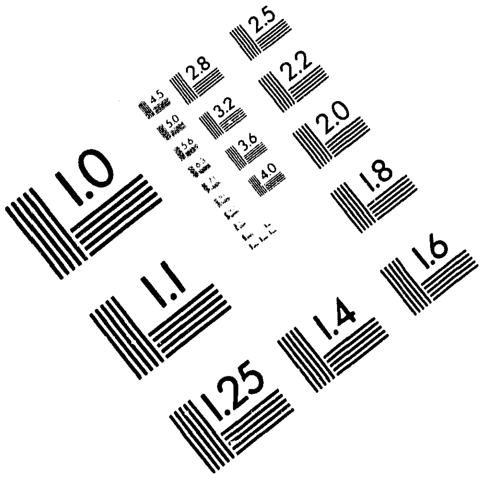
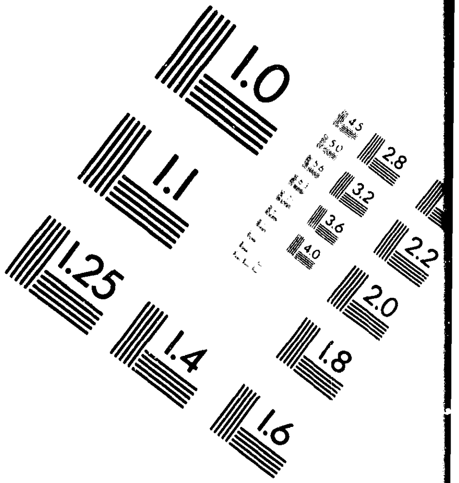
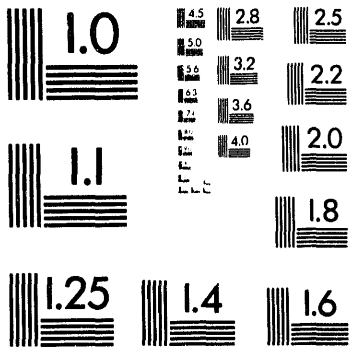
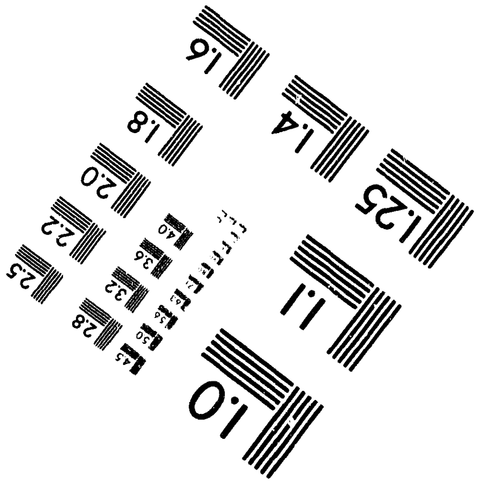
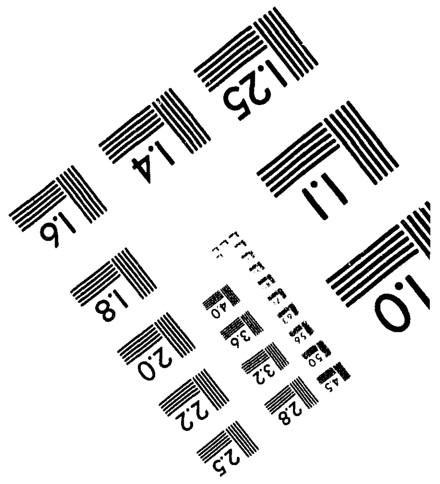



Centimeter
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 mm
1 2 3 4 5
Inches


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

1 of 1

ANL/DIS/CP--83044
Conf-9405175--1

A Highly Concurrent Transaction Management Model

Lawrence J. Henschen
Northwestern University
2145 Sheridan Rd.
Evanston IL 60208
henschen@eecs.nwu.edu

Julia C. Lee
Argonne National Laboratory
9700 S. Cass Ave. DIS/900
Argonne IL 60439
lee@eid.anl.gov

ABSTRACT

We describe a methodology for transforming sets of database transactions into sets of conflict-free transactions which can be executed independently and in parallel. The method identifies conflicting tuples and uses buffers in high-speed memory to hold those tuples. The separate transactions are then modified to operate on both the database and the relevant buffers.

Keywords: database, transactions, concurrency

1. Introduction

A transaction in a database system is a sequence of database operations that is treated as an atomic unit regarding execution, consistency and the like. However, with the advent of parallel and concurrent systems, a major effort has been devoted to allowing concurrent execution and/or interleaving of the pieces of distinct transactions to increase system throughput. Classic transaction management systems have been developed based on concepts like serializability, correctness, transaction history and two-phase locking ([BHG87], [Pap86], [Ull88]). However, two-phase locking requires the management of locks and suffers a heavy degree of "blocking overhead". These difficulties will increase in importance as technology moves more and more towards concurrent and distributed systems. It is therefore important to develop new ways to increase concurrency in such systems.

Several researchers have considered alternatives ([Gar83], [MRBKS92], [AAJ92], [LKS91]). The main idea is to identify substeps within each transaction that need to be executed atomically. The breakpoints between substeps are points at which concurrency can be considered. However, these techniques place a severe burden on database users to identify steps and breakpoints. Moreover, without additional knowledge about other transactions that are or may be in the system, the improvement in concurrency may be limited.

Our method, inspired in part by work in deductive databases ([HN84], [MH89]), considers general database operations (read, write, modify). For a set of transactions to be executed concurrently, the method identifies the sets of conflicting tuples and arranges to have those tuples read into one or more buffers in high-speed memory. The transactions themselves are then suitably modified to reflect the fact that some of the tuples may be in buffers instead of in the database. Thus, at the cost of some reading into buffers a very high level of parallelism can be achieved. Thus, in a way, our methods attempts to "resolve" conflicts rather than simply "avoid" them as in existing methods. In this paper we are concerned with the concurrency issue. Related details are discussed in [Lee92] and [LH93].

Section2 gives a more detailed description of some of the characteristics of our methods. Section 3 gives some formal definitions and a more detailed description of the method itself. Section 4 contains some concluding remarks.

2. Characteristics of the proposed transaction management model

In this section we compare our model with other existing models by summarizing its characteristics. A more systematic description will be given in Section 3.

2.1 Database operation semantics

Our model follows the direction of [Gar83] [Lyn83] [FO89] of increasing concurrency by exploring the semantics of the transaction. However, our model is different from the existing models. The semantics on which our models are based are not from some ad hoc example but from more general and practical real-life database operations. The interleaving of the database operations that our model achieves does not depend on the analysis by each user of their transaction but is based on the existing database operations used by all the users of relational databases. If we assume that the semantics of the transaction are structured hierarchically [Lyn83], then our model achieves the interleaving at the lowest and finest level.

We studied the concrete "building-blocks" of transactions. We don't require the users to define semantic information required by the system other than submitting the code of the transaction as usual. Our model uses a general mapping mechanism which accepts any transaction code built with the "building block" database operations.

More specifically, the "building-block"s of our model are READ(), INSERT(), DELETE() and UPDATE(), which are the most popular relational database operations used by almost every transaction on relational databases.

2.2 Resolving conflicts and achieving high concurrency

We categorize the database operations into two types - Retrieval Operations and Update Operations. Some existing research results (e.g., [AK91]) have introduced similar categorizations. Most of them assume the categorization at the transaction level, that is they view certain transactions as "read-only" or "write-only" in order to allow a higher degree of concurrency. Our categorization is done inside the transactions. We assume a transaction could have both retrieval and update operations. We categorize the operations in order to analyze the impact of different operations on the database and on other transactions.

We studied the structure and parameters of the database operations. After careful study and analysis, we found that all the database operations within the scope of the semantics we studied can be interleaved by mapping them into another set of database operations within the same semantic scope in which conflict is eliminated and that a very high concurrency can thus be achieved with our model. We have proved the correctness of this result [Lee92] [LH93]. Resolving conflict of database operations is a unique characteristics of our model. As many researchers have noted, the conflict among database operations is the major obstacle to increasing concurrency and efficiency of transaction management systems.

The model regroups the retrieval database operations and update operations into two phases. The model then maps the two groups of database operations into two sets of new database operations which are "conflict-free" and can be executed totally in parallel. Our model not only addresses the conventional concurrency problem, but also can be applied to parallel processing and parallel databases. One difference between "concurrent" and "parallel" is that the set of transactions (or processes) and their order of execution are preserved explicitly in concurrent settings, whereas the set of transactions (or processes) in a parallel system may be split up and therefore preserved only implicitly.

2.3 The correctness

The correctness of our model is still based on the notion of serial execution. We believe that the only suitable basic for determining correctness of a method is to compare against the serial execution model.

There are some other notions of correctness which assume that the resulting database states don't have to be accurate. As long as they fall into some predefined boundary, they would be considered as correct or consistant[Gar83] [WA92]. This approach may be acceptable for some of the applications but not for many other applications. For example, most bank costumers would like to avoid a service charge from an overdrawn account by knowing the exact balance of that account rather than an estimated one.

Our model uses the idea of "Multiversion Serializability" introduced in [BHG87] [KS88] [AK91]. The model insures not only that the database states resulting from the parallel execution of multiple transactions are correct but also that the "image of the database" to all the transactions executed are correct. That is, all the transactions should see a consistent database when they are executed. We assume that the consistency constraints of each transaction are ensured by the user who writes the transaction, the same as all the references mentioned so far regarding transaction management.

We assume buffer-usage for storing intermediate database states to be used by the new sets of database operations. The usage of the buffer may give the impression that our method has a high memory space overhead. However, we argue that theoretically, all the multiversion approaches would have to use about the same amount of space when they get implemented. This fact is almost implied by the word "multiversion". Practically, most of the existing database systems use buffers for every database operation. That means that using buffers is not something unpopular or impractical.

2.4 Deadlock Free

Since the execution of transactions in our model is the dispatching of groups of conflict-free database operations, there is no possibility of deadlock. This result frees the model (or the system to be implemented) from implementing a "deadlock prevention" or "deadlock detecting" mechanism for concurrency control.

2.5 Recoverability

The model adopts the idea of countersteps [Gar83] [Lyn83] [FO89] for recovery in case of transaction failure. However, the "countersteps" or "counter operations" are well defined in our model, and the users don't have to be burdened with this additional responsibility.

The recovery problem of our model is quite different from conventional models. We will address the issue in a separate paper. However, we point out at this time that the model is recoverable, and complexity and efficiency of the recovery process is compatible with other recovery mechanisms proposed so far.

3. A transaction management model with semantics

In this section we will give some definitions and sketches of the algorithms related to our model and a more systematic description of our model.

3.1 Database and Database Operations

Definition 3.1: We adopt the definition of a database state (*DBT*) from [MH89].

Definition 3.2: A Database operation in the new model is defined as a four-tuple $O = (\mathcal{OP}, R, \mathcal{A}, B)$ where $\mathcal{OP}$ is the type of the operation, i.e. DELETE, INSERT, etc., R is the relation the operation is to affect, and $\mathcal{A}$ is the set of attribute values affected by the operation. B is the buffer associated with this database operation. It contains a set of tuples left after the database operation takes place. It may also contain some augmented operations to be performed on the tuples. That is $B = (BS, \mathcal{AO})$.

Definition 3.3: The attribute value set $\mathcal{A}$ of a database operation O is denoted as $\mathcal{A} = ((C_{in}, \mathcal{V}_{in}), (C_{out}, \mathcal{V}_{out}))$ where C_{in}, $\mathcal{V}_{in}$ are the input constant and variable attributes of the operation, and C_{out}, $\mathcal{V}_{out}$ are the output constant and variable attributes of the operation, respectively.

More specifically, the input constant and variable attributes define the set of tuples to be affected by the database operation, and the output constant and variable attributes defines the set of tuples left in the database after the operation taking place.

Examples of database operations are (READ, R ((**a**, **x**), (**a**, **x**)))(INSERT, R, ((), $(a_1, a_2, a_3, a_4, a_5, a_6)$)), (UPDATE, R, (($\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{y}, \boldsymbol{w}$), ($\boldsymbol{a}, \boldsymbol{c}, \boldsymbol{d}, \boldsymbol{w}$))) where $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}$ are generic constants, and $\boldsymbol{y}, \boldsymbol{w}$ are generic variables.

Definition 3.4: The input database state of a database operation is the database state before the database operation takes place.

Definition 3.5: The affected set AS_o of a database operation O is defined as the tuple set of $(C_{in}, \mathcal{V}_{in})$ of O applied to the input database state. That is
$AS_o = (C_{in}, \mathcal{V}_{in})_o\ (DBT)$

Definition 3.6: The resulting set RS_o of a database operation O is defined as the tuple set of $(C_{out}, \mathcal{V}_{out})$ of O applied to the affected set of O. That is
$RS_o = (C_{out}, \mathcal{V}_{out})_o\ (AS_o)$.

Definition 3.7: The output state of a database operation is the database state after the operation takes place. If the input state of O is $\mathcal{DBT}$, then the output state of O is $O(\mathcal{DBT})$. Notice that $O(DBT) = (DBT - AS_o) \cup RS_o$.

Definition 3.8: A retrieval database operation retrieves data from a database into the buffer, and it does not change the database state. For example (READ, R, ((a, x), (a, x))).

Definition 3.9: An <u>update database operation</u> changes the database state, and it does not care about the content of the buffer left after it takes place. For example (DELETE, R, (b, y), ())).

Definition 3.10: <u>The retrieved set of a database operation</u> is the set of tuples retrieved by the operation into the buffer associated with that operation, and we denote it as TS_o. For retrieval database operations, $TS_o = AS_o = RS_o = BS_o$. TS_o is defined only for retrieval operations.

Notice that AS_o, RS_o and TS_o of an operation O associated not only with the operation but also with an initial (or input) database state. Therefore, we will use notations such as $TS_o(DBT)$, $RS_o(DBT)$, etc., to represent the association.

Observation 3.1: Duplicating database tuples in any relation does not change the database state.

Observation 3.2: If O is a retrieval database operation, we have $O(DBT) = DBT$.

3.2 Local Conflict and Global Conflict of Database Operations

In order to study the concurrency feature of transactions we need to investigate the conflict among the database operations within a transaction. It is clear that for retrieval databse operation O only TS_o has meaning to the transaction, and for an update operation only $O(\mathcal{DBT})$ concerns us. According to this observation we try to categorize the conflicts and have the following definitions related to different conflicts.

Definition 3.11: We say that two database operations O_1 and O_2 are <u>locally conflicting</u> if

(1) O_2 is a retrieval operation, and O_1 is either a retrieval operation or an update operation;

(2) $TS_{o2}(DBT) \neq TS_{o2}(O_1(DBT))$.

Observation 3.3: Two retrieval operations O_1 and O_2 are not locally conflicting since we have, in this case, that $O_1(DBT) = DBT$.

Definition 3.12: We say that two database operations are <u>globally conflicting</u> with each other if:

(1) At least one of O_1, O_2 is an update operation;

(2) $O_2(O_1(DBT)) \neq O_1(O_2(DBT))$.

Observation 3.4: A retrieval operation O_1 is not globally conflicting with an update database operation O_2. Since $O_1(DBT) = DBT$, we have $O_1(O_2(DBT)) = O_2(O_1(DBT)) = O_2(DBT)$.

Observation 3.5: Two database operation O_1 and O_2 are not globally or locally conflicting if they operate on two different relations.

3.3 Resolving Conflict

Before we present the new transaction management model, we introduce two theorems related to transactions and to the conflict among the database operations within a transaction. The proofs can be found in [Lee92] and [LH93]. Before the theorems can be introduced we also need a clear definition of the meaning of conflict resolution.

Definition 3.13: If a retrieval operation O_2 is locally conflicting with an update operation O_1 in a sequence of database operations O_1, O_2, by resolving the conflict we mean to convert O_2 into O'_2 so that $RS_{O'2}(DBT) = RS_{O2}(O_1(DBT))$.

Definition 3.14: If two ordered update database operations O_1, O_2 are globally conflicting with each other, by resolving global conflict we mean to convert O_1, O_2 into $\bigcup O'_i$ where $1 \le i \le 4$ so that $\Pi O_i(DBT) = \Pi' O_i(DBT)$ where ΠO_i and $\Pi' O_i$ are any two permutations of $\bigcup O'_i$.

Theorem 3.1: The local conflict of any operation can be resolved.

The proof and resulting algorithm depend on the use of a buffer. There are four cases depending on the order of the conflicting operations - I-R, D-R, U-R, R-R (insert followed by read, etc.) Within each of these four cases there are subcases depending on the relationship (inclusion, equal, mutually exclusive) of the input subsets for the two operations. The complete algorithm and proof are given in [LH93]. We present an example here to give the main ideas.

Example: O1: (DELETE, R, ((a,b,x), ()), ((a,b,x), $\varnothing$))
O2: (READ, R, ((a,y,x),(a,y,x)), ((a,y,x), $\varnothing$))

Here the tuples to be deleted are a subset of the tuples to be read. The idea for this case is to read into a buffer all of the tuples for O2. After this, the DELETE operation can be performed in parallel on the database AND THE BUFFER. Finally, the tuples left in the buffer are used as the output corresponding to O2. After the initial read operation, the operations on the buffer and the database can be done in parallel. Of course, the operations in the buffer would be much faster than the ones on the database because the buffer would be in fast memory. The resulting conflict-free transaction is:

(READ-TO-BUFFER, R, ((*a,y,x*), (*a,y,x*)), ((*a,y,x*), ∅))

(DELETE, R, ((*a,b,x*), ()), ((*a,b,x*), ∅)) (DELETEB, R, ((*a,b,x*), ()), ((*a,b,x*), ∅))
USE the data in the BUFFER

Theorem 3.2: The global conflict of any two update database operations can be resolved.

This time there are seven cases - I-D, D-I,I-U, U-I, D-U, U-D and U-U. Again, each case can have several subcases depending on the input sets of the operations. The complete algorithm and proof are given in [LH93]. We present a simple example here.

Example.: O1: (UPDATE, R, ((*a,z,c,y*), (*a,n,d,y*)), ((*a,n,d,y*), ∅))
O2: (DELETE, R, ((*a,n,x,h*), ()), ((*a,n,x,h*), ∅))

If R contained the tuple (**a,b,c,h**), then O1 changes this to (**a,n,d,h**), which is deleted by O2. On the other hand, if O2 is performed first, this tuple survives, after which O1 changes it to (**a,n,d,h**) and leaves it in R. Our method uses a buffer into which is read all the tuples in the input set of O1. Those tuples are then removed from the database. We are then free to do the delete operation in the database. In parallel we can perform the update on the buffer followed by the delete operation on the buffer. (Recall, buffer operations are relatively fast.) Any tuples left in the buffer are then transferred back to R. This leads to the following transaction

(READ-TO-BUFFER, R, ((*a,z,c,y*), (*a,z,c,y*)), ((*a,z,c,y*), ∅))

(DELETE, R, ((*a,n,x,y*), ()), ((*a,n,x,y*), ∅)) (UPDATEB, R, ((*a,z,c,y*), (*a,n,d,y*)),((*a,n,d,y*), ∅))
(DELETEB, R, ((*a,n,x,h*), ()),((*a,n,x,h*),∅))
(INSERT, R, ((), (*a,n,d,y*)), ((*a,n,d,y*), ∅))

Theorem 3.3: The local conflict of any retrieval operation with any sequence of preceding update operations can be resolved.

Theorem 3.4: The global conflict among any sequence of update operations can be resolved.

These two theorems use the algorithms in Theorems 3.1 and 3.2 above within two nested loops looping on the first database operation and the second database operation chosen from the same set [Lee92]. The order of the algorithms are O(n^2) and O(n^3), respectively.

3.4 The transaction management model

We now have the base to introduce our transaction management model. We will follow the traditional model to define the execution or (history) of transactions because only the execution of a transaction tells the difference between models.

Definition 3.15: A database transaction in our model is defined as a four-tuple (T, $\mathcal{P}$, IDBT, ODBT) where T= $\{O_1, O_2, \ldots, O_n\}$, $\mathcal{P}$ is a total order on T, IDBT is the input database state of the transaction and ODBT is the output database state of the transaction.

We assume that IDBT and ODBT comply with the database constraints defined for the database as we pointed out in Section 1.

Definition 3.16: An execution of a set of transactions T= $\{T_1, T_2, \ldots, T_n\}$ in our model is a four tuple (f, T', $\mathcal{R}$, $\mathcal{X}$), where T' is a set of new subtransactions (or database operations), f is a many to many mapping from T to T' where T'= $\{T_0, T_1, T_2, \ldots, T_m\}$, $\mathcal{R}$ is a total order on T_i, and $\mathcal{X}$ is a mapping from T' to a database state. T_0 is a process which groups the transactions and their database operations according to certain criteria. For each T_i $0 < i$ in T' $T_i = (t_{i0}, t_i, t_{if})$ where t_{i0} is the pseudo-initial transaction for T_i, and t_{if} the pseudo-final transaction for T_i.

t_i in each T_i is $t_i = (O'_{ri}, O'_{ui}, \mathcal{P})$ where $O'_{ri} = \{o_{ri1}, o_{ri2}, \ldots, o_{ril}\}$ is a new set of retrieval operations mapped from the retrieval operation and the update operations of T by f, and $O'_{ui} = \{o_{ui1}, o_{ui2}, \ldots, o_{uik}\}$ is a new set of update operations mapped from the update operations of T by f, and $\mathcal{P}$ is the order relation which has the same definition for all the T_i's in T'. The definition of $\mathcal{P}$ is such that all new retrieval operations o_{rij} precede all the new update operations o_{uij} for a specific i and all j within each t_i. No special order among the retrieval operations or among the update operations is required. Each t_{i0} is a specific execution of f which includes the implementation of the algorithms which resolve the local and global conflicts of the database operations in the original T and maps

T into T'. Each t_{if} in T_i is $t_{if}= (RO'_{ui}, HF_i)$ where $RO'_i= \{ro_{ui1}, ro_{ui2}, \ldots, ro_{uik}\}$ is a set of reverse update operations (or counter operations) corresponding to O'_{ui} mapped from the operations in T by *f*, and HF_i is a house-keeping function which determines none, one, or more than one of the ro_{uij}'s which need to be executed and also passes information to $t_{(i+1)0}$ when needed.

o_{rij}'s and o_{uij}'s are defined as $(\mathcal{OP}_{uij}, R_{ij}, \mathcal{A}_{ij}, B_{ij})$ where the $\mathcal{OP}_{ij}$, R_{ij}, $\mathcal{A}_{ij}$ and B_{ij} are defined as in Definition 2.2 and Definition 2.3.

Notice that the use of B_{ij} and the corresponding BS_{ij} and $\mathcal{A}O_{ij}$ as defined in Definition 2.2 can be seen as creating input and/or output states of database operations as in the existing models of View Serializable and Multiple-Version Serializable [BHG87] [KS88].

Definition 3.17: The execution of a set of transactions $T= (T_1, T_2, \ldots, T_n)$ is correct if the retrieval operations of all the transactions retrieve the same data into the transaction as they do in a serial execution of the transactions, and the update operations leave a final database state which is the same as the final state of a serial execution of the transactions.

More intuitively, our model takes a set of transactions to be executed simultaneously, converts them into two sets of new database operations - one set of new retrieval database operations, possibly with augmented database operations, and a set of new update database operations, possibly with buffers. The model also generates the reverse operation sets based on the input database operations. The model then dispatches the retrieval database operations to be executed in parallel first and then dispatches the update database operations to be executed in parallel next. The model invokes the reverse operations according to the finishing status of the execution. Notice that it is possible that the operations of the original transaction are subgrouped before they get converted into new database operations. This is reflected in the definition as more than one t_{i0}, t_i, t_{if} subtransaction after conversion.

Consider a relation R with attributes: flight number, departure date, departure time, row number, seat number, customer name, destination (including intermediate stops). The relation contains, initially, two tuples

(385, 5/20/93, 1000, 6, A, Cleveland, St. Louis),

(385, 5/20, 1000, 10, A, Schuman, Kansas City).

Consider two transactions T1 and T2.(We will use simplified forms for operations. A "*" in the form represents the same attribute set as the previous one.)

T1: (READ, R, (385, 5/20/93, x_3, x_4, x_5, x_6, Chicago), (*)),((*), $\varnothing$)),

(INSERT, R, ((), (385, 5/20/93, 1030, 6, B, Henry, Chicago)), ((*), $\varnothing$)),
(DELETE, R, ((385, 5/20/93, 1000, x_4, x_5, Schuman, x_7), ()), ((*), $\varnothing$))
T2: (READ, R, ((385, 5/20/93, 1000, x_4, x_5, x_6, x_7), (*)), ((*), $\varnothing$)),
(UPDATE, R, ((385, 5/20/93, 1000, x_4, x_5, x_6, x_7)
(385, 5/20/93, 1030, x_4, x_5, x_6, Chicago)), ((*), $\varnothing$)),
(INSERT, R, ((), (385, 5/20/93, 1030, 6, C, Lee, Chicago)), ((*), $\varnothing$)).

The operations within T1 and T2 are in conflict with each other. Therefore, T2 needs to wait until T1 finishes or until the last database operation DELETE(385, 5/20/93, 1000, x_4, x_5, Schuman, x_7) finishes before it can start.

Our method allows T1 and T2 to execute in parallel if they arrive at about the same time by mapping the operations in T1 and T2 into a set of new operations using local buffers. The new retrieval operations include:

o_{r11} = (READ, R, ((385, 5/20/93, x_3, x_4, x_5, x_6, Chicago), (*)), ((*), $\varnothing$)),
o_{r12} = [(READ, R, ((385, 5/20/93, 1000, x_4, x_5, x_6, x_7), (*)), ((*), $\mathcal{AO}$))

where $\mathcal{AO}$ is:

[(DELETEB, R, ((385, 5/20/93, 1000, x_4, x_5, Schuman, x_7), ()))].

Readers are referred to [LH93] for the mapping.

The new update operations include:

o_{u11} = (INSERT, R, ((), (385, 5/20/93, 1030, 6, B, Henry, Chicago)), ((*), $\varnothing$)),
o_{u12} = (DELETE, R, ((385, 5/20/93, 1000, x_4, x_5, x_6, x_7), ()), ((*), $\varnothing$)),
o_{u13} = (INSERT, R, ((), (385, 5/20/93, 1030, x_4, x_5, x_6, Chicago)),
((385, 5/20/93, 1030, 6, A, Cleveland, Chicago), $\varnothing$))
o_{u14} = (INSERT, R, ((), (385, 5/20/93, 1030, 6, C, Lee, Chicago)), ((*), $\varnothing$)).

Readers are referred to [Lee92] for the mapping.

Other elements of the example according to the model defined above are: $O'_{r1} = (o_{r11}, o_{r12})$, $O'_{u1} = (o_{u11}, o_{u12}, o_{u13}, o_{u14})$, t_{10} is a particular execution of f, $t_1 = (O'_{ri}, O'_{u1}, \mathcal{P}_1)$, $t'_{1f} = (RO'_{ui}, HF_i)$ is the reverse operation set and house keeping functions for recovery, $T = (t_{10}, t_1, t_{1f})$. Notice that in this case only one new-sub-transaction t_1 and its associated pseudo-initial and pseudo-final transactions are mapped. We also assume that the set of new mapped database operations is small enough to be dispatched at the same time.

Readers can easily verify that the result database state after T is the same as the database state for T1 followed by T2 which is

R= ((385, 5/20/93, 1030, 6, A, Cleveland, Chicago),
(385, 5/20/93, 1030, 6, B, Henry, Chicago),
(385, 5/20/93, 1030, 6, C, Lee, Chicago)).

One can also verify that the data retrieved by the two new READ() operations at the beginning of t_1 are the same as the data retrieved by the original READ()s in the sequence of T1, T2.

In the model we assume that the completed definitions of each of the transactions are known at the time of submission of the transaction, which is the normal case for most existing database systems.

4. Closing remarks

We believe the methodology can be applied to other concurrent parallel situations in database technology [WT92]. For example, the algorithm in Section 3 could be used in a parallel batch transaction system or a concurrent system to resolve conflict among distinct transactions submitted within a given time window and to be executed in parallel. In fact, with minimal adjustment, the method can be applied also to the notion of "sagas" for long-lived transactions ([MS87], [Lyn83], [FO89]).

We have presented a new model for transaction management using the semantics explored in other papers to resolve the conflicts among the database operations. This makes our model very different from the existing models and achieves much higher concurrence than existing transaction models. We hope that this work will contribute ideas to research in the area of transaction management. We hope also that this model will give some idea on how the database management system could be implemented on parallel machines.

We would like to say a few more words about the recovery issue. We expect that the recovery process for this model is going to be more complex than the one for existing models in some cases. Reverse processes are needed when failure is detected. In some cases, it could be a re-processing of all the transactions involved. However, since normally the probability of failure is much less than the probability of success in general cases, the advantage of having a highly concurrent model with a more complex recovery process over having a model with very low degree of concurrency and a simpler recovery process should be justified. Besides, the reverse processes or operations are to be generated by the system, rather than by the user as in some other models. This is an advantage over the existing models which use reverse processes (or counter processes).

There is room for further study on related topics. This work has concentrated on concurrent control, not on the issue of recovery. The recovery methods related to this model in case of non-batch processing need to be carefully studied. This paper did not discuss the validation of database constraints related to transaction management and /or to the database operations discussed here, and this needs to be explored. The semantics of other types of database operations such as object-oriented database operations need to be explored further for the purpose of increasing concurrency. How to divide the code which does not access database and to attach the code to parallel database operations is another important and broad area to be studied.

Acknowledgments

Many thanks to Professor Sheuermann and Professor Clifton of Northwestern University for introducing some of the important references for this study and for suggestions on early drafts. The work discussed in this paper was partially supported by the U. S. Department of Energy under contract W-31-109-Eng-38.

References

[AAJ92] D. Agrawal, A. El Abbadi, R. Jeffers "Using Delayed Commitment in Lock ing Protocols for Real-time Databases"; Proceeding of the 1992 ACM SIGMOD; San Diego, California June 2-5 Page: 104-113; 1992.

[AK91] D.Agrawal, V. Krishnaswamy; "Using Multiversion Data for Non-interfering Execution of Write-Only Transactions" Proceeding of the 1992 ACM SIGMOD; Denver, Colorado May 29-31, 19 Page: 98-107; 1991.

[BHG87] P.A.Bernstein, V.Hadzilacos, N.Goodman;"Concurrency Control and

Recovery in Database Systems"; Addison-Wesley Publishing Company Page: 1-45; 1987.

[FO89] Abdel Aziz Farrag, M. Tamer Ozsu; "Using Semantic Knowledge of Transactions to Increase Concurrency"; ACM Transactions on Database Systems; ACM TODS Dec 1989.; Page: 503-525; 1989.

[Gar83] Hector Garcia-Molina; "Using Semantic Knowledge for Transaction Processing in a Distributed Database"; ACM transaction on Database Systems; ACM TODS June 1983.; Page: 186-213; 1983.

[HN84] Lawrence J. Henschen and Shamin A. Naqvi "On Compiling Queries in Recursive First-Order Database"; Journal of the Association for Computing Machinery; JACM Vol.31, No. 1, January 19 Page: 47-85; 1984.

[KS88] Henry F. Korth, Gregory D. Speegle; "Formal Model of Correctness Without Serializability"; Proceeding of SIGMOD International Conference on Management of Data; ACM Chicago IL June 1-3.; Page: 379-386; 1988.

[LH93] Julia C. Lee, Lawrence, J, Henschen; "The Semantics and the Conflicts of Relational Database Operations"; Submitted to Data & Knowledge Engineering Journal; 1993.

[LKS91] Eliezer Levy, Henry F. Korth, Abraham Sil "An Optimistic Commit Protocol for Distributed Transaction Management"; Proceeding of the 1992 ACM SIG MOD; Denver, Colorado May 29-31, 19 Page: 88-97; 1991.

[Lee92] Julia C. Lee "A Transaction Management Model with Relational Database Operation Semantics"; Ph.D Dissertation Northwestern University; Evanston Dec, 1992

[Lyn83] Nancy A. Lynch; "Multilevel Atomicity-A new correctness Criterion for Database Concurrency Control"; ACM Transactions on Database Systems; ACM TODS Dec 1983.; Page: 485-502; 1983.

[MH89] William W. McCune and Lawrence J. Henschen "Maintaining State Constraints in Relational Database"; Journal of the Association for Computing Machinery; JACM Vol.36, No. 1, January 19 Page: 46-68; 1989.

[MRBKS92]
Sharad Mehrotra, Rajeev Rastogi, Yuri Brietbart, Henry F. Korth, Avi Silberschatz "The Concurrency Control Problem in Multidatabases: Characteristics and Solutions"; Proceeding of the 1992 ACM SIGMOD; San Diego, California June 2-5 Page: 288-297; 1992.

[MS87] Hector Garcia-Molina, Kenneth Salem; "SAGAS"; Proceeding of SIGMOD,; ACM San Francisco, CA, May 27- Page: 249-259; 1987.

[Mal89] Carl Malamud; "INGRES Tools for building an Information Architecture"; Van Nostrand Reinhold; Page: 57-188; 1989.

[Pap86] C. Papadimitrio; "The Theory of Database Concurrency Control"; Computer Science Press,; Computer Science Press; Page: -; 1986.

[Ull88] Jeffrey D. Ullman; "Chapter 9 Transaction Management"; Database and Knowledge-base Systems; Computer Science Press; Page: 467-542; 1988.

[WA92] M. H. Wong & D. Agrawal; "Tolerating Bounded Inconsistency for Increasing Concurrency in Database Systems"; Proceeding of ACM 11th Principles of Database Systems; San Diego, CA 1992.; Page: 236-245; 1992.

[WT92] Paul Watson and Paul Townsend; "The EDS Parallel Relational Database System"; Parallel Database Systems; Lecture Notes in Computer Science Page: 212-222; 1992.