**Transaction Management:**

Integrity (... and how to maintain it)

- Data in the DB should be “correct” at all times
- Many different kinds of integrity constraints (ICs):
  - primary key constraint: there is at most one tuple for any given key value
  - foreign key constraint: referenced parent tuple exists
  - functional dependencies: referenced tuple is unique
  - domain/range constraints: dom(R,A) = integer
  - application-specific constraints: no employee should make more than his boss
- Specification of ICs: logic formulas, rules, query expressions, ...
- Checking and Enforcement of ICs in SQL: domain constraints, key constraints, assertions, triggers

**Integrity of Databases**

- **Def.** A database instance D is consistent if D satisfies all given ICs.
- While the database changes from D to D’ it may be temporarily inconsistent: E.g., withdraw $100 from checking:
  - checking.balance -= 100; checking.daily_withdraw += 100
- **Def.** A transaction is a collection of actions that preserve consistency.

\[ D \xrightarrow{T} D’ \]

consist\text{ent} \quad \text{consistent}

=⇒ transactions (and the user) “see” only consistent states!

**Reasons for Integrity Violations**

- Reasons for IC violations:
  - Failures (Hardware, Software Bugs)
  - Concurrency / Data Sharing
- **Failure model** here (= undesired, expected events):
  - System crash: main memory is lost, cpu halts, resets.
  - Not covered (when the failure model fails, i.e., undesired, unexpected events):
    - disk data is lost
    - memory loss, cpu happily goes on
    - worse things (…)
  => use redundancy, backups, hardware checks (memory, CPU), ...
- For our failure model:
  - System crash =⇒ Recovery (now)
  - Data sharing =⇒ Concurrency Control (later)
Transaction Operations

- **INPUT(X)**
  - retrieve the block of item X from disk
- **OUTPUT(X)**
  - output the block of item X to disk
- **READ(X)**
  - retrieve item X in variable t. If X is not in memory, INPUT(X)
- **WRITE(X)**
  - write item X. Does not have to be pushed to disk unless buffer manager decides so
- Example: a transaction that doubles A and B
  - constraint: A=B
  
  ```
  READ(A,t) ; t := t * 2
  WRITE(A,t) ;
  READ(B,t) ; t := t * 2
  WRITE(B,t) ;
  OUTPUT(A);
  OUTPUT(B);
  ```

Key Problem: Unfinished Transaction

| T1: | Read (A); t := t2
|     | Write (A); t := t2
|     | Read (B); t := t2
|     | Write (B);
|     | Output (A);
|     | Output (B); System crash!

```

<table>
<thead>
<tr>
<th>Memory</th>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: 8 =&gt; 16</td>
<td>A: 8 =&gt; 16</td>
</tr>
<tr>
<td>B: 8 =&gt; 16</td>
<td>B: 8</td>
</tr>
</tbody>
</table>
```

Undo Logging (Immediate Update)

```

<table>
<thead>
<tr>
<th>Memory</th>
<th>Disk</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: 8 =&gt; 16</td>
<td>A: 8 =&gt; 16</td>
<td>&lt;T1, Start&gt;</td>
</tr>
<tr>
<td>B: 8 =&gt; 16</td>
<td>B: 8</td>
<td>&lt;T1, A, 8&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;T1, B, 8&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;T2, COMMIT&gt;</td>
</tr>
</tbody>
</table>
```

- if system crashes now (T1 has not committed)
- => use the log to restore the previous consistent state
Undo Logging Rules

- for every action, generate undo log record with old value
- before T changes X on disk, write the log record to disk (WAL: write ahead logging)
- before writing <T, COMMIT> to disk log, write all changes of T to disk

<table>
<thead>
<tr>
<th>Memory</th>
<th>Disk</th>
<th>Undolog</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: 8 = 16</td>
<td>A: 8 = 16</td>
<td>&lt;TI, Start&gt;</td>
</tr>
<tr>
<td>B: 8 = 8</td>
<td>B: 8</td>
<td>&lt;TI, A, 8&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;TI, B, 8&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;TI, COMMIT&gt;</td>
</tr>
</tbody>
</table>

Undo Logging Recovery Rules

- S = {T | <T, START> in log, <T, COMMIT> (or <T, ABORT>) NOT in log}
- For each <T, X, v> in log in reverse order do
  - if T in S then Write(X,v)
  - Output(X)
- For each T in S do
  - write <T, ABORT> to log

Q: What if failure during recovery?

Undo Logging vs. Redo Logging

- **Undo logging:**
  - immediate update: transaction has to write all data before commit
  - undoes effect of incomplete transactions
  - write data; then write COMMIT
  - cannot bring backup DB copies up to date

- **Redo logging:**
  - deferred update: changes are not immediately written to disk
  - redoes effects of committed transactions (ignores incomplete transactions)
  - write COMMIT; then write data
  - need to keep all modified blocks in memory until commit
Redo Logging (Deferred Update)

- for every action, generate redo log record with **new value**
- **before** $X$ is modified on disk, all log records of $T$ for $X$
  (including commit) must be on disk

<table>
<thead>
<tr>
<th>Memory</th>
<th>Redolog</th>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: 8 $\rightarrow$ 16</td>
<td>$&lt;T1, Start&gt;$</td>
<td>A: 8 $\rightarrow$ 16</td>
</tr>
<tr>
<td>B: 8 $\rightarrow$ 12</td>
<td>$&lt;T1, A, 16&gt;$</td>
<td>B: 8</td>
</tr>
<tr>
<td></td>
<td>$&lt;T1, B, 16&gt;$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$&lt;T1, COMMIT&gt;$</td>
<td></td>
</tr>
</tbody>
</table>

Redo Logging Recovery Rules

- $S = \{T | <T, COMMIT> \in \text{log}\}$
- For each $<T, X, v> \in \text{log}$ **in forward order** do
  - if $T \in S$ then WRITE($X$, $v$) ; OUTPUT($X$)
- **Problem:** Recovery is **very slow**!
  - have to redo all transactions that ever run on the system
- **Checkpoints:** periodically do:
  - stop accepting transactions and finish all running ones
  - flush all log records to disk and all buffers to the DB
  - write $<\text{CHECKPOINT}>$ record on log
- upon recovery search until the first checkpoint

Buffering Requirements

- program requests WRITE($X$) = $\rightarrow$ new value for $X$ is placed in the main memory (buffer) copy of $X$
- the program or the buffer manager can (but are not required to) force $X$ to be output to disk using OUTPUT($X$)
- undo logging and redo logging **limit** the above possibilities:
  - **Undo** requires that all items $X$ written by a transaction are copied back to disk before the commit entry appears in log
  - **Redo** logging requires that no item $X$ is stored on disk before the log entries (including the commit entry) are on disk
  - $\Rightarrow$ undo/redo logging
## Undo/Redo Logging

- Undo/Redo logging does not have the buffer (memory) management restrictions of undo or redo.
- **Undo/Redo logging rules:**
  - For every write action generate a log record \( <T,X, \text{old}(X), \text{new}(X)> \).
  - This must appear in the disk log before the data update appears on disk.
- But now:
  - OUTPUT(X) can be flashed before or after T commits.
- **Undo/Redo recovery:**
  - backwards pass: construct set \( S \) of committed transactions; undo actions of non-committed transactions.
  - forward pass: redo actions of \( S \) transactions.

## Undoing Real World Actions (NOT)

**Problem:**
- "Output" commands to other media can not be undone.
  - Show output to user.
  - Dispense cash at ATM.

**Solution:**
- Execute real world actions only after commit.