Integrity or correctness of data

- Would like data to be "accurate" or "correct" at all times

<table>
<thead>
<tr>
<th>EMP</th>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>White</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>Green</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>Gray</td>
<td>21</td>
</tr>
</tbody>
</table>

Integrity or consistency constraints

- Predicates data must satisfy
- Examples:
  - x is key of relation R
  - x → y holds in R
  - Domain(x) = {Red, Blue, Green}
  - no employee should make more than twice the average salary
  - Application business logic generally creates pre-post transaction constraints on database
Simplified Definition:

- **Consistent state**: satisfies all constraints
- **Consistent DB**: DB in consistent state

Transaction: collection of actions that preserve consistency

![Diagram: Consistent DB → T → Consistent DB']

Big assumption:

If T starts with consistent state +
  T executes in isolation
  \( \Rightarrow \) T leaves consistent state
Correctness (informally)

• If we stop running transactions, DB left consistent
• Each transaction sees a consistent DB

How can constraints be violated?

• Transaction bug
• DBMS bug
• Hardware failure
e.g., disk crash alters balance of account
• Data sharing
e.g.: T1: give 10% raise to programmers
    T2: change programmers ⇒ systems analysts

How can we prevent/fix violations?

• Chapter 8: due to failures only
• Chapter 9: due to data sharing only
• Chapter 10: due to failures and sharing
Will not consider:

- How to write correct transactions
- How to write correct DBMS
- Constraint checking & repair

That is, solutions studied here do not need to know constraints

Chapter 8  Recovery

- First order of business:
  Failure Model

Events — Desired
Undesired — Expected
Unexpected
Our failure model

Desired events: see product manuals....

Undesired expected events:
- System crash
  - memory lost
  - cpu halts, resets

that's it!!

Undesired Unexpected: Everything else!

Examples:
- Disk data is lost
- Memory lost without CPU halt
- CPU implodes wiping out universe....

Undesired Unexpected: Everything else!
Is this model reasonable?

**Approach:** Add low level checks + redundancy to increase probability model holds

E.g., | Replicate disk storage (stable store)
| Memory parity
| CPU checks

---

**Second order of business:**

Storage hierarchy

![Diagram of storage hierarchy](image)

**Operations:**

- Input (x): block with x → memory
- Output (x): block with x → disk
- Read (x,t): do input(x) if necessary t ← value of x in block
- Write (x,t): do input(x) if necessary value of x in block ← t
Key problem: Unfinished transaction

Example: Constraint: \(A = B\)

\[\begin{align*}
T_1: & \quad A \leftarrow A \times 2 \\
& \quad B \leftarrow B \times 2
\end{align*}\]

\[\begin{align*}
T_1: & \quad \text{Read} (A, t); \quad t \leftarrow t \times 2 \\
& \quad \text{Write} (A, t); \\
& \quad \text{Read} (B, t); \quad t \leftarrow t \times 2 \\
& \quad \text{Write} (B, t); \\
& \quad \text{Output} (A); \\
& \quad \underline{\text{Output} (B); \\
& \quad \text{failure!}
\end{align*}\]

A: \(\_16\)  \\
B: \(\_16\)

\begin{tabular}{ll}
memory & disk \\
A: \(\_16\) & B: \(\_16\)  \\
A: \(\_16\) & B: \(8\)
\end{tabular}

• Need atomicity: execute all actions of a transaction or none at all
One solution: undo logging (immediate modification)

due to: Hansel and Gretel, 782 AD

Undo logging (Immediate modification)

T1: Read (A,t); t ← t×2 A=B
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Output (A);
Output (B);

One “complication”

• Log is first written in memory
• Not written to disk on every action
Two “complications”

- Log is first written in memory
- Not written to disk on every action

Undo logging rules

1. For every action generate undo log record (containing old value)
2. Before x is modified on disk, log records pertaining to x must be on disk (write ahead logging: WAL)
3. Before commit is flushed to log, all writes of transaction must be reflected on disk

Recovery rules, Take One:
Undo logging

- For every Ti with <Ti, start> in log:
  - If <Ti, commit> or <Ti, abort> in log, do nothing
  - Else for all <Ti, X, v> in log:
    - write (X, v)
    - output (X)
    - Write <Ti, abort> to log

IS THIS CORRECT??
Recovery rules: Undo logging

(1) Let S = set of transactions with 
   <Ti, start> in log, but no 
   <Ti, commit> (or <Ti, abort>) record in log
(2) For each <Ti, X, v> in log, 
   in reverse order (latest → earliest) do:
   - if Ti ∈ S then
     - write (X, v)
     - output (X)
(3) For each Ti ∈ S do
   - write <Ti, abort> to log

What if failure during recovery?
No problem! Undo idempotent

Redo logging (deferred modification)

T1: Read(A,t); t ← t×2; write (A,t);
    Read(B,t); t ← t×2; write (B,t);
    Output(A); Output(B)
Redo logging rules

(1) For every action, generate redo log record (containing new value)

(2) Before X is modified on disk (DB), all log records for transaction that modified X (including commit) must be on disk

(3) Flush log at commit

Recovery rules: Redo logging

• For every Ti with <Ti, commit> in log:
  – For all <Ti, X, v> in log:
    - Write(X, v)
    - Output(X)

**IS THIS CORRECT??

Recovery rules: Redo logging

(1) Let S = set of transactions with <Ti, commit> in log

(2) For each <Ti, X, v> in log, in forward order (earliest → latest) do:
  - if Ti ∈ S then Write(X, v)
  - Output(X) —— optional
Recovery is very, very SLOW!

Redo log:

First Record
(1 year ago)  T1 wrote A,B
Committed a year ago

Last Record

Crash

--> STILL, Need to redo after crash!!

Solution: Checkpoint (simple version)

Periodically:
(1) Do not accept new transactions
(2) Wait until all transactions finish
(3) Flush all log records to disk (log)
(4) Flush all buffers to disk (DB) (do not discard buffers)
(5) Write "checkpoint" record on disk (log)
(6) Resume transaction processing

Example: what to do at recovery?

Redo log (disk):

<table>
<thead>
<tr>
<th>...</th>
<th>&lt;T1,A,16&gt;</th>
<th>...</th>
<th>&lt;T1,commit&gt;</th>
<th>...</th>
<th>&lt;T2,B,17&gt;</th>
<th>&lt;T2,commit&gt;</th>
<th>...</th>
<th>&lt;T3,C,21&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Crash
Key drawbacks:

- **Undo logging:**
  cannot bring backup DB copies up to date,
  real writes at end of transaction needed

- **Redo logging:**
  need to keep all modified blocks in memory
  until commit

**Solution:** undo/redo logging!

Update $\Rightarrow$ <Ti, Xid, New X val, Old X val>
page X

**Rules**

- Page X can be flushed before or
  after Ti commit
- Log record flushed before
  corresponding updated page (WAL)
- Flush at commit (log only)
Non-quiesce checkpoint

Examples what to do at recovery time?

Example

[Space for text]

[Space for text]

[Space for text]
Recovery process:

- **Backwards pass** (end of log \(\rightarrow\) latest checkpoint start)
  - construct set \(S\) of committed transactions
  - undo actions of transactions not in \(S\)
- **Undo pending transactions**
  - follow undo chains for transactions in (checkpoint active list) \(\rightarrow\) \(S\)
- **Forward pass** (latest checkpoint start \(\rightarrow\) end of log)
  - redo actions of \(S\) transactions

Real world actions

E.g., dispense cash at ATM

\(T_i = a_1 \; a_2 \; \ldots \; a_{j} \; \ldots \; a_n\)

\(\downarrow\)

\$(

Solution

(1) execute real-world actions after commit
(2) try to make idempotent
ATM

Give$(amt, Tid, time)$

Media failure (loss of non-volatile storage)

Solution: Make copies of data!

Example 1 Triple modular redundancy

- Keep 3 copies on separate disks
- Output$(X)$ --> three outputs
- Input$(X)$ --> three inputs + vote
Example #2  Redundant writes, Single reads

- Keep N copies on separate disks
- Output(X) --> N outputs
- Input(X) --> Input one copy
  \begin{itemize}
  \item if ok, done
  \item else try another one
  \end{itemize}
\Rightarrow\text{Assumes bad data can be detected}

Example #3: DB Dump + Log

- If active database is lost,
  \begin{itemize}
  \item restore active database from backup
  \item bring up-to-date using redo entries in log
  \end{itemize}

When can log be discarded?

- not needed for media recovery
- not needed for undo after system failure
- not needed for redo after system failure
Summary

- Consistency of data
- One source of problems: failures
  - Logging
  - Redundancy
- Another source of problems:
  Data Sharing..... next