Improving Execution Concurrency for Long-Duration Database Transactions*

Alex N.J. Wu[†]

Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan 300

Arbee L.P. Chen

Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan 300

Abstract

This paper presents an approach for processing long-duration database transactions with high concurrency degree. The basic idea is based on the use of a repository which stores the data items that can be exposed before the associated transaction commits. The management of the repository is described. Since we allow a transaction to read and update the early exposed data items, if the data items are invalidated we have to rollback the transactions that have read these data items. In order to reduce the cost of rollback, a partial rollback mechanism is proposed. Further, transactions that have read the early exposed data may commit earlier than the corresponding uncommitted transactions. We describe the commit decision rule to determine whether earlier commitment can be allowed, by which the system throughput can be increased.

1 Introduction

Conventional database management systems have been designed primarily to support the transaction-oriented business applications. A transaction as used in conventional applications has two properties: atomicity and serializability. The atomicity property means that all the reads and writes in a transaction are regarded as a single atomic action. It ensures that either all the operations in a transaction must be complete or none of them be done. The serializability property [10, 7, 8, 9, 11] means that the effect of concurrent executions of more than one transaction is the same as that of executing the same set of transactions one at a time.

The conventional transaction management systems are not suitable for transactions whose duration is much longer than that of conventional transactions. Two-phase locking will cause a long-duration transaction to hold a lock and block other transactions which need to access the locked data in a conflicting mode until the long-duration transaction finishes. The atomicity property will cause a long-duration transaction to back out all the previous results if a failure occurs just before it finishes.

In this paper, we propose an approach to satisfy the requirements of speeding up the execution of the long-duration transactions and still preserving serializability to guarantee the data consistency. With these facilities, users can cooperate efficiently in a long last environment. Early exposure of partially completed data is essential for long-duration transactions. Stable data [1, 2] are data which have been read and updated by a transaction and will not be further referenced by the transaction before its completion. Our main idea on improving the execution concurrency of long-duration transactions is based on the early exposure of stable data, and the appropriate management of these data to satisfy the serializability property. We assume that there is a repository in which the stable data are stored. When a data item become stable, it is written to this repository. By means of accessing to the repository, the other transactions can read and update the stable data. Again, after the data item is further modified and become stable, it is written back to the repository for more data sharing.

It is noteworthy that when a transaction reads the stable data of an uncommitted transaction, it cannot commit before the uncommitted transaction. Since the result produced by a committed transaction cannot be undone, the commitment has to be controlled in an appropriate manner. On the other hand, once the uncommitted transaction aborts, the transactions which have ever read the stable data should be rolled back accordingly. When rolling back these transactions, it is only required to partially roll back to the point where the stable data were read [4]. This can be accomplished by writing a special mark (savepoint) in the log when the stable data are read.

From the above statements, data can be shared before becoming permanent, thus the concurrency degree can be improved. We provide a commit decision rule to decide whether the earlier commitment can be allowed. In that case, the total system throughput will be increased.

The remainder of the paper is organized as follows. Section 2 provides the related research work. The model of our approach and the mechanisms required are presented in Section 3. Section 4 describes the com-

---

*This research was partially supported by the Republic of China National Science Council under Contract No. NSC 83-0408-E-007-029.

[†] The author is working for Institute for Information Industry Technology Research Division

mit decision rule of this approach. In Section 5 we illustrate our approach by some cases. Section 6 concludes our work and states our future work.

2 Related Research

Conventional database management systems consider a transaction as an atomic unit of work. However, executing a long-duration activity as a single transaction can significantly delay the execution of the other transactions. To solve the problem, the notion of *saga* is proposed as a model for long-running activities. An activity is composed of a sequence of transaction steps $T_1, T_2, \ldots, T_n$. When T_i finishes, it commits and then T_{i+1} is invoked. If T_i fails, then T_i is aborted and the compensating transactions $C_{i-1}, \ldots, C_2, C_1$ are invoked to eliminate the effects of earlier committed transactions. In general, the specification of the compensating transactions must be provided by the application programmers.

A transaction can be formed as a hierarchy of subtransactions [6]. This *nested transaction* model is an important extension of the conventional transaction model, allowing more semantics to be captured and greater concurrency to be achieved. An extension of the nested transaction model is proposed, which governs the execution of subtransactions by rules with different coupling modes is proposed.

These approaches provide the features that a long transaction can be regarded as a set of short transactions. These subtransactions can be organized systematically to allow more parallelisms. However, the intertransaction concurrency is not well considered. In our approach, mechanisms are developed to improve intertransaction concurrency degree.

Some approaches involve the concept of public and private databases to manage data sharing among transactions [3, 2]. A transaction can check out data from the public database and the private databases of the other transactions. A primary advantage of this approach is to allow a designer to check out the partial design and complete the design. Kim [2] presents a model of engineering transactions which augments existing models by refining the notion of checkout environment and coupling it with the notion of nested transactions.

The recovery in nested transactions using savepoint is discussed in [4]. Savepoints are exploited to support a finer grained transaction UNDO to allow partial rollback of ongoing transactions. It also serves as a restart point when some problems are encountered.

Version control is one of the most important data modeling requirements in the next-generation database applications [12]. Data with different versions are also helpful in the teamwork environments. Cooperative users can communicate with each other through these different versions of data such that long occupation of data can be alleviated.

Klahold et al. [3] organizes different versions of a design object by a *version graph*. This approach presents a mechanism by which a transaction can operate on the versions. However, some of the operations, e.g., the merging of versions, need to be done manually.

The approach mentioned in [12] distinguishes a version into a *transient version*, a *working version* and a *released version*. The concept of versioning can also be found in our approach. Since we allow an early exposed data items to be shared by others transactions, they can be regarded as working versions. The working version in [12] can only be read, while our approach also allows updates on a working version.

3 Improving Transaction Concurrency

3.1 Basic Definitions

Before expounding this approach, the transaction model and some terminologies are defined as follows.

Definition 1 A transaction is a collection of ordered operations called *execution pattern* (EP) which changes a set of data from one consistent state to another. We can represent a transaction T as:

$$T : (s, EP_k) \rightarrow s'$$

The notation s denotes all the data items read and updated by an execution pattern (EP_k) of T. After the execution, s will be changed into s'. A transaction is terminated only when committed or aborted. Notice that depending on the execution flow of a transaction, a transaction can be associated with different EP's.

Definition 2 An operation OP is *compensatable* if there exists an inverse operation of OP such that

$$d = OP^{-1}(OP(d))$$

A transaction is said to be *compensatable* if all its operations are *compensatable*. In this paper, we consider that all the transactions are compensatable. Not every operation has an inverse operation in the real case. Thus we have to keep track of each modification of data. This information can be used as an undo log to achieve the compensatability.

Definition 3 A rollback caused by reading invalid stable data is called a *turnback*. In essence, a turnback performs a partial rollback to the point where the stable data were read. A transaction T_j is *turnback dependent on* T_i if the abortion of T_i causes the turnback of T_j. The set of all transactions which are turnback dependent on T_i is denoted by $T_{TD}(T_i)$.

3.2 Mechanisms

3.2.1 Dangling Object Repository

Dangling Object Repository (DOR) is a place in which all stable data are stored. Each stable data item is associated with a table *Tab* in the DOR. The *Tab* associated with data item x is denoted by $Tab.x$.

In addition to writing the updated value of a stable data item to the DOR, some important information should also be recorded. This information is stored in *Tab* with the schema shown in Table 1. **T**, **V** and **OP** indicate the new value of the data item updated by the operation of the transaction denoted by **T**. S implies that the transaction is running, aborted

Table 1: Schema of *Tab*

Tid	Value	OPer	Status	ChkInstp	ChkOutstp

or committed. **I** means the timestamp of the data item checked in by transaction **T**. **O** means the timestamp of the data item checked out by transaction **T**. $Tab.x[\mathbf{V}_i]$ denotes the **V** field of the ith entry in the *Tab* of data item x. The notation is similarly applied to other fields. When a transaction reads a data item from a *Tab* in the DOR, this *Tab* is locked to prevent the others from accessing the same data item. We assume a data item is always read for a later update, thus locks in our approach are all exclusive locks. A lock is released after the associated data items is written back to DOR. New record is written at the first available entry of the associated *Tab*. Suppose there are m entries in a *Tab*, we have the following:

$$\{\mathbf{T}_{k+1},\ldots,\mathbf{T}_m\} \subseteq T_{TD}(\mathbf{T}_k)$$
$$\textit{where } k \textit{ is an index of the}$$
$$\textit{entries in the } Tab, 1 \leq k \leq m-1.$$

When a transaction issues a read command on a data item which is locked by the lock manager, this command is forwarded to the DOR server. The DOR server checks the availability of the data item in the DOR, and lets the transaction check out the data item if it is in the DOR and is not locked at the moment. When a checkout or a checkin is performed, the timestamp should be recorded in the corresponding field.

3.2.2 Rollback and Turnback

Once a transaction T aborts, the transaction manager has to inform the DOR server to find all the transactions which are turnback dependent on T. We give an example shown in Figure 1 to illustrate how to find those transactions which have to be turned back.

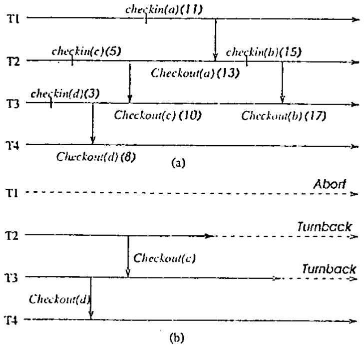

Figure 1: (a) A Snapshot of Executions (b) Transaction T1 Aborts

Figure 1.(a) manifests the checkins and checkouts by each transaction. The stamps associated with each checkin and checkout are also indicated. For example, a is checked in by T_1 at stamp 11 and checked out by T_2 at stamp 13. If T_1 aborts, T_1 has to roll back totally. However, T_2 has only to turn back to the savepoint associated with the checkout of a. The turnback of T_2 causes T_3 to turn back to the savepoint where T_3 checks out b, since the result of T_2 before stamp 13 will not be affected by the turnback of T_2. For the same reason, the result of T_3 before stamp 7 will not be affected by the turnback of T_3, transaction T_4 need not be turned back at all. Thus, $T_{TD}(T_1) = \{T_2, T_3\}$. Figure 1.(b) evinces the condition of rollback and turnback. The dash lines indicate the rollback and turnback portions. Note that if a transaction checked out more than one invalid stable data, it has to turn back to the savepoint associated with the earliest invalid data item.

From the above illustration, if T_i aborts, its turnback dependent transactions can be found by the following steps:

- all the transactions that have directly checked out the stable data items of T_i are ***turnback dependent on*** T_i.
- these transactions must be turned back to the savepoint associated with their checkouts from T_i; the checkout is called ***invalid checkout***.
- if a transaction has checked out a data item which was checked in after the ***invalid checkout***, it also needs to be turned back. Of course, the transaction is turnback dependent on T_i and the checkout becomes an ***invalid checkout***.
- the above procedure is applied repeatedly until no new transaction can be found to be turnback dependent on T_i.

3.2.3 Checkout Dependency Graph

We use the DOR to allow a transaction to expose its stable data before commitment. Accessing to the DOR should be concerned to avoid violating serializability property. Consider the following case: According to the Figure 2, we will get the following

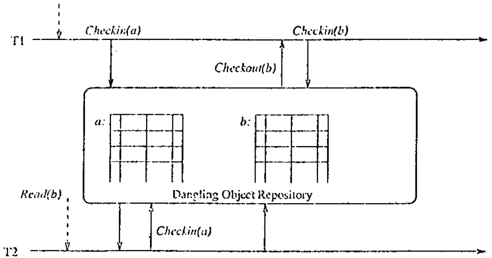

Figure 2: A Case which Violates the Serializability

schedule:

$R_1(a)\ R_2(b)\ \underline{W_1(a)\ R_2(a)}\ W_2(a)\ \underline{W_2(b)\ R_1(b)}$
$W_1(b)$ □

Obviously, the above schedule is not a serializable schedule. We should prevent the above situation. For this purpose, we use a *checkout dependency graph* to detect such a scenario.

Definition 4 A checkout dependency graph is denoted $G_i = (T, E)$, where T is the set of nodes corresponding to transactions that have checked out object i, and E is the set of edges e, where e is a directed edge from T_y to T_x if T_y checks out the data which is checked in by T_x. Each time when a transaction checks out a data item, the corresponding edge should be added. Once a transaction T_i commits, all the edges correspond to T_i are deleted.

Lemma 1 An execution which has checked out data from the DOR is serializable with respect to the DOR if the checkout dependency graph $G = \cup_i G_i$ is acyclic.

Once a transaction has to check out a data item, we have to test whether the checkout will result in a cycle in the checkout dependency graph. If yes, the transaction cannot perform the checkout since the cycle implies that the checkout will make the schedule nonserializable.

4 Commit Decision

A transaction once committed, by definition, cannot abort. It is required that if transaction T_j reads the value of a data item written by T_i, then T_j cannot commit until T_i commits. A system with such a property is said to be *recoverable* [5]. However:

```
Initial: a = 500
T1:                        T2:
begin-transaction          begin-transaction
  Read(a)
  a = a - 200
  Write(a)
                             Read(a)
                             if a > 100 then
                               a = a + 100
                             else
                               a = a + 200
                             endif
  Read(b)
  Read(c)
  c = c - b
  Write(c)
                             Write(a)
end-transaction            end-transaction
```

In this case, no matter T_1 commits or aborts, T_2 will always add 100 to a. Under this condition, we allow T_2 to commit before T_1. If T_1 successfully completes its execution then *400*, the value of a, is written back to the database. However, if T_1 aborts, *600* is written back instead. If we leave the decision, whether we should write *400* or *600* back to the database, to the *last active transaction* that holds the lock on a, we can commit T_2 before T_1.

4.1 Commit Decision Rule

To determine whether a transaction $T_j \in T_{TD}(T_i)$ can commit before T_i, we have to decide whether the execution patterns of T_j are identical no matter T_i commits or aborts. By Definition 1, a transaction changes one database state to another through an execution pattern. Different values of data may alter the execution pattern [5]. Consider the conditional branch of the form:

$$if\ (\ p\ )\ then\ OP_1\ else\ OP_2$$

where OP_1 and OP_2 are two different operations, and p is a predicate whose truth value depends on the content of data.

We say that two execution patterns are equal if they have the same number of operations and their corresponding operations are the same. Without loss of generality, we assume that T_1 reads a data item a with data value x, thereafter a is updated to y by T_1. Now T_2 checks out a after T_1 checked in it to the DOR. Two different execution patterns are possible for the following execution depending on whether T_1 commits or aborts. T_2 can be represented as follows:

$$T_2 : \begin{cases} ((y, v_1, v_2, \ldots, v_n), EP_1) \rightarrow s_0 & \text{if } T_1 \text{ commits} \\ ((x, v_1, v_2, \ldots, v_n), EP_2) \rightarrow s_1 & \text{if } T_1 \text{ aborts} \\ & \\ \text{where } v_1,\ \ldots,\ v_n & \\ \text{are read from the database.} & \end{cases}$$

Now, if T_2 wishes to commit but T_1 is still running, we have to check whether execution patterns EP_1 and EP_2 are equal. If the answer is yes, we can commit T_2, otherwise T_2 should wait until T_1 commits or aborts.

4.2 Equality Check of Execution Patterns

To achieve the effect of earlier commitment, we have to determine whether the possible execution patterns spanned by a transaction are equal. The checking procedure is invoked between a checkout and a checkin performed by a transaction. We give an example shown in Figure 3 to illustrate how to check the equality of the execution patterns.

Tid	Value	Operation	tag
T1	400	+100	1
T2	200	-200	1
T3	700	+500	-1

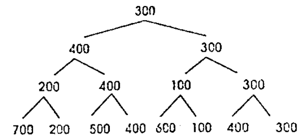

Figure 3: Equality Check of Execution Patterns

Initially, the value of a is *300*. The left hand side of Figure 3 indicates the operations on a performed by T_1, T_2 and T_3. The right hand side is a decision tree. The root of the decision tree is the initial value of a. The ith level of the decision tree corresponds to the ith entry in the *Tab.a*. Each node in the tree has two children. The left child is the value of a assuming the corresponding transaction commits, while the right child is the value of a assuming the corresponding transaction aborts.

As Figure 3 shows, the possible values of a read by T_2 are *400* and *300*. Assume no matter the value of a is *400* or *300*, T_2 always subtracts *200* from a. We can get the four nodes as shown in the second level. T_2 is then positively tagged in *Tab.a* to indicate that its execution patterns with respect to the checkout of

a are equal. After the operation of T_2, there are four possible values of a to be tested by T_3. We assume if the value of a is less than *250* then T_3 adds *500* to a, otherwise T_3 adds *100* to a. Since the possible values of a generated by T_2 are *200*, *400*, *100* and *300*, T_3 will span two different execution patterns. Therefore, T_3 is negatively tagged. After the checkin of a by T_3, we get the third level of the decision tree as shown.

If all the entries corresponding to the transaction in the *Tab*'s are positively tagged, we say that the execution patterns of this transaction are equal.

4.3 Finalizing the Data Values

We know so far that if a transaction $T_j \in T_{TD}(T_i)$, T_j may be allowed to commit before T_i. According to the ***durability*** property of a transaction, we have to make the effect of an operation performed by the committed transaction permanent. A transaction may ***delegate*** the responsibility for finalizing its effects on some of the objects to another transaction. To achieve this goal, the DOR server must reflect the final value of the data object to the database when all the transactions participate some *Tab* commit or abort. One important thing has to be determined is what have to be done when a transaction T aborts while some of $T_{TD}(T)$ had already committed or still in running.

If a transaction T aborts, the situations can be divided into the following categories:

- All the transactions belong to $T_{TD}(T)$ are still running, then $T_{TD}(T)$ have to be turned back to the corresponding checkin point. Those entries belong to $T_{TD}(T)$ in each *Tab* have also to be deleted.

- Some of the transactions belong to $T_{TD}(T)$ are running however some had already committed. In this case, the running transactions have to be turned back, however the **V** fields for the committed transactions should be recomputed by the following rule

$$\left\{\begin{array}{l}
Tab.d[\mathrm{T}_k] \text{ aborts and } Tab.d[\mathrm{T}_j] \in T_{TD}(\mathrm{T}_k) \\
\\
\textit{if } Tab.d[\mathrm{T}_j] \text{ is the first committed entry after} \\
\quad Tab.d[\mathrm{T}_k] \\
\textit{then} \\
\qquad Tab.d[\mathrm{V}_j] = Tab.d[\mathrm{OP}_j](Tab.d[\mathrm{V}_{k-1}]) \\
\textit{else} \\
\qquad Tab.d[\mathrm{V}_j] = Tab.d[\mathrm{OP}_j](Tab.d[\mathrm{V}_i]) \\
\\
\text{where j is the index corresponding to the} \\
\quad \text{committed transaction, i is the last committed} \\
\quad \text{transaction index before j}
\end{array}\right.$$

Since the data value for the committed transaction had to be recomputed after some transaction aborts. We have only to reflect the *Tab.d's* final entry's V field to the database when all the transactions in *Tab.d* had committed.

5 Transaction Executions

This section elaborates the transaction execution process under various situations. Before illustrating the following paragraph, we separate the transactions in an acyclic checkout dependency graph into the *head*, *body* and *tail* transactions which correspond to the root, internal nodes and leave nodes in the graph respectively. Transactions execution as shown in Figure 4 can expressed by Table 2. We start our discussion by this example:

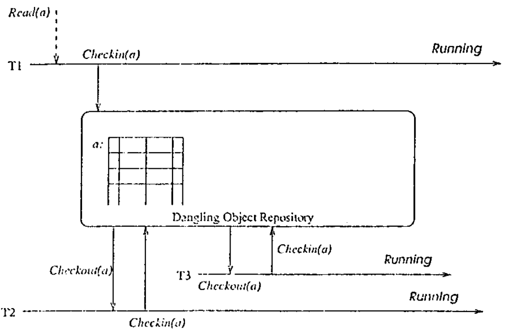

Figure 4: All Participating Transactions Are Running

Table 2: The Content of *Tab.a* While All Transactions Are Running

Tid	Value	OPer	Status	ChkInstp	ChkOutstp
T_1	1900	-100	Running	10	18
T_2	2100	+200	Running	20	26
T_3	2600	+500	Running	28	34

5.1 One Participating Transaction Aborts

To discuss the case that one participating transaction aborts, we delimit it into two subconditions: The abort transaction is a tail transaction or a non-tail transaction. It is rather easy to understand that when the abort ransaction is a tail transaction, we only have to rollback the tail transaction. The abort does not cause any effect on the other transactions. We discuss the case when the abort transaction is a non-tail transaction.

Transaction T_2 is a non-tail transaction. To abort T_2, we have to find all the transactions that are turnback dependent on it. In other words, we have to compute $T_{TD}(T_2)$ then every transaction belongs to $T_{TD}(T_2)$ has to be turned back accordingly. Table 3 shows the *Tab.a* after the rollback and turnback.

5.2 One Participating Transaction Commits

In this subsection, we discuss the case where one transaction commits. We discuss a case in the following paragraph, where the committed transaction is a non-head transaction. Suppose T_3 wishes to commit while T_1 and T_2 are still running. As described in

Table 3: The Content of *Tab.a* While A Transaction Aborts

Tid	Value	OPer	Status	ChkInstp	ChkOutstp
T_1	1900	−100	Running	10	18
T_2	2100	+200	Abort	20	26

Section 4, T_3 can commit only when its execution patterns are all equal. We assume that all the execution patterns of T_3 are identical, thus we can commit T_3 before T_1 and T_2. Since T_1 and T_2 are still running, *a* will not be reflected in the database by T_3. Instead, T_3 has to upward propagate its operation on *a*. After that we can get the *Tab.a* as shown in Table 4

Table 4: The Content of *Tab.a* While A Transaction Commits

Tid	Value	OPer	Status	ChkInstp	ChkOutstp
T_1	1900	−100	Running	10	18
T_2	2100	+200	Running	20	26
T_3	2600	+500	Commit	28	34

Now, if we assume transaction T_2 aborts after T_3 has already committed. Under this case, the DOR server has to inform all transactions contained in $T_{TD}(T_2)$ and status is running to turnback accordingly. Further, those transactions whose status is commit have to recompute their **V** field to reflect the new value of data by the rules mentioned in section 4.3. The reult is shown in Table 5.

Table 5: A Transaction Aborts After Its Turnback Dependent Transaction Commits

Tid	Value	OPer	Status	ChkInstp	ChkOutstp
T_1	1900	−100	Running	10	18
T_2	2100	+200	Abort	20	26
T_3	2400	+500	Commit	28	34

The cases demonstrated above represent the primitive cases. The real situation may be a combination of these primitive cases. However, they can be processed with the similar procedures.

6 Conclusion

This paper introduces an approach tailored to the requirements of the long-duration transactions. The existing methodologies can be classified into two categories. One is to view a long-duration transaction as a sequence of short transactions, and the other approach relaxes the requirements for atomicity and serializability of transactions.

The differences between our approach and the existing methodologies are that we view a long-duration transaction as a single atomic unit such that the properties of transactions are preserved. It is not difficult to find when a data item becomes stable in a simple transaction. However, when a transaction becomes complex, a flow analysis mechanism for finding the stable data is needed. Also, a performance comparison with other approaches is required to show the usefulness of our approach. These works are currently under investigation.

References

[1] Calton Pu, Gail E. Kaiser, Norman Hutchinson "Split-Transaction for Open-Ended Activities", *Proc. 14th International Conference on Very Large Database*, Aug, 1988.

[2] Won Kim, Raymond Lorie, Dan McNabb, Wil Plouffe "A Transaction Mechanism for Engineering Design Databases", *Proc. 10th International Conference on Very Large Database*, Aug, 1984.

[3] P. Klahold, G. Schlageter, R. Unland, W. Wilkes "A Transaction Model Supporting Complex Applications in Integrated Information Systems", *Proc. of the ACM SIGMOD*, Austin, Texas, 1985.

[4] Theo Haerder, Kurt Rothermel "Concept for Transaction Recovery in Nested Transactions", *Proc. of the ACM SIGMOD*, San Francisco, May 1987.

[5] Henry F. Korth, Eliezer Levy, Abraham Silberschatz "A Formal Approach to Recovery by Compensating Transactions", *Proc. 16th International Conference on Very Large Database*, Aug, 1990.

[6] J. Moss "Nested Transactions: An Approach to Reliable Distributed Computing" *MIT Labtory for Computer Science*, MITLCSTR-260, 1981.

[7] Weimin Du, Ahmed K. Elmagarmid, Won Kim "Maintaining Quasi Serializability in Multidatabase Systems", *International Conf. on Data Engineering*, 1991.

[8] Weimin Du, Ahmed K. Elmagarmid "Quasi Serializability: A Correctness Criterion for Global Concurrency Control in InterBase", *Proc. 15th International Conference on Very Large Database*, Aug, 1989

[9] Calton Pu, Avraham Leff "Replica Control in Distributed Systems: An Asynchronous Approach", *International Conf. on Data Engineering*, 1991.

[10] Henry F. Korth, Abraham Silberschatz "Database System Concepts", *McGRAW-Hill International Editions*, ISBN: 0-07-100804-7.

[11] Yuri Breitbart, Avi Silberschatz, Glenn R. Thompson "Reliable Transaction Management in a Multidatabase System", *Proc. of the ACM SIGMOD*, Atlantic City, NJ, May 1990.

[12] Won Kim, "Introduction to Object-Oriented Databases", *The MIT Press*, ISBN: 0-262-11124-1, 1991