Asynchronous Reading of Consistent Copy of a Large Database

Subhash Bhalla, Stuart Madnick

Sloan WP # 4152  Cisl WP #00-08
December, 2000

MIT Sloan School of Management
50 Memorial Drive
Cambridge, Massachusetts 02142-1347

ASYNCHRONOUS READING OF CONSISTENT COPY OF A LARGE DATABASE

SUBHASH BHALLA† AND STUART E. MADNICK‡

Abstract. To recover from media failures, a database is 'restored' from an earlier backup copy. A recovery log of transactions is used to roll forward from the backup version to the desired time (the current time). High availability requires - backup activity to be fast, and on-line with on-going update activity. It necessitates, obtaining a consistent copy of an entire database. Such concurrent generation of a database copy, interferes with system activity. It introduces blocking and delays for many update transactions. We study the performance of revised algorithms, to highlight the level of concurrent activity permitted by these algorithms, in parallel. Subsequently, the interference between global database copy activity and transaction updates is minimized based on a new algorithm for asynchronous generation of a consistent copy of the database.

Key words. High-speed backup, parallelism in Concurrency Control, system recovery

1. Introduction. Taking frequent backup is an essential part of database operations. Many applications require reading an entire database. Recently proposed algorithms [18, 19, 3] introduce delays for other executing transactions. Consider a few earlier examples [3], an accounts officer may want to read total deposits, or an inventory official may need to read the entire inventory to compile a stock report, or a computer operator may wish to take a database backup without suspension of services. If a conventional two-phase locking protocol is used to maintain database consistency, then the global-read transaction renders a considerable part of the database inaccessible for the update transactions. Other, new algorithms also have some drawbacks. For example, these algorithms, either do not guarantee serializability [18, 19], or these restrict certain type of transactions [3, 18, 19], and thus, allow very few concurrent updates.

A transaction is an update transaction, if it updates any part of the data within the database. A read-only transaction is a transaction that does not update any piece of data but reads a part of the database. A transaction is two-phase if it acquires locks during a growing phase, and releases locks during a shrinking phase. An update transaction acquires exclusive locks (write locks). A read-only transaction acquires the read (shared) locks, that can be shared among executing read-only transactions. A global-read transaction is an incremental transaction that acquires read locks for reading, a few at a time. It releases locks and acquires more locks in small steps, until the entire database has been read.

A global-read produces an inconsistent database version, in the presence of update transaction execution. Let us take an example, considered by Pu, and Amann et al. [18, 3]. Suppose, it is desired to calculate the total deposits in a bank, with a global-read that incrementally sums the checking and savings accounts. If a client, executes an update transaction to move a thousand dollars from a savings to a checking account during a global-read, the summation may be a thousand dollars short, in case, for example, if checking account is read by global-read, before it is updated (assuming, the savings account is read after it has been updated).

There have been a few recent research studies on reading a copy of the database

* Database Systems Laboratory, University of Aizu, Aizu-Wakamatsu, Fukushima 965-8580, JAPAN. (bhalla@u-aizu.ac.jp).
† Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA 02139, USA (smadnick@mit.edu).

A proposal for exploiting histories for database backup has been studied in [16]. We have considered transaction level backup and recovery. The rest of the chapter is organized as follows. In the following section, an early global-read algorithm has been studied with a view of making a system model for simulation and comparison. Section 3 considers a revised proposal of the algorithm. In section 4, and 5 new proposals for global-reading have been put forward. Sections 6, 7 and 8 consider Proof of correctness, performance issues, and highlight some of the related studies. Section 9 outlines summary and conclusions.

2. The Original Global-Read Algorithm. In the original proposal of Pu [18], a global-read transaction divides the database in two parts. The items read by global-read are treated to be black, and those not read as yet, are considered as white items. An ordinary transaction can update the database within one of the two portions, black or white. All other transactions are not permitted. In addition to the restriction, it was shown by the later study [3], that the algorithm does not guarantee serializability. The proposal considered by Amann and Jajodia [3] improves upon the original proposal. However, it prohibits more transactions from executing during the execution of a global-read transaction.

For a study of these algorithms, we made a simulation model for the initial proposal. This model has been tested for validity. The same model has been extended for simulation and comparison of results obtained by the revised proposals.

The original global-read algorithm changes the color of the entities within the database, along with the progress of the global-read transaction (Tgr). The other transactions executing in parallel are divided into 4 categories, based on the type of items in their write-sets (WS), as shown in Figure 2. Initially, all items are colored as white, at the beginning. The early transactions tend to be white as these update white entities. With the progress of Tgr, the number of gray transactions increases. During the final phase, most transactions tend to be black, as these update database items that have already been read by Tgr. From the point of view of serializability, all black transactions are considered to have occurred after the global-read. Similarly, the
white transactions are considered to occur before the global-read transaction, because their updates are incorporated before the global-read transaction \((T_w \rightarrow T_{gr} \rightarrow T_b)\).

### Table

<table>
<thead>
<tr>
<th>Other Parallel Transactions</th>
<th>Definition</th>
<th>Notion of Serialization with global-read Transaction ((T_{gr}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-only Transaction ((T_r))</td>
<td>(WS = \phi)</td>
<td>Concurrent</td>
</tr>
<tr>
<td>White update Transaction ((T_w))</td>
<td>(WS = \text{white entities})</td>
<td>Before global-read</td>
</tr>
<tr>
<td>Black update Transaction ((T_b))</td>
<td>(WS = \text{black entities})</td>
<td>After global-read</td>
</tr>
<tr>
<td>Gray Transaction ((T_g))</td>
<td>(WS = \text{white and black entities})</td>
<td>Rejected</td>
</tr>
</tbody>
</table>

Figure 2. Serialization of Parallel Transactions in Pu’s [18] algorithm

\[ T_w \rightarrow T_{gr} \rightarrow T_b; \quad \text{Reject } T_g \]

For sake of simplicity, we assume that the update transactions access the entities in the database uniformly. Let us consider a database with \(n\) entities, of which \(r\) have been painted as black. An update transaction that writes on \(k\) items, will not conflict with a \(T_{gr}\), if all its write-set entities are either, black or white. The probability of this event is -

\[
P_{n,k}(k) = \frac{C_r^k}{C^n_k} + \frac{C_{n-r}^k}{C^n_k}
\]

The probability of an abort in the case of gray transactions, after algebraic transformations [19] is given by -

\[
P(k) = \frac{2}{n+1} \sum_{r=k}^{n} \prod_{i=0}^{k-1} \frac{r-i}{n-i}
\]

These probabilities can be computed for various values of \(k\). The proportion of aborted transactions increases with \(k\). For \(k = 2, 3, 4, 5\), we obtain - 0.333, 0.500, 0.600, and 0.667, respectively [18]. For a normal range of values of \(k\) (between 5 - 10), more than 65 - 80 % aborts are likely to be encountered.

#### 2.1. Simulation Model

In order to evaluate the performance of the revised copy algorithm, a simulation model based on C language programs has been made. The simple model uses the following input parameters and output measurement variables (Table 1). The validity of the model was tested by adopting the above formulation for the original global-read algorithm [18]. The results obtained by the simulation model are similar to the results from the probability computations shown above in (2) (please see Figure 2.1 [21], [6]. Possibilities of existence of synchronization errors
were shown to exist within the algorithm by Pu [18]. A revised algorithm was proposed by Amann, et al. [3]. For example, a cyclic precedence could be setup, if a white transaction, read items updated by a black transaction (\( T_w \rightarrow T_{gr} \rightarrow T_b \rightarrow T_w \)). The revised algorithm aborts, few additional white transactions, for sake of correct synchronization activity. In our study, the simulation model was extended to incorporate additional output parameters. In the next section, we describe the revised proposal of Amann et al. [3]. In the subsequent section, we examine ways to achieve an asynchronous global-read transaction, with low overhead.

3. Revised Global-read Algorithm. The revised global-read algorithm [3] divides the database entities into two colors: white (not read as yet) and black (have been read). In addition to color, a shade is associated with data entities that have been read or written by a transaction (Figure 3.1). A transaction is colored as white, if it acquires update locks for white entities. A transaction is colored as black, if it acquires update locks for black entities. A gray transaction that attempts to update a mixture of both white and black entities is not permitted. Further access restrictions are shown in Figure 4.(b). The revised algorithm does not permit the following data operations, and transactions.
### Input Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of entities in the database</td>
<td>1000</td>
</tr>
<tr>
<td>Number of entities read by a transaction ((r))</td>
<td>1 - 6</td>
</tr>
<tr>
<td>Number of entities updated by a transaction ((k))</td>
<td>1 - 6</td>
</tr>
<tr>
<td>Distribution of the values of (r)</td>
<td>uniform</td>
</tr>
<tr>
<td>Distribution of the values of (k)</td>
<td>uniform</td>
</tr>
<tr>
<td>Distribution of number of transactions per unit time (k)</td>
<td>uniform</td>
</tr>
</tbody>
</table>

### Output Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Measurement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of white transactions</td>
<td>relative proportion</td>
</tr>
<tr>
<td>Number of black transactions</td>
<td>relative proportion</td>
</tr>
<tr>
<td>Number of aborted (gray) transactions</td>
<td>relative proportion</td>
</tr>
<tr>
<td>Number of aborted white transactions</td>
<td>relative proportion</td>
</tr>
<tr>
<td>(by the revised global-read algorithm)</td>
<td>relative proportion</td>
</tr>
</tbody>
</table>

Table 1. Input parameters and Output measurement variables.

- All gray transactions that write on both black and white items;
- A white transaction cannot write on an off-white entity; and
- A white transaction cannot read an off-black entity.

As shown in Figure 4.1., the revised global-read algorithm further increases the number of transactions to be aborted. The results from the extended simulation model are shown in Table 2.

<table>
<thead>
<tr>
<th>Other Parallel Transaction</th>
<th>Definition</th>
<th>Notion of Serialization with global-read Transaction ((T_{gr}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-only Transaction ((T_r))</td>
<td>(WS = \phi)</td>
<td>Concurrent</td>
</tr>
<tr>
<td>White update Transaction ((T_w))</td>
<td>(WS = \text{pure white entities})</td>
<td>Before global-read</td>
</tr>
<tr>
<td>Prohibited white transactions</td>
<td>(WS = \text{off white};) or (RS = \text{off black})</td>
<td>Rejected</td>
</tr>
<tr>
<td>Black update Transaction ((T_b))</td>
<td>(WS = \text{pure black entities})</td>
<td>After global-read</td>
</tr>
<tr>
<td>Gray Transaction ((T_g))</td>
<td>(WS = \text{white and black entities})</td>
<td>Rejected</td>
</tr>
</tbody>
</table>

Figure 3.2 Serialization of Transactions in the revised algorithm

\[ T_w \rightarrow T_{gr} \rightarrow T_b; \]

Reject \(T_g\), also reject \(T_w\), if \(WS = \text{off-white};\) or \(RS = \text{off-black}\)
State Transitions for colors of entities.

FIG. 3.1. Transaction access restrictions for colored and shaded data.

4. Save Some Strategy. We consider various ways for permitting more transactions to improve parallel execution among executing transactions. The gray transactions aim at updating white entities, and black entities. As a way, for executing these transactions, we propose to isolate their updates to white entities as a special case, and hide these from the global-read transaction. The database system needs to make a copy of the older version of data, so that the global-read transaction $T_g$ can read a consistent earlier version. This strategy allows $T_g$ to be treated as black transactions that seem to occur after the global-read, in the assumed serialization order for execution of database transactions. The $T_g$ are treated as $T_b$ by the algorithm for the subsequent processing activity. Similarly, the white transactions, that are rejected by the revised algorithm, are permitted by allowing the updated white
entities (and off-black entities) to keep an earlier version of the data. Thus, by making these transactions act as black transactions \( T_b \), their updates do not interfere with the \( T_{gr} \), which reads an earlier version of the data.

This proposal, requires additional storage of earlier versions of data. In Figure 7.1, the additional storage space needed for private buffers, is shown by shaded (gray) area. The given size of the database (1000 entities) was selected to create a simulation of a sufficiently high throughput environment (please see Figure 5.1). In real life systems, the relative proportion of the additional buffers may be less. In the worst case, an additional 100% disk space may be needed.

<table>
<thead>
<tr>
<th>( k )</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Percentage of white ( T_w ) transactions,</td>
<td>49.6</td>
<td>32.8</td>
<td>24.5</td>
<td>19.3</td>
<td>16.4</td>
<td>13.8</td>
</tr>
<tr>
<td>Percentage of black ( T_b ) transactions,</td>
<td>50.4</td>
<td>33.6</td>
<td>25.1</td>
<td>20.8</td>
<td>17.8</td>
<td>14.8</td>
</tr>
<tr>
<td>Percentage of gray transactions, ( T_{gr}(A) )</td>
<td>0.0</td>
<td>33.5</td>
<td>50.4</td>
<td>59.9</td>
<td>65.8</td>
<td>71.4</td>
</tr>
<tr>
<td>Percentage of rejected white transactions (B)</td>
<td>24.6</td>
<td>15.7</td>
<td>10.6</td>
<td>4.9</td>
<td>3.1</td>
<td>0.8</td>
</tr>
<tr>
<td>Total Percentage of rejected transactions (A +B)</td>
<td>24.6</td>
<td>44.0</td>
<td>55.7</td>
<td>61.9</td>
<td>66.9</td>
<td>71.6</td>
</tr>
</tbody>
</table>

Table 2. Proportion of aborted transactions (Revised global-read algorithm)

5. **Save All Strategy.** In the 'Save Some' approach, the effects of a few white transactions are picked up by the global-read transaction. We consider an approach for incorporating the changes made by all ongoing transactions. We have considered a different set of data entity classes. In Figure 8, the state transition diagram for data entities is shown. In this proposal, the above (gray and prohibited white) transactions are permitted and are termed as mixed transactions. A transaction that reads pure white or pure black entities, but only writes on pure white entities is called a white transaction. A black transaction may read pure black or pure white entities, but can only write on gray entities. All data items accessed and updated by a mixed (gray) transaction are colored as gray. The revised algorithm executes in two phases.

In phase 1, it generates a modified log during the execution of a global-read transaction. This log (called the color log) contains a color marking for each update transaction. On completion of the global-read, the data read by the global-read contains an inconsistent version of the database. In phase 2, modifications are applied to make the data consistent, as shown in Figure 5.3. An algorithm to generate the color log and for later generation of a consistent database version is described in the following section.

5.1. **Transformation of Database State.**

Based on the idea of 'State Transformation' by the committing transactions, we propose an implementation of a global-read transaction scheme that uses a history of committed transactions. A list of committed transactions is generated by the system as a log, during the execution of a global-read transaction. This pool of information
5.2. Proposed Approach.

In this section, we provide steps for implementation of a global-read transaction processing system based on the ideas of transactions’ logs and subsequent update activities. These remove some of the deficiencies associated with concurrent generation of a database copy. The proposed technique is capable of supporting multiple global-reads, and higher levels of replication of data, and moderate volume of update activity. The global-read transactions do not interfere with the update transactions. The various aspects of the proposed scheme are discussed below.
5.3. **Global-read Processing**

The global-read transaction ($T_{gr}$) divides the database entities into black and white colors. At the beginning, all database entities are white. Gradually, entities read by a $T_{gr}$ are colored black. In addition, all entities that are written by black or mixed transactions are colored as gray. White data entities that are read by black or mixed transactions are also colored as gray. Thus all data entities are colored as white, or black or gray. Normal update transactions are colored white or black depending on the color of data entities being updated by them. A transaction is termed as mixed, based on the following conditions,

- a transaction that updates a mixture of black and white data entities;
- a transaction that reads a data entity that is colored as gray; and
- a transaction that writes on a data entity that is colored gray.

The color of a transaction can be determined at the time of its commit by ex-
aminaing the color of data items in its, read-set and write-set. Normal read-only transactions are not colored and proceed normally subject to the constraints of the two-phase locking protocol. A revised scheme for processing a global-read algorithm is shown in Figure 5.3. Algorithm for generation of a consistent database copy after a global-read is presented in Figure 6.

5.4. Information Management

The color of all data entities is set as white, at the beginning of a $T_{gr}$. It changes to black immediately after a read access by a $T_{gr}$ transaction. Of the update transactions, only black or gray transactions can change the color of a data entity to gray. The change of color must occur before the black or gray transaction releases the shared
Asynchronous Reading of Consistent Copy of a Large Database

Before the start of a $T_{gr}$, a Transaction-id is assigned to each transaction on its arrival. It contains information related to site-of-origin, site-status, transaction type (read-only class/duration class/priority class; if known), time of the day, and local sequence number (site-status value indicates the Commit Sequence Number (CSN) of the most recent transaction committed).

```plaintext
/* Begin phase 1 -- Serial Log with Color */
BEGIN
  csn = 0;
  WHILE global-read
    FOR each transaction commit
      csn = csn + 1 and
      WRITE (csn, Transaction-id, Read-set, Write-set, color)
        in color log
    END WHILE
  last-csn = csn
END

/* Begin phase 2 -- Over write transactions */
BEGIN
  FOR csn 1, last-csn
    IF color = 'gray' WRITE transaction Write-set into DATABASE copy
  END

Figure 6. Proposed algorithms for Generation of Database Copy

In phase 1, the algorithm carries out an on-the-fly reading (no consistency) of the contents of the database. A color log of gray transactions $T_g$, is created. In phase 2, the algorithm generates a consistent copy of the database by overwriting the updates from its color log. The algorithm (through the color log), maintains a table of entries of completed gray transactions as a Committed Transaction List (CTL). This list contains entries for:

1. Transaction-id;
2. Read-set and associated details such as site at which access was made and the status of the site at the time of the Read-access;
3. Write sets and data values; and
4. Allotted Commit Sequence Number (CSN). These numbers are assigned in ascending order, i.e., Next CSN value = Previous CSN value + 1.

6. Proof of Correctness. While the earlier proposals avoid inconsistency by not allowing certain transactions to commit, this proposal permits a normal execution activity during the execution of the global-read transaction. Consistency is achieved by updating the inconsistent copy of the database by adopting a 'missed update transaction' approach. A log of the 'updates in progress' is used for creating a consistent version of the data, by over-writing on the inconsistent copy. All such updates that could have been missed partially or fully, are rewritten on the database copy, during phase 2. These updates by gray or black transactions, can generate inconsistency. Concurrent updates by white transactions are read by the global-read transaction,
The proposed algorithm considers only one global-read at a time, on a centralized database. Its extension to multiple global-reads and for considering a distributed system has been considered in [18]. It has been omitted here for reasons of simplicity. A formal proof in the case of the proposed algorithm is similar to the proof in [18]. It is being avoided, to reduce duplication. Informally, a white transaction writes on (only) pure white entities (Figure 7.1). These transactions do not affect the contents of the copied database. Each black and mixed transaction is selected (from the color log) and its updates are over-written (applied over the inconsistent copy in sequence). This eliminates the chances of data inconsistency incurred due to a lack of concurrency control. This approach (use of update history) is also used by database recovery algorithms. These use incremental logs with deferred updates, in a similar manner. A brief outline of the proof of consistency is presented for the proposed approach.

6.1. Proof of Consistency. An informal proof is being presented on similar lines as [3, 8, 18].

A transaction is a sequence of n steps:

\[ T = ( (T, a_1, e_1), \ldots, (T, a_i, e_i), \ldots, (T, a_n, e_n) ) \]

where \( T \) is the transaction, and \( a_i \) is the action at step i, and \( e_i \) is the entity acted upon at the step. A schedule \( S \) of transaction dependencies, can be written as:

Considering a mix of an update transaction and a read transaction, that access (read) a data entity ‘e’, Dependency(S) exists as \( (T_1, e, T_2) \), such that e is the output of \( T_1 \), and input of \( T_2 \), such that, either \( T_1 \), or \( T_2 \) updates data entity ‘e’. Two schedules, Dependency(S1) and Dependency(S2) are equivalent, if Dependendency(S1) = Dependendency(S2). A schedule \( S \) is consistent, if there exists an equivalent serial schedule. All schedules formed by well formed transactions following 2-phase locking are consistent [8].

Assertion 1: Given that, \( T_1 \) is a read-only transaction. A schedule \( S \) of transaction dependencies \( (T_1, e, T_2) \), can be transformed to schedule \( S' \), as \( (T_2, e, T_1) \), if all writes of \( T_2 \), are later overwritten on data read by \( T_1 \).

- In the given setup, white transactions precede, the black transactions (please see [3] for the proof).
- white transactions precede, the global-read transaction (by definition);
- For various items accessed by a gray transaction, two kind of dependency relations can exist. \( (T_{gr}, e_i, T_{gray}) \), and \( (T_{gray}, e_j, T_{gr}) \). The dependencies, \( (T_{gr}, e_i, T_{gray}) \), are transformed to \( (T_{gray}, e_j, T_{gr}) \) during phase 2, by Assertion 1.
- global-read transaction \( T_{gr} \) precedes black transactions (by definition).

For all the above transactions, global-read transaction is preceded by white transactions and is followed by black and gray (mixed) transactions.

7. Performance Considerations.

If the load on the system is low, then relatively few transactions are aborted as gray transactions by the earlier algorithms. The performance of the revised algorithm, also closely approximates the earlier algorithm, as few entities are shaded as
off-white, or off-black. However, in case of a heavy load of transactions, a number of update transactions will be rejected due to the presence of long running global-read transactions. Such rejects are at a peak, when global-read has read 50% of the data. Assuming that, update transactions seek their data items randomly, nearly 50% of all submitted update transactions need to face rejects, at the peak time. In practice, in a heavy load system, copying critical parts of the database will lead to many transaction rejects for such durations of time.

In the case of the proposed algorithm, no update transactions are blocked by the algorithm, during execution of phase 1, and phase 2. The main overheads are the delay in generation of database copy during the phase 2 processing. The size of the color log is not significant because a separate medium can normally be used to create serial logs. Also, the overhead can be avoided by merging the color log with normal transaction recovery logs. The generation of transaction logs are a matter of routine processing for database systems [5]. The addition of color markings within the conventional recovery logs does not add significant additional overheads.

The proposed algorithm does not introduce any blocking except requesting shared locks for a small portion of the database. On completion of a global-read, an off-line computation is capable of generating a consistent copy of the database. If the load on the system is light, then relatively few entities are colored as gray. This leads to a small size colored log, that is used for updation of the database copy. In this way, performance of the proposed algorithm is better than the original and the revised algorithms.

The normal recovery logs, that are generated as a part of the transaction processing activity, can be generated as color logs. A global-read can complete phase 1. The subsequent phase 2, can be completed off-line by using data from recovery logs (usually available on a separate volume or medium). In the case of non-availability of color logs, the off-line processing of phase 2, can be done by using normal data recovery log. There will be an increase in time of true copy generation. This is because, applying all updates (with no marking of color) will take more time.

The performance improvement alternatives high lighted by the earlier proposals [18, 19, 3] as turn-white or turn-black propose a re-read operation for the global-read. These approaches are not efficient as these increase the read time, and introduce recursive computational overheads.

8. Related Work

There have been many recent research studies that concern the global-reading of a consistent copy of a database [12, 13, 14]. Lomat [14] considers buffer-management level recovery and generation of backups. There have been a number of earlier studies that aim at transaction level backup and recovery [11, 10, 9]. Supporting long-lived read transactions under the rules of concurrency control based on 2-phase locking, has been studied by [7, 1, 2, 20]. Adoption of 2-phase locking, renders a large part of the database inaccessible, and is not efficient for long running transactions. Using multi-versions for supporting long running transactions has been studied by [4, 17].

An overview of related research activities is described and compared in [3]. Also, obtaining a copy of the entire database is a special case of Long-lived transactions.
The proposed approach differs from all the earlier proposals by offering to turn a non-stop read (blind read with no concurrency control), into a consistent database copy. The second phase of the algorithm can be carried out (optionally) in an off-line manner. Also, the overheads related with generation of a color log, can be avoided by large applications, by modifications of normal recovery logs, and adapting these for global-read. On the other hand, the small size applications can depend on normal recovery logs and avoid the color logs, and the overheads completely.

9. Summary and Conclusion. The paper presents a study and comparison of algorithms for reading a consistent version of a database. It also highlights an algorithm for processing global-read of an entire database copy. In this scheme, the other update transactions that execute in parallel face no blocking of update activity. To the best of our knowledge, it is an unsolved problem. Earlier algorithms achieve global-reading by restricting on-going transaction processing activity. The proposed algorithm is simple to implement. It preserves the consistency of the backup version, as per the notion of serializability.
REFERENCES