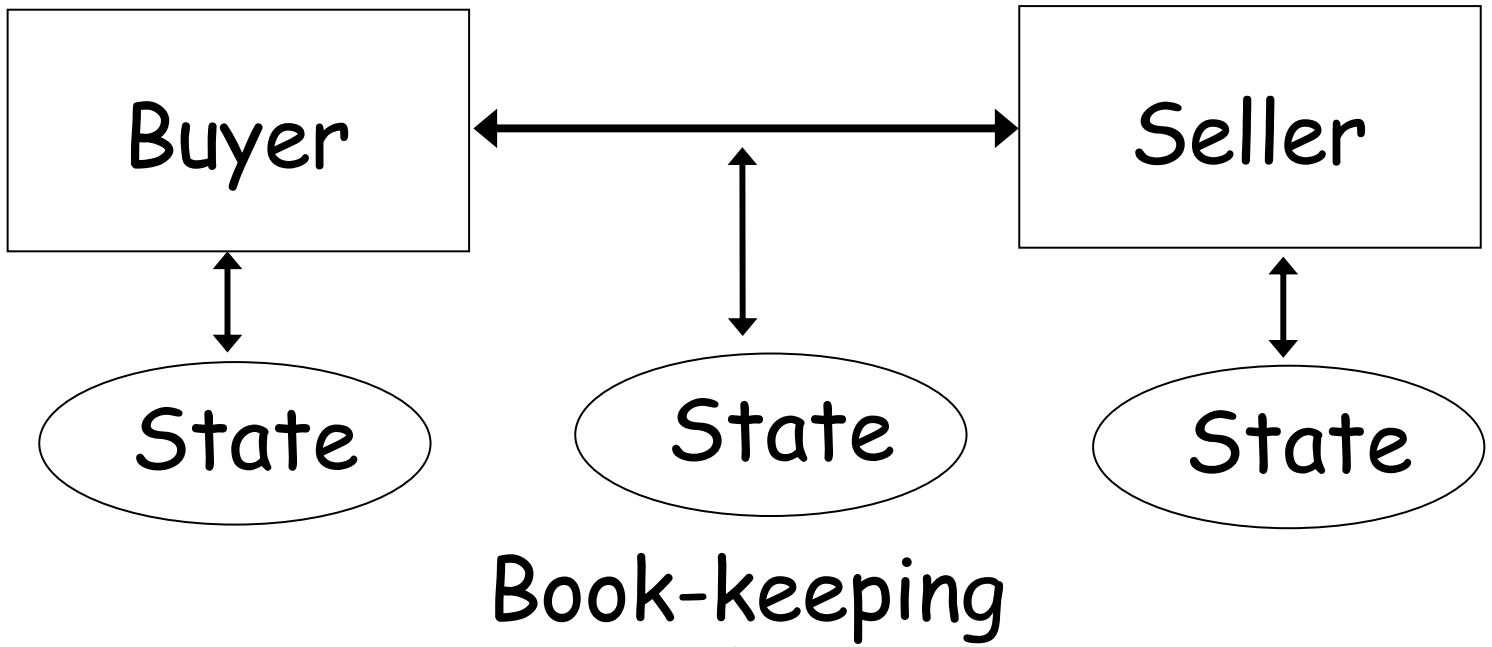
Transactions

Business transactions

- Business transaction: involves exchange between two or more entities (selling, buying, renting, booking ….).
- In computers: business transactions become electronic transactions (same ideas)



Money Transfer

In Application Program

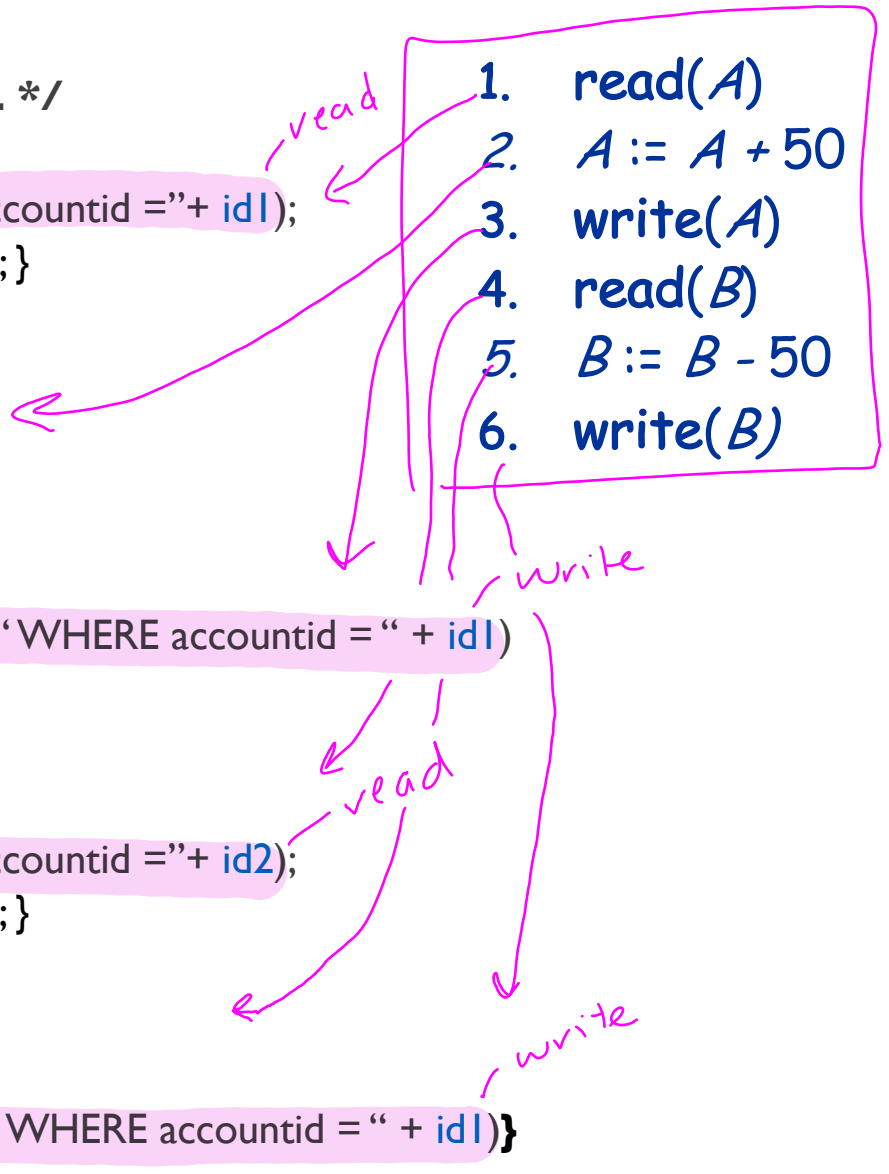
```
Transfer(id1,id2,value) {
 /* retrieve first balance through SQL */
  ResultSet rs =  stmt.executeQuery(
 "SELECT balance FROM accounts WHERE accountid ="+ id1);
  while ( rs.next() ) { bal = rs.getInt("balance"); }

  /* calculate new value */
  bal += value

 /* update balance through SQL */
  stmt.executeUpdate(
 "UPDATE accounts SET balance = " + bal +  " WHERE accountid = " + id1)

 /* do similar for second balance */
 ResultSet rs =  stmt.executeQuery(
  "SELECT balance FROM accounts WHERE accountid ="+ id2);
  while ( rs.next() ) { bal = rs.getInt("balance"); }

  bal -= value
  stmt.executeUpdate(
  "UPDATE accounts SET balance = " + bal + " WHERE accountid = " + id1)}
```



Electronic transaction

- An *electronic* transaction encapsulates operations that belong logically together
- Boundaries of the transaction have to be defined by the programmer (according to application semantics)
- Contains
  - DBS operations
  - Application code
- Examples:
  - Purchasing elements in shopping cart of online e-commerce side
  - Flight Reservation (might book more than one flight)

define logical units of work

Application vs. DBS

- A user's program
  - interaction with the database (select, update, delete…)
  - own computation

- A *database* ***transaction*** is the DBMS's abstract view of a user program:
  - a *sequence* of **read operations r(X)** and **write operations w(X)** on **objects (X)** (tuple, relation,…) of the DB
  - Read: bring object into main memory from disk, send value to application (same as copy value into program variable)
  - Write: bring object into main memory from disk and modify it. Write it back to disk (might be done sometime later)

Online transaction processing (OLTP)

- Database Schemas with 20-30 tables
- Pre-defined application tasks of reasonable size
  - 4-20 SQL operations cover one task = transaction
  - Individual Queries relatively simple (1-4 tables involved).
    - Often point queries (all information from one client /student)
    - Efficient execution possible
  - Update intensive: 15 to 50% of SQL statements are updates
- Examples
  - Banking Teller machines
  - Online banking (more queries than teller machine….)
  - Travel Reservations
  - E-Commerce
  - Stock Exchange
  - Cash Register in Supermarket
- Volumes: up to hundreds/thousands of transaction per minute
- Other application domains
  - More queries (and complex ones) – less updates analytical processing systems
  - Much simpler tasks – only one read or one write operation

ACID Properties

- The success of transactions was due to well-defined properties that are provided by the database system:
  - **ATOMICITY:**
    - A transaction is atomic if it is executed in its entirety or not at all
  - CONSISTENCY:
    - a transaction must preserve the consistency of the data
  - **ISOLATION:**
    - in case that transactions are executed concurrently: The effect must be the same as if each transaction were the only one in the system.
  - **DURABILITY**:
    - The changes made by a transaction must be permanent (= they must not be lost in case of failures)

- The application programmer only has to indicate
  - When a transaction starts
  - The sequences of SQL operations
  - When the transaction finishes (abort or commit request possible)
  - AID are guaranteed by the DBS

Money Transfer

```
In Application Program
Transfer(id1,id2,value) {
  /* begin txn */
  /* often automatic */
  ...
 /* increase balance 1st account*/
ResultSet rs =  stmt.executeQuery(
  "SELECT balance FROM accounts WHERE accountid =”+ id1);
   ...
 /* decrease balance 2nd account*/
 ResultSet rs =  stmt.executeQuery(
  "SELECT balance FROM accounts WHERE accountid =”+ id2);
  while ( rs.next() ) { bal = rs.getInt(“balance”); }
  bal -= value
  /* abort if necessary*/
  if bal < 0
    connection.rollback();
  ELSE {
    stmt.executeUpdate(
    ”UPDATE accounts SET balance = “ + bal + “ WHERE accountid = “ + id1)
  connection.commit();}
}
```

first update

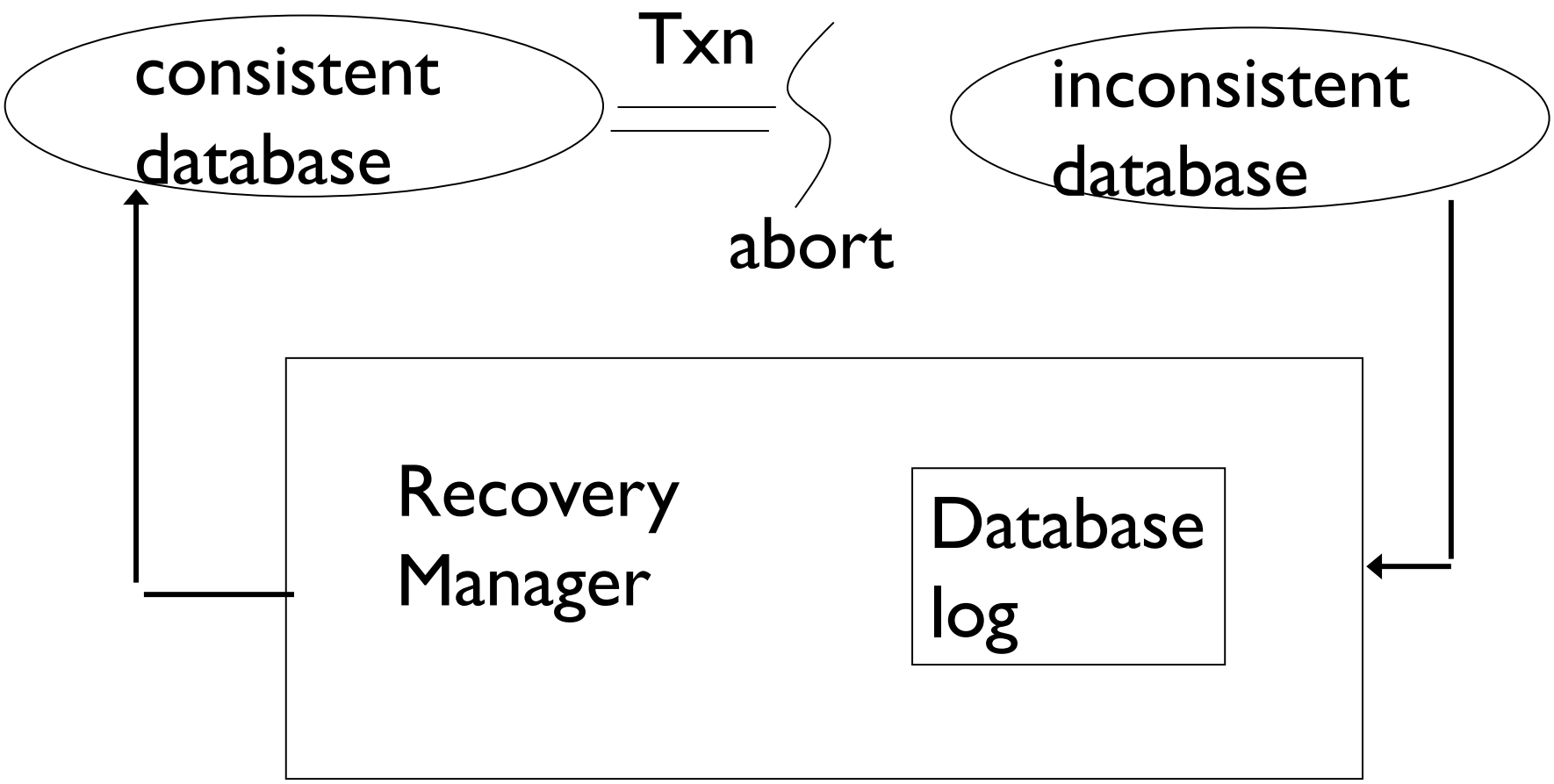
rollback the transaction - undo the update

connection object from JDBC

Atomicity: All or Nothing

- A transaction T might **commit** after completing all its actions.
  - If the user is informed about the commit they can be sure that all changes performed by T are installed in the DB.
- A transaction might **abort** (or be aborted by the DBMS) after executing some actions.
  - In this case the DBMS undoes all modifications so far.
  - After the abort the DB state is as if the transaction had never started.
  - After notification of abort the user knows that none of the transaction's modifications is reflected in the database.
- **Local recovery**: eliminating partial results in case of abort
  - before executing write(x)
    - store "before image" of x (somewhere in main memory ➔ called the log)
      - » log record: txn-id, rid, before-image
- Examples
  - committing transaction: read(x), write(x), write(y), read(z), commit
  - aborting transaction
    - read(x), write(x), write(y) … write^{-1}(y), write^{-1}(x) abort
    - write^{-1}(y) installs before-image of y taken from log

Atomicity

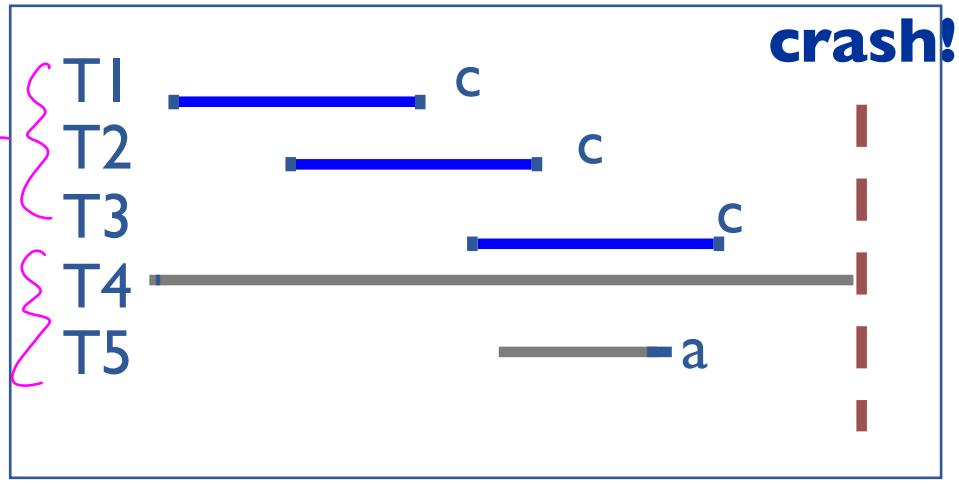

consistent database
Txn
abort
inconsistent database
Recovery Manager
Database log

Durability

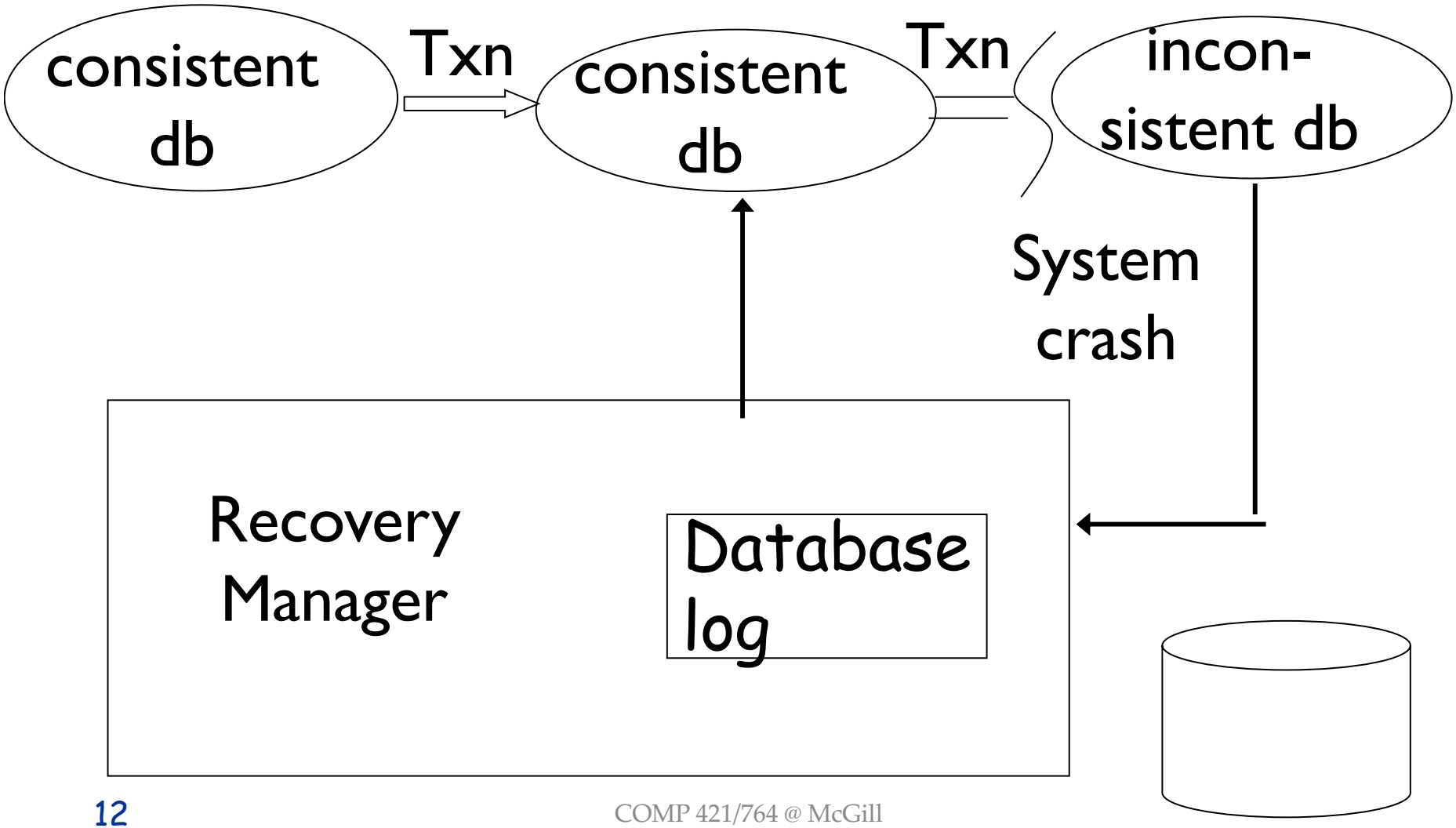
- **Durability**: There must be a guarantee that the changes introduced by a transaction will last, i.e., survive failures
  - once user has commit confirmation the change must persist in database despite possible crash

❖ Desired Behavior after system restarts:

- T1, T2 & T3 should be durable.
- T4 & T5 should be aborted (effects not seen).



Global Recovery



Recovery after System Crash

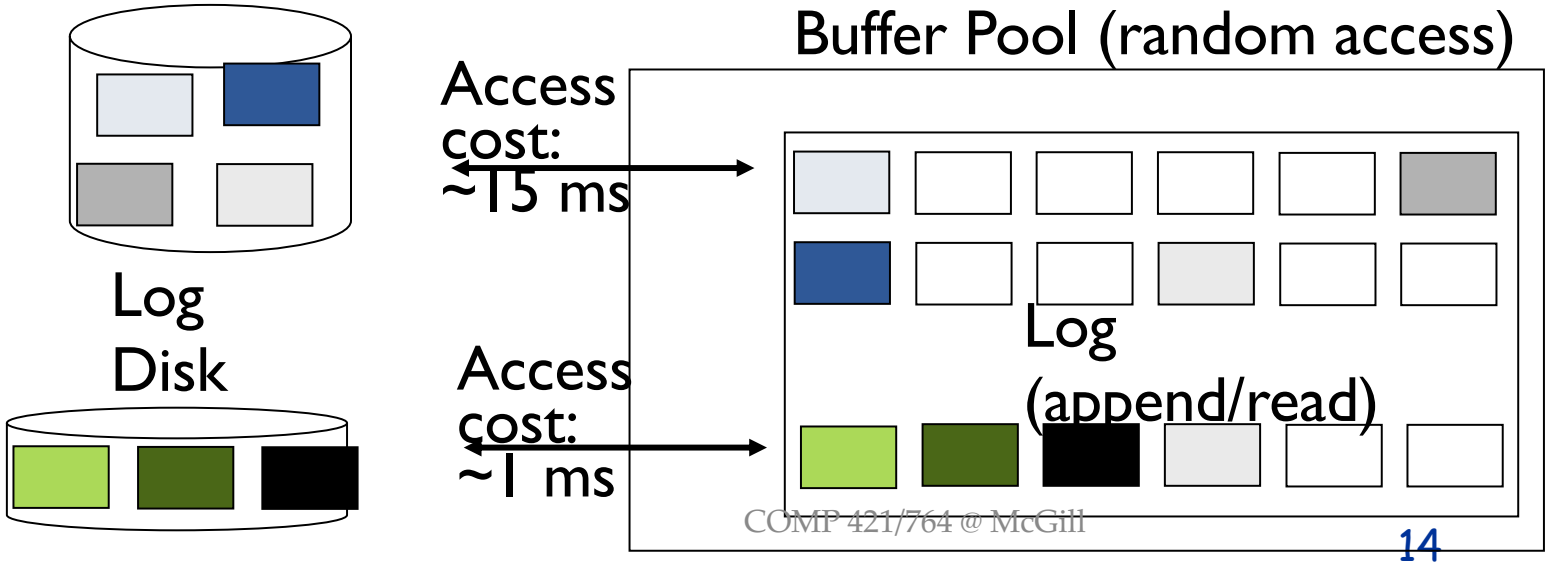
- restart server and perform **global recovery:**
  - transactions that committed before crash
    - make sure all changes are in database
  - transactions that aborted before crash
    - make sure no changes are in database
  - transactions that are active at time of crash
    - make sure no changes are in database

- Assumption:
  - disk does not crash ← everything on disk is persistant and safe

enough info needs to be on disk to recover

Updates vs. logs

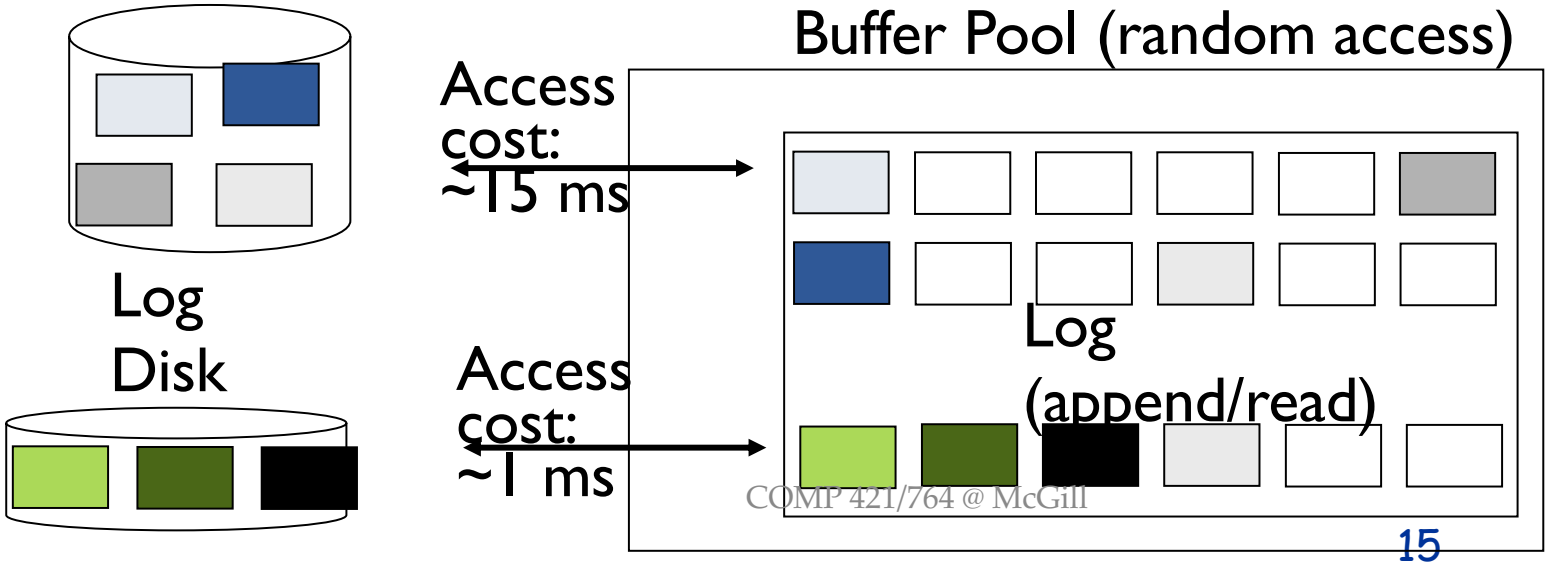
- Each update changes the record on the corresponding page
- Dirty (changed) pages are written to disk whenever the buffer manager decides (replacement policy)
- To be able to undo/redo in case of crash
  - Write log information to a log disk
  - Faster writes than random writes to standard disk



high performance system if log is on a different disk

WAL: write ahead logging

- for each write(x) of transaction T; x residing on DB page P:
  - Append log record with before- and after-image of x to log tail
    - before-image: you can undo changes of active transactions
    - after-image: you can redo changes of committed transactions
- Flush log to disk before flushing the P: then you can undo when you have to abort all necessary before images are on disk
- Flush log to disk before commit of update transaction T: then you can redo when something committed (read-only transactions don't create dirty pages and log records – so ignore)
- Log is typically an append; flush means all of the log is flushed in one shot; cheaper than standard flush



UNDO/REDO Recovery

- Recovery:
  - Undo: For each transaction T that did not commit
    - Find all of T's log records and install before images
  - Redo: For each transaction T that did commit
    - Find all of T's log records and install after image
    - If there were many committed transactions updating x?
- Challenges:
  - Doing things in the right order
  - How do I know which transactions committed
    - Write commit/abort logs, too
  - How do I avoid redoing all committed transactions since system start
    - Checkpointing
    - Write out dirty pages on a regular basis during normal processing
    - Helps reducing redo phase