Deletion

- Delete all instructors
  
  \[
  \text{delete from} \quad \text{instructor}
  \]

- Delete all instructors from the Finance department
  
  \[
  \text{delete from} \quad \text{instructor} \\
  \text{where} \quad \text{dept\_name} = 'Finance';
  \]

- Delete all tuples in the instructor relation for those instructors associated with a department located in the Watson building.
  
  \[
  \text{delete from} \quad \text{instructor} \\
  \text{where} \quad \text{dept\_name in} \ (\text{select} \ \text{dept\_name} \\
  \quad \text{from} \quad \text{department} \\
  \quad \text{where} \quad \text{building} = 'Watson');
  \]
Deletion (Cont.)

- Delete all instructors whose salary is less than the average salary of instructors

\[
delete \text{ from} \quad \text{instructor} \\
\text{where} \quad \text{salary} < (\text{select} \ \text{avg} (\text{salary}) \\
\text{from} \quad \text{instructor});
\]

- Problem: as we delete tuples from deposit, the average salary changes
- Solution used in SQL:
  1. First, compute \text{avg} (salary) and find all tuples to delete
  2. Next, delete all tuples found above (without recomputing \text{avg} or retesting the tuples)

Insertion

- Add a new tuple to \text{course}

\[
\text{insert into} \quad \text{course} \\
\text{values} (‘CS-437’, ‘Database Systems’, ‘Comp. Sci.’, 4);
\]

- or equivalently

\[
\text{insert into} \quad \text{course} (\text{course}_\text{id}, \text{title}, \text{dept}_\text{name}, \text{credits}) \\
\text{values} (‘CS-437’, ‘Database Systems’, ‘Comp. Sci.’, 4);
\]

- Add a new tuple to \text{student} with \text{tot_creds} set to null

\[
\text{insert into} \quad \text{student} \\
\text{values} (‘3003’, ‘Green’, ‘Finance’, \text{null});
\]
Insertion (Cont.)

- Add all instructors to the student relation with tot_creds set to 0
  
  ```
  insert into student
  select ID, name, dept_name, 0
  from instructor
  ```

- The `select from where` statement is evaluated fully before any of its results are inserted into the relation.
  Otherwise queries like
  
  ```
  insert into table1 select * from table1
  ```

  would cause problem

Updates

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others by a 5%
  
  - Write two `update` statements:
    
    ```
    update instructor
    set salary = salary * 1.03
    where salary > 100000;
    
    update instructor
    set salary = salary * 1.05
    where salary <= 100000;
    ```

  - The order is important
  
  - Can be done better using the `case` statement (next slide)
Case Statement for Conditional Updates

- Same query as before but with case statement

```sql
update instructor
    set salary = case
    when salary <= 100000 then salary * 1.05
    else salary * 1.03
    end
```

Updates with Scalar Subqueries

- Recompute and update tot_creds value for all students

```sql
update student S
    set tot_cred = (select sum(credits)
        from takes, course
        where takes.course_id = course.course_id and
        S.ID= takes.ID and
        takes.grade <> 'F' and
        takes.grade is not null);
```

- Sets tot_creds to null for students who have not taken any course
- Instead of sum(credits), use:

```sql
case
    when sum(credits) is not null then sum(credits)
    else 0
end
```
Integrity Constraints on a Single Relation

- not null
- primary key
- unique
- check \((P)\), where \(P\) is a predicate

Not Null and Unique Constraints

- not null
  - Declare \textit{name} and \textit{budget} to be \texttt{not null}
    
    \texttt{name varchar(20) not null}
    \texttt{budget numeric(12,2) not null}
  
- unique \((A_1, A_2, \ldots, A_m)\)
  - The unique specification states that the attributes \(A_1, A_2, \ldots, A_m\) form a candidate key.
  - Candidate keys are permitted to be null (in contrast to primary keys).
The check clause

- `check (P)`
  where P is a predicate

Example: ensure that semester is one of fall, winter, spring or summer:

```sql
create table section (  
course_id varchar (8),
sec_id varchar (8),
semester varchar (6),
year numeric (4,0),
building varchar (15),
room_number varchar (7),
time_slot_id varchar (4),
primary_key (course_id, sec_id, semester, year),
check (semester in ('Fall', 'Winter', 'Spring', 'Summer'))
);
```

Cascading Actions in Referential Integrity

- `create table course (  
course_id  char(5) primary key,
title      varchar(20),
department varchar(20) references department
)
```

- alternative actions to cascade: set null, set default
Integrity Constraint Violation During Transactions

- E.g.

```sql
create table person ( 
  ID char(10),
  name char(40),
  mother char(10),
  father char(10),
  primary key ID,
  foreign key father references person,
  foreign key mother references person)
```

- How to insert a tuple without causing constraint violation?
  - insert father and mother of a person before inserting person
  - OR, set father and mother to null initially, update after inserting all persons (not possible if father and mother attributes declared to be not null)
  - OR defer constraint checking (next slide)

Transaction

- Sequence of operations treated as a “single unit”
  - Either all happen, or none do
- Various syntaxes
  - SQL:1999: `begin atomic ... end`
  - Oracle: `set transaction ... commit`
- Default in most DBMSs: each statement is a transaction
Oracle Syntax

• Starting a transaction:
  – commit; -- End previous transaction
  – set transaction; -- Start the new transaction
  – set constraint all deferred; -- Check at commit
  – <statements>
  – commit; -- End the transaction
• Can rollback instead of commit
  – As if the transaction never happened

Second goal of transactions: Sequence of Operations

• Update should complete entirely
  – update stipend set stipend = stipend*1.03;
  – What if it gets halfway and the machine crashes?
• What about multiple operations?
  – Withdraw x from Account1
  – Deposit x into Account2
• Simultaneous operations?
  – Print paychecks while stipend being updated
**Concurrent Executions**

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **Increased processor and disk utilization**, leading to better transaction **throughput**
    - E.g. one transaction can be using the CPU while another is reading from or writing to the disk
  - **Reduced average response time** for transactions: short transactions need not wait behind long ones.

- **Concurrency control schemes** – mechanisms to achieve isolation
  - That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

**Example**

- Consider two transactions:

  - **T1:** `BEGIN A=A+100, B=B-100 END`
  - **T2:** `BEGIN A=1.01*A, B=1.01*B END`

- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together.

- Assume `A=100, B=100` at start. Result:
  - A. `A = 202, B = 0`
  - B. `A = 201, B = 1`
  - C. `A = 202, B = 1`
  - D. `A = 201, B = 0`
Consider a possible interleaving:

- **T1:** \( A = A + 100, \ B = B - 100 \)
- **T2:** \( A = 1.01^*A, \ B = 1.01^*B \)

Assume \( A = 100, \ B = 100 \) at start. Result:

A. \( A = 202, \ B = 0 \)
B. \( A = 201, \ B = 1 \)
C. \( A = 202, \ B = 1 \)
D. \( A = 201, \ B = 0 \)
Example (Contd.)

• Consider a possible interleaving:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = A + 100,</td>
<td>A = 1.01 * A,</td>
</tr>
<tr>
<td>B = B - 100</td>
<td>B = 1.01 * B</td>
</tr>
</tbody>
</table>

• Assume A = 100, B = 100 at start. Result:
  A. A = 202, B = 0
  B. A = 201, B = 1
  C. A = 202, B = 1
  D. A = 201, B = 0

Chris Clifton - CS34800

Example (Contd.)

• Consider a possible interleaving:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = A + 100,</td>
<td>A = 1.01 * A,</td>
</tr>
<tr>
<td>B = B - 100</td>
<td>B = 1.01 * B</td>
</tr>
</tbody>
</table>

• Assume A = 100, B = 100 at start. Result:
  A. A = 202, B = 0
  B. A = 201, B = 1
  C. A = 202, B = 1
  D. A = 201, B = 0

Chris Clifton - CS34800
Solution: Transaction

- Sequence of operations grouped into a transaction
  - Externally viewed as *Atomic*: All happens at once
  - DBMS manages so even the programmer gets this view
- Oracle: Requires additional argument
  - set transaction serializable

ACID properties

*Transactions have:*

- Atomicity
  - All or nothing
- Consistency
  - Changes to values maintain integrity
- Isolation
  - Transaction occurs as if nothing else happening
- Durability
  - Once completed, changes are permanent
Scheduling Transactions

• **Serial schedule**: Schedule that does not interleave the actions of different transactions.

• **Equivalent schedules**: For any database state, the effect (on the set of objects in the database) 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.

(If each transaction preserves consistency, every serializable schedule preserves consistency.)

Chris Clifton - CS34800
Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, “dirty reads”):

  T1: \( R(A), W(A), \) \( R(B), W(B), \) Abort
  T2: \( R(A), W(A), C \)

- Unrepeateable Reads (RW Conflicts):

  T1: \( R(A), R(A), W(A), C \)
  T2: \( R(A), W(A), C \)

Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

  T1: \( W(A), W(B), C \)
  T2: \( W(A), W(B), C \)
Example:

T1: Read(A)   T2: Read(A)
    A ← A+100     A ← A×2
    Write(A)      Write(A)
    Read(B)      Read(B)
    B ← B+100     B ← B×2
    Write(B)      Write(B)

Constraint: A=B

Schedule A

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read(A); A ← A+100</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>Write(A);</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Read(B); B ← B+100;</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(A); A ← A×2;</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>Write(A);</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>Read(B); B ← B×2;</td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td>250</td>
</tr>
<tr>
<td></td>
<td></td>
<td>250</td>
</tr>
</tbody>
</table>

Chris Clifton - CS34800
### Schedule B

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>T1</td>
<td>Read(A); A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>← A×2;</td>
<td></td>
<td>50</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write(A);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(B); B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>← B×2;</td>
<td></td>
<td>50</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(A); A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>← A+100</td>
<td></td>
<td>150</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write(A);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(B); B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>← B+100;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td></td>
<td>150</td>
<td>150</td>
</tr>
</tbody>
</table>

Chris Clifton - CS34800

### Schedule C

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>T1</td>
<td>Read(A); A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>← A+100</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write(A);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(A); A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>← A×2;</td>
<td></td>
<td>250</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write(A);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(B); B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>← B+100;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td></td>
<td>125</td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>Read(B); B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>← B×2;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td></td>
<td>250</td>
<td>250</td>
</tr>
</tbody>
</table>

Chris Clifton - CS34800
### Schedule D

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A); A ← A+100</td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A×2;</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B×2;</td>
<td></td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td></td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Chris Clifton - CS34800**

---

### Schedule E

Same as Schedule D but with new T2'.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2'</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A); A ← A+100</td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A×1;</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B×1;</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td></td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Chris Clifton - CS34800**

---

© 2016 Christopher W. Clifton
Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
  - Deadlock prevention
  - Deadlock detection

Logging and Recovery

- The following actions are recorded in the log:
  - *Ti writes an object*: the old value and the new value.
    - Log record must go to disk *before* the changed page!
  - *Ti commits/aborts*: a log record indicating this action.
- Log records are chained together by Xact id, so it’s easy to undo a specific Xact.
- Log is often *duplexed* and *archived* on stable storage.
- All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.
Recovering From a Crash

There are 3 phases in the *Aries* recovery algorithm:

- **Analysis:** Scan the log forward (from the most recent checkpoint) to identify all Xacts that were active, and all dirty pages in the buffer pool at the time of the crash.
- **Redo:** Redoes all updates to dirty pages in the buffer pool, as needed, to ensure that all logged updates are in fact carried out and written to disk.
- **Undo:** The writes of all Xacts that were active at the crash are undone (by restoring the *before value* of the update, which is in the log record for the update), working backwards in the log. (Some care must be taken to handle the case of a crash occurring during the recovery process!)

---

**Transaction State**

- **Active** – the initial state; the transaction stays in this state while it is executing.
- **Partially committed** – after the final statement has been executed.
- **Failed** – after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - Restart the transaction
    - can be done only if no internal logical error
  - Kill the transaction
- **Committed** – after successful completion.
Transaction State (Cont.)

- Active
  - Partially committed
  - Failed
- Committed
- Aborted