# CS525: Advanced Database Organization

**Notes 7: Recovery and Concurrency Control**
**Part I: Failure and Recovery**

Gerald Balekaki

Department of Computer Science

Illinois Institute of Technology

gbalekaki@iit.edu

July 31$^{st}$ 2023

Slides: adapted from courses taught by Shun Cheung, Emory University, Andy Pavlo, Carnegie Mellon University, Jennifer Welch, Texas A&M & Introduction to Database Systems by ITL Education Solutions Limited

# Concurrency and Recovery

- DBMS should enable reestablish correctness of data in the presence of failures
  - System should restore a correct state after failure (recovery)
- DBMS should enable multiple clients to access the database concurrently
  - This can lead to problems with correctness of data because of interleaving of operations from different clients
    - System should ensure correctness (concurrency control)

# Crash Recovery

- Recovery algorithms are techniques to ensure database consistency, transaction atomicity, and durability despite failures.
- Every recovery algorithm has two parts:
  - Actions during normal transaction processing to ensure that the DBMS can recover from a failure.
  - Actions after a failure to recover the database to a state that ensures atomicity, consistency, and durability

# Crash Recovery

- DBMS is divided into different components based on the underlying storage device.
- We must also classify the different types of failures that the DBMS needs to handle.

# Failure Classification

- **Type #1: Transaction Failures**
- **Type #2: System Failures**
- **Type #3: Storage Media Failures**

# Transaction Failures

- **Logical Errors**
  - A transaction cannot complete due to some internal error condition (e.g., integrity constraint violation).
- **Internal State Errors**
  - The DBMS must terminate an active transaction due to an error condition (e.g., deadlock)

# System Failures

- **Software Failure**
  - There is a problem with the DBMS implementation (e.g., uncaught divide-by-zero exception) and the system has to halt
- **Hardware Failure**
  - The computer hosting the DBMS crashes (e.g., power plug gets pulled).
  - Fail-stop Assumption
    - We assume that non-volatile storage contents are not corrupted by system crash.

# Storage Media Failures

- **Non-Repairable Hardware Failure**
  - A head crash or similar disk failure destroys all or parts of nonvolatile storage.
  - Destruction is assumed to be detectable.
- No DBMS can recover from this. Database must be restored from archived version

# Failure Modes: System failure example

- Example: Transaction: `transfer $100 from account A to account B`

```
READ A
A.balance = A.balance - 100
WRITE A
**system fails here**
READ B
B.balance = B.balance + 100
WRITE B
```

- Then `A` would lose his $100 !!!
- This problem is solved by `logging`
- `Transaction` needs to be executed correctly

# Modeling consistency of a database

- `Database element`: the unit of data accessed by the database system
  - Abstraction that will come in handy when talking about concurrency control and recovery
- `Database`: a collection of `database elements`
- Note:
  - Different DBMS uses different notion for database element
  - Possible units:
    - A relation
    - A disk block
    - A tuple in a relation

# Modeling consistency of a database

- `Database state`: the collection of values of all `database elements` in the database
- `Database state` can be changed by changing one or more of the `database elements` in the database
- A `database state` can be
  - Consistent: satisfy all constraints of the database schema and implicit constraints
  - Inconsistent

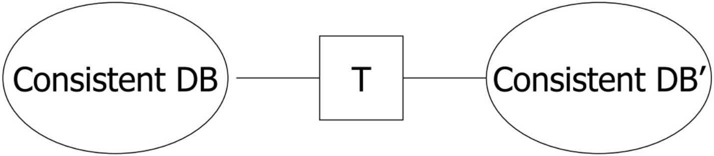
# Modeling consistency of a database

- `Transaction`: a sequence of changes to one or more database elements
- Example: Transaction: `transfer $100 from account A to account B`

```
READ A
A.balance = A.balance - 100
WRITE A
READ B
B.balance = B.balance + 100
WRITE B
```

# Modeling consistency of a database

- A more precise definition of transaction:
- `Transaction`: a sequence of changes to one or more database elements
- When all changes in a transaction are made to the database state:
  - The resulting database state is a **`consistent state`** (if the initial state is consistent)



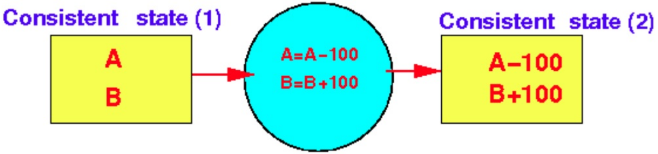
# Causes of inconsistent database states

- There are 2 causes of inconsistent database states
  1. System failure
  2. Concurrent execution

# How a system failure can result in an inconsistent DB state

- Consider the following transaction `transfer $100 from A to B`

```
READ A
A.balance = A.balance - 100
WRITE A
READ B
B.balance = B.balance + 100
WRITE B
```

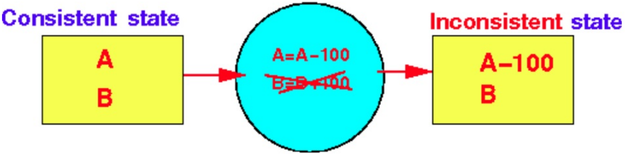



- There are 2 possible consistent states
  - Consistent state 1: `A` `B`
  - Consistent state 2: `A-100` `B+100`

# How a system failure can result in an inconsistent DB state

- Consider the database state that result from the following system failure

```
READ A
A.balance = A.balance - 100
WRITE A
**system fails here**
READ B
B.balance = B.balance + 100
WRITE B
```




- The resulting database state is: `Database state = A-100    B`
- Not one of the 2 possible consistent states

# How concurrent execution can cause inconsistent states

- Consider the following 2 transactions

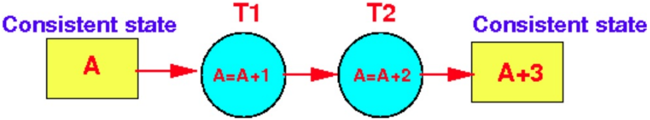
`T1= Deposit $1 to A`

```
READ A
A = A + 1
WRITE A
```

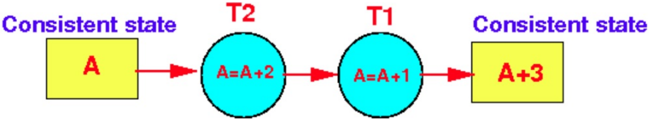
`T2= Deposit $2 to A`

```
READ A
A = A + 2
WRITE A
```

- The possible `consistent` database states for executing `T1` and `T2` are:
  - Case 1: `T1 before T2`




  - Case 2: `T2 before T1`

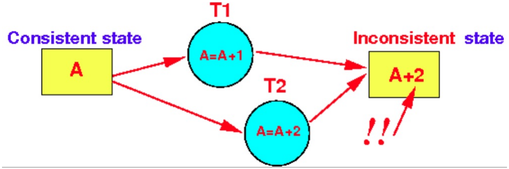


# How concurrent execution can cause inconsistent states

- Consider the following concurrent execution of `T1` and `T2`
- `T1= Deposit $1 to A, T2= Deposit $2 to A`

| T1 | T2 | |
|---|---|---|
| | | initially: A = 10 |
| **READ** A | | |
| | **READ** A | |
| A = A + 1 | | (A = 11) |
| | A = A + 2 | (A = 12) |
| **WRITE** A | | Writes 11 to A |
| | **WRITE** A | Writes 12 to A |

- Final database state: `A = 12 (= A + 2)`



# Correctness "theory" of database transactions

- Assuming that the database is in a `consistent state`
- Then, a transaction will transform the database into a (another) consistent state if:
  - There are no system failures
  - There are no other transactions executing in the database system

# Transactions

- `Transaction`: the (smallest) unit of execution of database operations (updates)
- Unit = whole thing, indivisible
- A `transaction` is:
  - executed completely or
  - nothing from the `transaction` is executed

# Transactions in SQL

- A new txn starts with the `BEGIN` command.
- The txn stops with either `COMMIT` or `ABORT`
  - If commit, the DBMS either saves all the txn's changes or aborts it.
  - If abort, all changes are undone so that it's like as if the txn never executed at all.
- Abort can be either self-inflicted or caused by the DBMS.

# Notation for a transaction

```
begin transaction
   ....
   .... operations performed by the transaction
   .... e.g.: read, compute, write
   ....
commit   // success
```

- or

```
begin transaction
   ....
   .... operations performed by the transaction
   .... e.g.: read, compute, write
   ....
abort   // failure
```

# Notation for a `transaction`: Result

- All operations between

```
begin transaction
   ....
   ....   ALL operations executed
   ....
commit
```

  will be executed

- None of the operations between

```
begin transaction
   ....
   ....   NO operations executed
   ....
abort
```

  has been executed

# Correctness Criteria: ACID

- A `transaction` (must) have the following properties
  - Atomicity
    - Either all operations of the transaction are properly reflected in the database or none are (i.e., either all or no commands of transaction are executed)
  - Consistency
    - Execution of a transaction in isolation preserves the consistency of the database.
  - Isolation
    - If two `transactions` are executing concurrently, each `transaction` will see the database as if the `transaction` was executing `sequentially` (in isolation) (i.e., transactions are running isolated from each other)
  - Durability
    - After a `transaction` completes successfully, the changes it has made to the database persist (permanent), even if there are system failures (i.e., modifications of transactions are never lost)