Review: TCP Flow Control

• AdvertisedWindow: number of bytes TCP receiver can receive
  \[ \text{AdvertisedWindow} = \text{MaxRcvBuffer} - (\text{LastByteRcvd} - \text{LastByteRead}) \]

• SenderWindow: number of bytes TCP sender can send
  \[ \text{SenderWindow} = \text{AdvertisedWindow} - (\text{LastByteSent} - \text{LastByteAcked}) \]

TCP Flow Control

- Sending app sends 350 bytes
- Recall:
  - We assume IP only accepts packets no larger than 100 bytes
  - MaxRcvBuf = 300 bytes, so initial Advertised Window = 300 bytes
TCP Flow Control

Sending Process

LastByteWritten(350)

201, 301, 350

LastByteAcked(200)

LastByteRcvd(350)

NextByteExpected(201)

\{1,100\}

\{1,200\}

\{(1,300)\}

\{101,300\}

\{201,300\}

\{201,350\}

\{101,200\}, \{301,350\}

Ack=201, AdvWin = 50

Data[1,100]

Data[101,200]

Data[201,300]

Data[301,350]

\{101,200\}, \{301,350\}

Yes! Sender can re-send 3rd packet since it's in existing window – won't cause receiver window to grow

Receiving Process

LastByteRead(100)

101, 200, 301, 350

LastByteSent(350)

101, 200, 301, 350

LastByteRcvd(350)

101, 201, 301, 350

101, 200

101, 350

301, 350

\{101,200\}, \{301,350\}

Data[101,200]

Data[201,300]

Data[301,350]

\{101,200\}, \{301,350\}

\{101,100\}, \{101,200\}, \{301,350\}

TCP Flow Control

Sending Process

LastByteWritten(350)

201, 301, 350

LastByteAcked(200)

LastByteRcvd(350)

NextByteExpected(351)

\{201,350\}

\{201,350\}

\{201,350\}

\{201,350\}

\{101,350\}

\{101,200\}, \{301,350\}

Ack=201, AdvWin = 50

Data[301,350]

\{101,200\}, \{301,350\}

Yes! Sender can re-send 3rd packet since it's in existing window – won't cause receiver window to grow

Receiving Process

LastByteRead(100)

101, 201, 301, 350

LastByteSent(350)

101, 200, 301, 350

LastByteRcvd(350)

101, 201, 301, 350

101, 200

101, 350

301, 350

\{101,200\}, \{301,350\}

Data[301,350]

\{101,200\}, \{301,350\}

\{101,100\}, \{101,200\}, \{301,350\}

TCP Flow Control

Sending Process

LastByteWritten(350)

201, 301, 350

LastByteAcked(200)

LastByteRcvd(350)

NextByteExpected(351)

\{201,350\}

\{201,350\}

\{201,350\}

\{201,350\}

\{101,350\}

\{101,200\}, \{301,350\}

Ack=201, AdvWin = 50

Data[301,350]

\{101,200\}, \{301,350\}

Yes! Sender can re-send 3rd packet since it's in existing window – won't cause receiver window to grow

Receiving Process

LastByteRead(100)

101, 201, 301, 350

LastByteSent(350)

101, 200, 301, 350

LastByteRcvd(350)

101, 201, 301, 350

101, 200

101, 350

301, 350

\{101,200\}, \{301,350\}

Data[301,350]

\{101,200\}, \{301,350\}

\{101,100\}, \{101,200\}, \{301,350\}

TCP Flow Control

Sending Process

LastByteWritten(350)

201, 301, 350

LastByteAcked(200)

LastByteRcvd(350)

NextByteExpected(351)

\{201,350\}

\{201,350\}

\{201,350\}

\{201,350\}

\{101,350\}

\{101,200\}, \{301,350\}

Ack=201, AdvWin = 50

Data[301,350]

\{101,200\}, \{301,350\}

Yes! Sender can re-send 3rd packet since it's in existing window – won't cause receiver window to grow

Receiving Process

LastByteRead(100)

101, 201, 301, 350

LastByteSent(350)

101, 200, 301, 350

LastByteRcvd(350)

101, 201, 301, 350

101, 200

101, 350

301, 350

\{101,200\}, \{301,350\}

Data[301,350]

\{101,200\}, \{301,350\}

\{101,100\}, \{101,200\}, \{301,350\}
TCP Flow Control

Sending Process

- LastByteWritten(350)
- LastByteAcked(200)
- LastByteSent(350)
- LastByteRcvd(350)

Receiving Process

- LastByteRead(100)
- LastByteWritten(350)
- LastByteRcvd(350)
- NextByteExpected(351)

Data[301,350]

\{[201,350]\}

\{[101,200], [301,350]\}

- Ack=201, AdvWin = 50
- Data[201,300]

\{101,200], [301,350]\}

\{101,350]\}

- Ack=31, AdvWin = 50

- Sender gets 3rd packet and sends Ack for 351
- AdvWin = 50

\{201,350]\}

\{[201,350]\}

\{[101,350]\}

Sender DONE with sending all bytes!

Quiz 18.1: Flow-Control

- Q1: True _ False _ Flow control is responsible for detecting packet losses and retransmissions
- Q2: True _ False _ Flow control always allows a sender to resend a lost packet
- Q3: True _ False _ With TCP, the receiving OS can deliver data to the application out-of-sequence (i.e., with gaps)
- Q4: True _ False _ Flow control makes sure the sender doesn’t overflow the receiver

Quiz 18.1: Flow-Control

- Q1: True _ False _ Flow control is responsible for detecting packet losses and retransmissions
- Q2: True _ False _ Flow control always allows a sender to resend a lost packet
- Q3: True _ False _ With TCP, the receiving OS can deliver data to the application out-of-sequence (i.e., with gaps)
- Q4: True _ False _ Flow control makes sure the sender doesn’t overflow the receiver
Goals for Today

• What is a database?

• Transactions (ACID semantics)

• Dealing with concurrency (Transaction Scheduling)

Note: Some slides and/or pictures in the following are adapted from lecture notes by Mike Franklin.

What is a Database

• A large organized collection of data

• Models real world, e.g., enterprise
  – Entities (e.g., teams, games)
  – Relationships, e.g.,
    Cal plays against Stanford in The Big Game

Large Databases

• Yahoo: User data: 700 Terabytes

• AT&T: 330 Terabytes

• Australian Bureau of Stats: 250 TB

What matters for these databases?
Large Databases

- Yahoo: User data: 700 Terabytes - BASE
- AT&T: 330 Terabytes – ACID
- Australian Bureau of Stats: OLAP, ROLAP

What matters for these databases?

Key Concept: Structured Data

- A **data model** is a collection of entities and their relationships
- A **schema** is an instance of a data model
  - E.g., describes the fields in the database; how the database is organized
- A **relational data model** is the most used data model
  - **Relation**, a table with rows and columns
  - Every relation has a **schema** which describes the fields in the columns

Example: University Database

- Conceptual schema:
  - **Students** (sid: string, name: string, email: string, age: integer, gpa: real)
  - **Courses** (cid: string, cname: string, credits: integer)
  - **Enrolled** (sid: string, cid: string, grade: string)
    - FOREIGN KEY sid REFERENCES Students
    - FOREIGN KEY cid REFERENCES Courses

- External Schema (View):
  - **Course_info** (cid: string, enrollment: integer)
    - Create View Course_info AS
      - SELECT cid, Count(*) as enrollment
      - FROM Enrolled
      - GROUP BY cid

Example: A Relation on Student Attributes (aka Relation Instance)

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>email</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@eecs</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.8</td>
</tr>
</tbody>
</table>

Remember the foundations of math: functions and relations:
- Functions z = f(x,y)
- Relations R(x,y,z) ∈ {true, false}

Relations are more general than functions, allow multiple values for z.
Relations can be thought of as predicates on a list of arguments, or as sets of tuples satisfying the predicate.
What is a Database System?

- A Database Management System (DBMS) is a software system designed to store, manage, and facilitate access to databases.
- A DBMS provides:
  - Data Definition Language (DDL)
    - Define relations, schema
  - Data Manipulation Language (DML)
    - Queries – to retrieve, analyze and modify data.
  - Guarantees about durability, concurrency, semantics, etc

Key Concepts: Queries, Query Plans, and Operators

System handles query plan generation & optimization; ensures correct execution.

Key concept: Transaction

- An atomic sequence of database actions (reads/writes)
- Takes DB from one consistent state to another
Example

- Here, consistency is based on our knowledge of banking “semantics”
- In general, up to writer of transaction to ensure transaction preserves consistency
- DBMS provides (limited) automatic enforcement, via integrity constraints (IC)
  - e.g., balances must be $\geq 0$

Administrivia

- Please fill out ETS survey (link posted on Piazza)
- Watch your slip days!
  - 10% per day penalty after all four are used

From Multiprogramming to Transactions

- Users would like the illusion of running their programs on the machine alone
  - Why not run the entire program in a critical section?
- Users want fast response time and operators want to increase machine utilization → increase concurrency
  - Interleave executions of multiple programs
- How can DBMS help?

2min Break
Concurrent Execution & Transactions

- Concurrent execution essential for good performance
  - Disk slow, so need to keep the CPU busy by working on several user programs concurrently

- DBMS only concerned about what data is read/written from/to the database
  - Not concerned about other operations performed by program on data

- **Transaction** – DBMS’s abstract view of a user program, i.e., a sequence of reads and writes.

- Locks are an important part of concurrency control.

Database Locks

- A “lock” in a DB is a higher-level construct than in an OS.

- Think of it as a Readers-Writers Monitor. Why?

- Locks are handled by a lock manager process, and involve updates to a lock table accessible to all clients.

- Low-level locks exist too, but are called “latches”.

- Setting and releasing locks is a relatively expensive process.

Locking Granularity

- What granularity to lock?
  - Database
  - Tables
  - Rows

- Fine granularity (e.g., row) → high concurrency
  - Multiple users can update the database and same table simultaneously

- Coarse granularity (e.g., database, table) → simple, but low concurrency

Need for Transactions in Distributed Systems

- Example: assume two clients updating same value in a key-value (KV) store at the same time
  - Client A subtracts 75; client B adds 25

  ![Diagram of transaction example](image)
Solution?

- How did we solve such problem on a single machine?
  - Critical section, e.g., use locks
  - Let’s apply same solution here...

Discussion

- How does client B get the lock?
  - Polling: periodically check whether the lock is free
  - KV storage system keeps a list of clients waiting for the lock, and gives the lock to next client in the list

- What happens if the client holding the lock crashes?

- Network latency might be higher than update operation
  - Most of the time in critical section spent waiting for messages

- What is the lock granularity?
  - Do you lock every key? Do you lock the entire storage?
  - What are the tradeoffs?

Better Solution

- Interleave reads and writes from different clients

- Provide the same semantics as clients were running one at a time

- **Transaction** – database/storage system’s abstract view of a user program, i.e., a sequence of reads and writes

“Classic” Example: Transaction

BEGIN; --BEGIN TRANSACTION

UPDATE accounts SET balance = balance - 100.00 WHERE name = 'Alice';

UPDATE branches SET balance = balance - 100.00 WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Alice');

UPDATE accounts SET balance = balance + 100.00 WHERE name = 'Bob';

UPDATE branches SET balance = balance + 100.00 WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Bob');

COMMIT; --COMMIT WORK

Transfer $100 from Alice’s account to Bob’s account
The ACID properties of Transactions

- **Atomicity**: all actions in the transaction happen, or none happen
- **Consistency**: transactions maintain data integrity, e.g.,
  - Balance cannot be negative
  - Cannot reschedule meeting on February 30
- **Isolation**: execution of one transaction is isolated from that of all others; no problems from concurrency
- **Durability**: if a transaction commits, its effects persist despite crashes

A  Atomicity
C  Consistency
I  Isolation
D  Durability
B  Basically
A  Available
S  Soft State
E  Eventual Consistency

Poor availability
Poor durability
Poor consistency

Atomicity

- A transaction
  - might commit after completing all its operations, or
  - it could abort (or be aborted) after executing some operations
- Atomic Transactions: a user can think of a transaction as always either executing all its operations, or not executing any operations at all
  - Database/storage system logs all actions so that it can undo the actions of aborted transactions

Consistency

- Data follows integrity constraints (ICs)
- If database/storage system is consistent before transaction, it will be after
- System checks ICs and if they fail, the transaction rolls back (i.e., is aborted)
  - A database enforces some ICs, depending on the ICs declared when the data has been created
  - Beyond this, database does not understand the semantics of the data (e.g., it does not understand how the interest on a bank account is computed)
Isolation

- Each transaction executes as if it was running by itself
  - It cannot see the partial results of another transaction

- Techniques:
  - Pessimistic – don’t let problems arise in the first place
  - Optimistic – assume conflicts are rare, deal with them after they happen

Durability

- Data should survive in the presence of
  - System crash
  - Disk crash → need backups

- All committed updates and only those updates are reflected in the database
  - Some care must be taken to handle the case of a crash occurring during the recovery process!

Quiz 18.2: Databases

- Q1: True  False  A relational data model is the most used data model
- Q2: True  False  Transactions are not guaranteed to preserve the consistency of a storage system
- Q3: True  False  A DBMS uses a log to implement atomicity
- Q4: True  False  Durability isolates the reads and writes of a transaction from all other transactions
This Lecture

- Deal with **(I)solation**, by focusing on concurrency control
- Next lecture focus on (A)tomicity, and partially on (D)urability

---

Example

- Consider two transactions:
  - T1: moves $100 from account A to account B
    \[
    \text{T1: } A := A-100; B := B+100;
    \]
  - T2: moves $50 from account B to account A
    \[
    \text{T2: } A := A+50; B := B-50;
    \]
- Each operation consists of (1) a read, (2) an addition/subtraction, and (3) a write
- Example: \( A = A-100 \)
  - Read(A); // R(A)
  - \( A := A - 100; \)
  - Write(A); // W(A)

---

Example (cont’d)

- Database only sees reads and writes

| T1: \( A := A-100; B := B+100; \) | \( T1: R(A), W(A), R(B), W(B) \) |
| T2: \( A := A+50; B := B-50; \) | \( T2: R(A), W(A), R(B), W(B) \) |

- Assume initially: \( A = 1000 \) and \( B = 500 \)
- What is the legal outcome of running T1 and T2?
  - \( A = 950 \)
  - \( B = 550 \)

---

Example (cont’d)

- What is the outcome of the following execution?

| T1: \( A := A-100; B := B+100; \) | Initial values: \( A := 1000 \) \( B := 500 \) |
| T2: \( A := A+50; B := B-50; \) |

| \( T1: R(A), W(A), R(B), W(B) \) | \( T2: R(A), W(A), R(B), W(B) \) |

| \( A := A+50; \) \( B := B-50; \) |

| \( A = 950 \) | \( B = 550 \) |

- What is the outcome of the following execution?

| T1: \( R(A), W(A), R(B), W(B) \) | \( T2: R(A), W(A), R(B), W(B) \) |

| \( A := A+50; \) \( B := B-50; \) |

| \( A = 1050 \) | \( B = 450 \) |
Example (cont’d)

- What is the outcome of the following execution?

  T1: R(A), W(A), R(B), W(B)
  T2: A=900, R(A), W(A), R(B), W(B)
  T1: A=950, B=450
  T2: A=1050, B=550

- What is the outcome of the following execution?

  T1: R(A), W(A), R(B), W(B)
  T2: A=900, R(A), W(A), R(B), W(B)
  T1: A=1000, B=450
  T2: A=900, B=550

Lost $50!

Transaction Scheduling

- Why not run only one transaction at a time?
- Answer: low system utilization
  - Two transactions cannot run simultaneously even if they access different data

Goals of Transaction Scheduling

- Maximize system utilization, i.e., concurrency
  - Interleave operations from different transactions
- Preserve transaction semantics
  - Semantically equivalent to a serial schedule, i.e., one transaction runs at a time
Two Key Questions

1) Is a given schedule equivalent to a serial execution of transactions?

2) How do you come up with a schedule equivalent to a serial schedule?

Anomalies with Interleaved Execution

- May violate transaction semantics, e.g., some data read by the transaction changes before committing
- Inconsistent database state, e.g., some updates are lost
- Anomalies always involves a "write"; Why?

Transaction Scheduling

- **Serial schedule**: A schedule that does not interleave the operations of different transactions
  - Transactions run serially (one at a time)
- **Equivalent schedules**: For any storage/database state, the effect (on storage/database) and output of executing the first schedule is identical to the effect of executing the second schedule
- **Serializable schedule**: A schedule that is equivalent to some serial execution of the transactions
  - Intuitively: with a serializable schedule you only see things that could happen in situations where you were running transactions one-at-a-time

Anomalies with Interleaved Execution

- Read-Write conflict (Unrepeatable reads)
  
  \[
  T_1: R(A), R(A), W(A) \\
  T_2: R(A), W(A) 
  \]

- Violates transaction semantics
- Example: Mary and John want to buy a TV set on Amazon but there is only one left in stock
  - (T1) John logs first, but waits...
  - (T2) Mary logs second and buys the TV set right away
  - (T1) John decides to buy, but it is too late...
Anomalies with Interleaved Execution

- Write-read conflict (reading uncommitted data)
  
  \[ T1: R(A), W(A), \quad W(A) \]
  
  \[ T2: \quad R(A), \quad \ldots \]

- Example:
  - (T1) A user updates value of A in two steps
  - (T2) Another user reads the intermediate value of A, which can be inconsistent
  - Violates transaction semantics since T2 is not supposed to see intermediate state of T1

Anomalies with Interleaved Execution

- Write-write conflict (overwriting uncommitted data)
  
  \[ T1: W(A), \quad W(B) \]
  
  \[ T2: \quad W(A), W(B) \]

- Get T1’s update of B and T2’s update of A
- Violates transaction serializability
- If transactions were serial, you’d get either:
  - T1’s updates of A and B
  - T2’s updates of A and B

Conflict Serializable Schedules

- Two operations conflict if they
  - Belong to different transactions
  - Are on the same data
  - At least one of them is a write

- Two schedules are conflict equivalent iff:
  - Involve same operations of same transactions
  - Every pair of conflicting operations is ordered the same way

- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule

Conflict Equivalence – Intuition

- If you can transform an interleaved schedule by swapping consecutive non-conflicting operations of different transactions into a serial schedule, then the original schedule is conflict serializable

- Example:
  
  \[ T1: R(A), W(A), \quad R(B), W(B) \]
  
  \[ T2: \quad R(A), W(A), \quad R(B), W(B) \]

  \[ \text{Swap} \]

  \[ T1: R(A), W(A), \quad W(B) \]
  
  \[ T2: \quad R(A), W(A), \quad R(B), W(B) \]
Conflict Equivalence – Intuition (cont’d)

• If you can transform an interleaved schedule by swapping consecutive non-conflicting operations of different transactions into a serial schedule, then the original schedule is conflict serializable.

Example:

\[
\begin{align*}
T_1 &: R(A), W(A), R(B), W(B) \\
T_2 &: R(A), W(A), R(B), W(B)
\end{align*}
\]

• Is this schedule serializable?

\[
\begin{align*}
T_1 &: R(A), W(A) \\
T_2 &: R(A), W(A)
\end{align*}
\]

Dependency Graph

• Dependency graph:
  – Transactions represented as nodes
  – Edge from Ti to Tj:
    » an operation of Ti conflicts with an operation of Tj
    » Ti appears earlier than Tj in the schedule

• Theorem: Schedule is conflict serializable if and only if its dependency graph is acyclic.

Example

• Conflict serializable schedule:

\[
\begin{align*}
T_1 &: R(A), W(A), R(B), W(B) \\
T_2 &: R(A), W(A), R(B), W(B)
\end{align*}
\]

\[
\begin{align*}
\text{Dependency graph}
\end{align*}
\]

• No cycle!
Example

• Conflict that is \textit{not} serializable:

\begin{center}
\begin{tabular}{c}
T1: R(A), W(A), \quad R(B), W(B) \\
T2: \quad R(A), W(A), R(B), W(B)
\end{tabular}
\end{center}

T1 \quad T2

\textit{Dependency graph}

• Cycle: The output of T1 depends on T2, and vice-versa

Notes on Conflict Serializability

• Conflict Serializability doesn’t allow all schedules that you would consider correct
  – This is because it is strictly syntactic - it doesn’t consider the meanings of the operations or the data

• In practice, Conflict Serializability is what gets used, because it can be done efficiently
  – Note: in order to allow more concurrency, some special cases do get implemented, such as for travel reservations, ...

• Two-phase locking (2PL) is how we implement it

Serializability ≠ Conflict Serializability

• Following schedule is \textit{not} conflict serializable

\begin{center}
\begin{tabular}{c}
T1: R(A), W(A), \\
T2: \quad W(A), \quad W(A) \\
T3: \quad W(A)
\end{tabular}
\end{center}

\textit{Dependency graph}

• However, the schedule is serializable since its output is equivalent with the following serial schedule

\begin{center}
\begin{tabular}{c}
T1: R(A), W(A), \\
T2: \quad W(A), \quad W(A) \\
T3: \quad W(A)
\end{tabular}
\end{center}

• Note: deciding whether a schedule is serializable (not conflict-serializable) is NP-complete
Summary

• Transaction: a sequence of storage operations

• ACID:
  – Atomicity: all operations in a transaction happen, or none happens
  – Consistency: if database/storage starts consistent, it ends up consistent
  – Isolation: execution of one transaction is isolated from another
  – Durability: the results of a transaction persists

• Serial schedule: A schedule that \textcolor{red}{\textbf{does not interleave}} the operations of different transactions
  – Transactions run serially (one at a time)