Project #: C-36-689  Cost share #:
Center #: 10/24-6-R7553-0A0  Center shr #:
Contract#: AGREEMENT SIGNED 7/14/92  Mod #: ADMIN.
Prime #: NAS7-1194
Subprojects ?: N
Main project #:

Project unit: COMPUTING  Unit code: 02.010.300
Project director(s): OMIECINSKI E R  COMPUTING (404)894-3160

Sponsor/division names: INTEGRATED PARALLEL TECH INC  / PLEASANTON CA
Sponsor/division codes: 208  / 043

Award period: 920717  to 941016 (performance)  941016 (reports)

Sponsor amount  New this change  Total to date
Contract value  0.00  63,994.00
Funded  0.00  63,994.00
Cost sharing amount  0.00

Does subcontracting plan apply ?: N
Title: TIME WARP CONCURRENCY CONTROL FOR DATABASE SYSTEMS

PROJECT ADMINISTRATION DATA

OCA contact: William F. Brown  894-4820
Sponsor technical contact  Sponsor issuing office
CALVIN BUZZELL  (510)734-8854
CALVIN BUZZELL  (510)734-8854
INTEGRATED PARALLEL TECHNOLOGY  INTEGRATED PARALLEL TECHNOLOGY
5994 W. LAS POSITAS BLVD.  5994 W. LAS POSITAS BLVD.
SUITE 209  SUITE 209
PLEASANTON, CA 94588  PLEASANTON, CA 94588

Security class (U,C,S,TS) : U  ONR resident rep. is ACO (Y/N): N
Defense priority rating : NONE  NONE supplemental sheet
Equipment title vests with: Sponsor X  GIT
NONE PROPOSED
Administrative comments -  ADMIN. REVISION TO INCORP. BUDGET REVISION.
GEORGIA INSTITUTE OF TECHNOLOGY
OFFICE OF CONTRACT ADMINISTRATION

NOTICE OF PROJECT CLOSEOUT

Closeout Notice Date 02/02/95

Project No. C-36-689 Center No. 10/24-6-R7553-0A0

Project Director OMIECINSKI E R School/Lab COMPUTING

Sponsor INTEGRATED PARALLEL TECH INC/PLEASANTON CA

Contract/Grant No. AGREEMENT SIGNED 7/14/92 Contract Entity GTRC

Prime Contract No. NAS7-1194

Title TIME WARP CONCURRENCY CONTROL FOR DATABASE SYSTEMS

Effective Completion Date 941016 (Performance) 941016 (Reports)

Closeout Actions Required: Y/N Submitted

Final Invoice or Copy of Final Invoice Y
Final Report of Inventions and/or Subcontracts Y
Government Property Inventory & Related Certificate Y
Classified Material Certificate N
Release and Assignment Y
Other N

Comments __________________________________________________________________________

Subproject Under Main Project No. 

Continues Project No. 

Distribution Required:

Project Director Y
Administrative Network Representative Y
GTRI Accounting/Grants and Contracts Y
Procurement/Supply Services Y
Research Property Management Y
Research Security Services N
Reports Coordinator (OCA) Y
GTRC Y
Project File Y
Other N

Date Submitted

Y

Y

N

Y

N

N

NOTE: Final Patent Questionnaire sent to PDPI.
Design and Implementation of a Time Warp Parallel Database System

Edward Omiecinski    Srinivas Gukal    Ashok Savasere
Kiran Achyutuni
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
e-mail: edwardo@cc.gatech.edu
October 16, 1994

1 Introduction

Applying multiprocessor machines to database query processing has been an active area of research. The motivation for using a multiprocessor machine stems from the need to store larger and larger amounts of data. An adequate level of performance (e.g., throughput) is required in the face of ever increasing database sizes. One way to increase the throughput of a database system is through inter-transaction parallelism. By that we mean that several transactions can be executed in parallel by a number of processors. To maintain the consistency of the database with many transaction running in parallel, it is necessary to guarantee the serializability of transactions. An execution is serializable if it is computationally equivalent to a serial execution. Concurrency control methods guarantee the serializability of transactions.

The two most studied concurrency control methods are 2-phase locking and timestamp ordering. 2-phase locking is the concurrency control method implemented in almost all commercial single-processor database systems. With 2-phase locking, locks can be requested in the first phase and released in the second phase. To resolve serializability conflicts, transactions are blocked. With timestamp ordering, transactions are assigned a unique timestamp. Transactions are then processed such that their execution is equivalent to a serial execution of transactions based on increasing timestamp value. Conflicts are resolved by aborting and restarting a transaction. Both these approaches have their drawbacks. With 2-phase locking, blocking may significantly reduce the potential for concurrency. With timestamp ordering, a transaction will have to be restarted after being aborted. This causes a lot of unnecessary work to be done by the database system.
The Time Warp concurrency control mechanism is similar to the timestamp ordering approach. However, with Time Warp, transactions are never aborted to resolve conflicts. Instead, transactions are rolled back to a point where the conflict no longer exists, i.e., a partial rollback is done. Under this scheme, transactions never get a new timestamp value. Hence, when a set of transactions begin execution, there will be only one equivalent serial schedule for those transactions. This presents a stronger condition than necessary for database transaction processing. Since partial rollbacks are necessary, old versions of the database items and old states of the transactions must be kept for a period. This is not necessary for either of the previously described methods.

Time Warp has an advantage with long running transactions that execute operations in a serial order. A transaction can be partitioned into sub-transactions which can be executed in parallel. The parallelism speeds up transaction response time. Here, the sub-transactions are assigned a timestamp that represents the serial ordering of operations. The Time Warp method is then responsible for guaranteeing that these operations appear to execute serially, as defined by the operation ordering. In this case, if a conflict occurs, a new (larger) timestamp cannot be assigned to a subtransaction without violating the operation ordering. This is also the case with the timestamp ordering method. Time Warp rolls back sub-transactions; however, for the timestamp method, the entire transaction is rolled back and assigned a new timestamp.

As an example, consider a banking application where we want to add the interest accumulated to each savings account. Suppose that accounts with balances over 10,000 dollars receive 6 percent interest, while all others receive 5 percent interest. We have a DEPOSIT relation that contains tuples with the balance information for each savings account. We can write this in a SQL-like syntax as follows:

```
Begin Program;

EXEC SQL Begin Transaction T1;
    EXEC SQL update Deposit
      set balance = balance*1.06
      where balance > 10000;
EXEC SQL Commit T1;

EXEC SQL Begin Transaction T2;
    EXEC SQL update Deposit
      set balance = balance*1.05
      where balance <= 10000;
EXEC SQL Commit T2;

End Program;
```
The top-level transaction, i.e., the main program, will access all of the tuples in the Deposit relation. This is an example of a long duration transaction. A typical short duration transaction might be one that transfers money from one account to another or simply withdraws an amount from a single account. To reduce the response time of our long duration transaction, we could execute the subtransactions in parallel. However, the order in which the two subtransactions are done is important. That is, the result of executing the two subtransactions in parallel should be the same as executing T1 followed by T2. If the order of the two subtransactions were reversed, then an account whose initial balance was less than 10,000 dollars but whose updated balance (produced by subtransaction T2) was 10,000 dollars or more would also be updated by subtransaction T1 as well. Those accounts would receive 11.3 percent interest, producing an incorrect execution. The Time Warp approach is well suited for such an application.

In comparison, the 2-phase locking scheme guarantees serializability but it cannot guarantee that the execution of the subtransactions will be in the specified order. The serial schedule that is equivalent to a 2-phase locking schedule has the transactions ordered on the time that they reached their lock point. Thus, we cannot exploit parallelism between the subtransactions without expensive coordination between the processors. Therefore, we are restricted to a single level transaction and this adversely affects the response time as well as the throughput of the database system.

In this paper, we describe the prototype design and implementation of a parallel database system based on the Time Warp concurrency control method. The parallel database system was implemented on a nCUBE 32-node machine. In the next section, we describe the Time Warp concurrency control in more detail. In Section 3 we present the design. The implementation is described in detail in the remaining sections.

2 Time Warp Concurrency Control

The Time Warp concurrency control was first described by Jefferson and Motro in [1]. This section describes the modified Time Warp mechanism used in our implementation of the parallel database system. The original Time Warp mechanism is augmented with dependent record based rollback. That is, it supports undo read and undo write as well. The description of the mechanism is illustrated using a comprehensive example of transaction execution. Each transaction is allocated a buffer. Each buffer contains all the records accessed by the corresponding transaction. Different versions of a record in different buffers are all connected using doubly-linked lists. A table of all the record identifiers that have been modified or read by the as yet uncommitted transactions, is maintained for each relation. Each entry in this table contains a record identifier and a pointer to the first buffer that contains a version of this record. The transaction buffers, which contain the records for a particular table, are all linked in the order of their timestamps. The table contains a pointer to the first corresponding transaction buffer. Let ts(T) indicate the time stamp of transaction (T)
and \( c(T) \), the search condition of \( T \). When activated, the transaction goes through the following steps.

1. Initialize its transaction buffer \( b(T) \).

2. Post its timestamp and the search expression in the "Global Buffer".

3. Search the appropriate index and get the satisfying record identifiers. Let the satisfying records be \( \text{set}(T) \).

4. For each record identifier \( \text{rid} \) in \( \text{set}(T) \) in step 2,
   
   (a) Latch the entry.
   
   (b) Check for the corresponding entry in the table of pointers.
   
   (c) If the record identifier is present in the table,
       
       i. The links are followed till the transaction buffer with the highest timestamp that is lower than \( ts(T) \).
       
       ii. The last version of the record is now checked for the condition \( c(T) \).
       
       iii. If \( c(T) \) is satisfied, update \( \text{rid} \) if needed and insert in \( b(T) \). Make the necessary changes in pointers.
   
   (d) If the record identifier is not present in the table,
       
       i. Update \( \text{rid} \) if needed and insert in \( b(T) \).
       
       ii. Make an entry in the table of pointers for \( \text{rid} \).
   
   (e) Unlatch the entry.
   
   (f) Delete \( \text{rid} \) from \( \text{set}(T) \).

5. Traverse all the corresponding transaction buffers in the order of their time stamps. For each transaction buffer \( b(T_1) \),
   
   (a) Search the transaction buffer sequentially checking each record for the condition \( c(T) \).
   
   (b) For each satisfying record,
       
       i. Latch the entry.
       
       ii. Follow the steps in (4.c).
       
       iii. Unlatch the entry.
   
   (c) Save the serial number of the last record checked in \( b(T_1) \) in the "Global Buffer".

6. Wait for any modifications sent by other transactions. Make necessary changes.

7. When it is time to commit,
(a) Write the redo log records to the disk.
(b) Insert the changes in all the corresponding indices.
(c) Delete all entries in the transaction buffer, making necessary pointer changes.
(d) Free the transaction buffer.

2.1 Propagating Changes

This section details steps (4.c.iii) and (6) above. When a transaction \( T \) inserts a record \( (rid) \) in its buffer \( b(T) \), a few pointers may have to be modified. Also, transactions with higher timestamps may need to be notified of the changes. Each entry in a transaction buffer has a (possibly modified) record, a forward pointer and a backward pointer. The forward pointer refers to the next (possibly modified) version of the same record, if one exists, in the buffer of a transaction with the least timestamp that is higher than \( ts(T) \). In a similar manner, the backward pointer indicates the earlier version of the record. If there are no earlier versions, other than the one on stable storage, the backward pointer points to the entry in the table of pointers. When a transaction \( T \) accesses a record \( (rid) \), the pointers should be suitably changed. Three cases might arise here. \( T \) may be the first transaction to modify \( (rid) \), \( T \) may be the transaction with the highest timestamp or transactions with timestamps higher than \( ts(T) \) may have accessed \( (rid) \) before \( T \). The first case is simple. Since \( T \) is the first transaction to access \( (rid) \), step (4.d) is executed. The \( (rid) \) is inserted in the table of pointers and connected with the entry for \( (rid) \) in \( b(T) \).

Step (4.c) is executed in the other two cases. If \( T \) is the transaction with the highest timestamp, \( (rid) \) is entered in \( b(T) \) and connected with the last entry. If \( ts(T) \) is not the highest, it implies that transactions with higher time-stamps have accessed \( (rid) \). These accesses should be nullified because of the current entry in \( b(T) \). \( T \) follows the pointers and marks void all the versions of \( (rid) \) in transaction buffers with higher timestamps. This can be safely done since \( T \) holds the exclusive lock on \( (rid) \).

Once a transaction makes an entry in its buffer, it has to check if this new version satisfies the condition of any other transaction with a higher time-stamp. Each entry in a transaction buffer has a unique serial number. When a transaction \( T1 \) searches the buffer of another transaction \( T \) in step (5), it makes a note in the "Global Buffer" about the last entry in \( b(T) \) that it checked. Now if \( T \) makes a new entry for \( (rid) \) in its buffer \( b(T) \), it searches the global buffer for any transaction \( T2 \), that satisfies the following conditions.

1. \( ts(T2) \) is greater than \( ts(T) \),
2. \( (rid) \) satisfies \( c(T2) \) and
3. The last entry in \( b(T) \) which \( T2 \) checked is less than the entry for \( (rid) \).
If all three conditions are satisfied, (T) passes (rid) to (T2). (T2) executes step (4) for (rid) and makes the necessary entry.

3 Design of the Manager Modules

3.1 Structure of the System

The nCUBE contains two types of processors - the host processor and the node processors. The users interact with the node processors via the host processor. A standard Unix operating system runs on the host processor, while a microkernel controls the node processors. The node processors are connected as a hypercube. In the time warp parallel database system, the node processors are divided into two categories: the Transaction manager nodes (TMNode) and the Data Manager nodes (DMNodes). The primary distinguishing feature between the nodes is that the DMNodes have local disks attached to them, whereas the TMNodes do not have any local disks. In order to keep all the processors in the system busy, the work is divided up between the TMNodes and the DMNodes. The TMNodes manage the various transactions running in the system. The DMNodes manage the disks that are connected to the nodes.

Figure 1 shows where the various processes of the time warp parallel database system run on nCUBE system. Each of the processes will be described in detail in the remaining part of this section.
3.2 The User Interface
The User Interface (UI) is an SQL-like interface that receives the queries and sends it to the Central Transaction Manager. It is also provides facilities to view the results.

3.3 The Central Transaction Manager
The Central Transaction Manager (CTM) is the principal coordinator of the database system. It is the first process started on the host as soon as the system is brought on-line. The CTM starts the support processes on the node processes. It then waits for the requests from user interfaces (UI). When a query is received, the CTM routes the query to a Transaction Manager (TM) running on one of the TMNodes, depending on the the load of the TMNodes.

The CTM also receives the results of the query and sends it to the UI, which in turn pretty prints the results. If there is an error in processing the query, the error message is also conveyed to the UI.

The CTM generates timestamps for transactions. The timestamp of a transaction is 4 bytes in length. The higher order two bytes of the timestamp of a transaction is generated by the CTM. The lower order two bytes are generated by the Application Transaction Manager (ATM) (see below).

The CTM determines the order in which the transactions commit. A transaction accessing a table can commit when all of its predecessors, which have accessed the same table, have committed. The CTM sends a commit token to the TM of the transaction when it is the turn of that transaction to commit.

3.4 The Transaction Manager
The Transaction Manager (TM) runs on the TMNodes. It’s primary responsibility is to listen for requests from the CTM and receive the corresponding query. When the query is received, the TM spawns an Application Transaction Manager (ATM) to exclusively handle the query.

3.5 The Application Transaction Manager
The Application Transaction Manager (ATM) is responsible for executing a single query handed to it by the TM. The ATM parses the query and builds the abstract syntax tree (AST). If there are any errors in the query, the CTM is notified. The query is then broken down into sub-queries, each of which can be executed in parallel. A sub-transaction is started for each sub-query. The sub-transactions are given consecutive timestamps, so that a later sub-transaction can see the results of the earlier sub-transactions. As mentioned before, the timestamp of a transaction consists of 4 bytes. All the sub-transactions of a transaction have the same value in their 2 higher order bytes. The lower order 2 bytes are distinct for every sub-transaction, and these are generated by the ATM. Since the timestamps are consecutive, no other transaction can see the intermediate results.
of the sub-transactions. For each sub-transaction, an entry is made in the transaction list. The transaction buffers are also initialized for the sub-transactions. The TM starts an Application Data Manager (ADM) process for each sub-transaction.

When all sub-transactions belonging to a query are ready to commit, they wait for the commit message from the TM. The TM in turn waits for the commit message from the CTM. After the TM receives the commit message and the sub-transactions are ready to commit, the TM sends a message to the Recovery Manager (RM) to write the log and update the database. The results returned by all the ADMs in the transaction buffers are then returned to the CTM, which in turn sends it to the UI. The results are filtered before sending it to the CTM by deleting all data in the fields not mentioned in the query. The field information is also sent so that the UI can display the results properly.

3.6 The Data Manager

The Data Manager (DM) runs on the DMNodes. It is primarily responsible for spawning the Application Data Manager (ADM) on the DMNodes to execute the sub-transactions of a transaction. The DM spawns an ADM for each sub-transaction.

3.7 Application Data Manager

An ADM manages the execution of a single sub-transaction. The ATM initializes the ADM and sends it a pointer to the sub-query AST. The ADM first determines which table to access. Based on the partitioning information and the fields in the condition, the ADM then decides which disks to search. The ADM next uses the current indices on the table and the form and complexity of the search condition to decide the type of search (serial or indexed) and the particular index to use (in case of index search). One Application I/O Manager (AIM) is started per disk to search the disks in parallel. While the AIMS execute, the ADM waits for any messages for selective rollbacks. A selective rollback message is received when the Buffer Manager (BM) detects a timestamp order violation in the access of a data item by different transactions. When a selective rollback message is received, the ADM performs necessary actions for the rollback and then forwards the message to the next appropriate ADM.

When all the AIMS are completed, the ADM waits for the commit token from the TM. After the commit token is received, the ADM returns the results in the transaction buffers to the TM and exits.

3.8 Application I/O Manager

An AIM is started by an ADM to execute a sub-transaction on a part of a table on one disk. The ADM also notifies the AIM about the type of search and the index to use in case of an indexed
search. The AIM uses the Access Manager (AM) to determine the records satisfying the search condition. For each call, the AM returns one record identifier which satisfies the index condition. The AIM then requests the Buffer Manager (BM) to check if someone else had already modified the record. If not, the AIM requests the corresponding Disk I/O Manager to fetch the data page to the main memory. The record is then checked to see if it satisfies the full search condition. If the condition is satisfied, then the BM is notified and the record is copied to the transaction buffer. If required, the copied record is updated.

In the second stage, the AIM uses the BM to search the global record directory to check if an earlier transaction had updated some records which might satisfy the condition. Any updated record that satisfies the condition is also copied to the transaction buffer. After the second stage, the AIM notifies the respective ADM that the search has been completed and returns the results in the transaction buffer.

3.9 Disk I/O Manager

In the initialization phase, the TM starts as many DIMs as the number of disks. Each DIM process manages all the I/O requests for one disk. The DIM process is unaware of the structure or contents of the data pages. It receives requests to load a physical page to an area in the main memory or to store the data in the memory area to a particular physical page. The DIM manages a queue where pending requests are stored in an FCFS order. Facilities are provided for the DIM to deal with duplicate requests. The DIM can also load an entire extent, instead of individual pages, which saves additional seek and latency times.

3.10 Access Manager

The Access Manager (AM) is used to search indices. The AIM provides the search condition and the index to search to the AM. The AM uses "crabbing" to avoid latching internal index pages for a long duration. Only the latch to the parent page is held while searching a child page. Each call to the AM returns one record identifier (RID) that satisfies the search condition. The first call returns the RID of the first record that satisfies the search condition. Along with the RID, the AM also returns information about the place in the index where the RID was located. Later calls to the AM include this information, so that the search can start from the page where the previous RID was located.

3.11 Buffer Manager

The Buffer Manager (BM) manages all the shared memory structures in the system. There are two modules in the BM. The first module, the Data Buffer Manager (DBM), manages the shared buffer area for the data pages. The data pages required by any process are loaded into the shared
The DBM provides exclusive access to the shared data pages to the different requesting processes by using semaphores. Each data page in the buffer is timestamped with the last access to it. The pages are replaced based on a least-recently-used policy. The DBM also writes back all the dirty pages when the system is shut down.

The transaction buffers and the associated shared data structures are maintained by the second module, called the Transaction Buffer Manager (TBM). The results of each transaction are kept in a separate set of transaction buffers. The TBM allocates, maintains and at the commit time, frees the transaction buffers. The TBM also manages the page directory. It maintains a list of records accessed and for each record, the list of transactions that accessed it.

3.12 Recovery Manager

The Recovery Manager (RM) is in-charge of logging the changes by update transactions and notifying the TBM when the transaction buffers of a transaction can be released. The Transaction Manager notifies the RM when a transaction completes. The RM writes all the changes made by that transaction to a log file. Only the changes to the data records are logged. The indices are rebuilt during crash recovery. The transaction buffers of a committed transaction (T) are released after all the active transactions at the commit time of (T) complete.

4 Shared Data Structures

Figure 2 shows the shared data structures used in each DMNode. The transaction list contains all the active transactions in the DMNode. A different list is maintained for each table. The
transactions are committed in the order of the list. Each entry for a transaction contains the
timestamp, pointers to the AST and the list of transaction buffers and the address of the UI that
started this transaction. Figure 2 shows two transactions T1 and T3 accessing table 1 and one
transaction T2 working on table 5.

The shared data buffer contains a set of pages in the main memory where the database pages are
loaded. Each disk is allotted a minimum number of pages in the shared buffer. This ensures that
a hot spot disk does not take up all the shared area, thereby starving the transactions accessing
the other disks. The shared buffer area also contains a large common pool of pages where pages
from any disk can be loaded. Data pages from a hot spot disk, that exceed the minimum pages
allotted, can use this common pool. All the processes in the system can access the shared data
buffer. Mutually exclusive access is provided by using semaphores.

The transaction buffers are used to hold the results of the transactions. Though shown sep­
arately in the figure, they are allotted from the common shared data buffer pool. A transaction
buffer can be modified only by its owner. However, other transactions can read the modified values
in the transaction buffer. This design avoids maintaining latches for access to transaction buffers.

The page directory contains one entry for each page in the database (a hash table can be used
instead to save space in case of large tables). All the records accessed are listed here. Each record
entry, in turn, has a list of all transactions that accessed it. The entry for a transaction contains
the type of access (Read or Write) and a pointer to the corresponding transaction buffer where the
record is stored. A transaction can use these pointers to access the updated values in the buffer of
an earlier transaction.

5 The Rollback Operation

In the time-warp mechanism, if an earlier transaction modifies a record already accessed by a
later transaction, then the later transaction should be rolled back so that it sees the value written
by the earlier transaction. This partial rollback can be costly if the earlier transaction had already
accessed a large number of records. To reduce the costs, we follow a selective rollback approach in
our design. Here the earlier transaction is rolled back for only that record where the timestamp
ordering is violated.

Figure 2 shows the shared data structures used to implement selective rollback. The transaction
list maintains the list of transactions accessing each table in the timestamp order. The page
directory contains the list of transactions that had accessed a record. When a transaction (T)
accesses a record, the page directory is checked if it causes any order violation. The ADM of any
later transaction that has to be rolled back is notified. The ADM removes the earlier copy of
that record from the transaction buffer. If the new value still satisfies the search condition, the
ADM inserts the new value in the transaction buffer. For example, in Figure 2, if transaction
(T1) updates record 8 on page 3, the order is violated since (T3), a later transaction, had already accessed this record. Hence (T1) notifies the ADM for (T3) about the conflict. This notification is also sent to all the later transactions accessing the same table, whose conditions are satisfied by the modified value. Several checks are placed to avoid race conditions that occur when two transactions simultaneously try to send notifications for the same record.

6 Interfaces

In this section, we describe the interfaces provided for the database administrator and the users of the database.

6.1 Administrator Interface

The Administrator Interface helps the administrator manage the database. This interface contains several utilities shown Figure 3. A new table can be created by selecting the "Create Table" utility. The utility requires information about the number of fields, their types and sizes. The primary key should also be specified. The "Load File" utility can be used to load a batch of tuples from a text file. The tuples should be formatted and the fields should be in the same order as the order specified while creating the table.

A table can be partitioned across several disks for faster access. There are several ways in which the data can be partitioned. The strategies provided here are hash partitioning, round-robin partitioning and range partitioning. The entire table can also be loaded in a single disk without any partitioning (no partition strategy). For each strategy, additional information should be provided. For example, range partitioning requires the field name and the range of values of the field for each partition.

The "Build Index" utility allows to build indices on an existing table. Both primary and
secondary indices can be built. Tables and indices can be dropped from the database using the "Drop Data" utility. The "Print Data" utility is used for debugging purposes. The data in any index or table in the table can be printed to a text file. All the above utilities are used when the database is off-line. The administrator interface also provides an utility to bring the database on-line.

6.2 User Interface

The user interface (UI) provides a SQL-like interface to the user. All user interaction with the system is through the UI. There can be several UI processes running, one for each user. An UI process, on initialization, checks if the TM is alive and on-line by sending a "Ping" to the TM. The TM, if active, sends back an acknowledgement. The user can now enter transactions. Each transaction may contain several sub-transactions. Rudimentary error checking (e.g., checking for the first keyword) is done in the UI. The entire transaction is then sent to the TM. The UI then waits for the results, periodically checking if the TM is still active. In case the transaction contains any syntactic or semantic errors, the TM sends an error message to the UI, which is then displayed to the user. If the transaction executes successfully, the results are sent back to the UI. The results contain two parts for each sub-transaction, the field information and the list of tuples retrieved. The field information is used to structure the data received and display the results. The results of each sub-transaction is displayed separately. Facilities are provided to switch from the results

Figure 4: User interface and transaction manager interaction
of one sub-transaction to those of another sub-transaction. The user can also set the number of tuples displayed per screen and scroll the results forward or backward.

7 Data Organization

The database is organized in extents. Each extent contains seven pages so as to keep the internal fragmentation low. When a file (table or index) grows, additional space is allocated in extents. An extent is used to cluster records based on the primary key. Extents can be of two types, a regular extent or an overflow extent. A regular extent is an extent that belongs to a single file. If a file requires only one additional page due to overflow in a clustered extent, it is allotted a page from an overflow extent. Hence an overflow extent can have pages from several files. A page map is maintained to determine which extents a file currently occupies. Each extent is represented by a byte. The highest order bit is used to distinguish regular extents from overflow extents. The other seven bits indicate which of the seven pages in the extent are occupied.
7.1 Data Page Organization

The data page is organized as shown in Figure 5. A data page contains the following areas: (a) header; (b) data; and (c) directory. The header contains information for maintaining the pages. This includes page type, next page number, previous page number, number of valid records on the page, etc. Currently the page organization supports fixed length records. The directory structures start from the bottom of the page and grow towards the data area. Each directory entry points to the beginning of a data record. For example, entry 1 points to record 1. The first bit of the directory entry indicates whether the corresponding record is valid or not. When a data record is deleted, the valid bit in its directory entry is set to invalid. Any data record does not span more than one page. Hence the maximum length of a record is limited by the page size.

7.2 Index Organization

B+ tree index structures can be built on any field of a record. A primary index is built only on a unique field. The data records are normally clustered on the same field. Any number of secondary indexes can be built on the other fields of the record.

The fan-out of the index is computed from the length of the indexed field. The internal pages are similarly organized for both the primary and secondary indexes. However, the leaf pages are slightly different. Since the primary index does not allow duplicate keys, the leaf page records are as follows: \( <\text{key}, \text{tid}> \). The secondary indexes allow duplicate keys. Hence the leaf record is as follows: \( <\text{key, tid}_1, \text{tid}_2, \ldots, \text{tid}_n> \). Currently each leaf record accommodates 25 record identifiers. If this number is exceeded, additional entries are allocated on overflow pages.

8 Future Work

The current implementation supports single table queries and updates. While this limits the capability of the database system, it nevertheless demonstrates the feasibility of the time warp concurrency control method. We are planning to enhance the implementation to support multiple tables and join queries. We intend to use differential files to ensure the correctness of joins. Let the timestamp of the join be T1, and that of an update transaction be T2, where T2 is less than T1. But if the join operation executes before the update transaction, then the result of the join is not entirely correct. Since the join operation is a long transaction, we want to avoid its rollback. With time warp, T2 commits before T1. Therefore, the updates of T1 are maintained in a differential file. When T1 is ready to commit, the differential file is applied to the join result and a new join result is computed. This approach is more efficient than aborting and/or rolling back the join transaction.
9 Conclusions

In this paper, we have described the design and implementation of a parallel database system with a time warp concurrency control method. The primary feature of time warp is partial rollback. Time warp concurrency control can deliver a better performance than either two phase locking and timestamp methods. This is because time warp avoids the blocking of two phase locking, and the costly rollbacks of classic timestamping. Time warp concurrency control method is best suited for a parallel database system that executes a large number of long transactions.

References