XI. Transactions

Lecture Topics

- Properties of Transactions
- Failures and Concurrency
- Transactions in SQL
- Implementation of Transactions
- Degrees of Isolation

Problems Caused by Failures

Accounts(Anum, CId, BranchId, Balance)

update Accounts
set Balance = Balance + 5
where BranchId = 12345

If the system crashes while processing this update, some, but not all, tuples with BranchId = 12345 may have been updated.

Problems (cont.)

Transfer money between accounts:

update Accounts
set Balance = Balance - 100
where Anum = 8888

update Accounts
set Balance = Balance + 100
where Anum = 9999

If the system crashes between these updates, money may be withdrawn but not redeposited.

Problems Caused by Concurrency

User 1:

update Accounts
set Balance = Balance - 100
where Anum = 8888

update Accounts
set Balance = Balance + 100
where Anum = 9999

User 2:

select Sum(Balance)
from Accounts

Value returned to User 2 may not reflect the true sum of the account balances.

Transaction Properties

Transactions are **durable**, **atomic** units of work.

- **Atomic:** indivisible, all-or-nothing.
- **Durable:** survives failures.

A transaction occurs either entirely, or not at all. A transaction's changes are never partially visible to other transactions. If a transaction occurs, its effects will not be erased or undone by subsequent failures.

Transaction Properties

Concurrent transactions must appear to have been executed sequentially, i.e., one at a time, in some order.

If T_i and T_j are concurrent transactions, then either:

- T_i will appear to precede T_j , meaning that T_j will "see" any updates made by T_i , and T_i will not see any updates made by T_j , or
- T_i will appear to follow T_j , meaning that T_i will see T_j's updates and T_j will not see T_i's.

Abort and Commit

A transaction may terminate in one of two ways: by aborting, or by committing.

- When a transaction **commits**, any updates it made become durable, and they become visible to other transactions. A commit is the “all” in “all-or-nothing” execution.
- When a transaction **aborts**, any updates it made have made are undone (erased), as if the transaction never ran at all. An abort is the “nothing” in “all-or-nothing” execution.

A transaction that has started but has not yet aborted or committed is said to be **active**.

Transactions in SQL

A new transaction is begun when an application first executes an SQL command.

Two SQL commands are available to terminate a transaction:

- **commit work**: commit the transaction
- **rollback work**: abort the transaction

A new transaction begins with the next SQL command after **commit work**or **rollback work**.

Implementing Transactions

Implementation of transactions in a DBMS has two aspects:

- **Concurrency Control:** guarantees that committed transactions appear to execute sequentially
- **Recovery Management:** guarantees that committed transactions are durable, and that aborted transactions have no effect on the database

Types of Failures

Transactions can simplify recovery from failures.

We will consider two types of failures: system failures and media failures.

- **System Failure:**
  - the database server is halted abruptly
  - processing of in-progress SQL command(s) is halted abruptly
  - connections to application programs (clients) are broken.
  - contents of memory buffers are lost
  - database files are not damaged.
- **Media Failure:**
  - one or more database files become damaged or inaccessible
  - a media failure may cause a system failure, or possibly an orderly system shutdown

Failures and Transactions

After a failure occurs:

- Transactions that were **active** when the failure occurs will be aborted automatically.
- Transactions that had **committed** before the failure will be durable, i.e., changes they made the the database will not be lost as a result of the failure.

A failure cannot cause a transaction to be partially-executed.

Recovery Management

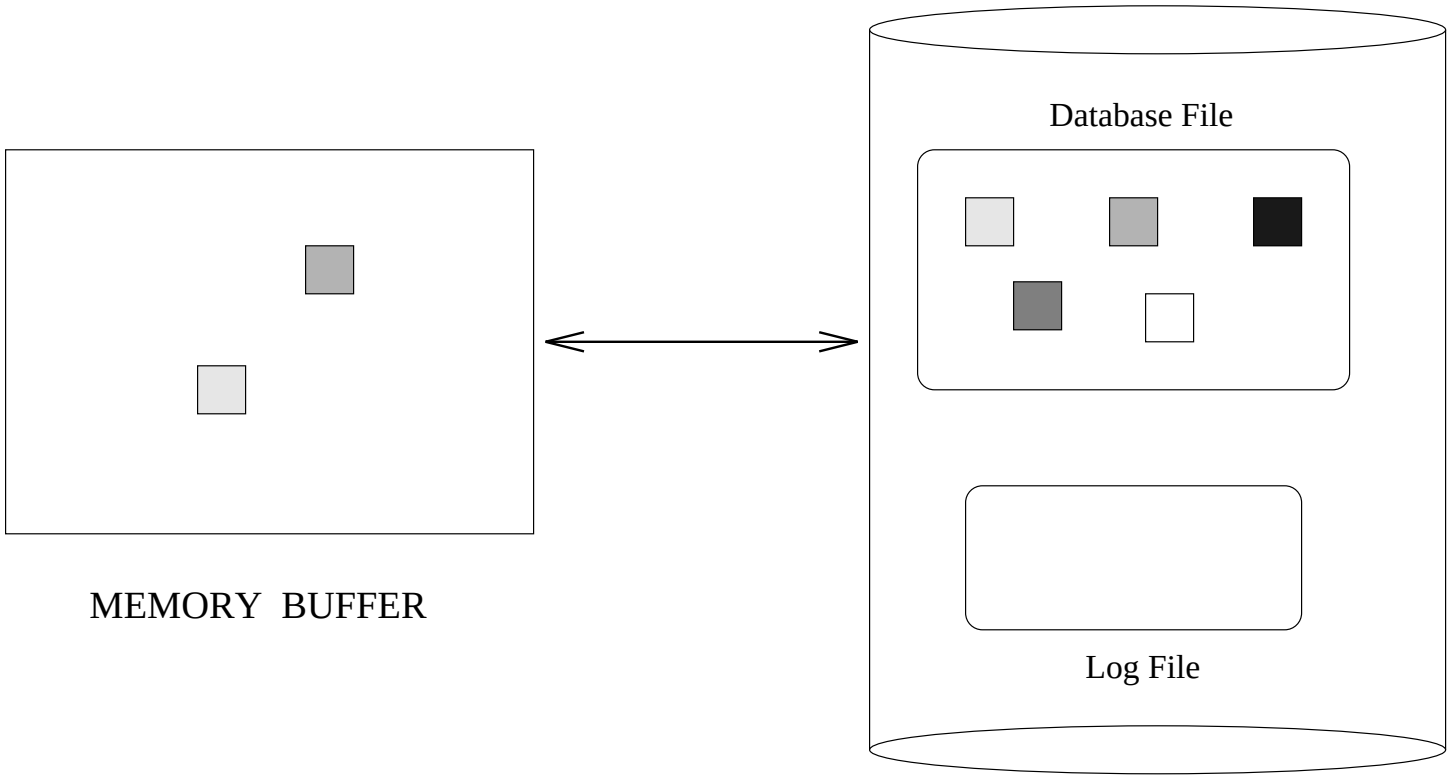
Recovery management is usually accomplished using a **log**.

A log is a read/append data structure. Normally it is stored in a file.

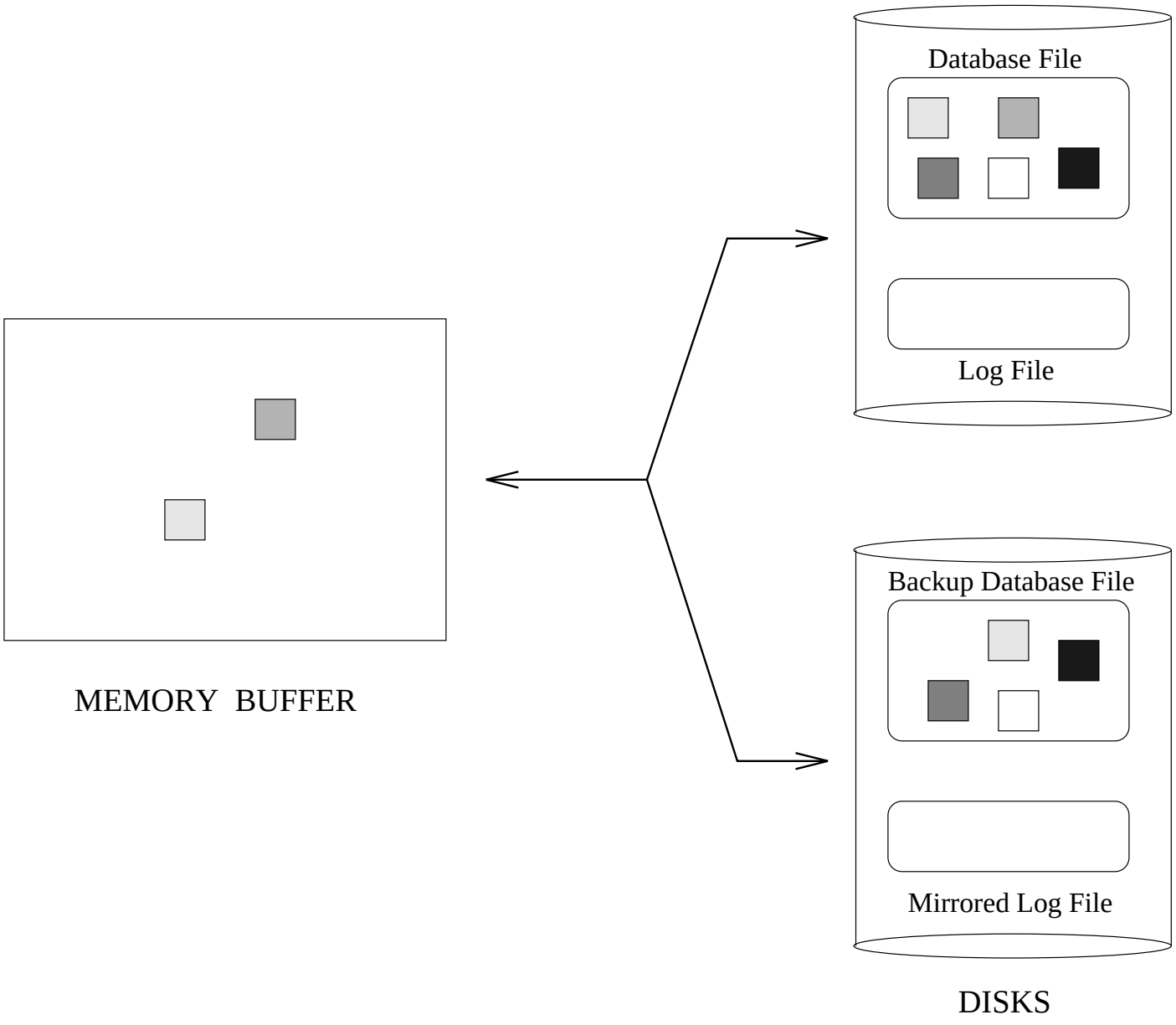
When transactions are running, **log records** are appended to the log. Log records contain several types of information:

- **UNDO information:** old versions of objects that have been modified by a transaction. UNDO information can be used to undo database changes made by a transaction that aborts.
- **REDO information:** new versions of objects that have been modified by a transaction. REDO records can be used to redo the work done by a transaction that commits.
- **BEGIN/COMMIT/ABORT** records are recorded whenever a transaction begins, commits, or aborts.

A Storage Model



Backups, Mirroring, and Multiple Disks



Using the Log

The log can be used to:

- recover from a system failure
  The database server can use the log to determine which transactions were active when the failure occurred, and to undo their database updates. Also, it may use the log to recreate the committed updates that may have been lost.
- recover from a media failure.
  The database server can use the log to determine which transactions committed since the most recent backup, and to redo their database updates.
- abort a single transaction.
  The database server can use the log to undo any database updates made by the aborted transaction.

Logging Example

log head	$\rightarrow$	T_0,begin
(oldest part of the log)		T_0,X,99,100
		T_1,begin
		T_1,Y,199,200
		T_2,begin
		T_2,Z,51,50
		T_1,M,1000,10
		T_1,commit
		T_3,begin
		T_2,abort
		T_3,Y,200,50
		T_4,begin
(newest part of the log)		T_4,M,10,100
log tail	$\rightarrow$	T_3,commit

Recovering from Failures

After a system failure:

1. Scan the log from tail to head:
    - Create a list of committed transactions
    - Undo updates of active transactions
2. Scan the log from head to tail:
    - Redo updates of committed transactions.

After a media failure:

1. If the database was damaged, restore the database from the most recent backup.
2. Scan the log from tail to head:
    - Create a list of committed transactions
3. Scan the log from head to tail:
    - Redo updates of committed transactions.

Checkpoints

As the log grows, the time required to recover from a failure also grows.

Checkpoints can be used to reduce the amount of log data that must be scanned after a system failure.

A simple checkpoint algorithm:

1. prevent new transactions from starting, and wait for active transactions to finish
2. copy modified blocks from memory buffer to database files
3. write a CHECKPOINT record in the log
4. allow new transactions to begin

After a system failure, the tail-to-head log scan can stop when a CHECKPOINT record is reached.

Concurrency Control

A database server will often be processing several transactions at the same time. This is generally much faster than processing transactions **serially**, i.e., one at a time.

The database system must make sure that concurrent transactions appear to be processed serially.

An execution of a set of transactions is said to be **serializable** if it is equivalent to a serial execution of the same transactions.

Some notation:

- $r_i[x]$ means that transaction T_i reads object x
- $w_i[x]$ means that transaction T_i writes (modifies) object x

Serializability

An interleaved execution of two transactions:

$$H_a = w_1[x]\ r_2[x]\ w_1[y]\ r_2[y]$$

An equivalent serial execution (T_1 , T_2):

$$H_b = w_1[x]\ w_1[y]\ r_2[x]\ r_2[y]$$

An interleaved execution with no equivalent serial execution:

$$H_c = w_1[x]\ r_2[x]\ r_2[y]\ w_1[y]$$

H_a is serializable because it is equivalent to H_b , a serial schedule. H_c is not serializable.

Two-Phase Locking

Most database systems use **locking** to guarantee that only serializable executions occur.

Before a transaction may read or write an object, it must have a lock on that object.

- a **shared lock** is required to read an object
- an **exclusive lock** is required to write an object

There is no ”lock” command in SQL. Instead, locks are acquired automatically by the database system.

The database system uses the following rules when acquiring locks for transactions:

- If two or more transactions hold locks on the same object, those locks must all be shared locks.
- A transaction’s locks are not released until it commits or aborts, i.e., until it is finished.

This algorithm is called **(strict) two-phase locking**.

> If a database system uses two-phase locking, transaction executions are guaranteed to be serializable.

Transaction Blocking

A transaction must have a lock on each object it wishes to read or write. However, the database system may be unable to acquire a lock without violating the two-phase locking rules.

For example:

- T_1 reads object x
- T_2 attempts to write x

T_2 cannot be given the necessary lock on x because of the rule prohibiting a shared and exclusive lock on the same object by different transactions.

When a transaction cannot obtain a lock, it is **blocked** (made to wait) until the lock can be obtained.

In the example above, T_2 will have to wait until T_1 commits or aborts.

Deadlocks

When two-phase locking is used, it is possible that deadlocks may occur.

For example:

- T_1 reads object x
- T_2 reads object y
- T_2 attempts to write object x (it is blocked)
- T_1 attempts to write object y (it is blocked)

If deadlock occurs, the database system must abort one of the transactions involved. This is called an involuntary abort.

Isolation Levels

For some applications, the guarantee of serializable executions may carry a heavy price. Performance may be poor because of blocked transactions and deadlocks.

TBITS SQL allows serializability guarantee to be relaxed, if necessary. Four **isolation levels** are supported, with the highest being serializability:

- Level 3: (Serializability)
  - read and write locks are acquired and held until end of transaction
- Level 2: (Repeatable Read)
  - identical to Level 3 unless insertion and deletion of tuples is considered
  - “phantom tuples” may occur

Isolation Levels (cont.)

- Level 1: (Cursor Stability)
  - shared (read) locks are not held until the end of the transaction
  - exclusive (write) locks are held until the end of the transaction
  - non-repeatable reads are possible, i.e., a transaction that reads the same object twice may read a different value each time

- Level 0:
  - neither read nor write locks are acquired
  - no updates, insertions, or deletions are permitted
  - transaction may read uncommitted updates

Transactions in Distributed Servers

A transaction is committed when its commit record is in the log.

In a distributed server, a transaction may execute at several sites, each with its own log.

A single transaction must not commit at some sites and abort on others.

Distributed servers must run an **agreement protocol** to make sure that all sites agree on the fate of each transaction.

Most systems use an agreement protocol called **two-phase commit**.

The Two-Phase Commit Protocol

One site acts as the coordinator. The following steps are taken to commit a transaction:

1. The coordinator sends a ”prepare” message to the other sites
2. Each site decides whether it wants to commit or abort the transaction and sends its vote to the coordinator
   - if abort, it writes an abort record in its log, and votes for abort
   - if commit, it writes a prepare record in its log, and votes for commit
3. If all sites vote commit, the coordinator writes a commit record in its log, otherwise it writes an abort record. The coordinator sends its decision to all of the sites.
4. Each site that voted to commit records the decision in its log and sends an acknowledgment to the coordinator.