superior to shadow copies in almost every way
Our goal is to build **reliable systems from unreliable components**. We want to build systems that serve many clients, store a lot of data, perform well, all while keeping availability high.

**transactions** — which provide **atomicity** and **isolation** — make it easier for us to reason about failures.

Our job in lecture is to understand how a system implements these two abstractions. How do our systems guarantee atomicity? How do they guarantee isolation?

**atomicity:** We have this working for one user and one file via *shadow copies*, but they perform poorly.

**isolation:** We don’t really have this yet. (Coarse-grained locks perform poorly; fine-grained locks are difficult to reason about.)
Our goal is to build **reliable systems from unreliable components**. We want to build systems that serve many clients, store a lot of data, perform well, all while keeping availability high.

**Transactions** — which provide **atomicity** and **isolation** — make it easier for us to reason about failures.

Our job in lecture is to understand how a system *implements* these two abstractions. How do our systems guarantee atomicity? How do they guarantee isolation?

**Atomicity**: **Logging**, which is going to provide us with much better performance at the cost of some added complexity.

**Isolation**: We don’t really have this yet (coarse-grained locks perform poorly; fine-grained locks are difficult to reason about).
begin    // T1
write(A, 100)
write(B, 50)
commit  // A=100; B=50

begin    // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit  // A=80; B=70

begin    // T3
write(A, read(A)+30)

\textbf{crash!} 

\textbf{problem:} after crash, A=\textbf{110}, but T3 never committed

we need a way to revert to A's previous committed value
begin  // T1
write(A, 100)
write(B, 50)
commit  // A=100; B=50

begin  // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit  // A=80; B=70

begin  // T3
write(A, read(A)+30)
let's try to read the value of A from this log

```python
read(log, var):
    commits = []
    // scan backwards
    for record r in log[len(log) - 1] .. log[0]:
        // keep track of commits
        if r.type == COMMIT:
            commits.add(r.tid)
        // find var’s last committed value
        elif r.type == UPDATE and
            r.tid in commits and r.var == var:
            return r.new_value
```

```
begin  // T1
    write(A, 100)
    write(B, 50)
commit  // A=100; B=50

begin  // T2
    write(A, read(A)-20)
    write(B, read(B)+20)
commit  // A=80; B=70

begin  // T3
    write(A, read(A)+30)
```

---

<table>
<thead>
<tr>
<th>TID</th>
<th>T1</th>
<th>T1</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>OLD</td>
<td>UPDATE</td>
<td>UPDATE</td>
<td>COMMIT</td>
</tr>
<tr>
<td>NEW</td>
<td>A=0</td>
<td>B=0</td>
<td></td>
</tr>
</tbody>
</table>

| OLD | A=100 | B=50 |

```
commits = []
```
let's try to read the value of $A$ from this log

```
read(log, var):
  commits = []
  // scan backwards
  for record r in log[len(log) - 1] .. log[0]:
    // keep track of commits
    if r.type == COMMIT:
      commits.add(r.tid)
    // find var's last committed value
    elif r.type == UPDATE and 
      r.tid in commits and r.var == var: 
      return r.new_value
```

commits = [T1]

```
begin // T1
write(A, 100)
write(B, 50)
commit  // A=100; B=50

begin // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit  // A=80; B=70

begin // T3
write(A, read(A)+30)
```
let's try to read the value of A from this log

```python
def read(log, var):
    commits = []
    # scan backwards
    for record r in log[len(log) - 1] .. log[0]:
        # keep track of commits
        if r.type == COMMIT:
            commits.add(r.tid)
        # find var’s last committed value
        elif r.type == UPDATE and r.tid in commits and r.var == var:
            return r.new_value

# commits = [T1]
```
 brief interlude: we’re going to change this example slightly, to illustrate one additional point

begin  // T1
write(A, 100)
write(B, 50)
commit  // A=100; B=50

begin  // T2
write(A, read(A)-20)
write(A, read(A)-30)
commit  // A=50; B=50

begin  // T3
write(A, read(A)+30)

let’s try to read the value of A from this log

read(log, var):
  commits = []
  // scan backwards
  for record r in log[len(log) - 1] .. log[0]:
    // keep track of commits
    if r.type == COMMIT:
      commits.add(r.tid)
    // find var’s last committed value
    elif r.type == UPDATE and
      (r.tid in commits or r.tid == current_tid)
      and r.var == var:
      return r.new_value


Now back to our original example.

```
begin  // T1
write(A, 100)
write(B, 50)
commit  // A=100; B=50

begin  // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit  // A=80; B=70

begin  // T3
write(A, read(A)+30)
crash! 🙁
```

Let's try to read the value of A from this log:

```python
def read(log, var):
    commits = []
    # scan backwards
    for record r in log[len(log) - 1] .. log[0]:
        # keep track of commits
        if r.type == COMMIT:
            commits.add(r.tid)
        # find var's last committed value
        elif r.type == UPDATE and
            (r.tid in commits or r.tid == current_tid)
            and r.var == var:
            return r.new_value
```

After a crash, the log is still correct; uncommitted updates will not be read.

Katrina LaCurts | lacurts@mit.edu | 6.033 2021
begin // T1
write(A, 100)
write(B, 50)
commit // A=100; B=50

begin // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit // A=80; B=70

begin // T3
write(A, read(A)+30)
crash!

problem: reads can be very slow
begin // T1
write(A, 100)
write(B, 50)
commit // A=100; B=50

begin // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit // A=80; B=70

begin // T3
write(A, read(A)+30)

read(var):
    return cell_read(var)

write(var, value):
    log.append(current_tid, “UPDATE”, var, read(var), value)
    cell_write(var, value)
BEGIN // T1
write(A, 100)
write(B, 50)
commit
// A=100; B=50

BEGIN // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit
// A=80; B=70

BEGIN // T3
write(A, read(A)+30)
crash!

CELL STORAGE

TID | T1 | T1 | T1 | T2 | T2 | T2 | T3
OLD | A=0 | B=0 | A=100 | B=50 | A=80 | B=70 | A=110
NEW | A=100 | B=50 | A=100 | B=50 | A=80 | B=70 | A=110

read(var):
return cell_read(var)

write(var, value):
log.append(current_tid, “UPDATE”, var, read(var), value)
cell_write(var, value)

recover(log):
commits = []
for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
        commits.add(r.tid)
    if r.type == UPDATE and r.tid not in commits:
        cell_write(r.var, r.old_val) // undo

commits = []
```
begin  // T1
write(A, 100)
write(B, 50)
commit  // A=100; B=50

begin  // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit  // A=80; B=70

begin  // T3
write(A, read(A)+30)
crash!
```

<table>
<thead>
<tr>
<th>TID</th>
<th>T1</th>
<th>T1</th>
<th>T1</th>
<th>T2</th>
<th>T2</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>OLD</td>
<td>A=0</td>
<td>B=0</td>
<td></td>
<td>A=100</td>
<td>B=50</td>
<td></td>
<td>A=80</td>
</tr>
<tr>
<td>NEW</td>
<td>A=100</td>
<td>B=50</td>
<td></td>
<td>A=80</td>
<td>B=70</td>
<td></td>
<td>A=110</td>
</tr>
</tbody>
</table>

**cell storage**

- **A**: 80
- **B**: 70

---

**read(var):**
```
return cell_read(var)
```

**write(var, value):**
```
log.append(current_tid, “UPDATE”, var, 
        read(var), value)
cell_write(var, value)
```

**recover(log):**
```
commits = []
for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
        commits.add(r.tid)
    if r.type == UPDATE and r.tid not in commits:
        cell_write(r.var, r.old_val) // undo

commits = []
```
<table>
<thead>
<tr>
<th>TID</th>
<th>OLD</th>
<th>NEW</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A=0</td>
<td>B=0</td>
<td>A=100</td>
<td>B=50</td>
<td>A=80</td>
<td>B=70</td>
</tr>
</tbody>
</table>

**cell storage (on disk)**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>80</td>
<td>70</td>
</tr>
</tbody>
</table>

**read(var):**
```
return cell_read(var)
```

**write(var, value):**
```
log.append(current_tid, "UPDATE", var, read(var), value)
cell_write(var, value)
```

**recover(log):**
```
commits = []
for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
        commits.add(r.tid)
    if r.type == UPDATE and r.tid not in commits:
        cell_write(r.var, r.old_val) // undo

commits = [T2, T1]
```
```
<table>
<thead>
<tr>
<th>TID</th>
<th>T1</th>
<th>T1</th>
<th>T1</th>
<th>T2</th>
<th>T2</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>OLD</td>
<td>A=0</td>
<td>B=0</td>
<td></td>
<td>A=100</td>
<td>B=50</td>
<td></td>
<td>A=80</td>
</tr>
<tr>
<td>NEW</td>
<td>A=100</td>
<td>B=50</td>
<td></td>
<td>A=80</td>
<td>B=70</td>
<td></td>
<td>A=110</td>
</tr>
</tbody>
</table>
```

```
begin // T1
write(A, 100)
write(B, 50)
commit // A=100; B=50
begin // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit // A=80; B=70
begin // T3
write(A, read(A)+30)
crash!
```

**problem:** read performance is now great, but writes got (a little bit) slower and recovery got (a lot) slower

**read(var):**
```
return cell_read(var)
```

**write(var, value):**
```
log.append(current_tid, "UPDATE", var, read(var), value)
cell_write(var, value)
```

**recover(log):**
```
commits = []
for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
        commits.add(r.tid)
    if r.type == UPDATE and r.tid not in commits:
        cell_write(r.var, r.old_val) // undo
```
read(var):
    if var in cache:
        return cache[var]
    else:
        // may evict others from cache to cell storage
        cache[var] = cell_read(var)
        return cache[var]

write(var, value):
    log.append(current_tid, update, var, read(var), value)
    cache[var] = value

flush(): // called “occasionally”
    cell_write(var, cache[var]) for each var

recover(log):
    commits = {}
    for record r in log[len(log)-1] .. log[0]:
        if r.type == commit:
            commits.add(r.tid)
        if r.type == update and r.tid not in commits:
            cell_write(r.var, r.old_val) // undo
begin // T1
write(A, 100)
write(B, 50)
commit // A=100; B=50

begin // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit // A=80; B=70

begin // T3
write(A, read(A)+30)

read(var):
if var in cache:
    return cache[var]
else:
    // may evict others from cache to cell storage
    cache[var] = cell_read(var)
    return cache[var]

write(var, value):
    log.append(current_tid, update, var,
               read(var), value)
    cache[var] = value

flush(): // called “occasionally”
    cell_write(var, cache[var]) for each var

suppose we flushed the cache after T1 committed, but have not flushed it since then
begin // T1
write(A, 100)
write(B, 50)
commit // A=100; B=50

begin // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit // A=80; B=70

begin // T3
write(A, read(A)+30)
crash!

read(var):
if var in cache:
    return cache[var]
else:
    // may evict others from cache to cell storage
    cache[var] = cell_read(var)
    return cache[var]

write(var, value):
log.append(current_tid, update, var,
    read(var), value)
cache[var] = value

flush(): // called “occasionally”
cell_write(var, cache[var]) for each var

recover(log):
commits = []
for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
        commits.add(r.tid)
    if r.type == UPDATE and r.tid not in commits:
        cell_write(r.var, r.old_val) // undo

commits = []
<table>
<thead>
<tr>
<th>TID</th>
<th>T1</th>
<th>T1</th>
<th>T1</th>
<th>T2</th>
<th>T2</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>TID</td>
<td>UPDATE</td>
<td>UPDATE</td>
<td>COMMIT</td>
<td>UPDATE</td>
<td>UPDATE</td>
<td>COMMIT</td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>A=0</td>
<td>B=0</td>
<td></td>
<td>A=100</td>
<td>B=50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>A=100</td>
<td>B=50</td>
<td></td>
<td>A=80</td>
<td>B=70</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>A=80</td>
<td>B=70</td>
<td></td>
<td>A=110</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**cell storage (on disk)**

$$\begin{array}{cc}
A & 80 \\
B & 50 \\
\end{array}$$

**cache (memory)**

### read(var):
- if var in cache:
  - return cache[var]
- else:
  - // may evict others from cache to cell storage
  - cache[var] = cell_read(var)
  - return cache[var]

### write(var, value):
- log.append(current_tid, update, var, read(var), value)
- cache[var] = value

### flush():
- // called “occasionally”
- cell_write(var, cache[var]) for each var

### recover(log):
- commits = []
- for record r in log[len(log)-1] .. log[0]:
  - if r.type == COMMIT:
    - commits.add(r.tid)
  - if r.type == UPDATE and r.tid not in commits:
    - cell_write(r.var, r.old_val) // undo

- commits = []
<table>
<thead>
<tr>
<th>TID</th>
<th>T1</th>
<th>T1</th>
<th>T1</th>
<th>T2</th>
<th>T2</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>OLD</td>
<td>A=0</td>
<td>B=0</td>
<td></td>
<td>A=100</td>
<td>B=50</td>
<td></td>
<td>A=80</td>
</tr>
<tr>
<td>NEW</td>
<td>A=100</td>
<td>B=50</td>
<td></td>
<td>A=80</td>
<td>B=70</td>
<td></td>
<td>A=110</td>
</tr>
</tbody>
</table>

**begin**  
// T1  
write(A, 100)  
write(B, 50)  
commit  // A=100; B=50

**begin**  
// T2  
write(A, read(A)-20)  
write(B, read(B)+20)  
commit  // A=80; B=70

**begin**  
// T3  
write(A, read(A)+30)  
crash!

---

**cell storage**  
(on disk)  
A 80  
B 50  
**cache**  
(memory)

---

**read(var):**  
if var in cache:  
    return cache[var]  
else:  
    // may evict others from cache to cell storage  
    cache[var] = cell_read(var)  
    return cache[var]

**write(var, value):**  
log.append(current_tid, update, var, read(var), value)  
    cache[var] = value

flush(): // called “occasionally”  
    cell_write(var, cache[var]) for each var

**recover(log):**  
    commits = []  
    for record r in log[len(log)-1] .. log[0]:  
        if r.type == COMMIT:  
            commits.add(r.tid)  
        if r.type == UPDATE and r.tid not in commits:  
            cell_write(r.var, r.old_val) // undo

    commits = []

all other updates were committed; B’s value won’t ever be changed
begin // T1
write(A, 100)
write(B, 50)
commit // A=100; B=50

begin // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit // A=80; B=70

begin // T3
write(A, read(A)+30)
crash!

problem: recovery is still slow
### Solution: Write Checkpoints and Truncate the Log

Given the transaction logs and the values of variables in the cache and cell storage, we can determine the final state of the variables. Here is a step-by-step solution:

1. **Begin T1**
   - Write A to 100
   - Write B to 50
   - **Commit**
     - A = 100; B = 50

2. **Begin T2**
   - Write A to A - 20
   - Write B to B + 20
   - **Commit**
     - A = 80; B = 70

3. **Begin T3**
   - Write A to A + 30
   - **Crash**

To recover the state of the database, we can use the log and the cache. The recovery process involves:

- **Read VAR**:
  - If var in cache: return cache[var]
  - Else: may evict others from cache to cell storage
    - cache[var] = cell_read(var)
    - return cache[var]

- **Write VAR, VALUE**:
  - log.append(current_tid, update, var, read(var), value)
  - cache[var] = value

- **Flush**: called “occasionally”
  - cell_write(var, cache[var]) for each var

- **Recover LOG**: 
  - commits = []
  - For record r in log[len(log)-1] .. log[0]:
    - If r.type == COMMIT:
      - commits.add(r.tid)
    - If r.type == UPDATE and r.tid not in commits:
      - cell_write(r.var, r.old_val) // undo
  - For record r in log[0] .. log[len(log)-1]:
    - If r.type == UPDATE and r.tid in commits:
      - cell_write(r.var, r.new_value) // redo

By following these steps, we can ensure the integrity of the database and recover the state of the variables correctly.
our goal is to build **reliable systems from unreliable components**. we want to build systems that serve many clients, store a lot of data, perform well, all while keeping availability high.

**transactions** — which provide **atomicity** and **isolation** — make it easier for us to reason about failures.

our job in lecture is to understand how a system *implements* these two abstractions. how do our systems guarantee atomicity? how do they guarantee isolation?

**atomicity**: provided by **logging**, which gives better performance than shadow copies* at the cost of some added complexity. *shadow copies are used in some systems.

**isolation**: we don’t really have this yet (coarse-grained locks perform poorly; fine-grained locks are difficult to reason about).
(write-ahead) logs provide atomicity with better performance than shadow copies. The primary benefit is making small appends for each update, rather than copy an entire file over for every change.

Cell storage is used with the log to improve read performance, and caches and truncation can be used to improve write and recovery performance.