Transactions

Databases

Ira Assent

Intended learning outcomes

- Be able to
  - reason about transactions and describe the ACID properties
  - Identify problems for executions that violate ACID properties

Recap: Information Retrieval (IR)

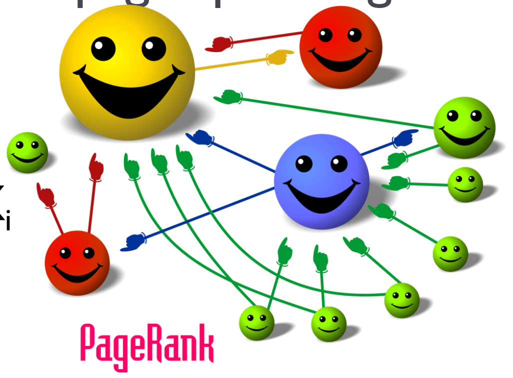
- Information retrieval
  - retrieving documents in response to unstructured keyword search query
- Vector space model
  - Each vector position is weight for term in vocabulary
  - Popular choice: TF-IDF score: $w_{ij} = TF_{ij} \times IDF_i$
    - TF high weight for frequent terms *i* in a document *j*
    - IDF high weight for rare words *i* in few documents (logarithm for reduced impact)
- High noise-to-signal ratio: evaluation
  - Recall: how many of the relevant documents are found?
    - Hits / relevant documents (TP+FN)
  - Precision: how many found documents are relevant?
    - Hits / documents retrieved (TP+FP)
  - F-score: harmonic mean of precision and recall

ID	Term	Document: position
1.	example	1:2, 1:5
2.	inverted	1:8, 2:1
3.	index	1:9, 2:2, 3:3
4.	market	3:2, 3:13

		Relevant?	
		Yes	No
Retrieved?	Yes	☺ Hits TP	☹ False Alarms FP
	No	Misses FN ☹	Correct Rejections TN ☺

Recap: Web Search

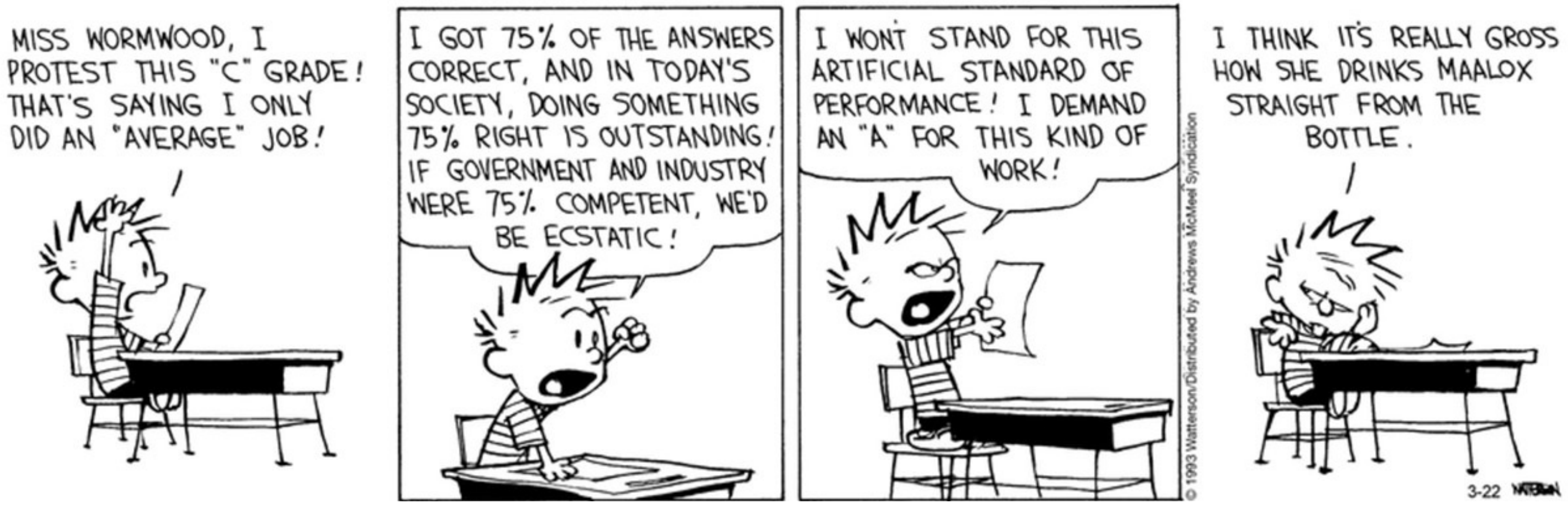
- Web pages connected via hyperlinks
- Search results with high precision at possible expense of recall
- PageRank: page measure of query-independent importance
- pagerank P(A) indicates A's authority attributed by other webpages
  - P(A) is sum of normalized pageranks of all webpages pointing into A
  - $P(A) = \sum_i P(Xi)/O(Xi)$
    - X_i is the set of webpages pointing to A
    - $O(X_i)$ is the number of out-links from page X_i



- Normalized: being pointed to by a page with many outlinks counts less
- Algorithm: assign equal weights initially, iteratively compute page rank until convergence (damping factor)

A look behind the scenes of a DBMS…

- Answering the question "How does the DBMS actually do this?"
  - "Invisible" in using MySQL where we see the result of the following approaches
    - As we will see in more detail later: we can turn off some of the support (for efficiency), but we cannot control what happens
  - We start by looking at multi-user case (recall: DBMS advantage over using files)
    - Several users query the same database to get information
      - University: annoying if users wait until database available to see grades
    - Several users can modify the same database to update information
      - Again, queuing for exclusive access to enter an exam grade not ideal
        - E.g. during exam season…

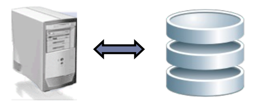

MISS WORMWOOD, I PROTEST THIS "C" GRADE! THAT'S SAYING I ONLY DID AN "AVERAGE" JOB!
I GOT 75% OF THE ANSWERS CORRECT, AND IN TODAY'S SOCIETY, DOING SOMETHING 75% RIGHT IS OUTSTANDING! IF GOVERNMENT AND INDUSTRY WERE 75% COMPETENT, WE'D BE ECSTATIC!
I WON'T STAND FOR THIS ARTIFICIAL STANDARD OF PERFORMANCE! I DEMAND AN "A" FOR THIS KIND OF WORK!
I THINK IT'S REALLY GROSS HOW SHE DRINKS MAALOX STRAIGHT FROM THE BOTTLE.
© 1993 Watterson/Distributed by Andrews McMeel Syndication
3-22 WATTERSON

Transactions: the Setting

- Transactions are the DBMS basic building block for handling queries and modifications
  - Use this to understand which of them
    - can be done at the same time or have to be shielded from one another
    - are successfully completed or need to be "cleaned up" in case of issues
- Example
  - You and your spouse each deposit $100 from different ATM's at about the same time
    - The DBMS better make sure no deposit gets lost

Transactions as the logical unit

- **Transaction: logical unit** of database processing
  - Includes one or more operations on the database
  - May be stand-alone specified in a high-level language like SQL and submitted interactively
    - You and your workbench MySQL
  - Or may be embedded within a program
    - Typical case



- **Transaction boundaries**
  - All database operations between BEGIN and END TRANSACTION are considered a single transaction, i.e., one logical unit
    - MySQL: `START TRANSACTION` or `BEGIN` to start transaction
    - `COMMIT` to end current transaction, making its changes permanent
    - `ROLLBACK` to end current transaction, canceling its changes
    - `SET autocommit` disables or enables the default autocommit mode for the current session

https://dev.mysql.com/doc/refman/8.0/en/commit.html

Example: Computing aggregates in a dynamic world

- Assume a Sells(bar, beer, price) relation
  - Bars selling beer at certain prices
- Suppose that J Bar sells only Bud for $2.50 and Miller for $3.00.
- Sally is querying Sells for the highest and lowest price Joe charges, executes

```
(max) SELECT MAX(price) FROM Sells
           WHERE bar = 'J Bar';
(min) SELECT MIN(price) FROM Sells
           WHERE bar = 'J Bar';
```

- At about the same time, Joe decides to stop selling Bud and Miller, but to sell only Heineken at $3.50

```
(del) DELETE FROM Sells
           WHERE bar = 'J Bar';
(ins) INSERT INTO Sells
           VALUES('J Bar', 'Heineken', 3.50);
```

Execution order?

A. (max) (del) (ins) (min)

B. (max) (min) (del) (ins)

C. (del) (ins) (max) (min)

D. (del) (max) (ins) (min)

```
Sally
(max)  SELECT MAX(price) FROM Sells
                 WHERE bar = 'J Bar';
(min)  SELECT MIN(price) FROM Sells
                 WHERE bar = 'J Bar';
Joe
(del)  DELETE FROM Sells
       WHERE bar = 'J Bar';
(ins)  INSERT INTO Sells
       VALUES('J Bar', 'Heineken', 3.50);
```

Example: Strange Interleaving

- Suppose the steps execute in the order (max)(del)(ins)(min)

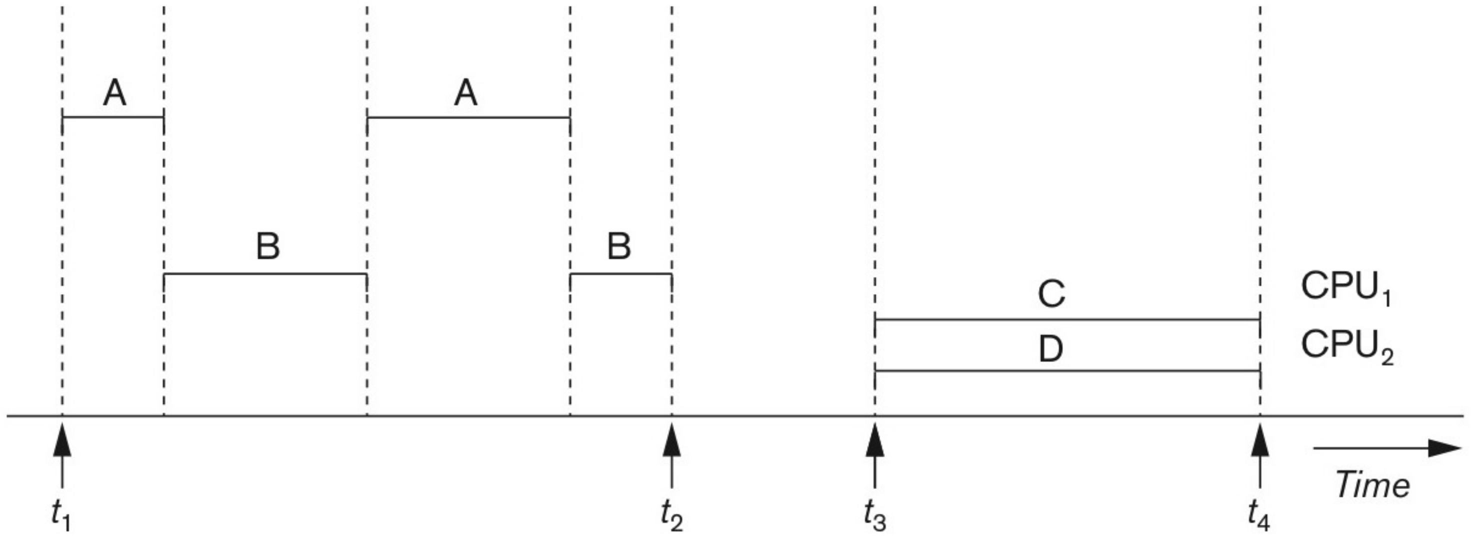
Joe's Prices:	2.50, 3.00	2.50, 3.00		3.50
Statement:	(max)	(del)	(ins)	(min)
Sally's Result:	3.00			3.50

- Sally sees MAX < MIN!
- Although (max) must come before (min) and (del) before (ins), there are no other constraints on their order
- Fixing the problem with transactions
  - If we group Sally's statements (max)(min) into one transaction, then she cannot see this inconsistency
  - She see's Joe's prices at some fixed time
    - Either before or after he changes prices, or in the middle, but the MAX and MIN are computed from the same prices

before or after transaction, never in-between

Terminology for Transaction Processing

- Single-User System: at most one user at a time can use the system
  - Slow, but safe
- Multiuser System: many users can access the system concurrently
  - Fast, typical use case, need to handle interactions
- Concurrency: more than one process / user at a time
  - In the interest of speed (and space), this is what we want
- Interleaved processing: one model of concurrency
  - Processes "take turns" being executed on the CPU
  - (opposite: parallel execution, e.g. on the GPU)



Abstraction

Simple model of database for purposes of discussing transactions

- **Database** modelled as collection of named data items
- **Granularity** of data is ”size” of data items
  - a field, a record , or a whole disk block
  - Concepts for transaction processing are independent of granularity
- Basic operations are **read** and **write**
  - **read_item(X)**
    - Reads a database item named X
      - E.g. into a program variable
  - **write_item(X)**
    - Writes the value of database item X
  - For the purpose of discussion transactions, we will ignore all other operations!
    - Because whatever else the program / user does, does not affect the database state

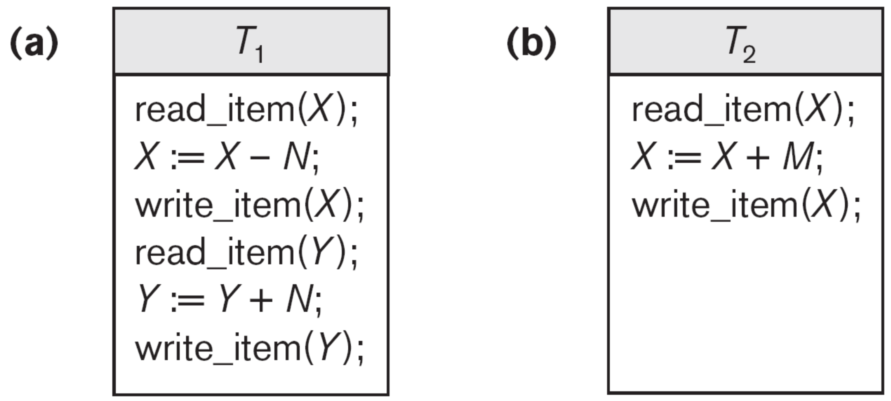
```
SELECT *
   FROM Students
   WHERE id = ‘42’;
```
becomes read_item(X) where X is the row corresponding to Student id 42

```
SELECT *
   FROM Students;
```
becomes read_item(X) where X is table Student

```
UPDATE People
   SET office = ‘Ny-343'
   WHERE userid = ‘ira’;
```
becomes write_item(X) where X is office field for user ira in People table

Example transactions from textbook

- Example transactions



(a) T_1

```
read_item(X);
X := X – N;
write_item(X);
read_item(Y);
Y := Y + N;
write_item(Y);
```

(b) T_2

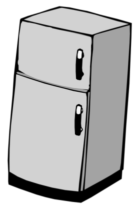
```
read_item(X);
X := X + M;
write_item(X);
```

- E.g. in T_1 "`X:=X-N;`" has no impact on database
  - Only supports our understanding of what the program (transaction) does, not what the database contains (i.e., its state)

- From the point of view of the database state, we do no not need to know about anything but read_item and write_item

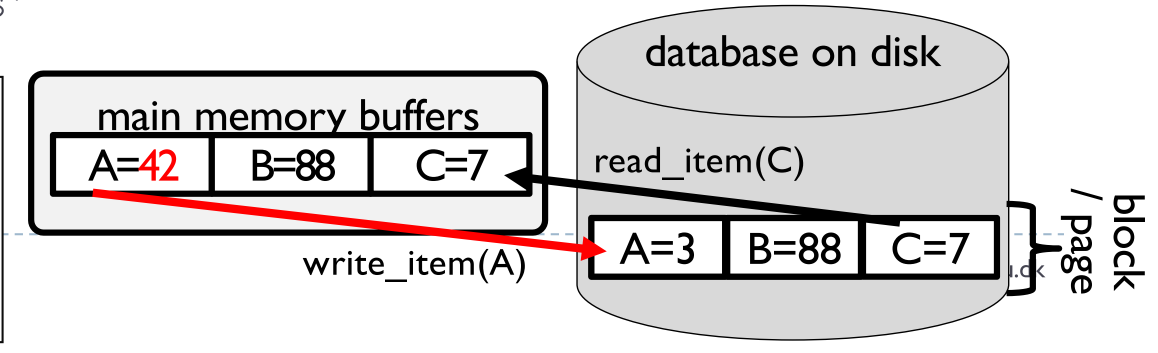
Reading and writing

- **read_item(X)**
  - Find address of disk block that contains item X
  - Copy that disk block into a buffer in main memory (unless already there)
  - Copy item X from buffer to program variable named X (or some other name)
- **write_item(X)**
  - Find address of disk block that contains item X
  - Copy that disk block into a buffer in main memory (unless already there)
  - Copy item X from program variable named X into its correct location in buffer
  - Store updated block from buffer back to disk (either immediately or at later point)



- DBMS loads data from disk (I/O) to main memory, writes changes to disk
  - Operating System (OS) executes these operations (more later)
- data item (read or written) is field of some record
  - E.g. read value of `FirstName where userid='3'` or write value of `Price where Product='Chips'`

T_1	T_2
BEGIN	
read_item(A)	
Write_item(A)	
…	
	BEGIN
	read_item(C)
	…



Change of mind

Joe executes (del)(ins), but after executing them, thinks better of it and issues a ROLLBACK statement.

What happens if Sally executes her transaction after (ins) and before the rollback?

```
Sally
(max)    SELECT MAX(price) FROM Sells
                  WHERE bar = 'J Bar';
(min)    SELECT MIN(price) FROM Sells
                  WHERE bar = 'J Bar';
Joe
(del)    DELETE FROM Sells
         WHERE bar = 'J Bar';
(ins)    INSERT INTO Sells
         VALUES('J Bar', 'Heineken', 3.50);
```

1. She gets an incorrect value that never existed in the database
2. She gets the correct result as if Joe had never executed (del)(ins)
3. She gets the correct result after Joe has executed his statements
4. She gets an incorrect value that needs to be undone

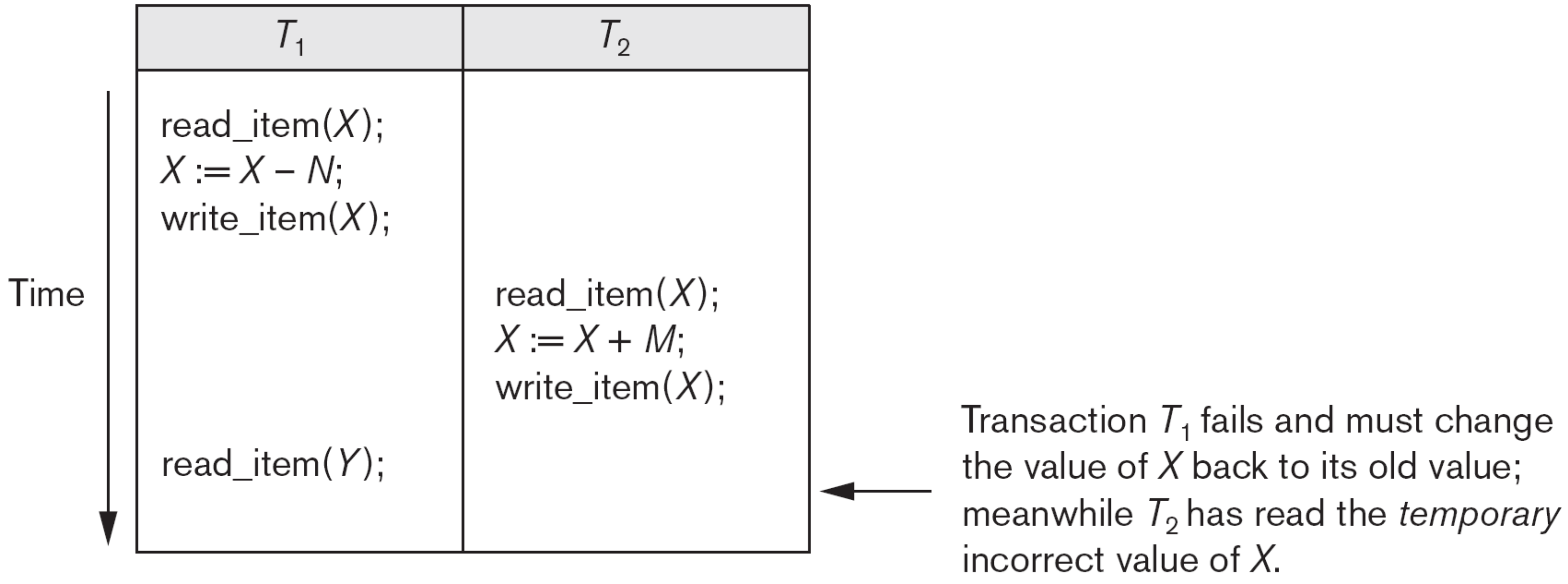
Dirty Read (temporary update)

- If Joe executes (del)(ins), but after executing these statements, thinks better of it and issues a ROLLBACK statement
- And if Sally executes her transactions after (ins) but before the rollback, she sees a value, 3.50, that never existed in the database
- Solution to avoid Dirty Read
  - If Joe executes (del)(ins) as a transaction, its effect cannot be seen by others until the transaction executes COMMIT
    - If the transaction executes ROLLBACK instead, then its effects can never be seen

```
Sally
(max)    SELECT MAX(price) FROM Sells
                 WHERE bar = 'J Bar';
(min)    SELECT MIN(price) FROM Sells
                 WHERE bar = 'J Bar';
Joe
(del)    DELETE FROM Sells
         WHERE bar = 'J Bar';
(ins)    INSERT INTO Sells
         VALUES('J Bar', 'Heineken', 3.50);
```

before or after transaction, never in-between

Textbook example: temporary update problem



	T_1	T_2

Time ↓	read_item(X);
	$X := X - N$;
	write_item(X);	

		read_item(X);
		$X := X + M$;
		write_item(X);

	read_item(Y);	

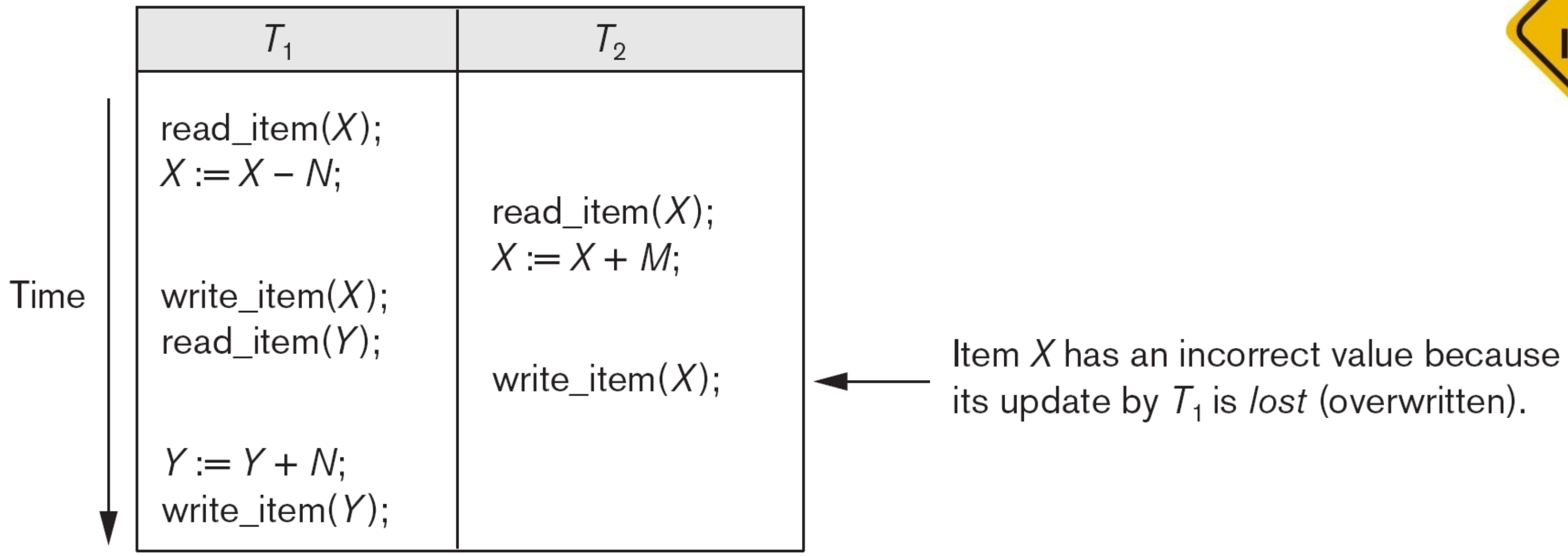
Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

- T_1 writes X, which is read by T_2, but then T_1 fails and needs to be undone
  - T_2 should have never seen this value
  - Temporary update / dirty read

Overview over potential multi-user issues

- **Temporary Update (or Dirty Read) Problem**
  - A transaction updates a database item and then fails for some reason
  - Updated item is accessed by another transaction before it is changed back to its original value
    - *Joe doing a Rollback after Sally read the value…*
- **Lost Update Problem**
  - two transactions access same item rendering its value incorrect
    - *You and your spouse both withdrawing money from your joint bank account…*
- **Incorrect Summary Problem**
  - A transaction calculates an aggregate on a number of records while another transaction updates some of these records
    - aggregate function may be based on values from both before and after update
    - *Like min/max with del/ins example with Sally and Joe*
- **Unrepeatable Read Problem**
  - A transaction reads the same item twice
  - Another transaction changes the value between first and second read
    - *You put an item in your shopping cart and when you proceed to checkout it is sold out…*

Textbook example: lost update problem



T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;

write_item(X);
read_item(Y);	

	write_item(X);
$Y := Y + N$; write_item(Y);	

Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

- T_1 reads X, then writes X, T_2 reads "old" value of X, then also writes X
  - Update (write) by T_1 is lost (overwritten by T_2)

Example: unrepeatable read problem

T1	T2
`read_item(X);`	
	`read_item(X);`
`read_item(Y);`	
`Y:=Y+X;`	
	`X:=X-1;`
	`write_item(X);`
`read_item(X);`	

- T_1 reads X twice
- Between these reads, T_2 changes value of X
- So T_1 gets inconsistent view of X's value
- Often, such reads are part of e.g. finding a seat for a theatre concert, then checking out

What problem is illustrated here?

A. Unrepeatable read problem

B. Lost update problem

C. Incorrect summary problem

D. Temporary update problem

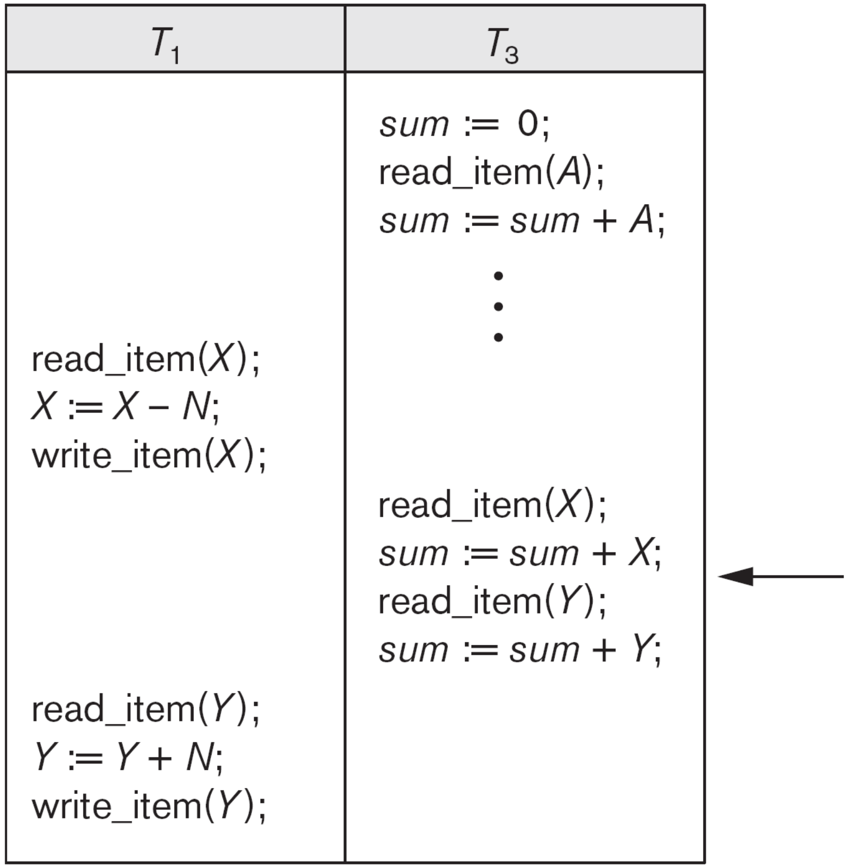
T_1	T_3
	sum := 0; read_item(A); sum := sum + A; ⋮

read_item(X);
X := X − N;
write_item(X);	

	read_item(X);
	sum := sum + X;
	read_item(Y);
	sum := sum + Y;

read_item(Y);
Y := Y + N;
write_item(Y);	

Textbook example: incorrect summary problem



T_1	T_3

	sum := 0;
	read_item(A);
	sum := *sum* + A;
	⋮

read_item(X);
X := X − N;
write_item(X);	

	read_item(X);
	sum := *sum* + X;
	read_item(Y);
	sum := *sum* + Y;

read_item(Y);
Y := Y + N;
write_item(Y);	

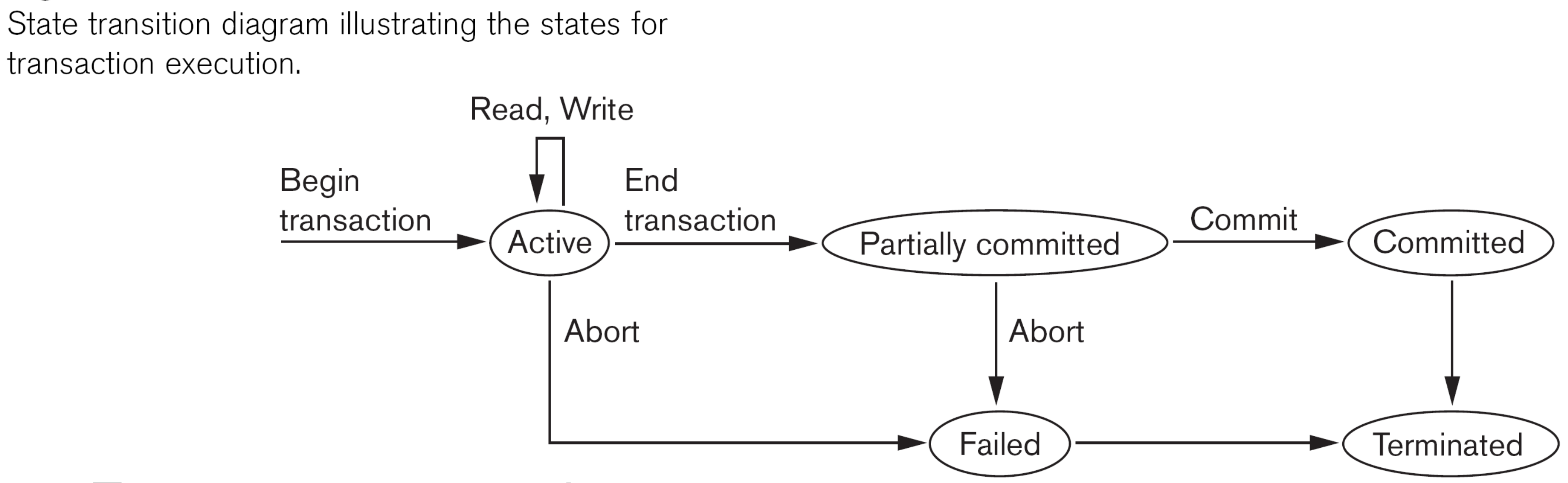
T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

- T_3 computes an aggregate of X and Y, but reads X after T_1 changes it, and Y before T_1 changes it
  - Sum not correct, should be either before or after T_1's changes

Keeping track of transactions

- A **transaction** is an atomic unit of work that is either completed in its entirety or not done at all
  - For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts
- **Transaction states**:
  - Active state
  - Partially committed state
  - Committed state
  - Failed state
  - Terminated State

State Transition Diagram

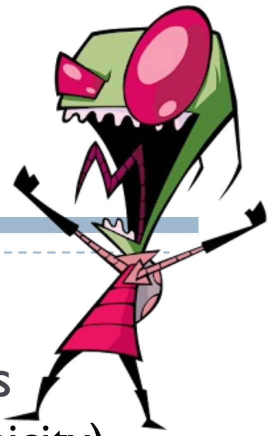


- Transactions are started
  - Are **active** when they "work", i.e., read, write
  - When they are "done", they become **partially committed**
    - When the transaction can be successfully completed by the system, they are **committed**, and after clean up, they are **terminated** (out of the system)
    - When it cannot be committed, it is instead aborted and becomes **failed**, and after clean up, will also be **terminated** (out of the system)

Failing transactions

- Computer failure (system crash)
- Hardware or software error occurs during transaction execution
  - If hardware crashes, memory contents may be lost
  - Disk may fail, so read or write cannot be executed
- Transaction or system error
  - Some operation in transaction may cause it to fail
    - E.g. integer overflow or division by zero
  - erroneous parameter values or logical programming error
  - user may interrupt the transaction during its execution
- Local errors or exception conditions detected by transaction
    - E.g. data for transaction not be found
    - E.g. insufficient account balance in banking
- Concurrency control enforcement (next lecture)
- Physical damage /issue
  - E.g. aircondition fails, wrong tape mounted, disk stolen

ACID Transactions



- DBMS supports **ACID** transactions
  - **Atomic** : Either the whole transaction process is done or none is
    - Any transaction does all its work on the database or none at all (atomicity)
      - so that there are no partial updates to the data or the like
  - **Consistent** : Database constraints are preserved
    - Any transaction that starts on a consistent database leaves it again in a consistent state; temporary inconsistency while it is still working is permitted;
      - so that there are no data integrity issues or the like
  - **Isolated** : It appears to user as if only one transaction process executes at a time
    - No transaction interferes with another
      - So that transactions do not produce conflicting results; looks like first one of the transactions has the database to itself until done, then the other
  - **Durable** : Effects of a process do not get lost if the system crashes
    - So that the changes made by a transaction actually persist and do not disappear
      - E.g. data is not actually written to the disk or the like

ACID is very powerful

- ACID properties describe fundamental characteristics of transactions that support efficient multi-user databases
- Grouping operations (statements, queries) into transactions means they are logically protected
- We will study how to achieve these properties
  - Concurrency control techniques focus on having several transactions not interfere with one another and maintaining integrity constraints (isolation, consistency)
  - Recovery techniques focus on making sure in case of a crash that we get back to a valid state of the database (atomicity, durability)
- From user point of view: well-behaved
  - We look at algorithms behind the scene

How to recover?

How can we undo/redo (parts of) a transaction in case of a failure?

1. Restart the database from scratch and ask all transactions to run again
2. Keep track of all changes in a separate file on disk
3. Check all integrity constraints and undo/redo if violated
4. Define triggers to recover to valid state

System Log

- **Log** or **Journal**: The log keeps track of all transaction operations that affect the values of database items
  - This information may be needed to permit recovery from transaction failures
  - The log is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure
  - In addition, the log is periodically backed up to archival storage (tape) to guard against such **catastrophic failures**
    - E.g. harddisk dies

The recovery manager

- Recovery manager keeps track of
  - **begin_transaction**
    - marks the beginning of transaction execution
  - **read** or **write**
    - read or write operations on the database items executed as part of transaction
  - **end_transaction**
    - marks end limit of transaction execution (no more reads and writes)
    - Now check whether changes made by transaction can be permanently applied to the database or whether the transaction has to be aborted
  - **commit_transaction**
    - Signals successful end of transaction so that any changes (updates) can be safely committed to the database
  - **rollback** (or **abort**)
    - signals that transaction has ended unsuccessfully, so that changes or effects of transaction to the database must be undone

Recovery

- Recovery techniques use the following operators:
  - **undo**: Similar to rollback except that it applies to a single operation rather than to a whole transaction
  - **redo**: This specifies that certain *transaction operations* must be *redone* to ensure that all the operations of a committed transaction have been applied successfully to the database

Transactions in SQL

- SQL supports transactions, often behind the scenes
  - Each statement issued at the generic query interface is a transaction by itself
  - In programming interfaces like JDBC, a transaction begins the first time an SQL statement is executed and ends with the program or an explicit end
  - Many DBMS: `BEGIN` or `START TRANSACTION`
- The SQL statement `COMMIT` causes a transaction to complete
  - It's database modifications are now permanent in the database
- The SQL statement `ROLLBACK` also causes the transaction to end, but by aborting
  - No effects on the database
- Failures like division by 0 can also cause rollback, even if the programmer does not request it

Transactions in MySQL

- Generally works as described
- Transaction support in InnoDB (default)
- MySQL: default "auto commit"
  - `SET autocommit={0 | 1}` to change
- DDL: cause implicit commit+cannot roll back

https://dev.mysql.com/doc/refman/8.0/en/commit.html

https://dev.mysql.com/doc/refman/8.0/en/sql-transactional-statements.html

Execute these statements one after the other in the order given

- Assume that the initial score for movieid 1 is 42, what is the score for movieid 1 at the last SELECT ?

A. 42
B. 15
C. 5
D. 10

```
SET autocommit=0;
START TRANSACTION;
SELECT * FROM movie;
UPDATE movie SET score=10 WHERE movieid=1;
rs.updateInt("score",15);
END TRANSACTION;
START TRANSACTION;
SELECT * FROM movie;
UPDATE movie SET score=5 WHERE movieid=1;
ROLLBACK;
SELECT * FROM movie;
```

Intended learning outcomes

- Be able to
  - reason about transactions and describe the ACID properties
  - Identify problems for executions that violate ACID properties

  - Remember: Tuesday video screening of Concurrency control lecture with the one and only Katrine!

Acknowledgements: some slide material by Jeff Ullman, Torben Bach Pedersen, and teachcoop

What was this all about?

Guidelines for your own review of today's session

- Single-user access and multi-user access to a database differ in that…
- We may run into the following issues when allowing for multi-user access…
  - Transactions are…
  - They allow us to…
  - In SQL, we can specify…
- ACID properties are…
  - Transaction states are…
  - Concurrency means…
    - It allows us to…
  - Recovery means…
    - It allows us to…