Transactions and Failure Recovery

Instructor: Matei Zaharia
Spark SQL Wrap-Up

SQL

Data Frames

Logical Plan

Optimizer

Physical Plan

Code Generator

RDDs

Catalog

Data Source API

HDFS

cassandra

HBASE

elasticsearch

PostgreSQL

HIVE

...
Data Sources

Uniform way to access structured data

» Apps can migrate across Hive, Cassandra, JSON, Parquet, …

» Rich semantics allows query pushdown into data sources

users[users.age > 20]

select * from users
Extensible Optimizer

Uses Scala pattern matching (see demo!)
Spark Usage Today

Languages Used in Databricks Notebooks

- Python 68%
- SQL 18%
- Scala 11%
- R 3%

>90% of API calls run via Spark SQL engine
Outline

Defining correctness
Transaction model
Hardware failures
Recovery with logs
Outline

Defining correctness

Transaction model

Hardware failures

Recovery with logs
Focus of This Part of Course

Correctness in case of failures & concurrency

» There’s no point running queries quickly if the input data is wrong!
Correctness of Data

Would like all data in our system to be “accurate” or “correct” at all times
  » Both logical data model and physical structs

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>52</td>
</tr>
<tr>
<td>Green</td>
<td>3421</td>
</tr>
<tr>
<td>Chen</td>
<td>1</td>
</tr>
</tbody>
</table>
Idea: Integrity or Consistency Constraints

Predicates that data structures must satisfy

Examples:

» X is an attribute of relation R
» Domain(X) = \{student, prof, staff\}
» If X = prof in a record then office != NULL in it
» T is valid B-tree index for attribute X of R
» No staff member 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 All Issues

Example 1: transaction constraints

When a salary is updated,
    new salary > old salary

When an account record is deleted,
    balance = 0
Constraints (As We Use Here) May Not Capture All Issues

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

| account | acct # | ... | balance | is_deleted |
Constraints (As We Use Here) May Not Capture All Issues

Example 2: database should reflect real world
Constraints (As We Use Here) May Not Capture All Issues

Example 2: database should reflect real world
In Any Case, Continue with Constraints...
In Any Case, Continue with Constraints...

Observation: DB can’t always be consistent!

Example: $a_1 + a_2 + \ldots + a_n = \text{TOT}$ (constraint)

Deposit $100$ in $a_2$:

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

Deposit $100$ in $a_2$:  
\[
a_2 \leftarrow a_2 + 100
\]
\[
\text{TOT} \leftarrow \text{TOT} + 100
\]

<table>
<thead>
<tr>
<th></th>
<th>$a_2$</th>
<th></th>
<th>TOT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>50</td>
<td></td>
<td>1000</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c|c|c}
   & \text{new} & \text{old} \\
\hline
 a_2 & 50 & \ldots \\
\text{TOT} & 1000 & 1000 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
   & \text{new} & \text{old} \\
\hline
 a_2 & 150 & 50 \\
\text{TOT} & 1000 & 1000 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
   & \text{new} & \text{old} \\
\hline
 a_2 & 150 & 50 \\
\text{TOT} & 1100 & 1000 \\
\end{array}
\]
Transaction: Collection of Actions that Preserve Consistency
Big Assumption:

If $T$ starts with a consistent state

$+ \quad T$ executes in isolation

$\Rightarrow \quad T$ leaves a consistent state
Correctness (Informally)

If we stop running transactions, the database is left consistent

Each transaction sees a consistent DB
More Detail: Transaction API

Client App

Start Transaction
Read
Write
Commit

DB
More Detail: Transaction API

Client App

DB

Both clients and system can abort transactions
How Can Constraints Be Violated?

Transaction bug

DBMS bug

Hardware failure
  » e.g., disk crash alters balance of account

Data sharing
  » e.g.: $T_1$: give 10% raise to programmers,
    $T_2$: change programmers $\Rightarrow$ marketers
We Won’t Consider:

How to write correct transactions

How to check for DBMS bugs

Constraint verification & repair

» That is, the solutions we’ll study do not need to know the constraints!
Failure Recovery

First order of business: Failure Model
Failure Models

- Events
  - Desired
    - Undesired
      - Expected
      - Unexpected

Our Failure Model

processor

memory

CPU

M

D

disk
Our Failure Model

**Desired Events:** see product manuals….

**Undesired Expected Events:**
- System crash ("fail-stop failure")
  - CPU halts, resets
  - Memory lost

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 the universe....
Is This Model Reasonable?

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

E.g.,

- Replicate disk storage (stable store)
- Memory parity
- CPU checks
Second Order of Business:

Storage hierarchy

Memory  Disk
Operations

Input(x): block containing x → memory

Output(x): block containing 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: $A = B$

T1: $A \leftarrow A \times 2$

$B \leftarrow B \times 2$
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
B: 8

memory

A: 8
B: 8
disk
T₁: Read (A,t); t ← t×2
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Output (A);
Output (B);

memory

A: 8
B: 8

disk

A: 8
B: 8
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!

memory

A: 8 16
B: 8 16

disk

A: 8 16
B: 8
Need: Atomicity

Execute all the actions in a transaction together, or none at all
One Solution

Undo logging (immediate modification)

Due to: Hansel and Gretel, 1812 AD

Updated to durable undo logging in 1813 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);

A:8
B:8

memory

A:8
B:8

disk

log
Undo Logging (Immediate modification)

T₁: Read (A,t); \( t \leftarrow t \times 2 \)       \( A=B \)
    Write (A,t);
    Read (B,t); \( t \leftarrow t \times 2 \)
    Write (B,t);
    Output (A);
    Output (B);

memory

\[
\begin{array}{ll}
A: 8 & 16 \\
B: 8 & 16 \\
\end{array}
\]

disk

log

\[
\begin{array}{ll}
A: 8 & \\
B: 8 & \\
\end{array}
\]

\[
\langle T₁, \text{start} \rangle \\
\langle T₁, A, 8 \rangle
\]
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);

<table>
<thead>
<tr>
<th></th>
<th>memory</th>
<th>disk</th>
<th>log</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>8</td>
<td>16</td>
<td>&lt;T1, start&gt;</td>
</tr>
<tr>
<td>B</td>
<td>8</td>
<td>16</td>
<td>&lt;T1, A, 8&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;T1, B, 8&gt;</td>
</tr>
</tbody>
</table>
Undo Logging  (Immediate modification)

T₁: Read (A,t);  \( t \leftarrow t \times 2 \)  \( \text{A=B} \)
Write (A,t);
Read (B,t);  \( t \leftarrow t \times 2 \)
Write (B,t);
Output (A);
Output (B);

<table>
<thead>
<tr>
<th>Memory</th>
<th>Disk</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: 8 16</td>
<td>A: 8 16</td>
<td>A: 8 16</td>
</tr>
<tr>
<td>B: 8 16</td>
<td>B: 8 16</td>
<td>B: 8 16</td>
</tr>
</tbody>
</table>

\(<T₁, \text{start}>\)
\(<T₁, \text{A, 8}>\)
\(<T₁, \text{B, 8}>\)
Undo Logging  (Immediate modification)

$T_1$:  Read $(A, t)$;  $t ← t \times 2$  \hspace{1cm} A=B

Write $(A, t)$;
Read $(B, t)$;  $t ← t \times 2$
Write $(B, t)$;
Output $(A)$;
Output $(B)$;

- Memory:
  - $A$: 8
  - $B$: 8

- Disk:
  - $A$: 16
  - $B$: 16

- Log:
  - $<T_1, \text{start}>$
  - $<T_1, A, 8>$
  - $<T_1, B, 8>$
  - $<T_1, \text{commit}>$
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
One “Complication”

Log is first written in memory

Not written to disk on every action

memory

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

DB

A: 8
B: 8

Log

BAD STATE
# 2

<T1, B, 8>
<T1, commit>
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 record is flushed to log, all writes of transaction must be on disk
Recovery Rules: Undo Logging

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

Can our writes of <Ti, abort> records be done in any order (in Step 3)?

» Example: T1 and T2 both write A
» T1 executed before T2
» T1 and T2 both rolled-back
» <T1, abort> written but NOT <T2, abort>? 
» <T2, abort> written but NOT <T1, abort>?
What If We Crash During Recovery?

No problem! → Undo is idempotent

(same effect if you do it twice)
Any Downsides to Undo Logging?
Any Downsides to Undo Logging?

Have to do a lot of I/O to commit (write all updated objects to disk first)

Hard to replicate database to another disk (must push all changes across the network)
To Discuss

Redo logging

Undo/redo logging
Redo Logging

First send Gretel up with no rope, then Hansel goes up safely with rope!
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 (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 (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  (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 (in DB), all log records for transaction that modified X (including commit) must be on disk

3. Flush log at commit

4. Write END record after DB updates are flushed to disk
Recovery Rules: Redo Logging

(1) Let \( S = \) set of transactions with \(<Ti, \text{commit}>\) and no \(<Ti, \text{end}>\) in log

(2) For each \(<Ti, X, v>\) in log, in forward order (earliest \(\rightarrow\) latest) do:
   - if \( Ti \in S \) then
     \[
     \begin{cases} 
     \text{Write}(X, v) \\
     \text{Output}(X) 
     \end{cases}
     \]

(3) For each \( Ti \in S \), write \(<Ti, \text{end}>\)
Combining <Ti, end> Records

Want to delay DB flushes for hot objects

Say X is branch balance:
T1: ... update X...
T2: ... update X...
T3: ... update X...
T4: ... update X...

Actions:
write X
output X
write X
output X
write X
output X
write X
output X
Combining \(<\text{Ti}, \text{end}>\) Records

Want to delay DB flushes for hot objects

Say X is branch balance:
T1: ... update X...
T2: ... update X...
T3: ... update X...
T4: ... update X...

Actions:
- write X
- output X
- write X
- output X
- write X
- output X
- write X
- output X

combined \(<\text{end}>\) record
Solution: Checkpoints

Simple, naïve checkpoint algorithm:
1. Stop accepting 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>Checkpoint</th>
<th>...</th>
<th>&lt;T2,B,17&gt;</th>
<th>...</th>
<th>&lt;T2,commit&gt;</th>
<th>...</th>
<th>&lt;T3,C,21&gt;</th>
<th>Crash</th>
</tr>
</thead>
</table>
Any Disadvantages to Redo Logging?
Any Disadvantages to Redo Logging?

Need to keep all modified blocks in memory until commit

» Might take up a lot of space, or waste time
Problems with Ideas So Far

**Undo logging:** need to wait for lots of I/O to commit; can’t easily have backup copies of DB

**Redo logging:** need to keep all modified blocks in memory until commit
Solution: Undo/Redo Logging!

Update = <Ti, X, new X val, old X val>

(X is the object updated)
Undo/Redo Logging Rules

Object X can be flushed **before or after** Ti commits

Log record (with undo/redo info) must be flushed before corresponding data (WAL)

Flush log up to commit record at Ti commit
Example: Undo/Redo Logging

What to Do at Recovery?

Undo/redo log (disk):

<table>
<thead>
<tr>
<th>...</th>
<th>&lt;checkpoint&gt;</th>
<th>...</th>
<th>&lt;T1, A, 10, 15&gt;</th>
<th>...</th>
<th>&lt;T1, B, 20, 23&gt;</th>
<th>...</th>
<th>&lt;T1, commit&gt;</th>
<th>...</th>
<th>&lt;T2, C, 30, 38&gt;</th>
<th>...</th>
<th>&lt;T2, D, 40, 41&gt;</th>
<th>...</th>
<th>Crash</th>
</tr>
</thead>
</table>

Crash
Example: Undo/Redo Logging

What to Do at Recovery?

Undo/redo log (disk):

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>&lt;checkpoint&gt;</td>
<td>&lt;T1, A, 10, 15&gt;</td>
<td>&lt;T1, B, 20, 23&gt;</td>
<td>&lt;T1, commit&gt;</td>
<td>...</td>
<td>&lt;T2, C, 30, 38&gt;</td>
<td>...</td>
<td>&lt;T2, D, 40, 41&gt;</td>
</tr>
</tbody>
</table>

T1 committed, so
REDO all its updates
Example: Undo/Redo Logging
What to Do at Recovery?

Undo/redo log (disk):

T1 committed, so REDO all its updates

T2 didn’t commit, so UNDO all its updates
Non-Quiescent Checkpoints

LOG

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

end ckpt

for dirty memory undo pages flushed
Non-Quiescent Checkpoints

checkpoint process:
for $i := 1$ to $M$ do
  output(buffer $i$)

[transactions run concurrently]
Example 1: How to Recover?

<table>
<thead>
<tr>
<th>LOG</th>
<th>...</th>
<th>T₁, a</th>
<th>...</th>
<th>Ckpt</th>
<th>T₁</th>
<th>Ckpt</th>
<th>end</th>
<th>...</th>
<th>T₁, b</th>
</tr>
</thead>
</table>

no T₁ commit
Example 1: How to Recover?

**LOG**

| ... | T₁₋a | ... | Ckpt T₁ | ... | Ckpt end | ... | T₁₋b |

- **Undo T₁** (undo a,b)
- **no T₁ commit**
Example 2: How to Recover?

<table>
<thead>
<tr>
<th>L</th>
<th>O</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>T₁</td>
<td>a</td>
</tr>
<tr>
<td>...</td>
<td>ckpt-s</td>
<td>T₁</td>
</tr>
<tr>
<td>...</td>
<td>T₁</td>
<td>b</td>
</tr>
<tr>
<td>...</td>
<td>ckpt-end</td>
<td>T₁</td>
</tr>
<tr>
<td>...</td>
<td>c</td>
<td>T₁</td>
</tr>
<tr>
<td>...</td>
<td>cmt</td>
<td>T₁</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Example 2: How to Recover?

Redo T1: (redo b, c)
What if a Checkpoint Does Not Complete?

| LOG    | ckpt start | ckpt end | T1 b | ckpt-start | T1 c | ...
|--------|------------|----------|------|------------|------|------

Start recovery from last complete checkpoint
Undo/Redo Recovery Process

Backward pass (end of log → latest valid 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’s active list) - S

Forward pass (latest checkpoint start → end of log)
  » redo actions of all transactions in S