Integrity or correctness of data

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

**EMP**

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>White</td>
<td>52</td>
</tr>
<tr>
<td>Green</td>
<td>3421</td>
</tr>
<tr>
<td>Gray</td>
<td>1</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}
  - α is valid index for attribute x of R
  - no employee should make more than twice the average salary
Definition:

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

Constraints (as we use here) may **not** capture “full correctness”

Example 1: Transaction constraints

- When salary is updated, new salary > old salary
- When account record is deleted, balance = 0

**Note**: could be “emulated” by simple constraints, e.g.,

<table>
<thead>
<tr>
<th>account</th>
<th>Acct #</th>
<th>balance</th>
<th>deleted</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Constraints** (as we use here) may **not** capture “full correctness”

**Example 2** Database should reflect real world

\[ \begin{array}{c}
\text{DB} \\
\text{Reality}
\end{array} \]

In any case, continue with constraints...

**Observation:** DB cannot be consistent always!

**Example:** \( a_1 + a_2 + \ldots + a_n = TOT \) (constraint)

Deposit $100 in \( a_2 \): \[
\begin{align*}
a_2 & \leftarrow a_2 + 100 \\
TOT & \leftarrow TOT + 100
\end{align*}
\]

**Example:** \( a_1 + a_2 + \ldots + a_n = TOT \) (constraint)

Deposit $100 in \( a_2 \): \[
\begin{align*}
a_2 & \leftarrow a_2 + 100 \\
TOT & \leftarrow TOT + 100
\end{align*}
\]

<table>
<thead>
<tr>
<th>a_2</th>
<th>50</th>
<th>150</th>
<th>150</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOT</td>
<td>1000</td>
<td>1000</td>
<td>1100</td>
</tr>
</tbody>
</table>
Transaction: a collection of actions that preserve consistency

Big assumption:
If T starts with consistent state +
T executes in isolation
⇒ 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

CPU — processor
memory — M
  disk — D
Desired events: see product manuals....

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

that's it!!

Undesired Unexpected: Everything else!

Undesired Unexpected: Everything else!
Examples:
- Disk data is lost
- Memory lost without CPU halt
- CPU implodes wiping out universe....

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]

Operations:
- Input (x): block with x → memory
- Output (x): block with x → disk
- Read (x,t): do input(x) if necessary
  \[ t \leftarrow \text{value of } x \text{ in block} \]
- Write (x,t): do input(x) if necessary
  \[ \text{value of } x \text{ in block} \leftarrow t \]

Key problem: Unfinished transaction

Example: Constraint: A=B
- T1: \[ A \leftarrow A \times 2 \]
- \[ B \leftarrow B \times 2 \]
T1: Read (A,t); \( t \leftarrow t \times 2 \)
Write (A,t);
Read (B,t); \( t \leftarrow t \times 2 \)
Write (B,t);
Output (A);
Output (B); failure!

- 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

- Improved in 784 AD to durable undo logging
undo logging (immediate modification)

\[ T_1: \text{Read}(A,t); \quad t \leftarrow t \times 2 \quad A = B \]
Write \((A,t)\);
Read \((B,t)\);  \( t \leftarrow t \times 2 \)
Write \((B,t)\);
Output \((A)\);
Output \((B)\);

---

one “complication”

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

---

One “complication”

- 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: 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

To discuss:
• Redo logging
• Undo/redo logging, why both?
• Real world actions
• Checkpoints
• Media failures

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)

A: 8 16
B: 8 16
memory

A: 8 16
B: 8
DB

LOG

<T1, start>
<T1, A, 16>
<T1, B, 16>
<T1, commit>

Output
16
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:

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

First Record
(1 year ago)

T1 wrote A, B
Committed a year ago

Last Record

→ 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):

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>&lt;T1,A&gt;</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>&lt;T1,commit&gt;</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>&lt;T2,A&gt;</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>&lt;T2,commit&gt;</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Crash</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```
Key drawbacks:

- **Undo logging**: cannot bring backup DB copies up to date
- **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

LOG

Start-ckpt
active TR: T1, T2, ...
end ckpt ...

for undo
dirty buffer
pool pages
flushed

Examples

what to do at recovery time?

LOG

no T1 commit

T1, ...

T1-

a

Ckpt

T1

Ckpt-

T1-

end

b

Undo T1 (undo a,b)

Example

LOG

T1, ...

dkpt-s

T1-

ckpt-

cmt

T1-

Redo T1: (redo b,c)
Recovery process:
- Backwards pass (end of log ➜ 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) - S
- Forward pass (latest checkpoint start ➜ end of log)
  - redo actions of S transactions

Real world actions
E.g., dispense cash at ATM
\[ T_i = a_1 a_2 \ldots a_j \ldots a_n \]
\[ s \]

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

Give$(amt, Tid, time)$

lastTid: 

time: 

$\downarrow$
give$(amt)$

$\downarrow$

Media failure (loss of non-volatile storage)

A: 16

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
  - if ok, done
  - else try another one
- Assumes bad data can be detected

Example #3: DB Dump + Log

- If active database is lost,
  - restore active database from backup
  - bring up-to-date using redo entries in log

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