On-the-Fly, Incremental, Consistent Reading of Entire Databases

Calton Pu

Department of Computer Science
University of Washington

Abstract

We describe an algorithm to read entire databases with locking concurrency control allowing multiple readers *or* an exclusive writer. The algorithm runs concurrently with the normal transaction processing (on-the-fly), and locks the entities in the database one by one (incremental). We prove the algorithm produces consistent pictures of the database. We also show that the algorithm aborts a minimal number of updates in the class of on-the-fly, incremental, consistent algorithms.

On-the-fly, incremental algorithms to read entire databases consistently can improve system availability and reliability. Most existing systems either require the transaction processing to stop, or produce potentially inconsistent results. Our algorithm does not change the database physical design, so it can be adapted to existing systems by expanding their lock table. Finally, we extend the algorithm in a straightforward way to read entire distributed databases.

1 Introduction

In many situations we would like to read (i.e. access without modification) an entire database. For example, a bank officer may want to know the total amount of deposits, or a computer operator may need to make a backup copy of the database (usually called a *checkpoint*). The data in a database must satisfy certain assertions called *consistency constraints*. In order to preserve data consistency under concurrent access, the usual locking concurrency control allows multiple readers *or* an exclusive writer. A common assumption in the literature is that a consistent and complete picture can be obtained only with a quiescent database. The reason is that 2-phase locking [3] —necessary for consistency— would require a naive reader of the entire database to lock all data at least for a moment, thus updates must stop.

Our work differs from existing literature for three main reasons:

1. Our algorithm reads the database entities one by one (it is *incremental*), avoiding deadlocks and allowing update activities to proceed concurrently (it works *on-the-fly*).
2. Its interference with update transactions is shown to be minimal in the class of incremental, on-the-fly algorithms.
3. We extend the algorithm to produce consistent pictures of entire distributed databases.

In addition, there are two characteristics facilitating its implementation. First, our algorithm consumes modest hardware resources; it does not maintain extra copies of the database and produces only sequential output. Second, no additional disk storage is required, so only modifications on the concurrency control is needed to adapt the algorithm to existing database systems.

We should note that there is no problem checkpointing databases that permit concurrent readers *and* a writer. In principle, any database that maintains two versions of its data can provide this level of concurrency [2]. However, for efficiency reasons, most practical databases write in-place. Our work is aimed at these systems.

The paper is organized as follows. The algorithm is described in section 2. In section 3, we prove the consistency of its output and show the interference with update transactions is minimal. Section 4 outlines the extensions to distributed databases and further improvements. Comparison with related work and applications of the algorithm are included in section 5. Finally, the results are summarized in section 6.

[1]This work was partially supported by the National Science Foundation under Grant No. MCS-8004111.

2 The Algorithm

2.1 Definitions and Introduction

In order to describe our problem and its solutions more precisely, some terms need to be defined. A database is a set of *entities* [3]. Each entity can be individually read through shared locks or written under an exclusive lock. We will reserve the term *checkpoint* to denote a query reading the entire database. Normal transactions on the database will be referred to as either *update transactions* or *read-only transactions* [5].

Consider a naive checkpoint strategy: the entities are read one by one. It is easy to see that a checkpoint processed this way may not be consistent. For example, suppose we want to calculate the total amount of deposits in a bank by summing up the checking accounts first and then the savings accounts. If a client moved a million dollars from his savings to his checking account during the checkpoint, the result would be one million short of the real amount. The key idea of this paper is that *only* this kind of updates can make this naive checkpoint inconsistent.

The algorithm has three parts. First, the checkpoint reads entities one by one. Second, entities in the database are divided by the checkpoint into two subsets: entities not yet read (*white*), and the ones already processed (*black*). Third, update transactions writing both white and black entities are not allowed to commit, because they may not be serializable with respect to the checkpoint. In this section, for simplicity of presentation, the algorithm processes only one checkpoint at a time on a centralized database.

2.2 Basic Checkpoint Algorithm

The following data structures are needed in the volatile storage, as an addition to the lock table:

- One *entity color bit* per entity. (Entities can only take one of two "colors", black or white.)
- One *paint bit* per database, used in a trick to repaint all entity color bits.
- Accompanying the paint bit we have a *checkpoint semaphore* to guarantee only one checkpoint runs at any time.

At database (lock table) initialization time, the paint bit is copied onto all entity color bits. Checkpoints can start only after all entity color bits agree with the paint bit. We also assume the update transactions will not start until the initialization is complete. In case of a crash, the recovery consists simply of a re-initialization.

The basic checkpoint algorithm in figure 1 has several properties. First, the checkpoint locks and reads the entities one by one, so the checkpoint will not cause deadlocks. Second, the checkpoint does not use additional disk access other than the necessary entity reading. Third, in order to adapt the algorithm to an existing database, its physical design (disk format) does not have to be changed.

The basic checkpoint terminates when all entities are painted black. This will happen some time because every loop in step 2 paints another entity black. The while loop will not be blocked until all remaining white entities are exclusively locked. At that time, the checkpoint queues a lock request and eventually will succeed given a fair lock management.

2.3 Concurrency Control

As we have seen in our banking example, the naive checkpoint algorithm alone may produce inconsistent pictures. The checkpoint's consistency is maintained by ensuring that all update transactions writing both white and black entities (*gray transactions*) are aborted. In order to enforce this rule, if a checkpoint is in progress, every update transaction needs to pass an additional *color test* before it can execute and commit. After the acquisition of all exclusive locks (before commit), the color bits of exclusively locked entities have to be checked. If all color bits are the same, the update can proceed, otherwise it is aborted. Please note that if no checkpoint is executing, all entity color bits are the same and the updates always pass the color test.

```
{ Pre-condition: all entity color bits are the same as the paint bit (black). }
step 1:   P(semaphore)          { Checkpoint runs in a critical section. }
          Change the paint bit.        { This re-paints all entities white. }
step 2:   WHILE there are white entities {This loop paints the white entities black. }
          DO BEGIN
               IF all white entities are exclusively locked { Unordered set optimization. }
                    request shared lock on a white entity and wait until lock is granted
               ELSE       lock any sharable white entity;
                    read entity, change entity color, release entity lock.
          END WHILE       { All entity are black, the same as the paint bit. }
step 3:   V(semaphore)          { Let the next checkpoint go. }
```

Figure 1: Basic Checkpoint

Informally we argue that the remaining transactions and the checkpoint are consistent:

1. When the checkpoint paints all entity color bits white, all uncommitted update transactions become white (writing only white entities).
2. All white transactions terminate and are serialized before the checkpoint. (The checkpoint reads the white entities after the white transactions have released their exclusive locks.)
3. When the checkpoint terminates, all uncommitted update transactions must be black (only black entities remain).
4. All black transactions are serialized after the checkpoint. (The checkpoint has read all black entities before they were painted.)
5. Other read-only transactions do not conflict with the read-only checkpoint.

This informal argument is summarized in table 1. An important observation is that once an update transaction has the exclusive lock on an entity, that entity's color will not change during the transaction. This happens because the checkpoint can only paint an entity it has a shared lock on. A more formal proof of checkpoint consistency follows in section 3.

2.4 Entity Creation and Deletion

The creation of new entities require special attention from the concurrency control mechanism. As we have seen in section 2.3, the gray transactions are aborted. So at commit time, we have three possibilities. The entity-creating transaction may be white, black, or colorless. We will consider each case in turn.

First, a white transaction has written on at least a white entity. In this case the entities it creates should be painted white. Since at least one white entity is exclusively locked by the white transaction, the checkpoint will wait for it and read all new white entities when the update commits.

Second, a black transaction has written on black entities only. Since the checkpoint will not come back and read the updated black entities, the newly created entities should be painted black so the checkpoint will not read them.

Third, a colorless transaction has not written on existing entities but has created new entities. The new entities must be painted black so the checkpoint will not read them. We should not paint these entities white because of a race condition at the checkpoint termination.

There is no difficulty with deletions. We assume the entities being deleted are locked exclusively. White transactions are serialized before the checkpoint, so deleted white entities will not be seen by the checkpoint, as expected. Black transactions are serialized after the checkpoint, so black entities are deleted after the checkpoint has read them. The checkpoint reflects all black entities it has read, as it should.

3 Proof of Consistency

3.1 More Definitions

Eswaran et al. [3] defined formally the terms to be introduced here. A transaction is a sequence $T = ((\mathbf{T}, a_i, e_i))_{i=1}^{n}$ of n steps where $\mathbf{T}$ is the transaction name, a_i is the *action* at step i and e_i is the entity acted upon at step i. Typical actions are lock, unlock, read and write. Any sequence obtained by collating the actions of transactions $T_1, \ldots, T_n$ is called a *schedule* for $T_1, \ldots, T_n$. A schedule is *serial* if actions from one transaction form a contiguous group without interleaving with actions from other transactions.

Informally, the *dependency* relation induced by schedule S, $DEP(\mathbf{S})$ is a set of ternary relations $(\mathbf{T}_1, e, \mathbf{T}_2)$ such that entity e is an output of T_1 and an input of T_2 where one of the transactions T_1 or T_2 updates the entity e. We say that two schedules, $\mathbf{S}_1$ and $\mathbf{S}_2$ are *equivalent* if $DEP(\mathbf{S}_1) = DEP(\mathbf{S}_2)$. A schedule $\mathbf{S}_1$ is *consistent* if it has an equivalent serial schedule.

A *well-formed* transaction locks an entity e only once, works on it, and unlocks it. A *two-phase* transaction requests locks during its growing phase, and after entering the shrinking phase, signaled by the first unlock, the transaction cannot issue a lock action on any entity. Finally, in a *legal* schedule, no two transactions can hold the same lock at the same time. We will need the theorem 8*d* from [3]: if transactions $\mathbf{T}_1, \ldots, \mathbf{T}_n$ are each well-formed and two-phase then any legal schedule is consistent.

3.2 Basic Checkpoint Consistency

We assume that normal transactions are well-formed and two-phase in order to avoid foreign inconsistencies. A checkpoint $\mathbf{Q}$ is well-formed but not two-phase, because it unlocks an entity before it locks the next one. Since the checkpoint is a read-only transaction, there is no conflict with other read-only transactions so it is sufficient to show that $\mathbf{Q}$ can be serialized with respect to update transactions except the gray ones. The proof divides the actions on the database into three periods: before $\mathbf{Q}$, during the execution of $\mathbf{Q}$, and after $\mathbf{Q}$.

Table 1: Summary of Checkpoint Serialization

transaction type	read-only	white updates	black updates	gray updates
checkpoint serialization	concurrent	before checkpoint	after checkpoint	incompatible

Lemma 1 *The schedule of transactions committed before the checkpoint* **Q** *has started is consistent.*

Proof: Since we assumed that all normal transactions are well-formed and two-phase, applying theorem 8*d* from [3] we have lemma 1.

Theorem 1 *The schedule* S_0 *of transactions committed during the execution of (and including)* **Q** *is consistent.*

The proof of theorem 1 builds a serial schedule S_3 based on S_0 and shows that $DEP(S_3) = DEP(S_0)$, thus proving the consistency of S_0. The idea is to characterize the class of gray transactions and show that their elimination is sufficient to achieve a serial schedule.

We start by extracting a sub-schedule S_1 from S_0, with the elimination of all **Q** actions. We need to remember the dependencies taken out in this occasion. First we define the subset $DEP(Q)$.

Definition 1 $DEP(Q)$ *is a set of ternary relations* (T_1, e, T_2) *from* $DEP(S_0)$ *satisfying*
either *1.1)* $T_2 = Q$ *and* T_1 *is a white transaction writing on e;*
or *1.2)* $T_1 = Q$ *and* T_2 *is a black transaction writing on e.*

Now we can capture the checkpoint dependencies in our first equality lemma.

Lemma 2 $DEP(Q) = DEP(S_0) - DEP(S_1)$

Proof: ($\subseteq$) By construction, relations defined by conditions 1.1) and 1.2) are both members of $DEP(S_0)$ but not $DEP(S_1)$.

($\supseteq$) Consider any relation (T_1, e, T_2) in $DEP(S_0)$ but not in $DEP(S_1)$. Either T_1 or T_2 has to be **Q**, otherwise it would be in $DEP(S_1)$. First case, (T_1, e, Q): T_1 has to be white because e is input to **Q**. Second case, (Q, e, T_2): T_2 has to be black because e comes from **Q**. □

Proceeding in our construction, we observe that by introducing a checkpoint and aborting some updates, no inconsistency is created.

Lemma 3 *The schedule* S_1 *is consistent.*

Proof: The consistency of S_1 is guaranteed by the underlying concurrency control mechanism because all transactions in S_1 are well-formed and two-phase. In addition, **Q** does not update any entity in the database. □

The next building block in our construction is the monotonic color change of entities:

Lemma 4 *There is no relation* (T_1, e, T_2) *in* $DEP(S_1)$, *such that* T_1 *is black and* T_2 *is white.*

Proof: An entity starts white, and **Q** atomically paints it black. So once a black transaction T_1 has seen an entity e black, no transaction in S_0 can see it white. □

Now we can separate white transactions from black ones:

Lemma 5 *There is a serial schedule* S_2 *equivalent to* S_1, *such that all white transactions precede all black transactions.*

Proof: S_1 is consistent by lemma 3, so there is an equivalent serial schedule S_{11}. For every entity e, lemma 4 guarantees the precedence of white transactions in S_{11}. Moreover, If a black transaction happens to precede a white transaction in S_{11}, they certainly access disjoint sets of entities. Therefore we can swap their contiguous positions in the schedule without changing the dependency. Doing this repeatedly we have S_2. More concisely,
S_2 = (white transactions, black transactions). □

The final step in our construction is to put the checkpoint **Q** back in the middle, making:
S_3 = (white transactions, **Q**, black transactions).
We still need to show that S_3 is equivalent to S_0, so we have a second equality lemma.

Lemma 6 $DEP(Q) = DEP(S_3) - DEP(S_2)$

Proof: ($\subseteq$) Because $DEP(S_2) = DEP(S_1)$, relations defined by conditions 1.1) and 1.2) cannot be in $DEP(S_2)$. However, those relations must be in $DEP(S_3)$ because **Q** is the same.

($\supseteq$) Similar to the proof of lemma 2, replacing S_0 with S_3, and S_1 with S_2. □

Finally we have theorem 1. Schedule S_3 is serial by construction. Lemmas 2, 5, and 6 show that $DEP(S_0) = DEP(S_3)$, so S_0 and S_3 are equivalent. Therefore S_0 is consistent.

The last theorem guarantees that no problem can occur after the checkpoint is complete.

Theorem 2 *The schedule of transactions committed after* **Q** *has terminated is consistent and those transactions can be serialized after* **Q**.

Proof: Since all remaining transactions are normal, there is a consistent schedule. We only have to show that there is no white transactions running after **Q** has terminated. Suppose there is a white transaction still running; it must have at least one exclusive lock on a white entity. This contradicts the hypothesis that **Q** has terminated and all entities have been painted black. □

Combining lemma 1, theorems 1 and 2, we conclude that the checkpoint can be serialized with respect to all normal transactions, except the gray ones which are aborted.

3.3 Minimal Update Interference

In the previous section we have shown that there is a price in breaking the 2-phase locking, namely that all gray transactions which write on two or more entities are aborted. Update transactions which write only on one entity receive the color of that entity and are never aborted because of the checkpoint. Read-only transactions are not affected by our concurrency control modification at all.

It is not hard to see that this price is necessary for all incremental algorithms that read and write each entity only once.

Theorem 3 *In order to obtain a consistent checkpoint of the database incrementally, without locking the entire database, and reading/writing each entity only once, it is necessary and sufficient to distinguish the entities already read by the checkpoint (white) from those to be read (black) and abort the transactions which update both white and black entities.*

Proof: The sufficient part has been done in section 3.2. The necessary part is based on the earlier banking example. We need to show that any incremental (non-two-phase locking) checkpoint algorithm, which allows a gray transaction **G** to commit, cannot guarantee the checkpoint consistency.

Consider such a non-two-phase checkpoint algorithm. There must be a part of database that is locked during the early stage of a checkpoint **Q** and unlocked before some other part of the database is locked. (Otherwise **Q** must be two-phase and lock the entire database at some time.) Let us call the first part being unlocked "black" and the next part being locked "white". The allowed gray transaction **G** first writes on a white entity e_w while **Q** is reading the black part, and waits for **Q** to unlock the black part, then writes a black entity e_b. Since the checkpoint reads and writes every entity only once, we have both $(\mathbf{Q}, e_b, \mathbf{G})$ and $(\mathbf{G}, e_w, \mathbf{Q})$ in $DEP(\mathbf{G})$. Consequently **G** cannot be serialized either before or after **Q**.

4 Extensions

4.1 Distributed Databases

In this section we extend the basic checkpoint algorithm to read a distributed database. Again we assume an underlying concurrency control which handles the consistency of logical entities. Issues such as physical replication and partial unavailability of database are beyond the scope of this paper.

The basic checkpoint can be processed in parallel by dividing the set of entities into subsets. A coordinator (figure 2) would start the checkpoint and the server processes (figure 3) to read the subsets; the coordinator then waits and synchronizes the completion of server processes. The initialization and concurrency control modifications are similar to the centralized case.

Although we have replicated the paint bit using the critical section created by the checkpoint semaphore, there is only one paint bit value. Only one paint value implies that only one checkpoint can execute at any time in the distributed database.

The modification on the concurrency control is the same as in the centralized system. Since Section 2.3 applies directly to centralized concurrency control methods, we need only look at what is added by the distributed concurrency control. The distributed color test is a straightforward extension. All local processes test their own entity color bits for uniformity. At commit time, the transaction manager receives the commit votes with their colors. The transaction is committed only if all votes have the same color. Other possible extensions in a distributed concurrency control like replication do not impact our algorithm, which assumes that the possession of a shared lock is sufficient to guarantee entity consistency.

A harder question is what to do when a part of the distributed database is not available or crashed. As stated, our coordinator process will block, waiting for the server to

```
{ Pre-condition: all entity color bits are the same as the paint bit. }
step 1:   P(semaphore)          { Checkpoint runs in a critical section. }
          Change the paint bit.        { Painting all entities white. }
step 2:   Send the new paint bit to the server processes, starting them. See figure 3.
          Wait for their completion.
step 3:   All servers terminated: merge the results if necessary;
          V(checkpoint semaphore)         { Let the next checkpoint go. }
```

Figure 2: Coordinator Process

```
step 1:   Receive the new paint bit from the coordinator. See figure 2.
step 2:   WHILE there are white entities { Entity color bit different from new paint bit. }
              DO BEGIN
                  IF all white entities are exclusively locked
                      request a shared lock on a white entity and wait until lock is granted
                  ELSE lock any sharable white entity;
                      read entity, change entity color, release entity lock.
              END WHILE { All entity are black, the same as the paint bit. }
step 3:   Return to the coordinator.
```

Figure 3: Server Processes

read the unavailable part. If the definition of "entire database" includes the unavailable part, this is the correct procedure. When the unavailable part is reincorporated to the distributed database, a new server process should be started to complete the checkpoint. The recovery from a coordinator crash consists of certifying that the coordinator is indeed dead (which may be non-trivial) and initializing a new one.

4.2 Further Extensions

In this paper we have described algorithms to take a consistent picture of centralized and distributed databases. There are some restrictions that will be relaxed in a future report [12]. For example, the algorithm can be extended to handle concurrent checkpoints with multiple sets of data structures described in Section 2.2. Similarly, it is possible to use the same idea to read predefined partitions of a database.

Another optimization is based on the observation that the gray transactions which started as black transactions and request some white entities need not be aborted. They can be delayed until the requested white entities have been read by the checkpoint, painted black, and then continue as black transactions. Finally, a new family of algorithms can be derived by imposing a total order on the entities. The basic checkpoint marks its database traversal with entity color bits. The new algorithms need only a watermark to keep track of their progress, thus releasing the memory occupied by the entity color bits.

5 Comparison and Applications

5.1 Comparison with Related Work

There are two related areas of research: database checkpoints and replicated databases. Checkpointing a database has received considerable attention, including optimal checkpoint policies, on-the-fly checkpoints, and an early attempt on incremental checkpoint.

Studies on the performance of backup procedures [10,14] have assumed that (update) transaction processing is not allowed during the backup copying time. Several optimization criteria and optimal checkpoint policies are based on the above assumption [9], trading interrupted transaction time for short recovery time. In contrast, on-the-fly, incremental algorithms provide overall available consistency by limiting the distribution of entities being updated.

Two known on-the-fly algorithms to checkpoint databases are Lorie's shadow pages [11] and Gray's fuzzy dump [7]. However, these methods are specific to media recovery of databases and are not concerned with consistency constraints. An early attempt on incremental checkpoint algorithms (called *dynamic dumps*) was made by Rosenkrantz [13]. However, it is not clear that the dumps produced by his algorithms will always be transaction-consistent.

Replication can add availability and performance to (distributed) systems. A replica is created by copying from the original and kept consistent by propagating updates from the original. Attar et al. [1] use a read transaction locking the entire replica database to stop the updates and avoid deadlocks by delaying write locks. Fischer et al. [4] simulate a copy transaction system "forked" from the normal transaction system. The copy system completes ongoing transactions and refuses to start new transactions, obtaining consistency when updates cease. This technique would require considerable additional hardware investment for large databases in order to perform the updates on the copy at (roughly) the same speed as the original. More seriously, for distributed databases, significant communications cost must be added. In comparison, our algorithm does not update database entities and produces only sequential output.

5.2 Applications

Fischer et al. [4] have mentioned several applications using checkpoints, such as checking consistency constraints in a database, and media recovery. However, like their global checkpoint, our checkpoint is consistent but may not reflect any schedule based on chronological order. Consider a checkpoint that started at time t_1 and terminated at t_2. The checkpoint will reflect all updates committed before t_1, plus all white transactions which must terminate before t_2. In other words, the checkpoint may include "later" white transactions but not "earlier" black transactions. This characteristic should not affect applications like totals, statistics or consistency checking, where the actual transaction scheduling is not important.

In order to use a backup copy made by our checkpoints to recover from media failures, a log containing the committed transactions is still necessary. Logs in both shadow pages and fuzzy dump methods are logs of actions on "physical addresses" because their backup copies are not necessarily consistent. Since a backup copy made with our algorithms is transaction-consistent, we need only logs that contain transaction actions. There are two possibilities for recovery. First, we can redo the black transactions onto the backup copy to reach the database at t_2. Alternatively, one can undo the white transactions from the backup to find the database at t_1. In either case, in addition to actions, the log must include each transaction's color.

6 Conclusion

Highly available databases [8] have become increasingly popular, making batch operations decreasingly desirable. Gray et al. [6] mentioned typical dumping times of 10 minutes for a 100-megabyte database. Consistent checkpoints on entire databases processed on-the-fly can eliminate the down time due to backup copies. An interesting area of research is the performance and availability evaluations of on-the-fly, incremental algorithms according to different distributions of update pattern. Other applications include consistency checks, totals, and statistics over entire databases.

We have presented an algorithm which does not voluntarily abort, does not cause deadlocks, does not produce excess writes to disk, and terminates given a fair lock management. The basic algorithm introduces little interference into the transaction processing, with a minimal number of updates being aborted. Assuming an in-core lock table, this algorithm requires n bits of additional main memory for an n-entity database. The algorithm is extended to handle distributed databases with little additional overhead.

The adaptation of this algorithm to existing databases requires only modifications to the concurrency control. The physical design, specifically the disk format, need not be changed. The concurrency control checks the update transactions to avoid conflicts with the checkpoint. The simplicity of the data structures, crash recovery, concurrency control modification, and the checkpoint itself makes this algorithm and derivatives an attractive way to increase database availability.

7 Acknowledgment

This work grew out of my research in the Eden Project, which would not have been carried out without the guidance and support of Prof. Jerre Noe. I would like to thank Greg Andrews, Jean-Loup Baer, and Andy Proudfoot for their careful reading and extremely helpful comments on the first drafts of this paper. I also want to thank Phil Bernstein, Jim Gray, Won Kim, and Larry Snyder for their advice and encouragement.

References

[1] R. Attar, P.A. Bernstein, and N. Goodman.
Site initialization, recovery, and back-up in a distributed database system.
IEEE Transactions on Software Engineering, SE-10(6):645–650, November 1984.

[2] R. Bayer, H. Heller, and A. Reiser.
Parallelism and recovery in database systems.
ACM Transactions on Database Systems, 5(2):139–156, June 1980.
See also *Distributed Concurrency Control in Database Systems* by Bayer, Elhardt, Heller and Reiser, in the Proceedings of 6th Int. Conf. on Very Large Data Bases.

[3] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger.
The notions of consistency and predicate locks in a database system.
Communications of ACM, 19(11):624-633, November 1976.

[4] M.J. Fischer, N.D. Griffeth, and N.A. Lynch.
Global states of a distributed system.
In *Proceedings of Symposium on Reliability in Distributed Software and Database Systems*, July 1981.

[5] H. Garcia-Molina and G. Wiederhold.
Read-only transactions in a distributed database.
ACM Transactions on Database Systems, 7:209–234, June 1982.

[6] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger.
The recovery manager of the system R database manager.
ACM Computing Surveys, 13(2):223–242, June 1981.

[7] J.N. Gray.
Notes on data base operating systems.
In *Operating Systems - An Advanced Course*, Springer-Verlag, 1978.
Also IBM Research Report RJ 2188, Feb. 1978.

[8] Won Kim.
Highly available systems for database applications.
ACM Computing Surveys, 16(1), March 1984.

[9] C.M. Krishna, K.G. Shin, and Y.H. Lee.
Optimization criteria for checkpoint placement.
Communications of ACM, 27(10):1008–1012, October 1984.

[10] G.M. Lohman and J.A. Muckstadt.
Optimal policy for batch operations: backup, checkpointing, reorganization and updating.
ACM Transactions on Database Systems, 2(3):209–222, September 1977.

[11] R.A. Lorie.
Physical integrity in a large segmented database.
ACM Transactions on Database Systems, 2(1):91–104, March 1977.

[12] Calton Pu.
On-the-fly, incremental, consistent reading of entire databases.
In preparation.

[13] D.J. Rosenkrantz.
Dynamic database dumping.
In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 3–8, May 1978.

[14] A.N. Tantawi and M. Ruschitzka.
Performance analysis of checkpointing strategies.
ACM Transactions on Computer Systems, 2(2):123–144, May 1984.