Transaction and Review

CSCI435

Outline

- Basic concepts
- Desirable properties of transactions
- Review

Introduction

- ***Single-user system*** = at most one user at a time
- ***Multiuser system*** = many users can access the system concurrently
- ***Concurrent*** = at the same time
- Concurrent execution is either
  - ***Interleaved*** on a single CPU
  - ***Processed in parallel*** executed on multiple CPUs

Transactions

- ***Transaction*** = logical unit of DB processing that includes one or more ***access operations***
  - Read = retrieval
  - Write = insert
  - Update = delete
- A transaction is either
  - Embedded within a program
  - Submitted interactively in a high level language like SQL
- ***Transaction boundaries*** indicate when the transaction begins and ends
- Application program may contain several transactions separated by their transaction boundaries

Databases and transactions

- Think of a database as a collection of named data items
- Basic unit of data transfer from disk to main memory is one block
- ***Data item*** = what is read or written
- **Granularity** of data item: a field, a record , or a disk block
- Basic operations
  - **Read**
  - **Write**

Reading and writing

- **read_item(X)**: Reads database item named X into program variable also named X
  - Find address of block that contains X
  - Copy that block into buffer in main memory (if that block is not already in some main memory buffer)
  - Copy item X from buffer to program variable named X
- **write_item(X)**: Writes value of program variable X into database item named X
  - Find address of block that contains X
  - Copy that block into buffer in main memory (if that block is not already in some main memory buffer)
  - 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 some later point in time)

What transactions look like

(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);

How transactions fail (1)

- Computer failure = hardware or software error during transaction execution, contents of main memory may be lost
- Transaction or system error
  - Integer overflow or division by zero
  - Erroneous parameter values
  - Logical programming error
  - Interruption by user
- Local errors or exception conditions detected by the transaction
  - Necessary data not found
  - Condition causes cancellation, e.g., insufficient funds to withdraw
  - Programmed abort in the transaction causes it to fail

How transactions fail (2)

- Concurrency control enforcement may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock (Chapter 18)
- Disk failure
  - Data loss due to read or write malfunction or head crash
  - Can happen during a read or a write
- Physical problems and catastrophes
  - Power failure
  - Air-conditioning failure
  - Fire
  - Theft
  - Sabotage
  - Overwriting media by mistake
  - Mounting wrong media…

Lost update problem

- Two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect

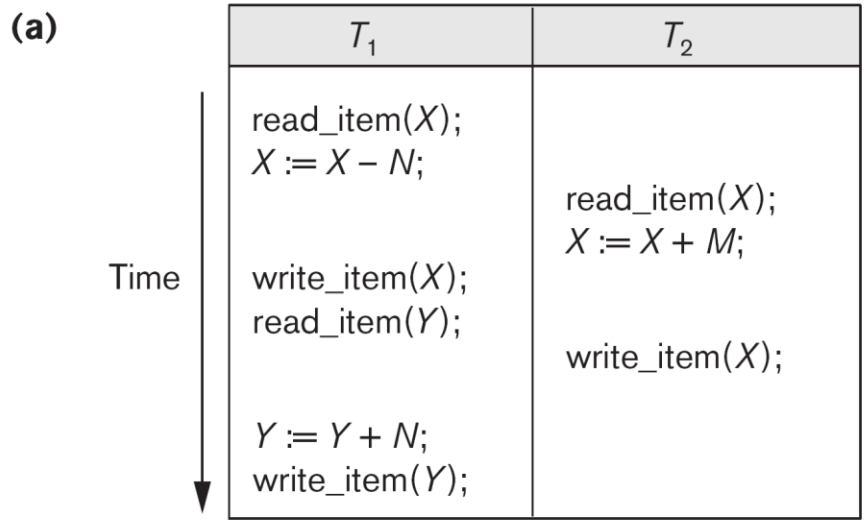
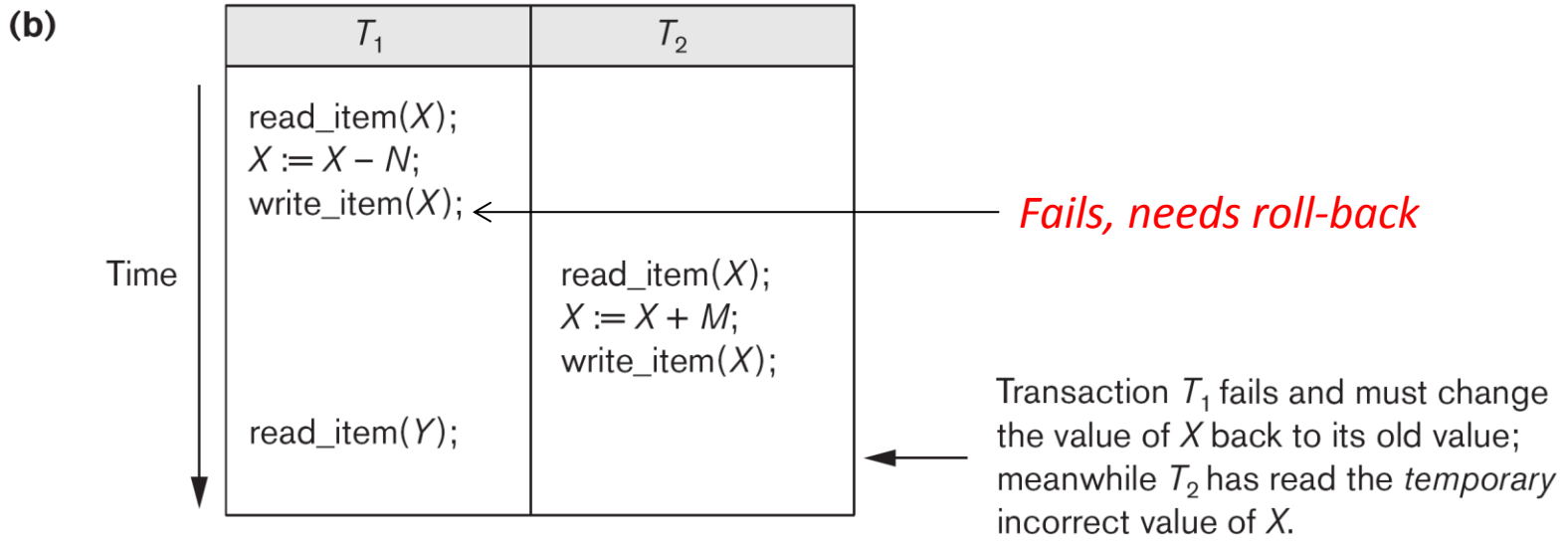


Figure 21.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

Temporary update problem (dirty read)

- One transaction updates a database item and then the other fails for some reason, but the updated item is accessed by another transaction before it is changed back to its original value

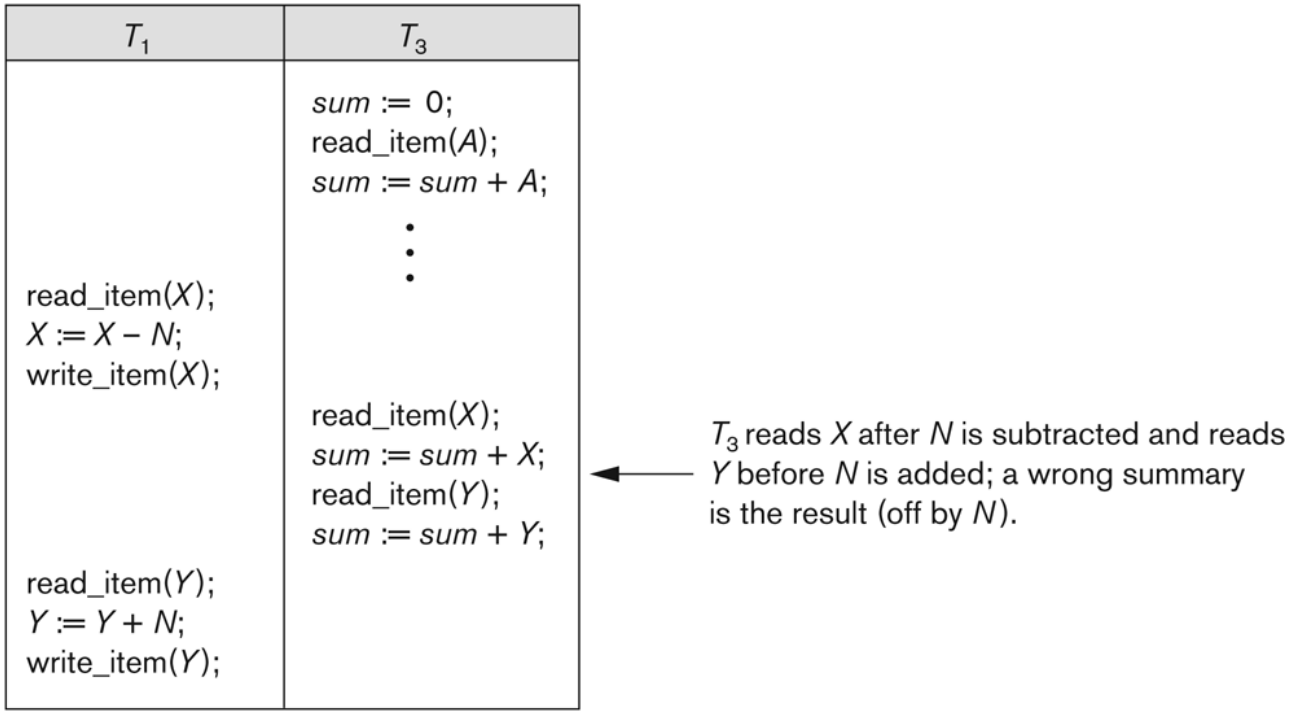


Incorrect summary problem

- One transaction is *calculating* an aggregate summary function on a set of records while other transactions are *updating* some of these

Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

(c)



So if transactions look like this…

(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);

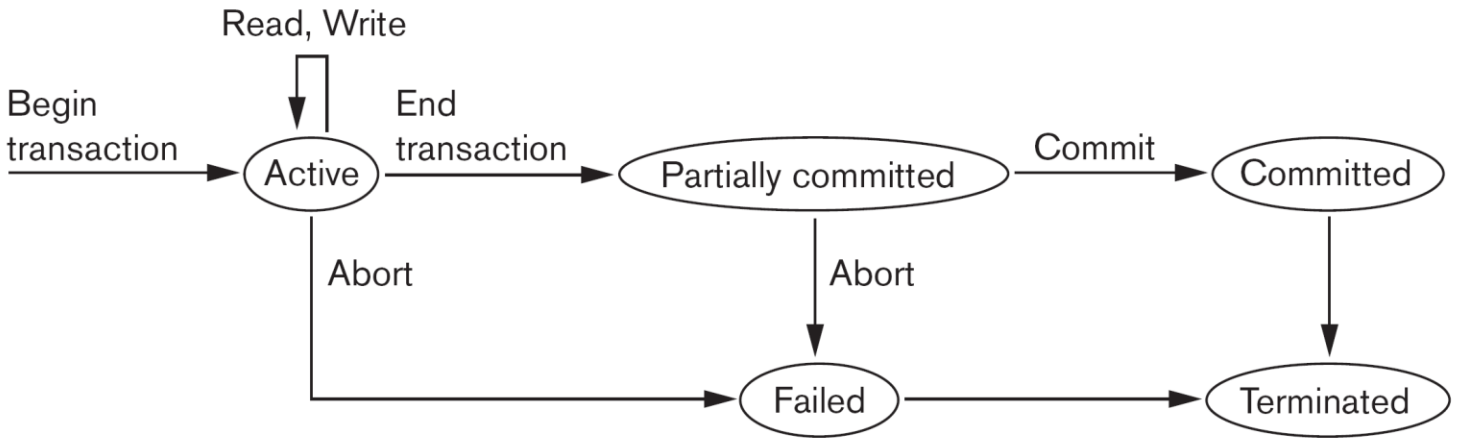
ACID properties of transactions

To protect against all these kinds of failure, we want every transaction to be

- ***Atomic*** = either ***done completely or not at all***
- ***Consistency preserving*** = correct execution of the transaction must take the database from one consistent state to another
- ***Isolated*** = should not make its updates visible to other transactions until it is committed
- When enforced strictly, this solves the temporary update problem and makes cascading rollbacks of transactions unnecessary
- ***Durable*** = once a transaction changes the database and the changes are committed, these changes must ***never*** be lost because of subsequent failure

State transition diagram

Figure 21.4
State transition diagram illustrating the states for transaction execution.



- Note the 5 possible transaction states
  - Active
  - Partially committed
  - Committed
  - Failed
  - Terminated
- To recover from failure, we must know what state all transactions were in

For recovery, track these operations

- ***begin_transaction*** = beginning of transaction execution
- **read** or **write** operations on database items executed as part of a transaction
- ***end_transaction*** = all read and write transaction operations have ended
- Marks the end limit of transaction execution
- May be necessary to check whether changes introduced by the transaction can be permanently applied to the database (or whether the transaction has to be aborted because it violates concurrency control or for some other reason)
- ***commit_transaction* =** signals successful end of transaction, any changes it executed can be safely committed to the database and will ***not*** be undone
- ***rollback*** (or ***abort***) = signals that transaction ended unsuccessfully,
- any changes or effects it applied to the database ***must*** be undone

Recovery techniques

What can we do to restore database to a consistent state?

- ***Undo*** = like rollback but applies to one ***operation***
- ***Redo*** = certain ***transaction operations*** must be *redone* to ensure that all the operations of a committed transaction have been applied successfully to the database

System log (journal)

- Keeps track of all transaction operations that affect the values of database items
  - May be needed for recovery from transaction failure
  - Kept on disk, so it is affected only by disk or catastrophic failure
  - Periodically backed up to archival storage (tape) to guard against catastrophic failures

Entries in the system log

Here, T refers to a unique ***transaction-id*** generated automatically by the system and used to identify each transaction:

- **[start_transaction,T]** records that T has started execution
- **[write_item,T,X,old_value,new_value]** records that T changed the value of database item X from old_value to new_value.
- **[read_item,T,X]** records that T read the value of database item X
- **[commit,T]** records that T completed successfully, and affirms that its effect can be committed (recorded permanently) to the database
- **[abort,T]** records that transaction T has been aborted

Recovery using log records

- After a system crash, can recover to a consistent database state by examining the log and using techniques described in Chapter 19
- Because the log contains a record of every write operation that changes the value of some database item, it is possible to **undo** the effect of these write operations of a transaction T by tracing backward through the log and resetting all items changed by a write operation of T to their old values
- Can also **redo** the effect of the write operations of a transaction T by tracing forward through the log and setting all items changed by a write operation of T (that did not get done permanently) to their new values

Commit and rollback

- Transaction T reaches its ***commit point*** when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database has been recorded in the log
- Beyond its commit point, a transaction is said to be committed, and its effect is assumed to be permanently recorded in the database
- Transaction then writes an entry [commit,T] into the log
- ***Roll back*** needed for transactions that have a [start_transaction,T] entry in the log but no commit entry [commit,T] in the log

Redoing transactions

- Transactions that have written their commit entry in the log must also have recorded all their write operations in the log (otherwise they would not be committed)
- Their effect on the database can be redone from the log entries
- At the time of a system crash, only the log entries that have been written back to disk are considered in the recovery process (because the contents of main memory may be lost)
- ***Force-writing*** the log file
- Occurs before committing a transaction
- Any portion of the log that has not been written to the disk yet must now be written to the disk

Summary

- Transaction and system concepts
- ACID properties

Final Review

Materials

- Chapter 4 and 5 (SQL)
- Chapter 15 and 16 (Normalization)
- Chapter 18 (Index)
- Chapter 21 (Transaction)

Data retrieval in SQL

- Basic syntax

  SELECT *attribute* [,…] attribute names whose values are to be retrieved

  FROM *table* [,…] relation names required to process the query

  WHERE *condition* [,…];
- Produces a ***temporary*** table
- Subquery

JOIN

- Cross join
- Inner join
- Natural Join
- Outer Join
- Example:
  - SELECT * FROM T1, T2;
  - SELECT * FROM T1 INNER JOIN T2 ON *condition;*
  - SELECT * FROM T1 NATURAL JOIN T2;
  - SELECT * FROM T1 LEFT OUTER JOIN T2 ON *condition;*

Aggregate functions

- SQL has many built-in functions including the aggregates COUNT, SUM, MAX, MIN, and AVG
- Example: Find the maximum salary, the minimum salary, and the average salary among all employees.

```
SELECT MAX(SALARY), MIN(SALARY), AVG(SALARY)
FROM EMPLOYEE;
```

- How many employees in the company? in the 'Research‘ department?

```
SELECT COUNT (*)FROM EMPLOYEE;
SELECT COUNT (*)
FROM EMPLOYEE, DEPARTMENT
WHERE DNO=DNUMBER AND DNAME='Research’;
```

GROUP BY

- Syntax: **GROUP BY** {*attribute* | *attribute* [,…]}
- Example: For each department, retrieve the department number, the number of employees in the department, and their average salary.

  SELECT DNO, COUNT (*), AVG (SALARY)

  FROM EMPLOYEE

  GROUP BY DNO;
- This separates EMPLOYEE tuples into groups of tuples with the same value for the ***grouping attribute*** DNO
- COUNT and AVG are applied to each group of tuples separately
- SELECT clause includes only the grouping attribute and the functions to be applied on each group of tuples (no access to detail in tuples themselves)

HAVING

- To retrieve values only for those ***groups that satisfy certain conditions***
- **HAVING** specifies a selection condition for groups (rather than one for individual tuples)
- Syntax: HAVING *condition*
- Example: For each project *on which more than two employees work*, retrieve the project number, project name, and the number of employees who work on that project.

```
SELECT PNUMBER, PNAME, COUNT(*)
FROM PROJECT, WORKS_ON
WHERE PNUMBER=PNO
GROUP BY PNUMBER, PNAME
HAVING COUNT (*) > 2;
```

What Questions Look Like

- Given a question, write a SQL query
- Given a SQL query, decide its output
- Given a SQL query, describe what it asks about

What is the Key

- Superkey
- Key
- Candidate key
- Primary key
- Algorithm to determine a key

Normalization

- **1NF:**
- **2NF:**
- **3NF:**
- **BCNF:**

Normalization

- **1NF:** Attribute value must be single value
- **2NF:** 1NF + not partially dependent on the composite primary key
- **3NF:** 2NF + no transitive FD
- **BCNF:** 3NF, not have multiple overlapping candidate keys

- Given a Relation R, you should determine if it satisfy any above normal forms.

Algorithm for a key from a set of FDs

- **Input**: relation *R* and a set of FDs *F* on the attributes of *R*
- Let *K* = *R*
- For each attribute *A* in *K*

  Compute (*K* – *A*)+ with respect to *F*

  If (*K* – *A*)+ contains all the attributes in *R*

  then set *K* to *K* – {*A*}
- Example: *R(*A,B,C,D), A -> BCD, BC ->AD, D -> C
  - *K* = {A,B,C,D}
  - (*K* – A)+ = {A,B,C,D} so *K* = {B,C,D}
  - (*K* – B)+ = {C,D} so no change
  - (*K* – C)+ = {B,C,D} so no change
  - (*K* – D)+ = {B,C} so *K* = {B,C,D}
  - *R*(A,<u>B</u>,<u>C</u>,<u>D</u>)

  (*K* – B)+ = {A,B,C,D} so *K* = {A,C,D}
  (*K* – C)+ = {A,B,C,D} so *K* = {A,D}
  (*K* – D)+ = {A,B,C,D} so so *K* = {A}
  R(<u>A</u>,B,C,D)

- Given a relation R, a set of FDs F, you should be able to find all keys, including superkeys.

Concept of Index

- ***Dense index***
- ***Sparse index***
- ***Primary index***
- ***Secondary index***

- You should be able to explain them in clear English

How big is an index search?

- Given data file EMPLOYEE(NAME, SSN, ADDRESS, JOB, SAL, ...) with record size R = 100 bytes, block size B = 1024 bytes, r = 30000 records

- **Blocking factor** bfr = $\lfloor B/R \rfloor = \lfloor 1024/100 \rfloor = 10$ records/block
- Number of file blocks b = $(r/bfr) = \lceil 30000/10 \rceil = 3000$ blocks
- For dense index on SSN, field size VSSN = 9 bytes, record pointer size PR = 6 bytes
- Index entry size RI=(VSSN+ PR) = (9 + 6) = 15 bytes
- **Index blocking factor** BfrI = $\lfloor B / RI \rfloor = \lfloor 1024/ 15 \rfloor = 68$ entries/block
- Number of index blocks bi = $(r / BfrI) = \lceil 30000/68 \rceil = 45$ blocks
- Average search cost in block accesses
- Linear (no index): (r/2) = 30000/2 = 15000 block accesses
- Binary (no index): log2b= log230000= 12 block accesses
- Index and binary search: log2bi +1= log245+1= 7 block accesses

- You should be able to carry out such calculations

What is Transaction?

- ***Transaction*** = logical unit of DB processing that includes one or more ***access operations***
  - Read = retrieval
  - Write = insert
  - Update = delete
- A transaction is either
  - Embedded within a program
  - Submitted interactively in a high level language like SQL
- ***Transaction boundaries*** indicate when the transaction begins and ends
- Application program may contain several transactions separated by their transaction boundaries

What are ACID properties?

- ***Atomic***
- ***Consistency preserving***
- ***Isolated***
- ***Durable***

What are ACID properties?

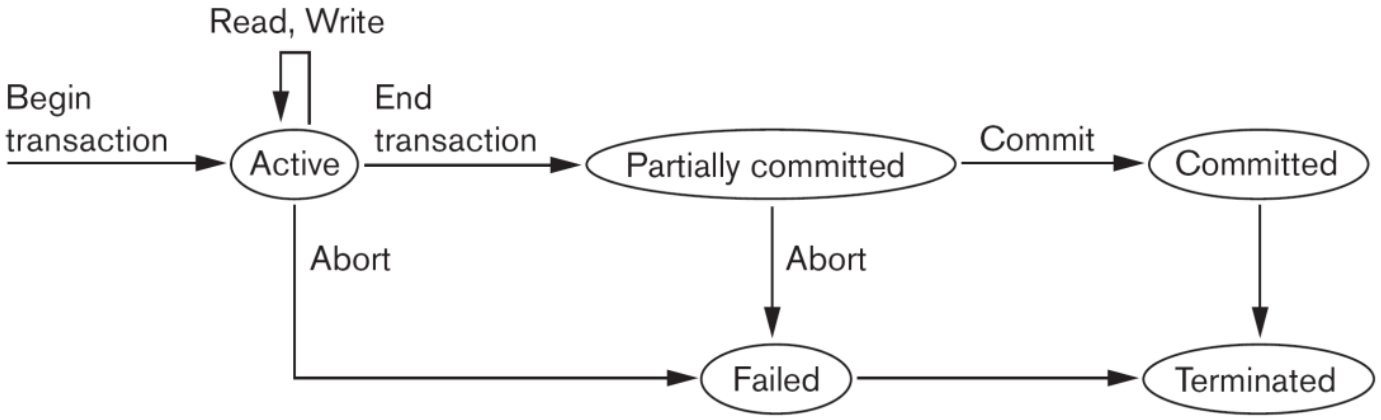
- ***Atomic*** = either ***done completely or not at all***
- ***Consistency preserving*** = correct execution of the transaction must take the database from one consistent state to another
- ***Isolated*** = should not make its updates visible to other transactions until it is committed
- ***Durable*** = once a transaction changes the database and the changes are committed, these changes must ***never*** be lost because of subsequent failure

- You should be explain these properties clearly in English

How does Database Handle Transaction?

- State transition diagram

Figure 21.4
State transition diagram illustrating the states for transaction execution.



- System Log

Good Luck

- May 21 (Wed), 3:00pm-5:00pm
- 1000G
- Close book