Concurrency Control and Recovery in Database Systems

- **Concurrency control**
  - is the activity of coordinating the actions of processes that operate in parallel, access shared data, and therefore potentially interfere with each other

- **Recovery**
  - is the activity of ensuring that Sw and Hw failures do not corrupt persistent data

- **Transaction**
  - is a sequence of actions on a shared database

- The goal is to ensure that transactions execute atomically (no interference)
DataBase Systems

- **State**
  - Set of named data items with values

- **Main operations**
  - Read(x) – query – returns the value of the data item x
  - Write(x,v) – update – sets the data item x to value v

- We abstract from the granularity of data items
  - Words in main memory
  - Pages of a disk
  - Records in a file
  - Fields of a record

---

Execution

- Operations should be executed sequentially, but some operations can be executed concurrently
  - The final effect must be the same of some sequential execution

1. Read(x)
2. Write(x,1) | Read(y)
3. Write(y,0)

The order is inessential
### Transaction Primitives

- **Start**
  - begin a new transaction
  - assign a unique transaction identifier
- **Commit**
  - successfully terminate the transaction
  - its effects are made permanent
- **Abort**
  - invoke the abnormal termination of the transaction
  - its effects are obliterated
  - can be imposed by failures

### Terminology

- **Unique entry / exit points**
  - a transaction is defined by a Start operation, followed by a possibly concurrent execution of (Read / Write) operations followed by either a Commit or Abort
- **Committed transactions**
- **Aborted transactions**
- **Active transactions**
  - started but not yet committed (or aborted)
- **Uncommitted transactions**
  - active or aborted
Database Transactions

- Sequences of operations / actions which preserve database consistency
- ACID properties:
  - Atomicity
  - Consistency
  - Isolation
  - Durability
- Interest on serializability
### Database Transactions

- Sequences of operations / actions which preserve database consistency

- **ACID properties:**
  - Atomicity
  - Consistency
  - Isolation
  - Durability

- Interest on serializability

---

A transaction is a correct transformation of the state: its actions, taken as a group, do not violate any integrity constraint associated with the states.

Transactions can be executed concurrently, but it appears to each transaction T that others are executed either before T or after T.
Database Transactions

- Sequences of operations / actions which preserve database consistency
- ACID properties:
  - Atomicity
  - Consistency
  - Isolation
  - Durability
- Interest on serializability

Once a transaction successfully commits, its changes to the states survive failures

Interest on *serializability*

- any \( T_1 | ... | T_n \) can be seen as \( T_{i1}; ... ; T_{in} \)
Recoverability

- When T aborts its effects are wiped out
  - Effects on data (updates)
  - undo
  - restore the original value as if T never executed
  - Effects on each T’ which read values updated by T (dependencies)
  - cascading aborts
    - each T’ must be aborted
    - but remember that committed transactions cannot be aborted
    - delaying commits can be necessary to ensure recoverability

Recoverability: Example

1. When T1 aborts, Write(x,2) must be undone
2. Undoing Write(x,2) can compromise T2 that read from x
3. But T2 did already commit... it should have been delayed!
Avoiding Cascading Aborts

- Cascading aborts are unpleasant, but often necessary to ensure recoverability
  - Require bookkeeping of dependencies
  - Uncontrollably many transactions might be forced to abort
- Not necessary if every transaction reads values written by committed transactions
  - Delay Read(x) until all transactions that issued Write(x,v) are either aborted or committed

Strict Execution

- Undoing Write(x,v)
  - which value should be restored?
    - the initial value?
    - but other transactions can have issued Write(x,v')
  - Before image
    - the value of x just before the operation to be undone
    - Later overwritten values should not be updated
- Delay Write(x,v) until all transactions that issued Write(x,v') are either aborted or committed
Strict Execution: Example

- Write(x,1)
- Write(y,3)
- Write(y,1)
- Commit
- Read(x)
- Abort

... Write(x,2) Write(x,3) Abort

Suppose initially x=1. When T1 aborts, the value of x should be 3 not 1

The before image of T2 is not meaningful after T1 aborts

When T2 aborts, the value of y should be 3 (the before image)

Non-Recoverable Interactions

- It is not always possible to reverse effects
  - e.g. input / output statements
- Deferred output approach
  - output messages of T only after T commits
  - not always feasible (user input might depend on output messages)
More on Serializability

- When two transactions execute concurrently, their operation are interleaved
  - Even if they commit, interference can cause incorrect results
    - Lost updates
    - Inconsistent retrieval
- Avoid interleaving at all
  - correct but inefficient
- Restrict to **serializable** executions

Lost Updates: Example

\[
T\_Deposit(a,n):
\]
Start
\( t:=\text{Read}(a) \)
\( t:=t+n \)
Write\((a,t)\)
Commit

\[
T\_Deposit(a,10) \mid T\_Deposit(a,20):
\]
Start
Read\((a)\) // returns \( v \)
Start
Read\((a)\) // returns the same \( v \)
Write\((a,v+10)\)
Write\((a,v+20)\)
Commit
Commit

At the end the account is increased by 20 (not by 30 as in any serial execution)
**Inconsistent Retrieval: Example**

T\_Transfer(£, $, n):
- Start
- \( s := \text{Read}(£) \)
- \( s := s - n \)
- Write(£, s)
- \( t := \text{Read}($) \)
- \( t := t + n \times \text{rate}[£, $] \)
- Write(£, t)
- Commit

T\_Total(£, $):
- Start
- \( s := \text{Read}(£) \)
- \( t := \text{Read}($) \)
- \( t := t + s \times \text{rate}[£, $] \)
- output(t)
- Commit

T\_Transfer(a, b, 5)|T\_Total(a, b):
- Start
- \( \text{Read}(a) \) // returns v
- Start
- \( \text{Read}(a) \) // returns the same v
- Write(£, v - 5)
- \( \text{Read}(b) \) // returns w
- Write(£, w + 5 \times r)
- \( \text{Read}(b) \) // returns w + 5 \times r
- Commit
- Commit

The output does not correspond to the total amount

**DataBase System Model I**

- **Transaction Manager (TM)**
  - receives requests for transactions and operations
  - forwards them to the scheduler
- **Scheduler**
  - controls the relative order of execution
- **Recovery Manager (RM)**
  - responsible for commits and aborts
- **Distributed Architecture**
  - sites connected by a network
Some ACID Flavours I

- **Flat Transactions**
  - Traditional model, strictly meets ACID properties
  - Not adequate to model complex situations
    - A task’s abort causes all activities to be rolled back

- **Flat with Savepoints**
  - Introduces programmable savepoints
    - Rollback command restore savepoints without aborting
    - Transactions can be rolled back individually

- **Chained transactions**
  - Add savepoints via the *chain* operation
    - Commits and start a new transaction with same locks
Some ACID Flavours II

- Nested Transactions
  - Hierarchical tree of transactions
    - Top-level is ACID
    - Other nodes are ACI (subtransactions)
      - If a parent aborts all its children abort (even if committed)
      - If a subtransaction aborts its parent can still commit
      - A subtransaction commit is seen outside only when its parent commits (protected nested transactions)
  - The structure can exploit parallelism within transactions
  - Localize failures to subtransactions

Some ACID Flavours III

- Multi-level Transactions
  - Nested transaction that allow
    - pre-commits to export partial results to other transactions at the same level
    - perfect compensations to abort after pre-commit
  - Abstraction layers
    - Operations at level n are implemented using operations of the lower level immediately below
      - no shortcuts
Some ACID Flavours IV

- **Distributed Transactions**
  - Flat but executed in a distributed environment
  - Partitioned in sub-transactions on network’s nodes
    - Decomposition reflects neither the hierarchical structure of the application, nor its functional structure
    - Decomposition depends on data distribution over the nodes
  - Decisions made by a sub-transaction affect the entire transaction
  - Commit protocols are needed
- **All these models are traditional**
  - see e.g. [Gray, Reuter 1993]

Some Non ACID Flavours I

- **Long-Lived Transactions (LLT)**
  - Execution takes a considerable amount of time
    - hours, days, weeks, ...
  - Not possible to block all short transactions
- **Open Nested Transactions** [Gray 1981]
  - Compensations are not perfect
  - Relaxed multi-level
    - No abstraction layers
    - Shortcuts are allowed
Some Non ACID Flavours II

- **Sagas** [Garcia-Molina, Salem 1987]
  - Special case of open-nested
  - Only two levels of nesting
  - LLT broken in a non-atomic sequence of ACID sub-transactions $T_1...T_n$
    - Each $T_i$ has a (non-perfect) compensation $C_i$
    - Partial execution of the sequence are undesirable
      - If it cannot be completed, then must be compensated
  - Executed as either $T_1...T_n$ or $T_1...T_kC_k...C_1$
  - Partial effects are visible to other transactions
    - no notification / compensation to siblings

Some Non ACID Flavours III

- **Flexible** [Elmagarmid et al. 1990]
  - Same task implemented by different (functionally equivalent) sub-transactions
  - Programmable failure / success dependencies
    - $T_1$: Rent a car from AVIZ
    - $T_2$: Rent a car from HERTS if $T_1$ fail
  - Pre-committed actions and compensations
    - Mixed (with or without compensations) transactions coexist within a single global transaction
  - Incorporates time for flexible scheduling
Some Non ACID Flavours IV

- **Split and Join Transactions** [Pu et al. 1988]
  - Open-ended activities
    - Long-running, not foreseeable concurrent interactions
      - Computer-Aided Design/Computer-Aided Manufacturing
      - Editing and Sw development
  - Split command
    - Divide a running transaction in two independent ones
    - Can export results, release/share resources
  - Join command
    - Merges two running transactions into one
    - All resources become shared
    - Results are visible only after the commit of the joint transaction

---

Some Non ACID Flavours V

- **Multi-Transactions**
  - To be used in distributed objects systems with heterogeneous databases
  - Complex transactions by composing building blocks (as needed by applications)
  - Mixture of flat, nested, open-nested are possible
  - Countermeasure, Retriable, Timed, ...
Transactions in GC?

- ACID would generate too many aborts
- Local consistency vs global consistency
- Asynchronous commit
- Heterogeneous policies
- Several entry / exit points
- Messages in place of shared data
- Dynamic topology of the network
- Mobility
- For an overview:
  - A framework for analyzing mobile transaction models (Journal of Database Management 12(13), 2001)

Recap

- We have seen
  - Basic terminology, notions and aspects of transactions
  - Sources of troubles
  - Importance of serializability and compensation
  - ACID models
  - Relaxed-ACID models
References I

- Concurrency control and recovery in database systems (Addison-Wesley 1987)
  - P. Bernstein, N. Goodman, V. Hadzilacos
- Transaction processing: concepts and techniques (Morgan Kaufmann 1993)
  - J. Gray, A. Reuter

References II

- The transaction concept: virtues and limitations (Proc. VLDB’81, IEEE, pp. 144-154)
  - J. Gray
- Sagas (Proc. SIGMod’87, ACM, pp. 249-259)
  - H. Garcia-Molina, K. Salem
- A multidatabase transaction model for interbase (Proc. VLDB’90, Morgan Kaufmann)
  - A.K. Elmagamid, Y. Leu, W. Litwin, M. Rusinkiewicz
- Split-transactions for open-ended activities (Proc. VLDB’88, Morgan Kaufmann, pp. 26-37)
  - N. Hutchinson, G.E. Kaiser, C. Pu