the count for future use. If the user has collected statistics on the involved tables, the join plan will always be consistent. The cost model used minimizes the total resource utilization, assuming exclusive use of the system. The optimizer minimizes the sum of the CPU utilization, the disk array utilization (calculated using maximum disk array throughput), and BYNET utilization.

The optimizer determines whether using any indexes, either primary or secondary, is more efficient than a full file scan. Any table selection and projection can be done by doing a full file scan. Depending on the query constraints, and the available indexes, the optimizer might choose to access the table using these indexes. Depending on the join clause, there are several options that the optimizer evaluates in choosing a join method. Assume that we are doing an equi-join on two tables, table S that is the smaller of the two tables, and table L that is the larger of the two table. If the join columns are primary indexes of both tables, no redistribution is required. In this case, all the rows in the two tables that are likely to match already exist on the same AMP. If the join column of either or both tables is not the primary index, the optimizer will normally choose from the following binary join plans, even though several other are also evaluated:

(1) Project, select, and hash redistribute by the join column table S into a spool file.
Project, select, and hash redistribute by the join column table L into a spool file.
Perform a product (nested-loop) join of the above two spool files.

(2) Project, select, hash redistribute and sort by the join column table S into a spool file.
Project, select, hash redistribute and sort by the join column table L into a spool file.
Perform a merge join of the above two spool files.

(3) Project, select, duplicate, and sort table S by the join column into a spool file.
Project, select, and sort table L by the join column into a local spool file.
Perform a merge join of the above two spool files.

Option 1 is used normally when the small table S has only a few rows per AMP. Option 2 is used when both tables S and L have a large number of rows. Otherwise, option 3 is used. The figure below shows what binary join plans the optimizer chooses if the join columns are not indexed. The results of the join plans are from an analytical model done to study the effect of different optimizer cost formulas.

3.3 *Query Execution*

The parser and optimizer break down the original SQL query into several basic operations, called *steps*, to accomplish the desired result. The dispatcher is responsible for sending these steps to the AMPs. Based on the query, the steps can be either: (1) all-AMP steps, which are received and processed at all the AMPs, or (2) be targeted to a single AMP, or (3) be targeted to a Vproc-group of AMPs

As an example of a query that is executed by a group of AMPs, consider a simple SELECT:

SELECT * FROM TEST_TABLE WHERE USI = 1;

In this example, assume that column USI is a unique secondary index. As explained previously, the unique secondary indexes are hash redistributed by the index value. The parser and optimizer generate an AMP step that is dispatched to the AMP that contains the USI row value. The USI row value contains the hash code and rowid of the primary data row, which most likely resides on another AMP. An AMP to AMP step is sent to the primary row AMP, which joins this transaction's Vproc-Group. Thus, queries such as this example will scale with the size of the system, as more PMAs (and AMPs) are added.

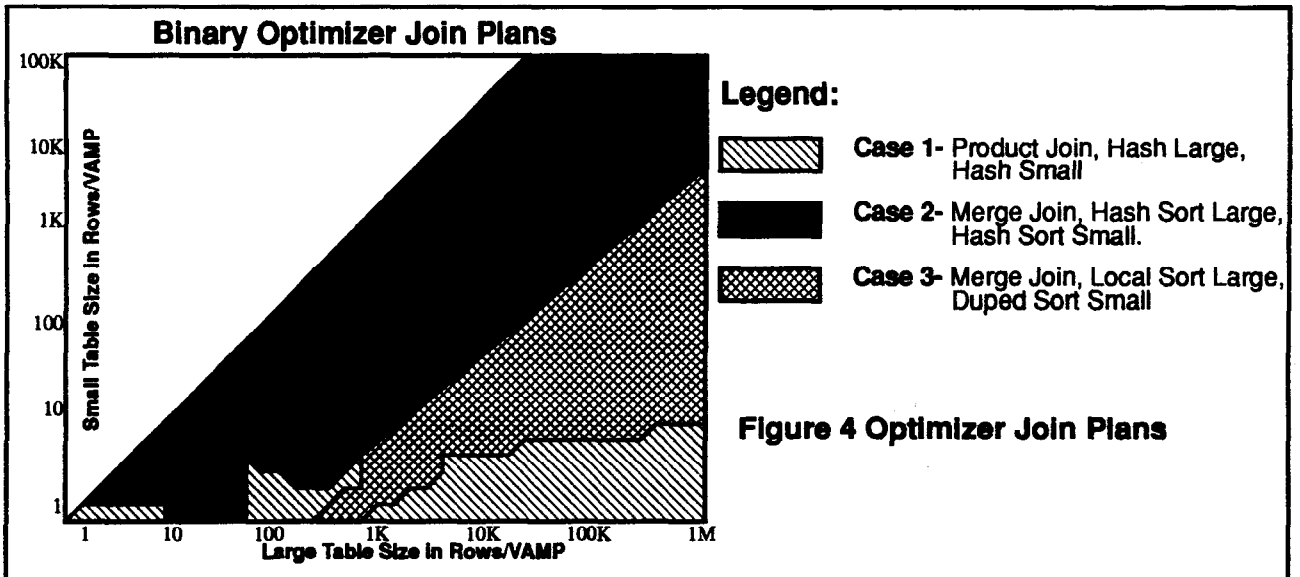

Figure 4 Optimizer Join Plans

Modelled query: **SELECT * FROM LARGE_TABLE, SMALL_TABLE WHERE LARGE_TABLE.NON_KEY_COL = SMALL_TABLE.NON_KEY_COL;**

The following join example (from Figure 4) is used to demonstrate how the AMP steps are generated.

SELECT * FROM LARGE_TABLE, SMALL_TABLE WHERE

LARGE_TABLE.NON_KEY_COL = SMALL_TABLE.NON_KEY_COL;

Assume the row sizes in the two tables are such that Case 2 of the Figure 1 join plan is chosen. This join plan contains 6 steps: Step(1) A lock table step to place a read lock on both tables. Step(2) A retrieve step to project and select rows from the LARGE_TABLE, redistribute the rows using the join column, and sort the rows by the join column, before writing them out to a spool file. Step(3) A retrieve step to perform exactly the same functions on the SMALL_TABLE that were done for the LARGE_TABLE in step 2. Step(4) A join step to perform a merge join of the two spool files generated in steps 2 and 3 into another spool file. Step(5) A retrieve step to read and return the results to the user. Step(6) Finally, an end-transaction step to release the locks, and clean up any spool files. The end-transaction step is sent only to the AMPs involved in the transaction, i.e., to the AMPs in the Vproc-group for the transaction. This avoids the overhead of processing the step on non-involved AMPs.

Our experience with the early DBC/1012 showed that sending this step to all AMPs decreased throughput of OLTP transactions that normally touch one or two AMPs by a factor of two or more, thus the concept of the Vproc-group was introduced. In this example, steps 2 and 3 would be dispatched in parallel to all the AMPs. Before steps 2 or 3 can start sorting the rows, an intra-step synchronization point has to be reached. This synchronization point causes the other AMP steps to wait until the last AMP step is done redistributing the rows. Using the operating system messaging calls and coordination channels, all the AMPs issue a message call, requesting to know whether it is the last AMP to finish. The last AMP to finish will send a message to all the other AMPs to start the sort. In step 5, the final merge of the result rows is done in the BYNET controller. This merging in the BYNET controller is done to achieve compatibility with the original DBC/1012 Ynet interconnection network, which performs the merge operation in its' hardware logic. Each AMP runs a copy of the lock manager. The lock manager handles locks for database objects in its own AMP. Locks are placed at three granularity levels: *database* level locks, *table* level locks, and *hash code* level locks. Since the hash code is a 32 bit value, the hash level locks are fine grain.

There is no escalation of locks. Should the lock table overflow, the transaction is aborted. Global deadlock detection is done in one of the IFPs. Periodically, the AMP lock managers pass their local wait-for graphs to the global deadlock detector. Using these wait-for graphs, system wide deadlocks can be detected, and the newest transaction is aborted.

Transaction logging uses before image logging. This is done locally at each AMP. Before a row is updated, the before image copy is written out to the log, called the transient journal. This transient journal is written out to stable storage at transaction commit time together with the end of transaction marker. On the 3700, this means copying the transient journal to the backup PMA using file system mirroring, as explained earlier. Since the system does not use write ahead logging, at transaction commit time the datablocks are also forced out to stable storage. To commit a transaction across multiple AMPs, a modified version of two-phase commit is used. An end transaction step is broadcast to the participating AMPs. Each AMP will place an end transaction entry in the transient journal. Using the system provided channeling mechanisms, all AMPs report the success (or failure) of their operations. The last AMP to reach this point forces the second phase of the commit. In case of a system crash, the recovery process on each AMP scans its transient journal. A transaction with end of transaction marked on at least one AMP is subject a to commit, otherwise it is subject to an abort. One AMP, designated as the recovery coordinator, collects transaction votes from each AMP and determines the fate of each active transaction.

3.4 *Media Recovery*

The Teradata database supports media recovery on failures using *permanent journalling* which is optionally specified when creating a table. Journalling allows users to recover a database or a table to a previous consistent state before or after changes have been made. Journal tables are created to record the changed rows and then used during recovery operations to rollforward or rollback to a previous consistent state. The Teradata database can generate journals of changed rows which can include before-images, after-images or both. Modified images are stored in a user-specified journal table on the create table command. The location of the modified image rows is a function of the table attributes, fallback/non-fallback table and single/dual Journalling. For a non-fallback table, after-images are written to another AMP in the same cluster as the one containing the data being changed. If the dual option is specified, the after-images are also written on the same AMP as the one containing the changed data rows. The before-images for a non-fallback data table are written on the same AMP as the modified data rows. For a fallback data table, the before-images and after-images are always written to the same AMP as the changed data rows.

Journalling provides protection against both application failure as well as data integrity failure of the DBMS. Users can request that the named checkpoint be placed on a database or a table. In case of an application failure, a user can request that the data be brought to a named checkpoint state. This is done using the rollback operation that backs out the changes made since the checkpoint was taken using the before-images. In case of data integrity failure, such as a file system failure on an AMP, data is first restored from a tape archive, and is then brought to a named consistent state using the rollforward operation that applies after-images up to the named checkpoint state. The location of the modified image rows can be tailored to both modes of failures. For users concerned only with the application failures, the before images are placed on the same AMP that generated the change. This avoids the overhead of sending the image to another AMP. For users concerned with data integrity or media failure, after images for non-fallback tables are placed on another AMP, protecting against a failure of the generating AMP. Permanent journalling is complemented by archiving on an external tape device, entire databases, individual data tables or journal tables. Archiving a table not protected by after-image journalling places a read lock on the table, and this precludes updates to the table. Archiving a table with after-image Journalling places a read lock only on the blocks that are currently archived. This allows for concurrent update and archive operations on the same table

3.5 *Loading and Extracting Massive Amounts of Data*

The Teradata DBMS provides two modes for extracting the data. The *Regular Export Mode* is intended for selecting small to medium amounts of data. Suppose a user with an open LAN or Channel connection to an IFP submits a query, for example the "SELECT... ORDER BY..." SQL statement. The IFP's parser decomposes it into steps, passes them to IFP's dispatcher, which passes them to the AMPs. Each AMP produces a result in an ordered spool file. The last AMP to finish producing the spool signals to the dispatcher the completion of the step. The dispatcher then initiates the merge operation and the ordered data from the AMPs is returned to the user through the single LAN or channel connection. The merge operation is performed by the message system which, when merging, composes a totally ordered stream out of partially ordered spool files. Observe that in the regular mode, AMPs work in parallel when producing the result; however, the answer set is sent to the user through a single LAN or channel connection and this gages the throughput.

The *Fast Export Mode* eliminates this gaging by allowing an ordered answer set to be returned to the host using several LAN or channel connections. The result data is re-partitioned among AMPs such that AMP_1 has block 1 of totally ordered data, AMP_2 has block 2 of totally ordered data, etc. in round robin fashion. Suppose that the host has N connections to the machine. The fast export utility on the host will use all N connections in parallel to retrieve data blocks from the AMPs. Due to data re-partitioning, the total ordering is easily preserved. The re-partitioning is achieved in two stages as follows. In the first stage AMPs sort their respective result spool files. Then a control AMP, AMP_C, is selected. Each AMP sends to AMP_C the first key value from each block of its spool file. As the result AMP_C can approximate the distribution of the sorting key. Suppose there are M AMPs. AMP_C partitions the distribution into M equal portions and assigns the first portion to AMP_1, the second to AMP_2, etc. The assignment is broadcast to all AMPs which perform the re-distribution. The result is placed in another spool with equal-size blocks. This finishes the first stage. In the second stage, AMP_1 communicates its block count to AMP_2. AMP_2 adds this to its block count and passes it to AMP_3. This continue until each AMP knows the relative order of its blocks. Then AMP_1 steps sequentially through its blocks and redistributes them to all the AMPs in round robin fashion. Suppose it finished on some AMP_k. AMP_2 starts redistributing its blocks in round robin fashion starting from AMP_{k+1}, etc. The last AMP to complete notifies the dispatcher that re-partitioning hash been done. The dispatcher notifies the host, which starts retrieving the data blocks in parallel on different channels.

Teradata DBMS provides an efficient mechanism for loading and updating massive amounts of data. Assume that a host with multiple connections to the machine has a large data file to be loaded to a table. The host submits to the parser on an IFP the description of the data file together with a mapping of the fields in the file to the fields of the target table. The parser broadcasts this description to the AMPs. Each AMP establishes a loading session and prepares two tasks: the *deblocker* task and the *receiver* task. The deblocker task is provided with the description of the data file and its mapping to the table. After this is done, the host starts sending the datablocks from the file to the deblocker tasks on the AMPs. When sending it uses all of its connections. Datablocks are addressed to the AMPs in round robin fashion. After it receives a datablock from the host, each deblocker task, converts each record into a row in the target table. It also calculates, based on the primary index of the table, the destination AMP of each row. Such a preprocessed datablock is then handed to the row redistribution service described in previous sections. The row redistribution services are asked to deliver the rows to the receiver tasks. The service examines each row and sends it to a buffer on the destination AMP. After a buffer is filled, it is presented to the receiver task. The receiver task places it in the target table. Occasionally the host will send a checkpointing request. The row redistribution service flushes its buffers and presents the rows to the receiver tasks with an indication to take a checkpoint. The receiver tasks will then commit all the rows to the table.