Report Number: WUCS-88-31

1988-10-01

A Transaction System for the NCUBE

Kenneth C. Cox and Gruia-Catalin Roman

We present the design of a transaction system which supports tuple-oriented database operations in the concurrent environment. An implementation of this system for the NCUBE Corporation NCUBE-7 hypercube processor is described. The implementation includes both the basic kernel to support the database operations and two software packages to assist users of the system.

A TRANSACTION SYSTEM FOR THE NCUBE

Kenneth C. Cox and Gruia-Catalin Roman

WUCS-88-31

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

A Transaction System for the NCUBE

Kenneth C. Cox and Gruia-Catalin Roman

Department of Computer Science
WASHINGTON UNIVERSITY
Saint Louis, Missouri 63130

Abstract

We present the design of a transaction system which supports tuple-oriented database operations in a concurrent environment. An implementation of this system for the NCUBE Corporation NCUBE-7 hypercube processor is described. The implementation includes both the basic kernel to support the database operations and two software packages to assist users of the system.

TABLE OF CONTENTS

1. Introduction

A transaction system is a software package which maintains a database and provides functions whereby user programs can access the database. The system described in this paper is designed to operate in a concurrent environment where many independent processes can simultaneously access the system. The system maintains a distributed database of content-addressable entities called *tuples*, which may be thought of as vectors of arbitrary data.

The basis of the system is a *kernel* which maintains the database and provides low-level functions for access. Two packages of functions for users are also provided; the first is intended for experimenters and provides only minimal support, while the second is intended for less sophisticated users and encapsulates very complex operations to provide a simple interface to the database. An implementation of the system on the NCUBE Corporation NCUBE-7 hypercube processor is in progress. The system is being implemented as a number of C functions and programs. The major design decisions for the implementation are discussed.

This paper begins with a logical overview of the transaction system. The system kernel, a package of functions and programs which supports the system and provides low-level operations on the database, is then described. The next section discusses implementation of the kernel on the NC-7. The higher-level software packages and their implementation are then discussed.

2. Transaction System Overview

The transaction system maintains a distributed database in a concurrent environment, in which many mutually independent processes run simultaneously. *User processes* are user applications programs which need to access the information in the database. *Database processes*, also called *sites*, maintain the database and service user process access requests. Processes communicate by means of message-passing; we assume message-passing may be performed asynchronously with other computations.

The entities stored in the database are content-addressable *tuples*. Each tuple consists of an arbitrarily-long sequence of *data elements*. In effect, the tuple data is a vector in which each element can have any of a number of primitive types; the length of the vector and the types used in the vector are called the tuple *structure*. Each tuple also has a unique *site* and *position*. Together these form the *tuple identifier*, which is unique over the entire system. The tuple identifier and structure are determined when the tuple is created, and do not change during the tuple's existence. The *contents* of the tuple (that is, the data stored in the tuple) is permitted to change.

The basic operations which can be performed on the database are *assert* (add a tuple), *query* (inspect a tuple), *modify* (alter a tuple), and *retract* (remove a tuple). The interaction of a user process with the system is called a *transaction*. A transaction is *atomic*: the database accesses and modifications associated with the transaction appear to be performed without any interleaving with other transactions.

A *pattern* is a predicate which is used for tuple access. Patterns must specify the structure of the tuple. They may optionally specify constraints on the identifier and contents; a number of simple relational operations are permitted, e.g. accept the tuple if a particular element is less than 10. The process of searching the database for a tuple which satisfies a pattern is called *matching*. Matching may be performed by *query*, *modify*, and *retract* operations, which collectively are called *matching transactions*. A matching operation finds at most one tuple which matches the pattern; if no matching tuple is found, the match is said to fail.

3. Transaction System Kernel

The system kernel is implemented in two parts: a package of C functions which allow user programs to access the transaction system, and a program which manages the tuples at a site. Both the user package and the site program share a set of datatype packages. These definitions give the structure of tuples, tuple identifiers, patterns, interprocess messages, and similar data, and permit manipulation of these types in an object-oriented fashion. These definitions are omitted from this paper to save space, and the types *site*, *tuple-id*, *tuple*, *pattern*, and so on are taken as primitives. Each data type has at least one value which cannot represent an actual object, called the *failure* for the type.

3.1. Kernel Functionality

The basic interaction between a user process and a site is called a *dialog*. A dialog consists of either two or three messages. The user program first sends a *request* to a site. At some later time the site performs the request and returns a *reply*. The reply may complete the dialog, or the user process may be required to send a *confirmation*. These actions occur asynchronously, so the user process is free to continue with other processing after issuing a request. A *dialog-id* is a type of object which identifies a dialog (request, reply and confirmation). Any object of type *dialog-id* is unique over the entire transaction system. When a user program issues a request, it receives a *dialog-id*.

Each site maintains a list of tuples. When a tuple is asserted, it is appended to the list. A position counter is used to assign a new, unique position to each tuple as it is asserted. Tuple matching starts at a particular position and traverses the list until a matching tuple is found or there are no more tuples. Tuples at a particular sites are thus ordered; further, sites have an order, so the entire collection of tuples can be placed in a single ordering. Note, however, that assertions or deletions may add or remove tuples anywhere in this ordering.

Requests may be issued in a number of *modes* which alter the manner in which the site processes the request. These modes are: *operation mode*, which determines the behavior of matching requests if the initial scan of the list fails; *lock mode*, which permits the request to lock a tuple and require a confirmation before completing the operation; and *block mode*, which determines the behavior of the request when it reaches a locked tuple.

The operation mode may be either *immediate* or *pending*. In immediate mode, the request scans the tuple list only once, then replies. If no matching tuple is found, the reply indicates failure of the match. In pending mode, the tuples are scanned; if no matching tuple is found, the request is retained by the site until a matching tuple becomes available and the reply is made. A pending request may be *released* to convert it to the immediate form, causing it to reply even if no matching tuple is available.

The lock mode is either *non-lock* or *lock*. Lock mode specifies the manner in which the tuple is locked. When a non-lock request matches a tuple, the operation is immediately performed and the reply is sent; the dialog is then complete. When a lock request matches a tuple, the locking is performed and a reply is sent. The user process must complete the dialog by sending a confirmation. A confirmation may be either a *commit* or a *cancel*. A commit performs the operation and unlocks the tuple; a cancel unlocks the tuple.

The block mode is one of *block*, *non-block*, or *ignore*. Both block and non-block modes have parameters describing what the request considers to be a lock; block mode has additional parameters which describe how the tuple mark is modified if the request blocks. Block mode causes the request to block, or stop scanning the tuple list, if it finds a tuple that (according to the parameters) is locked; the request modifies the tuple locking information, so other requests can test to see if a request is blocked at a tuple. Non-block mode causes the transaction to skip any locked tuples; it does *not* re-examine the tuple if it reaches the end of the tuple list. Ignore mode causes the transaction to ignore the tuple locking.

When a tuple is unlocked or altered, any requests blocked at the tuple are permitted to continue the scan of the tuple list. The requests resume scanning starting with the tuple; if the tuple was retracted, they proceed to the next. (Note that the request may immediately block on the same tuple, if when it resumes scanning it still matches the pattern and the tuple is still locked.)

Any request may be *aborted* or *deleted*. The effect of an abort or delete is to prevent the request operation from being performed, provided it has not already been performed; aborts and deletes do not undo operations. If the abort of a request is successful, a failure reply in a form appropriate to the original request is sent. Note that whether the abort succeeds or fails (because the request had already been performed), exactly one reply is sent for the request. The delete does not send a reply.

3.2. User Package

The functions of the user package are summarized in Table 1. The notation gives the type of the function and of its arguments and the input/output status of each argument; of course this does not represent the actual C

declaration. The functions fall into six categories. Functions which *send requests* initiate a dialog and return a *dialog-id.* Functions which *test for replies* allow examinination of the replies that have arrived. Functions which *receive replies* get the data from the replies. Functions which *send confirmations* are used when the dialog requires a cancel or commit. Functions which *change transaction status* send abort, delete, or release messages.

Functions to send requests:

Sassert (**in**Site:*site*; **in**Tuple:*tuple*; **in**Lock:*lock-mode*) : *dialog-id*

Squery(**in**Site:*site*; **in**Pos: *position*; **in**Pattern:*pattern*; **in**Op:*operation-mode*;
inLock:*lock-mode*; **in**Block:*block-mode*) : *dialog-id*

Smodify(**in**Site:*site*; **in**Pos: *position*; **in**Pattern:*pattern*; **in**Op:*operation-mode*;
inLock:*lock-mode*; **in**Block:*block-mode*; **in**Mod:*modification*) : *dialog-id*

Sretract(**in**Site:*site*; **in**Pos: *position*; **in**Pattern:*pattern*; **in**Op:*operation-mode*;
inLock:*lock-mode*; **in**Block:*block-mode*) : *dialog-id*

Sbounds (**in**Site:*site*) : *dialog-id*

Functions to test for replies:

Tavail(**in**Did:*dialog-id*) : *boolean*

Tfirstreply(**in**DT:*dialog-type*) : *dialog-id*

Tnextreply(**in**DT: *dialog-type*) : *dialog-id*

Functions to receive replies:

Rassert(**in**Did:*dialog-id*; **out**Site:*site*; **out**Pos:*position*; **out**Status:*integer*) : *boolean*

Rquery(**in**Did:*dialog-id*; **out**Site:*site*; **out**Pos:*position*; **out**Tuple:*tuple*; **out**Status:*integer*) : *boolean*

Rmodify(**in**Did:*dialog-id*; **out**Site:*site*; **out**Pos:*position*;
outTuple_1:*tuple*; **out**Tuple_2:*tuple*; **out**Status:*integer*) : *boolean*

Rretract(**in**Did:*dialog-id*; **out**Site:*site*; **out**Pos:*position*; **out**Tuple:*tuple*; **out**Status:*integer*) : *boolean*

Rbounds (**in**Did:*dialog-id*; **out**Low: *position*; **out**High: *position*) : *boolean*

Functions to send confirmations:

Scancel(**in**Did:*dialog-id*) : *boolean*

Scommit(**in**Did:*dialog-id*) : *boolean*

Functions to change transaction status:

Sabort(**in**Did:*dialog-id*) : *boolean*

Sdelete(**in**Did:*dialog-id*) : *boolean*

Srelease(**in**Did:*dialog-id*) : *boolean*

Miscellaneous functions:

LoadDB(**in**Channel: *integer*; Order: *integer*; Flags: *array of bytes*) : *boolean*

EnterTrans() : *boolean*

ExitTrans() : *boolean*

Table 1. Kernel Functions.

3.2.1. Functions to send requests

These functions generate a new, unique *dialog-id* which identifies the dialog. The *dialog-id* is returned as the function value. This *dialog-id* must be used to receive the reply, send a confirmation, or send messages which change the transaction status.

Sassert
Sends a request that the Tuple be added to the database at the given Site. If a Lock is required, it is applied when the tuple is asserted and the transaction requires a confirmation to complete the action.

Squery
Sends a request that a query be performed at the Site for a tuple matching Pattern, starting at the position Pos. Op is either immediate or pending mode. Lock and Block affect the searching as described above.

Smodify
Sends a request to modify a tuple. The first six arguments have the same meaning as in **Squery**. Mod specifies a single operation to be performed on *one* tuple data field. Modifications do not alter the tuple identifier.

Sretract
Sends a request to retract a tuple; the arguments have the same meaning as in **Squery**.

Sbounds
Sends a request to Site to reply with a pair of positions such that (at the time the reply is made) all tuples at the Site have positions falling in the range specified by the pair.

3.2.2. Functions to test for replies

These functions allow examination of the replies which are available to the user process.

Tavail
Returns **true** if the reply for the dialog associated with Did is available, **false** otherwise.

Tfirstreply
Initiates a search of the available replies of the indicated dialog type. Returns the dialog-id of the first such reply, or a failure indication if none is available.

Tnextreply
Continues a search for replies of a particular type from the last such reply returned by **Tfirstreply** or **Tnextreply**. Returns the dialog-id of the next reply, or a failure indication if none is available.

3.2.3. Functions to receive replies

These functions receive replies, returning the reply information through the arguments. All return a boolean, which is **true** if the reply was available. The values returned through the arguments are valid when the function returns **true**. On return, the *Status* field is a value indicating if the transaction **succeeded**, succeeded and requires **confirmation**, **failed**, or was **aborted**.

Rassert
Receives the reply to the assert request of the dialog associated with Did, returning the Site and Pos of the asserted tuple. Together these form the tuple id.

Rquery
Receives the reply to the query request of Did, returning the Site and Pos of a matching tuple and the Tuple data itself. These are all failures if no match could be found.

Rmodify
Receives the reply to the modify request of Did. The Site and Pos of the modified tuple are returned, as well as both the old $Tuple_1$ and new $Tuple_2$ data.

Rretract
Receives the reply to the retract request of Did. The return values are the same as in **Rquery**.

Rbounds
Receives the reply to the bounds request of Did. The Low and High values bracket the range of tuple positions at the site.

3.2.4. Functions to send confirmations

These functions must be used with transactions issued in *lock* mode to complete the operation, remove the lock, and allow any blocked transactions to continue. They return a boolean, which is **false** if the transaction does not exist, was not issued in lock mode, or has not yet replied.

Scancel
Send a cancel for the transaction.

Scommit
Send a commit for the transaction.

3.2.5. Functions to change transaction status

These functions return a boolean, which is **false** if the operation could not be performed.

Sabort
If possible, immediately stop the transaction. If the abort succeeds, the reply associated with the trans-id will be a failure indication which is recognizable as an abort (and not just a failed match). The function returns **false** only if the transaction does not exist.

Sdelete
If possible, immediately stop the transaction. No reply is made if the abort succeeds.

Srelease
Change the transaction status from pending to immediate. The function returns **false** if the transaction does not exist or if it was not issued in pending mode.

3.2.6. Miscellaneous functions

LoadDB
This function is used only by the program running on the NCUBE host. It loads the database program on the subcube associated with the Channel, which is of the indicated Order. The Flags contains 2^{Order} bytes; the database program is loaded on logical node i iff the i^{th} entry of Flags is non-zero.

ExitTrans
Initializes the transaction system package data structures. This function must be called by the user program before performing any other transaction system functions.

ExitTrans
Completes any outstanding transactions for the calling database process by aborting them; any locks held by the process are canceled. This should be called by the user process when it finishes interacting with the system.

4. Kernel Implementation on NCUBE

4.1. User package

Each user process has a table which maps the logical sites to the physical database-node addresses. With this package, the generation of this table (as well as starting the appropriate process on each NCUBE node) is the user's responsibility. Each user process also has an internal counter for each of the database nodes which it uses to generate the trans-ids. System-wide uniqueness is guaranteed by concatenating the counter with the address of the user process node to create the trans-id; both the address and the counter portions can be recovered from the trans-id.

When processing an S-class (send request) function, the user process first performs any necessary error-checking. It then generates a trans-id and prepares a message containing an integer encoding the transaction type, the trans-id, and the function arguments. The message is sent to the database node corresponding to the site. The trans-id and information about the transaction (including the transaction modes) are stored on the *outstanding list*. Finally, the function returns the trans-id.

When processing any of the package functions, the user process first reads any transaction-system messages (replies) which have arrived at the node. Any that do not correspond to outstanding transactions are discarded. The others are added to the *available list* and the entry on the outstanding list is changed to show the reply is available. The **Tfirstreply** and **Tnextreply** functions implement a list-traversal mechanism on the available list. The **Tavail** function simply examines the outstanding list.

When an **R**-class (receive reply) function is performed, the outstanding list is examined. If the reply is available, the reply contents are copied into the function arguments and the reply is removed from the available list. If the transaction was issued in non-lock mode, it is removed from the outstanding list; otherwise the outstanding list entry is modified to indicate the reply has been received.

When an **Scancel** or **Scommit** is issued, the outstanding list is checked to ensure that the transaction reply has been received. If it has, the confirmation is sent and the entry is removed from the outstanding list.

When an **Sabort** is issued, the outstanding list is again checked. The function returns **false** if the transaction is not on the list. If it is on the list, the reply status is checked. If the reply has not yet arrived, an abort message is sent to the site. Note **Sabort** returns **true** even if the reply has arrived.

The function **ExitTrans** deletes all transactions on the outstanding list; depending on the transaction status, this may involve sending a delete message. (A delete message aborts the transaction and cancels any lock it holds, but no reply is made by the database process.) No further initialization is performed.

When an **Srelease** is issued, the outstanding list is checked. The function returns **false** if the transaction is not on the list or is not pending. Otherwise the reply status is checked, and if the reply has not yet arrived a release message is sent to the site. Finally, the transaction mode is changed to immediate.

4.2. Database processes

The database process loops, continually reading messages from user processes and performing the indicated action, perhaps sending a reply as a result. When it receives a message the process first reads all messages queued in the NCUBE system and places them in an internal queue.

Any abort, release, cancel or commit messages are processed first. Transaction requests from a particular user process are performed strictly in order of trans-id. If a request arrives out of order, it is held until the intervening requests are obtained and processed. The process keeps the largest trans-id (as determined by the counter portion) it has received from each of the user processes.

All transactions are stored on a master *outstanding list*; each entry contains all the information associated with the transaction. A number of other lists of transactions are described below. These are implemented as lists of

pointers to entries of the outstanding list, so the data for a transaction only appears once. These other lists have the property that a transaction appears on at most one other list. A part of the outstanding list entry indicates what other list, if any, the transaction is on.

The tuples are stored in a *tuple list*. In addition to the tuple data, each list element contains the *position*, a *mark* used for locking, and a *block list* of transactions. The process maintains a counter which is used to generate positions. When an *assert* is performed, the tuple is appended to the tuple list with a new position, a zeroed (unlocked) mark, and an empty block list.

To support pending transactions, the process maintains a *pending list* of unserviced requests. Any transaction issued in pending mode which does not produce results when first processed is appended to the list. After performing any transaction (including a pending one) which asserts, modifies, or unlocks a tuple, the pending list is scanned for a transaction which matches the tuple. If such a transaction is found, the transaction is removed from the list and performed. If the transaction had no affect on the tuple, the scanning of the pending list continues from the transaction. If it modified the tuple (including modification of the mark), a new tuple results and the scanning of the pending list starts from the beginning. If it retracted the tuple, scanning halts.

When a release request is processed, the largest trans-id is examined. If the transaction to be released has not been processed yet, the release request is held until the transaction is received, at which time it is changed to immediate mode. If the transaction is not on the outstanding list, the release is ignored. If the transaction is on the pending list, it is removed from the pending and outstanding lists and a reply containing a failure indication is sent. (The pending-list scan algorithm maintains the invariant 'transaction on pending list ≡ no matching tuple exists'; a final scan of the tuple list is therefore not necessary.) Finally, if it is blocked at a tuple its status is changed to immediate and when it is unblocked it is processed as such.

When a transaction pattern matches a tuple, the results depend on the blocking-mode in which the transaction was issued. If the transaction was issued in *ignore* mode, or the transaction was issued in *non-block* or *block* mode and the tuple is not locked, then the appropriate action is taken (either the operation is performed or a lock is applied) and a reply is sent. If the transaction was issued in *non-block* mode and the tuple is locked, the transaction proceeds to the next tuple. If the transaction was issued in *block* mode and the tuple is locked, the transaction becomes blocked.

When a transaction is blocked at a tuple, the transaction is appended to the tuple's *block list*. The execution of a cancel, commit, modify, or retract on the tuple frees the transactions on the block list to continue searching. If the tuple was retracted, searching continues with the next tuple in the tuple list; otherwise the transaction tests its pattern against the tuple again (and may immediately reblock). Processing of the blocked transactions occurs before examination of the pending list.

Tuple locking and transaction blocking are carried out using the tuple *mark*. The mark consists of two elements, a bitfield B and integer I. The transaction may examine both elements to determine if the tuple is locked.

The effect of a block or lock is to set some bits of the appropriate mark's bitfield and increment the mark's integer by some amount, while undoing a block or lock clears the same bits of the bitfield and decrements the integer by the same amount. The test for blocking examines the mark and blocks if particular bits are set or if the integer is greater than some value.

Issuing a transaction in either block and non-block mode requires the specification of a test to determine if the tuple is locked. This test consists of two values, a bitfield b_t and an integer i_t. Block mode also requires the specification of an alteration to be applied to the block mark if the transaction is blocked. This alteration consists of two values, a bitfield b_b and integer i_b.

Issuing a transaction in lock mode requires the specification of an alteration to be applied to the lock mark if the transaction succeeds in locking the tuple. This alteration consists of two values, a bitfield b_l and integer i_l.

Let $\otimes$ be bitwise AND, $\oplus$ be bitwise OR, and $\bar{a}$ be logical NOT (of a). Then the lock/block actions are:

A transaction considers a tuple locked if

$$((B \otimes b_l) \neq 0) \vee (I > i_l)$$

When a transaction blocks on a tuple, it performs

$$B \leftarrow B \oplus b_b \qquad I \leftarrow I + i_b$$

When a transaction is unblocked, it performs

$$B \leftarrow B \otimes \overline{b_b} \qquad I \leftarrow I - i_b$$

When a transaction locks on a tuple, it performs

$$B \leftarrow B \oplus b_l \qquad I \leftarrow I + i_l$$

When a transaction unlocks a tuple (by cancel or commit), it performs

$$B \leftarrow B \otimes \overline{b_l} \qquad I \leftarrow I - i_l$$

When a transaction applies a lock to a tuple, the transaction is placed on a *confirmation list*. When a confirmation message (cancel or commit) is received, this list is searched for the transaction; if it is not found the confirmation is ignored. If it is found, the transaction operation is applied to the tuple (for a commit), the tuple is unlocked, and the transaction is removed from the confirmation and outstanding lists. The block list for the tuple and the pending list are then processed as described above.

When an abort is received, the largest trans-id is examined to determine if the transaction has been processed. If it has not, the abort is held until the transaction is received, at which time the failure reply is made. If the transaction has been processed, the outstanding list is examined. If the transaction is found, the failure reply is made; otherwise no action is taken.

5. User Package 1

This package provides a simple set of protocols for dialogs, concealing the mechanisms of locking, blocking, and confirmations. A transaction consists of two messages, a *request* and a *reply*. After issuing a request, the user process is free to continue with other processing. A new data type, the *trans-id*, is used to identify the transaction. The functions available at this level are listed in Table 2; except for the absence of locking and blocking and the use of *trans-ids* instead of *dialog-ids*, the function arguments are the same as those of the kernel. The **LoadDB**, **EnterTrans**, and **ExitTrans** functions of the kernel are also still available.

Matching takes place at only one site, using a pattern and start position supplied by the user program. The pattern may specify constraints on the position. Matching transactions may be issued in either *immediate* or *pending* mode; pending transactions may be *released*. The user program is also able to *abort* transactions; however, the transaction operations is not undone if it has already been performed.

All requests are issued in *non-lock* and *non-block* mode. This means that when a matching tuple is located, the operation is performed and the reply made immediately. No locking is performed, so no confirmation is required. Implementation of these protocols is simple, requiring only a single table data structure to translate transaction ids into the corresponding dialog ids.

6. User Package 2

This package provides a more complex set of protocols for dialogs and further conceals the message-based and site-oriented implementation. A transaction consists of a request, made by calling a function. The function blocks until the request is completed or until a specified amount of time has elapsed. If the request completes, the results are returned through the function arguments. The transaction system handles the details of tuple site management for the user.

The available functions are listed in Table 3. A *timeout* is a specification of an amount of time which the package is to wait for a reply to a request. The amount specified may be 'infinite'. If the indicated time elapses before a reply is obtained, the transaction is aborted.

Functions to send requests:

S1assert (**in**Site:*site*; **in**Tuple:*tuple*;) : *trans-id*

S1query(**in**Site:*site*; **in**Pos: *position*; **in**Pattern:*pattern*; **in**Op:*operation-mode*) : *trans-id*

S1modify(**in**Site:*site*; **in**Pos: *position*; **in**Pattern:*pattern*; **in**Op:*operation-mode*;
inMod:*modification*) : *trans-id*

S1retract(**in**Site:*site*; **in**Pos: *position*; **in**Pattern:*pattern*; **in**Op:*operation-mode*) : *trans-id*

Functions to test for replies:

T1avail(**in**Tid:*trans-id*) : *boolean*

T1firstreply(**in**DT:*dialog-type*) : *trans-id*

T1nextreply(**in**DT: *dialog-type*) : *trans-id*

Functions to receive replies:

R1assert(**in**Tid:*trans-id*; **out**Site:*site*; **out**Pos:*position*; **out**Status:*integer*) : *boolean*

R1query(**in**Tid:*trans-id*; **out**Site:*site*; **out**Pos:*position*; **out**Tuple:*tuple*; **out**Status:*integer*) : *boolean*

R1modify(**in**Tid:*trans-id*; **out**Site:*site*; **out**Pos:*position*;
outTuple$_1$:*tuple*; **out**Tuple$_2$:*tuple*; **out**Status:*integer*) : *boolean*

R1retract(**in**Tid:*trans-id*; **out**Site:*site*; **out**Pos:*position*; **out**Tuple:*tuple*; **out**Status:*integer*) : *boolean*

Functions to change transaction status:

S1abort(**in**Tid:*trans-id*) : *boolean*

S1delete(**in**Tid:*trans-id*) : *boolean*

S1release(**in**Tid:*trans-id*) : *boolean*

Table 2. User Package 1.

---

assert (**in**Tuple:*tuple*; **in**Timeout:*integer*; **out**Tuid:*tuple-id*; **out**Status:*integer*) : *boolean*

query (**in**Pattern:*pattern*; **in**Timeout:*integer*; **out**Tuid:*tuple-id*; **out**Tuple:*tuple*; **out**Status:*integer*) : *boolean*

modify (**in**Pattern:*pattern*; **in**Mod:*modification*; **in**Timeout:*integer*; **out**Tuid:*tuple-id*;
outTuple$_1$:*tuple*; **out**Tuple$_2$:*tuple*; **out**Status:*integer*) : *boolean*

retract (**in**Pattern:*pattern*; **in**Timeout:*integer*; **out**Tuid:*tuple-id*; **out**Tuple:*tuple*; **out**Status:*integer*) : *boolean*

setcut (**in**Cut:*integer*) : *boolean*

Table 3. User Package 2.

6.1. NCUBE Implementation

6.1.1. Allocation of NCUBE nodes to processes

The first aspect of site management handled by this package is the allocation of NCUBE nodes to user and database processes, with the attendant mapping from logical sites to physical nodes. In a host program, the user specifies the total number of nodes to be used, the number of user process nodes, and the program(s) to be run as

user processes. The host package automatically loads the processes on the correct nodes and initializes the necessary internal tables.

6.1.2. Automatic tuple site selection

The second aspect of site management is the determination of the site at which a tuple is to be stored and the site(s) where tuples matching a pattern may be stored. This is accomplished by two mapping functions (hash functions). The first function F_t maps a tuple to a single site; the second function F_p maps a pattern to a non-empty set of sites. These functions are selected in such a way that

$$F_t(tuple) \in F_p(pattern)$$

for any *pattern* which matches *tuple*.

When a *tuple* is asserted, the system places it at site $F_t(tuple)$. When a *pattern* is to be matched, the system generates $F_p(pattern)$ and attempts matching at each of these sites; if a *tuple* matching *pattern* has been asserted, then by the above property of F_t and F_p its site is examined.

The use of this method speeds up the process of searching for a tuple. However, it also requires that at all times $F_t(tuple)$ be the site where *tuple* is stored. Since the tuple site does not change once the tuple is asserted, the manner in which the tuple contents may be modified is restricted. A tuple can only be modified if the new contents has the same F_t as the old contents.

The method used to guarantee this behavior uses the fact that tuples are vectors of data elements. A user program selects a number called the *cut*. Once a user process has asserted a tuple, it cannot change the value of the cut. Any data elements whose position in the tuple is at or before the cut can be modified; data elements whose position is after the cut cannot be modified. The functions F_t and F_p use only information about positions after the cut to generate their results; the desired behavior is thus guaranteed.

Although the required property of F_t and F_p appears difficult to realize, it is actually quite simple to develop appropriate functions. Assume that there are 2^n database nodes; then each is assigned a (logical) base-2 number consisting of n bits. Given a tuple to assert, an n-place vector of values is constructed by taking tuple data elements in order from after the cut (if there are not n tuple elements, the are repeated as necessary; if there are no tuple elements after the cut, the vector is filled with a constant). A hashing function which produces 0 or 1 is then applied to each vector element, producing an n-digit binary number which is used as the (logical) node address.

A similar method is used for tuple matching. Each position of the pattern after the cut is either specified (must be the atom 'pixel', must be a 7) or unspecified. The same method as above is used to construct an n-digit number in which the specified pattern positions are 0 or 1 (as produced by the hashing function) and the unspecified positions are left empty. The set of possible tuple sites is generated by filling in the empty positions in all possible ways with 0 or 1.

Adaptations for the above scheme when there are not exactly 2^n database nodes are simple. To further add efficiency, the number and type of the data elements are used to fill in several slots of the hashing vector; since this information is known for both tuples and patterns, it reduces the number of unspecified slots and hence the number of sites which must be searched.

6.1.3. Data structures for multiple-site searching

A *trans-id* is a reference to a *transaction record*, which contains the information associated with the transaction. As discussed below, this information includes the transaction type and operation-mode, a status indicator, the lower-level trans-ids, and the transaction reply.

When a transaction is requested a new transaction record is generated and added to the *transaction list*. The transaction status and mode are set appropriately and the transaction reply is made a failure indication. The protocols described in the next section are then followed to initiate the transaction.

The protocols generate a number of simultaneous dialogues. Each of these dialogues is implemented using the lower packages and thus has an associated *trans-id* and entry on the *outstanding list* of the user process. During the initiation of a transaction, these trans-ids are collected into a list and entered in the transaction record.

A transaction can have at most one reply. When a reply for a dialogue arrives, the associated entry in the transaction record's list is marked. The reply is then tested with the current transaction reply according to the protocols of the next section; the new reply may be discarded or the old reply may be replaced. Depending on the protocol, some of the outstanding dialogues may be aborted, canceled, or committed. When all the dialogues have completed (including through aborts), the reply is placed on the available list and the entry in the transaction record is changed to reflect this.

6.1.4. Protocols for multiple-site searching

The protocols described here are used for all database searches, even if only one site needs to be interrogated. The transaction involves sending a request to each site in F_p; this is called a *broadcast*. Similarly, multiple replies may be received before one is selected to be supplied to the user process; the selected reply is called the *response*.

When *lock* or *block* modes are specified, the parameters used are as follows: *lock* sets a particular bit in the mark bitfield, while *block* blocks if this bit is set but makes no changes itself. Note that this protocol permits only one transaction to hold a lock on a tuple.

A transaction is issued in one of eight configurations as determined by the transaction type, operation-mode, and search-mode. The type may be either **Query** or **Alter** (modify or retract). The operation-mode may be either **Immediate** or **Pending**. The search-mode may be either **Unordered** (any or only) or **Ordered** (first or next).

Query, Immediate, Unordered

The user process broadcasts in *immediate*, *non-lock*, *ignore* modes. As replies arrive the sites are noted. The first positive reply to arrive is taken as the response; delete messages are sent to all sites from which no reply has been received. If all sites reply negatively, the response is a failure indication.

Query, Immediate, Ordered

The user process broadcasts in *immediate*, *non-lock*, *ignore* modes. When a positive reply arrives, delete messages are sent to all sites following the replying site in the site ordering which have not replied. Among the positive replies to arrive, the first in the site ordering is taken as the response. If all sites reply negatively, the response is a failure indication.

Query, Pending, Unordered

The user process broadcasts in *pending*, *non-lock*, *ignore* modes. The first positive reply to arrive is taken as the response; delete messages are sent to all other sites.

Query, Pending, Ordered

The user process broadcasts in *pending*, *non-lock*, *ignore* modes. When a positive reply arrives, delete messages are sent to all sites following the replying site in the site ordering and release messages are sent to all sites preceding it. The replies from the preceding sites are examined, and the first (in the site-ordering) positive reply is taken.

Alter, Immediate, Unordered

The user process broadcasts in *immediate*, *lock*, *block* modes. As replies arrive the sites are noted. The first positive reply to arrive is taken as the response and a commit is sent to that site. Delete messages are sent to all sites which have not replied (recall that a delete message also cancels any existing lock). If all sites reply negatively, the response is a failure indication.

Alter, Immediate, Ordered

The user process broadcasts in *immediate*, *lock*, *block* modes. The first positive reply to arrive is taken as the *temporary response* and delete messages are sent to any sites following it. Each subsequent positive reply is compared with the temporary response. If it follows the temporary response in the ordering, it is discarded; if it precedes the temporary response, it replaces the temporary response and delete messages are sent to any sites following it (including that of the old temporary response). When all sites preceding the current temporary response have replied negatively, the temporary response is taken as the response and a commit is sent. If all sites reply negatively, the response is a failure indication.

Alter, Pending, Unordered

The user process broadcasts in *pending*, *lock*, *block* modes. The first positive reply to arrive is taken as the response and a commit is sent to the site; delete messages are sent to all other sites.

Alter, Pending, Ordered

The user process broadcasts in *pending*, *lock*, *block* modes. When a positive reply arrives, it is taken as the temporary response; delete messages are sent to all sites following it, and release messages are sent to all sites preceding it. The protocol described above for **Alter, Immediate, Ordered** is then used to select the response.

The action taken when the user releases a transaction depends on the status of the transaction. In all the **Pending** protocols described above, there is a point (the receipt of the first reply) at which release and/or delete messages are sent to the sites. If the release request occurs before this point, the release messages are sent and the corresponding **Immediate** protocol is followed. If the release request occurs after this point, it is ignored.

The action taken when the user aborts a transaction also depends on the status. In a **Query** type of transaction, delete messages are sent to all sites and the response to the transaction is taken as a failure-abort. In the **Alter** type, there is a point at which the response is selected from among the replies which have been received and a commit is sent to the site. If the abort occurs before this commit is sent, delete messages are sent to all sites and the response is taken as a failure-abort; otherwise, no messages are sent and the selected response is used. **false** if it timed out.