## TIMESTAMP BASED PROTOCOL :( Timestamp based concurrency control techniques)

A timestamp is a unique identifier created by the DBMS to identify a transaction. Typically, timestamp values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the transaction start time. We will refer to the timestamp of transaction T as TS(T). Concurrency control techniques based on timestamp ordering do not use locks; hence, deadlocks cannot occur.

### Timestamp Ordering Algorithm

- **Basic Timestamp Ordering**
- **Strict Timestamp Ordering**
- **Thomas's Write Rule**

The idea for this scheme is to order the transactions based on their timestamps. A schedule in which the transactions participate is then serializable, and the equivalent serial schedule has the transactions in order of their timestamp values. This is called timestamp ordering (TO). Notice how this differs from 2PL, where a schedule is serializable by being equivalent to some serial schedule allowed by the locking protocols. In timestamp ordering, however, the schedule is equivalent to the particular serial order corresponding to the order of the transaction timestamps. The algorithm must ensure that, for each item accessed by conflicting operations in the schedule, the order in which the item is accessed does not violate the serializability order. To do this, the algorithm associates with each database item X two timestamp (TS) values:

1. **Read_TS(X):** The read timestamp of item X; this is the largest timestamp among all the timestamps of transactions that have successfully read item X—that is, read_TS(X) = TS(T), where T is the youngesttransaction that has read X successfully.

2. **Write_TS(X):** The write timestamp of item X; this is the largest of all the timestamps of transactions that have successfully written item X—that is, write_TS(X) = TS(T), where T is the youngest transaction that has written X successfully.

### Basic Timestamp Ordering

Whenever some transaction T tries to issue a read_item(X) or a write_item(X) operation, the basic TO algorithm compares the timestamp of T with read_TS(X) and write_TS(X) to ensure that the timestamp order of transaction execution is not violated. If this order is violated, then transaction T is aborted and resubmitted to the system as a new transaction with a new timestamp. If T is aborted and rolled back, any transaction T1 that may have used a value written by T must also be rolled back. Similarly, any transaction T2 that may have used a value written by T1 must also be rolled back, and so on. This effect is known as cascading rollback and is one of the problems associated with basic TO, since the schedules produced are not recoverable. An additional protocol must be enforced to ensure that the schedules are recoverable, cascadeless, or strict. We first describe the basic TO algorithm here. The concurrency control algorithm must check whether conflicting operations violate the timestamp ordering in the following two cases:

1. Transaction T issues a write_item(X) operation:

a. If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some younger transaction with a timestamp greater than TS(T)—and hence after T in the timestamp ordering—has already read or written the value of item X before T had a chance to write X, thus violating the timestamp ordering.

b. If the condition in part (a) does not occur, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).

2. Transaction T issues a read_item(X) operation:

a. If write_TS(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some younger transaction with timestamp greater than TS(T)—and hence after T in the timestamp ordering—has already written the value of item X before T had a chance to read X.

b. If write_TS(X) <= TS(T), then execute the read_item(X) operation of T and set read_TS(X) to the larger of TS(T) and the current read_TS(X).

**Example: class nodes**

**Strict Timestamp Ordering**

A variation of basic TO called strict TO ensures that the schedules are both strict (for easy recoverability) and (conflict) serializable. In this variation, a transaction T that issues a read_item(X) or write_item(X) such that TS(T) > write_TS(X) has its read or write operation delayed until the transaction T that wrote the value ofX (hence TS(T) = write_TS(X)) has committed or aborted. To implement this algorithm, it is necessary to simulate the locking of an item X that has been written by transaction T until T is either committed or aborted. This algorithm does not cause deadlock, since T waits for T only if TS(T) > TS(T).

## Thomas's Write Rule

A modification of the basic TO algorithm, known as Thomas's write rule, does not enforce conflict serializability; but it rejects fewer write operations, by modifying the checks for the write_item(X) operation as follows:

1. If read_TS(X) > TS(T), then abort and roll back T and reject the operation.

2. If write_TS(X) > TS(T), then do not execute the write operation but continue processing. This is because some transaction with timestamp greater than TS(T)—and hence after T in the timestamp ordering—has already written the value of X. Hence, we must ignore the write_item(X) operation of T because it is already outdated and obsolete. Notice that any conflict arising from this situation would be detected by case (1).

3. If neither the condition in part (1) nor the condition in part (2) occurs, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).

## DATABASE RECOVERY MANAGEMENT

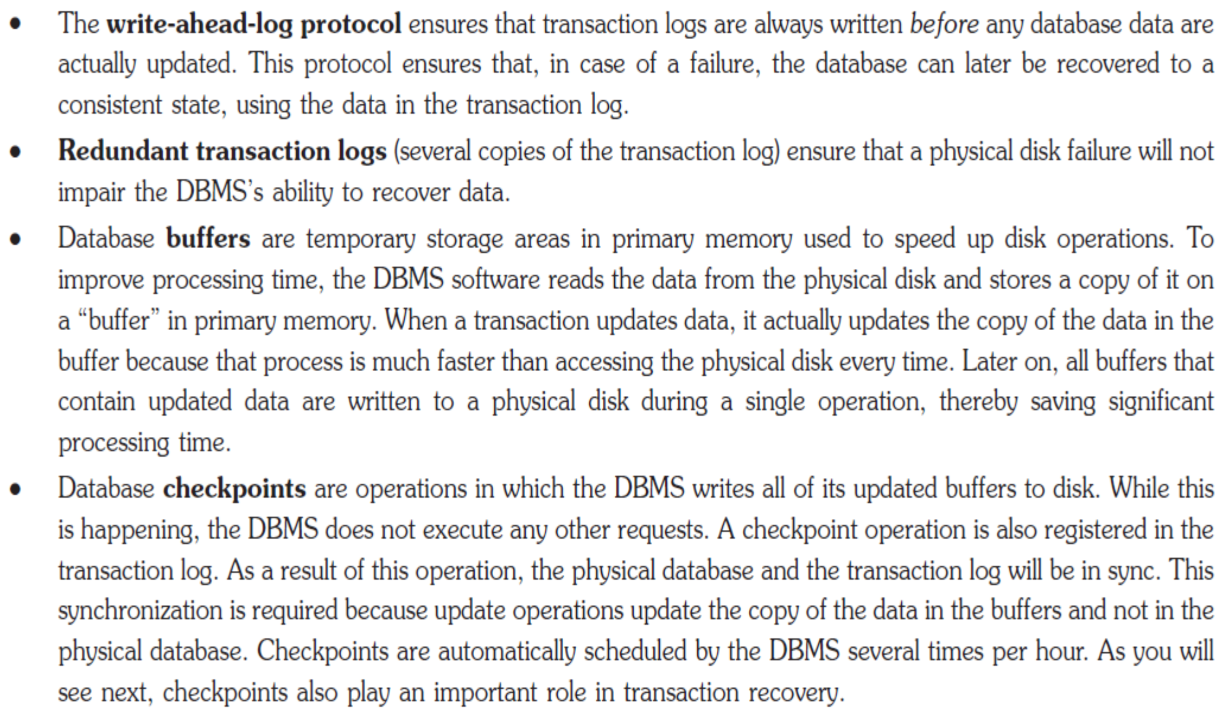
**Database recovery** restores a database from a given state (usually inconsistent) to a previously consistent state. Recovery techniques are based on the **atomic transaction property**: all portions of the transaction must be treated as a single, logical unit of work in which all operations are applied and completed to produce a consistent database. If, for some reason, any transaction operation cannot be completed, the transaction must be aborted and any changes to the database must be rolled back (undone). In short, transaction recovery reverses all of the changes that the transaction made to the database before the transaction was aborted.

- *Hardware/software failures*.
- *Human-caused incidents*. This type of event can be categorized as unintentional or intentional.
  - An unintentional failure is caused by carelessness by end-users. Such errors include deleting the wrong rows from a table, pressing the wrong key on the keyboard, or shutting down the main database server by accident.
  - Intentional events are of a more severe nature and normally indicate that the company data are at serious
  risk. Under this category are security threats caused by hackers trying to gain unauthorized access to data
  resources and virus attacks caused by disgruntled employees trying to compromise the database operation
  and damage the company.

- *Natural disasters*. This category includes fires, earthquakes, floods, and power failures.

## Transaction Recovery

Database transaction recovery uses data in the transaction log to recover a database from an inconsistent state to a consistent state. Before continuing, let's examine four important concepts that affect the recovery process:



- The **write-ahead-log protocol** ensures that transaction logs are always written *before* any database data are actually updated. This protocol ensures that, in case of a failure, the database can later be recovered to a consistent state, using the data in the transaction log.
- **Redundant transaction logs** (several copies of the transaction log) ensure that a physical disk failure will not impair the DBMS's ability to recover data.
- Database **buffers** are temporary storage areas in primary memory used to speed up disk operations. To improve processing time, the DBMS software reads the data from the physical disk and stores a copy of it on a "buffer" in primary memory. When a transaction updates data, it actually updates the copy of the data in the buffer because that process is much faster than accessing the physical disk every time. Later on, all buffers that contain updated data are written to a physical disk during a single operation, thereby saving significant processing time.
- Database **checkpoints** are operations in which the DBMS writes all of its updated buffers to disk. While this is happening, the DBMS does not execute any other requests. A checkpoint operation is also registered in the transaction log. As a result of this operation, the physical database and the transaction log will be in sync. This synchronization is required because update operations update the copy of the data in the buffers and not in the physical database. Checkpoints are automatically scheduled by the DBMS several times per hour. As you will see next, checkpoints also play an important role in transaction recovery.

## Transaction recovery

Procedures generally make use of **deferred-write** and **write-through techniques**.

When the recovery procedure uses a **deferred–write technique** (also called a **deferred update)**, the transaction operations do not immediately update the physical database. Instead, only the transaction log is updated. The database is physically updated only after the transaction reaches its commit point, using information from the transaction log. If the transaction aborts before it reaches its commit point, no changes (no ROLLBACK or undo) need to be made to the database because the database was never updated. The recovery process for all started and committed transactions (before the failure) follows these steps:

1. Identify the last checkpoint in the transaction log. This is the last time transaction data was physically saved to disk.

2. For a transaction that started and was committed before the last checkpoint, nothing needs to be done because the data are already saved.

3. For a transaction that performed a commit operation after the last checkpoint, the DBMS uses the transaction log records to redo the transaction and to update the database, using the "after" values in the transaction log. The changes are made in ascending order, from oldest to newest.

4. For any transaction that had a ROLLBACK operation after the last checkpoint or that was left active (with neither a COMMIT nor a ROLLBACK) before the failure occurred, nothing needs to be done because the database was never updated.

When the recovery procedure uses a **write-through technique** (also called an **immediate update)**, the database is immediately updated by transaction operations during the transaction's execution, even before the transaction reaches its commit point. If the transaction aborts before it reaches its commit point, a ROLLBACK or undo operation needs to be done to restore the database to a consistent state. In that case, the ROLLBACK operation will use the transaction log "before" values. The recovery process follows these steps:

1. Identify the last checkpoint in the transaction log. This is the last time transaction data were physically saved to disk.

2. For a transaction that started and was committed before the last checkpoint, nothing needs to be done because the data are already saved.

3. For a transaction that was committed after the last checkpoint, the DBMS redoes the transaction, using the "after" values of the transaction log. Changes are applied in ascending order, from oldest to newest.

4. For any transaction that had a ROLLBACK operation after the last checkpoint or that was left active (with neither a COMMIT nor a ROLLBACK) before the failure occurred, the DBMS uses the transaction log records to ROLLBACK or undo the operations, using the “before” values in the transaction log. Changes are applied in reverse order, from newest to oldest.