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>3421</td>
</tr>
<tr>
<td></td>
<td>Blue</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 (functional dependency)
  - 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.,

| account | Acct # | .... | balance | deleted? |
Constraints (as we use here) may not capture “full correctness”

Example 2  Database should reflect real world
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 \( a_2 \):

\[
\begin{align*}
\text{a}_2 & \leftarrow \text{a}_2 + 100 \\
\text{TOT} & \leftarrow \text{TOT} + 100
\end{align*}
\]
Example: \( a_1 + a_2 + \ldots + a_n = TOT \) (constraint)

Deposit $100 in \( a_2 \): \( a_2 \leftarrow a_2 + 100 \)
\( TOT \leftarrow TOT + 100 \)
Transaction: collection of actions that preserve consistency
Assumption:

If T starts with DB in consistent state +
   T executes in isolation
⇒ T leaves DB in 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 $\implies$ systems analysts
How can we **prevent/fix** violations?

- Due to failures **only**
- Due to data sharing **only**
- 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
Recovery

• First order of business: Failure Model
Events — Desired

Undesired — Expected

Unexpected
Our failure model

- CPU
- Memory (M)
- Disk (D)

Processor

memory ———— disk
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:

• Software bugs
• 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 the probability that model holds

E.g.,

- Replicate disk storage (stable store)
- Memory parity
- CPU checks
Second order of business:

Storage hierarchy

Memory                  Disk

Diagram of storage hierarchy with Memory and Disk connected by an arrow.
Operations:

- **Input (x):** block containing x → memory
- **Output (x):** block containing x → disk
- **Read (x,t):** do input(x) if necessary
  
  \[ t \leftarrow \text{value of x in block} \]
- **Write (x,t):** do input(x) if necessary
  
  \[ \text{value of x in block} \leftarrow t \]
**Key problem**  Unfinished transaction

**Example**

**Constraint:** $A = B$

$T_1$: $A \leftarrow A \times 2$

$B \leftarrow B \times 2$
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);

\[\text{failure!}\]

memory

disk
• 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);

memory

 disk

log

<T₁, start>
<T₁, A, 8>
<T₁, B, 8>
<T₁, commit>
One “complication”

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

**memory**

A: 8 16
B: 8 16
Log:
<T₁,start>
<T₁, A, 8>
<T₁, B, 8>

**DB**

A: 8
B: 8

**BAD STATE**

# 1

**Log**
One “complication”

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

<table>
<thead>
<tr>
<th>A:</th>
<th>B:</th>
<th>Log:</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>16</td>
<td>&lt;T₁,start&gt;</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>&lt;T₁, A, 8&gt;</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>&lt;T₁, B, 8&gt;</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>&lt;T₁, commit&gt;</td>
</tr>
</tbody>
</table>

DB

BAD STATE

# 2

memory

Log

A: 8 16
B: 8
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 for Undo logging

• For every Ti with <Ti, start> in log:
  - Either: Ti completed
    <Ti,commit> or <Ti,abort> in log
  - Or: Ti is incomplete

Undo incomplete transactions
Recovery rules for Undo Logging (contd.)

1. Let \( S = \) set of transactions with \(< Ti, \text{start}>\) in log, but no \(< Ti, \text{commit}>\) or \(< Ti, \text{abort}>\) record in log

2. For each \(< Ti, X, v>\) in log, in reverse order (latest \(\rightarrow\) earliest) do:
   - if \( Ti \in S \) then
     - write \((X, v)\)
     - output \((X)\)

3. For each \( Ti \in S \) do
   - write \(< Ti, \text{abort}>\) to log
What if failure during recovery?

No problem: Undo is idempotent
To discuss:

- Redo logging
- Undo/redo logging, why both?
- Real world actions
- Checkpoints
- Media failures
Redo logging (deferred modification)

\( T_1: \) 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)
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:
    \[
    \begin{align*}
    &\text{Write}(X, v) \\
    &\text{Output}(X)
    \end{align*}
    \]

⚠️ IS THIS CORRECT??
Recovery rules: Redo logging

(1) Let $S$ = set of transactions with $<Ti, \text{commit}>$ in log

(2) For each $<Ti, X, v>$ in log, in forward order (earliest $\rightarrow$ latest) do:
   - if $Ti \in S$ then
     \[
     \begin{cases} 
     \text{Write}(X, v) \\
     \text{Output}(X) \quad \text{optional} 
     \end{cases}
     \]
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 $\Rightarrow \langle Ti, Xid, New\ X\ val,\ Old\ X\ val\rangle$

page X
Rules

• Page X can be flushed before or after Ti commit
• Log record flushed before corresponding updated page (WAL)
Recovery Rules

- Identify transactions that committed
- Undo uncommitted transactions
- Redo committed transactions
Recovery is very, very **SLOW**!  

Redo log:

---

First Record (1 year ago)

T1 wrote A,B

Committed a year ago

---

\[\text{\ldots} \quad \text{\ldots} \quad \text{\ldots} \quad \text{\ldots} \]

Last Record

Crash

\[\text{\ldots} \quad \text{\ldots} \quad \text{\ldots} \quad \text{\ldots} \]

\[\Rightarrow \text{STILL, Need to redo after crash!!} \]
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):

| ... | \langle T1, A, 16 \rangle | ... | \langle T1, \text{commit} \rangle | ... | \text{Checkpoint} | ... | \langle T2, B, 17 \rangle | ... | \langle T2, \text{commit} \rangle | ... | \langle T3, C, 21 \rangle | ... | \text{Crash} |
|------|-------------------|------|------------------|------|-------------------|------|-------------------|------|-------------------|------|-------------------|

System stops accepting new transactions
Non-quiescent checkpoint for Undo/Redo logging

LOG

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

... end ckpt ...

for undo dirty buffer pool pages flushed
Example: Undo/Redo + Non Quiescent Chkpt.

<start T1>
<T1,A,4,5>
<start T2>
<commit T1>
<T2,B,9,10>
<start chkpt(T2)>
<T2,C,14,15>
<start T3>
<T3,D,19,20>
<end checkpt>
<commit T2>
<commit T3>

1. Flush log
2. Flush all dirty buffers. May start new transactions
3. Write <end checkpt>. Flush log
Examples

what to do at recovery time?

LOG

... | $T_1,-a$ | ... | Ckpt $T_1$ | ... | Ckpt end | ... | $T_1,-b$

no $T_1$ commit

✗ Undo $T_1$ (undo $a,b$)
Example

Example Log:

- **Redo T1**: (redo b, c)
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) - \( S \)
- **Forward pass** (latest checkpoint start \( \Rightarrow \) end of log)
  - redo actions of \( S \) transactions
Example: Redo + Non Quiescent Chkpt.

<start T1>
<T1,A,5>
<start T2>
<commit T1>
<T2,B,10>
<start chkpt(T2)>
<T2,C,15>
<start T3>
<T3,D,20>
<end chkpt>
<commit T2>
<commit T3>

1. Flush log
2. Flush data elements written by transactions that committed before <start chkpt>.
   May start new transactions.
3. Write <end chkpt>. Flush log
Example: Undo + Non Quiescent Chkpt.

<start T1>
<T1,A,5>
<start T2>
<T2,B,10>
<start chkpt(T1,T2)>
<T2,C,15>
<start T3>
<T1,D,20>
<commit T1>
<T3,E,25>
<commit T2>
<end checkpt>
<T3,F,30>

1. Flush log
2. \textit{Wait} for active transactions to complete. New transactions may start
3. Write \texttt{<end checkpt>}. Flush log
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
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
  - if ok, done
  - else try another one
leftrightarrow 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
Non-quiescent Archiving

- Log may look like:
  
  <start dump>
  <start checkpt(T1,T2)>
  <T1,A,1,3>
  <T2,C,3,6>
  <commit T2>
  <end checkpt>
  Dump completes
  <end dump>
When can log be discarded?

<table>
<thead>
<tr>
<th>log</th>
<th>db dump</th>
<th>last needed undo</th>
<th>checkpoint</th>
</tr>
</thead>
</table>

- 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