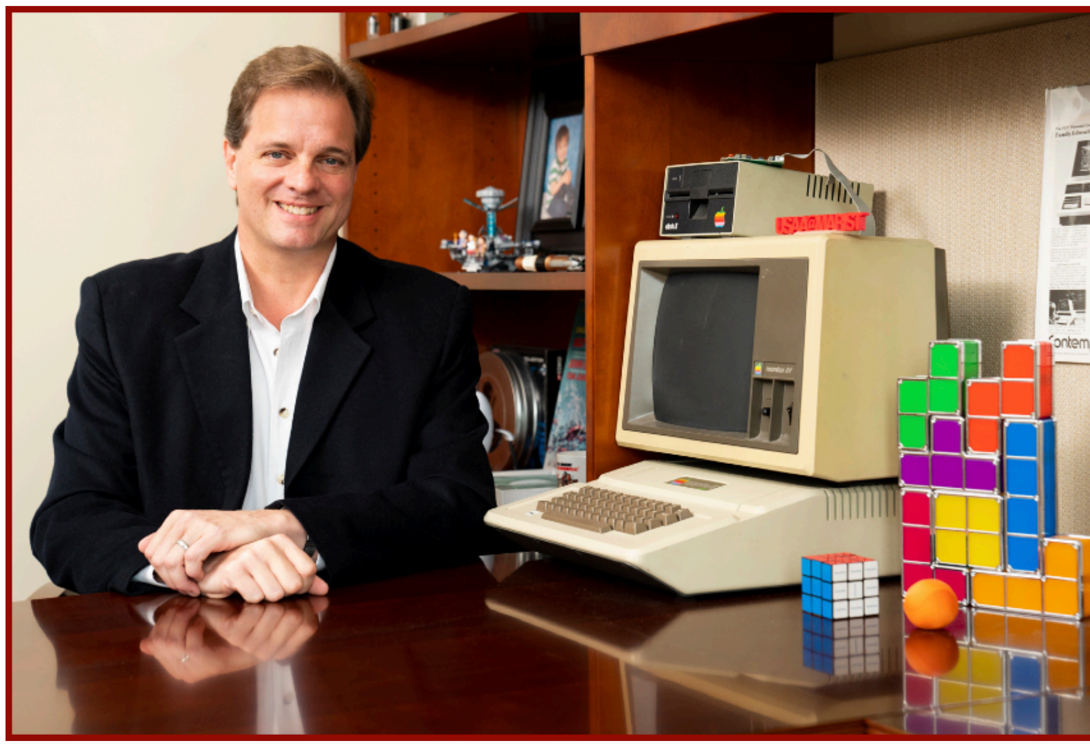
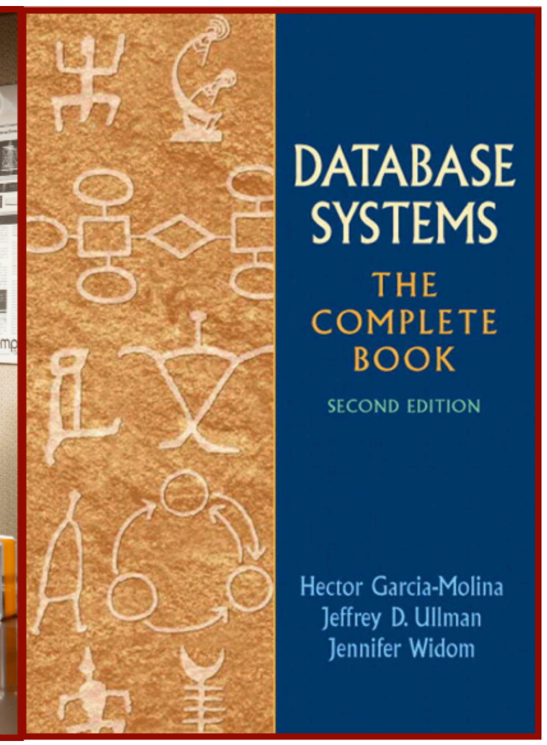
Locking, Transactions, and the WAL





Alan G. Labouseur, Ph.D.
Alan.Labouseur@Marist.edu

Transactions

A transaction (in a good data management system) is a set of operations that the database engine guarantees will be either done in its entirety or not at all. E.g., EON: Everything or Nothing

```
begin transaction
  SQL and other commands
  control structures
  insert/update/delete
commit / rollback
```

There are four important properties for transactions:

- **A**tomic
- **C**onsistent
- **I**ndependent
- **D**urable

We'll call them ACID for short.

ACID

Transactions should be . . .

- **A**tomic
  Indivisible. Everything or nothing. No partial work.
  The results of a transaction are seen in their **entirety** or **not at all**.

- **C**onsistent
  All transactions leave the database in a consistent state.
  Constraints that were true before the transactions are true after.

- **I**ndependent or **I**solated
  Transactions running concurrently act as if they were running sequentially, isolated and independent of each other.

- **D**urable
  The effects of completed transactions are resilient against failures.
  One complete, transaction results will not be lost regardless of what happens next.

Locking and Blocking

The "A", "C", and "I" in ACID are achieved with locking and blocking.

Lock Types

- read / shared — multiple
- write / exclusive — one at a time

Lock Terms

- short — lasts only as long as the access
- long — lasts until the end of the transaction

Lock Granularity

- Database — the whole database
- Table — one table in the database
- Page — one page (TSB) on the disk
- Row — one or more rows in the table
- Field — one or more columns in a row

Locking and Blocking

Locking Rules

- If a transactions wants to **read** an object, it must first acquire a read/shared lock.
- If a transactions wants to **write** an object, it must first acquire a write/exclusive lock.

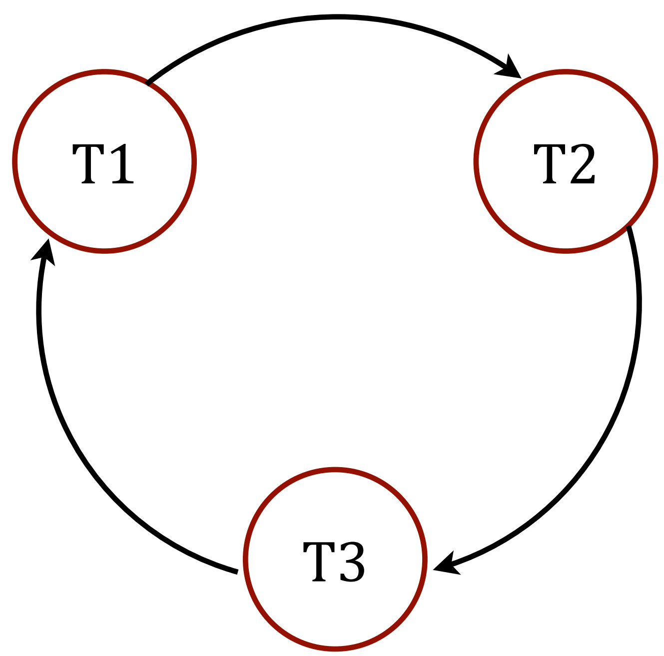
- The database engine will permit many shared locks per object, but **only one exclusive lock per object**.

(That might be a problem.)

Locking and Blocking

Deadlock

- A “deadly embrace” when two or more transactions are all waiting for others’ resources (objects on which they have or want locks).
- Represented as a **wait-for** graph.

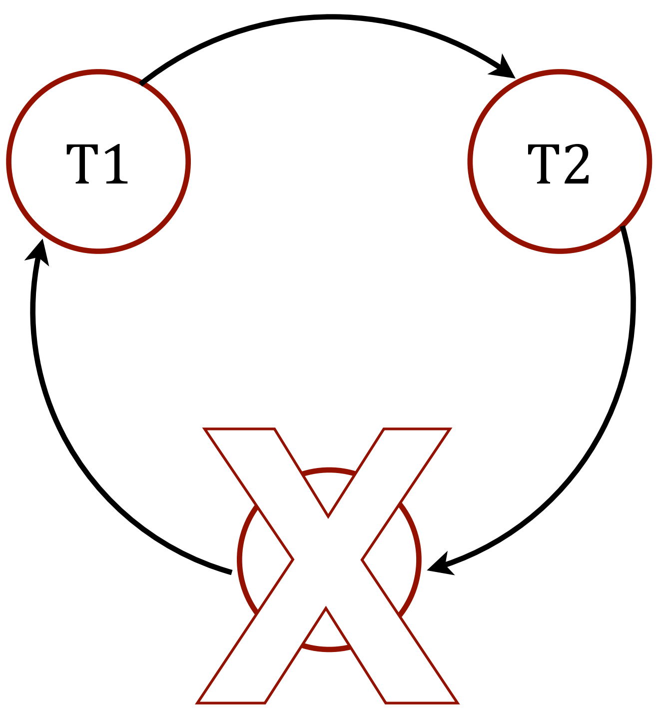


Q: How do we fix that?

Locking and Blocking

Deadlock

- A “deadly embrace” when two or more transactions are all waiting for others’ resources (objects on which they have or want locks).
- Represented as a **wait-for** graph.

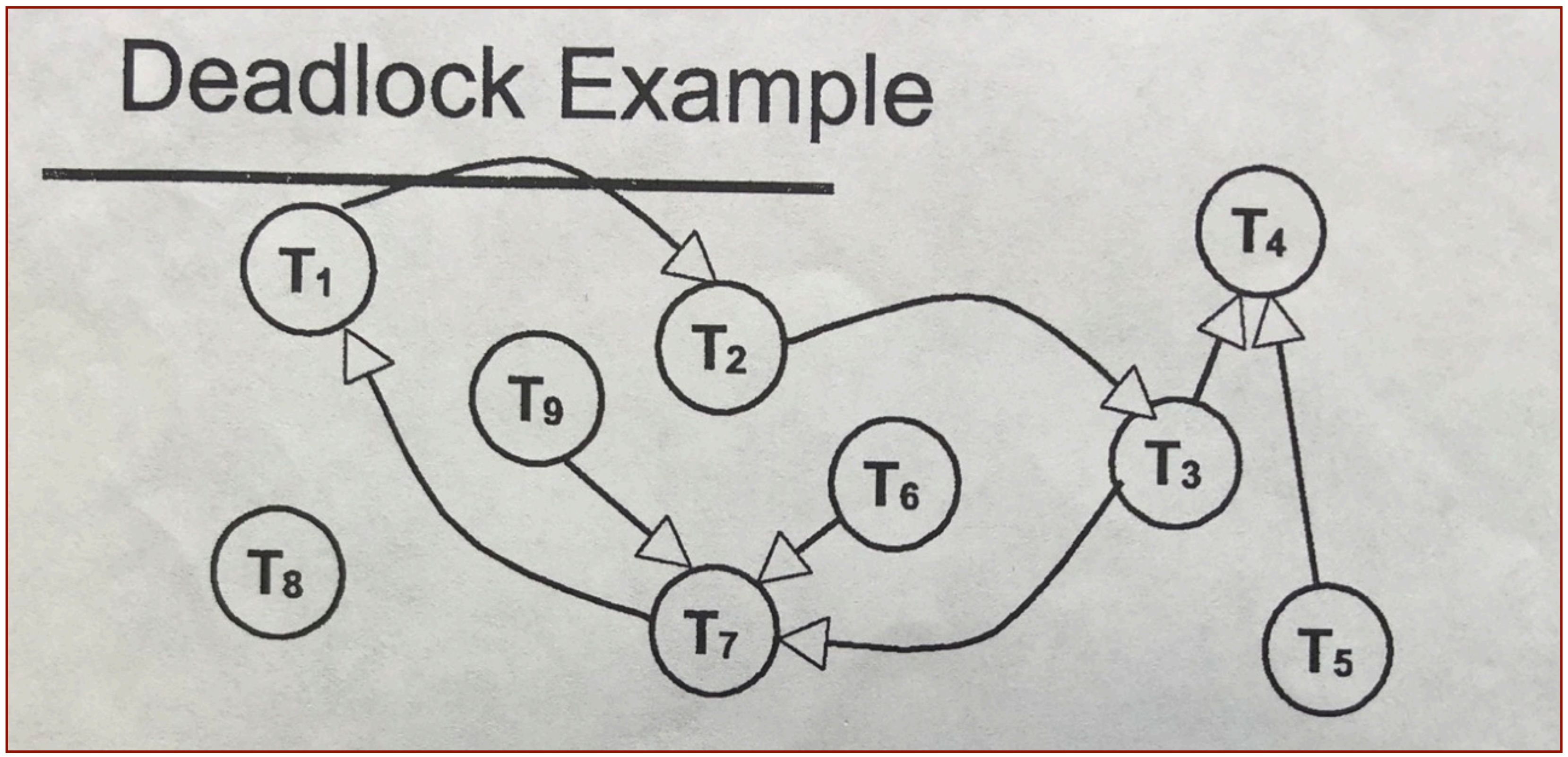


Q: How do we fix that?
A: Roll back (abort) one of the transactions, releasing their locks.

Locking and Blocking

Deadlock

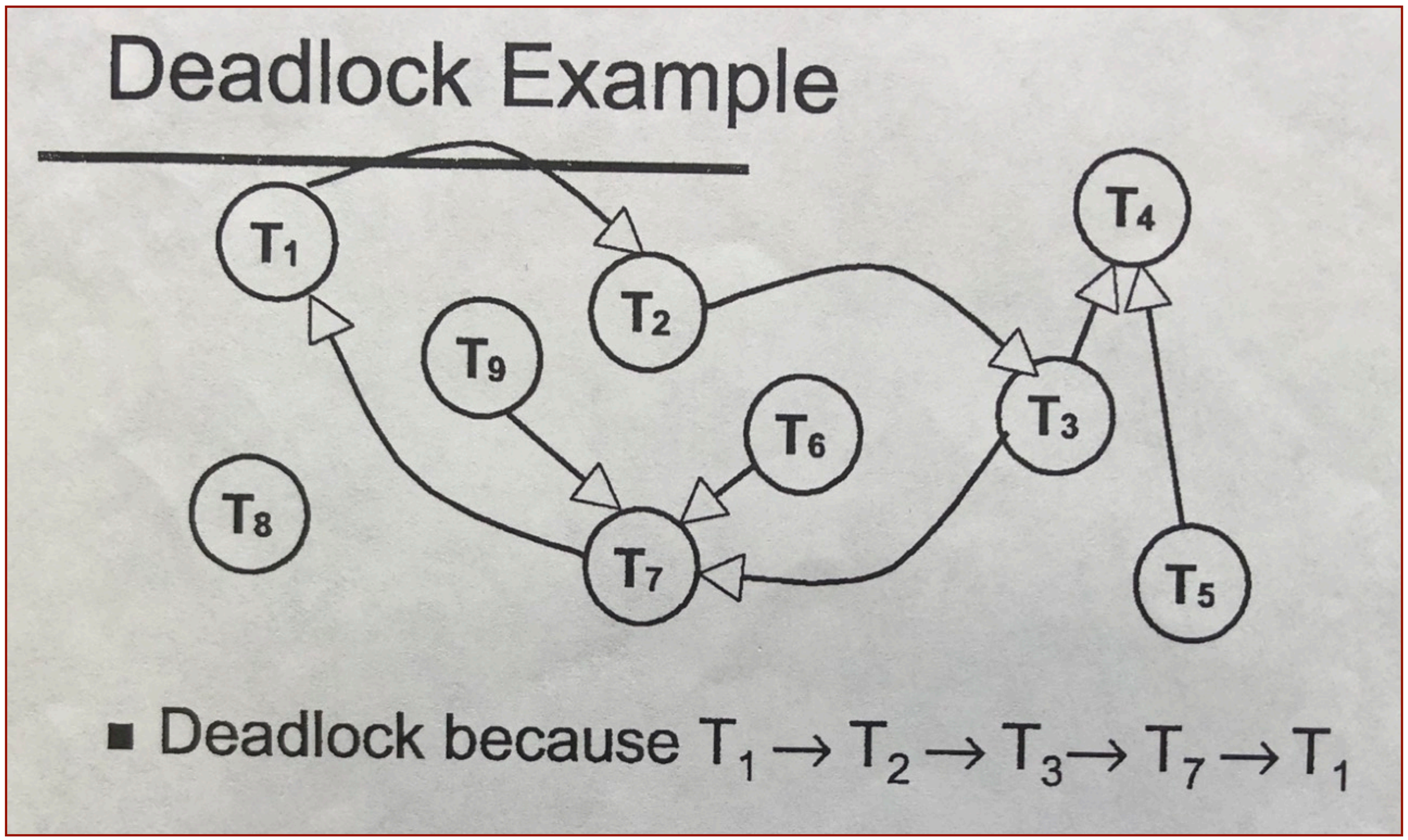
- Can be non-obvious.



Locking and Blocking

Deadlock

- Can be non-obvious.



A depth-first traversal of the wait-for-graph will reveal deadlock when a "back-edge" (denoting a cycle) is found.

Why Bother?

Consider bank account transactions for user "A".
Two accounts: A1 = \$900 A2 = \$100 (\$900 + \$100 = \$1000)

Consider two simultaneous transactions:

- T1 — Transfer \$400 from A1 to A2.
- T2 — Balance check to influence a credit offer.

Why Bother?

Consider bank account transactions for user "A".
Two accounts: A1 = \$900 A2 = \$100 (\$900 + \$100 = \$1000)

Transaction T1 (update transaction)

```
Update A set balance = balance - $400.00
   where A.aid = 'A1';
(Now balance = $500.00.)

Update A set balance = balance + $400.00
   where A.aid = 'A2';
(Now balance = $500.00.)

commit work;
```

Consistent

T1 — Transfer \$400 from A1 to A2. (\$500 + \$500 = \$1000)

Why Bother?

Consider bank account transactions for user "A".
Two accounts: A1 = \$900 A2 = \$100 (\$1000 total)

Transaction T2 (read-only transaction)

```
int bal, sum = 0.00;

select A.balance into :bal from A
   where A.aid = 'A1';
sum = sum + bal;

select A.balance into :bal from A
   where A = 'A2';
sum = sum + bal;

commit work;
```

T2 — Balance check to influence a credit offer.

Why Bother?

Consider bank account transactions for user "A".
Two accounts: A1 = \$900 A2 = \$100 (\$1000 total)

Transaction T1 (update transaction)	Transaction T2 (read-only transaction)
	`int bal, sum = 0.00;`

`Update A set balance = balance - $400.00`
`where A.aid = 'A1';`
(Now balance = \$500.00.)	

	`select A.balance into :bal from A`
	`where A.aid = 'A1';`
	`sum = sum + bal;`

	`select A.balance into :bal from A`
	`where A = 'A2';`
	`sum = sum + bal;`

`Update A set balance = balance + $400.00`
`where A.aid = 'A2';`
(Now balance = \$500.00.)	

	`commit work;`
`commit work;`	

(from O'Neil and O'Neil's awesome Database book)

What can go wrong?

Why Bother?

Consider bank account transactions for user “A”.
Two accounts: A1 = \$900 A2 = \$100 (\$1000 total)

Transaction T1 (update transaction)	Transaction T2 (read-only transaction)
	`int bal, sum = 0.00;`

`Update A set balance = balance - $400.00`
`where A.aid = 'A1';`
(Now balance = \$500.00.)	

	`select A.balance into :bal from A`
	`where A.aid = 'A1';`
	`sum = sum + bal;` (Now sum = \$500.00.)
	`select A.balance into :bal from A`
	`where A = 'A2';`
	`sum = sum + bal;` (Now sum = \$600.00.)

`Update A set balance = balance + $400.00`
`where A.aid = 'A2';`
(Now balance = \$500.00.)
	Inconsistent !

	`commit work;`
`commit work;`	

Figure 5.15 No Locks Held for the Transaction: View of Inconsistent Data

Why Bother? Because it preserves consistency.

Consider bank account transactions for user "A".
Two accounts: A1 = \$900 A2 = \$100 (\$1000 total)

Transaction T1 (update transaction)	Transaction T2 (read-only transaction)
	`int bal, sum = 0.00;`

`Update A set balance = balance - $400.00`
`where A.aid = 'A1';`
(This row is now locked, balance = \$500.00.)	

	`select A.balance into :bal from A`
	`where A.aid = 'A1';`
	(Same row as T1 just locked; must WAIT.)

`Update A set balance = balance + $400.00` `where A.aid = 'A2';` (This row now locked) (Now balance = \$500.00; transfer complete.)	
`commit work;` (Releases locks)	

	(Prior select can now achieve needed lock.)
	`select A.balance into :bal from A`
	`where A = 'A1';`
	`sum = sum + bal;` (Now sum = \$500.00.)

	`select A.balance into :bal from A`
	`where A = 'A2';`
	`sum = sum + bal;` (Now sum = \$1000.00.)

	`commit work;`

Figure 5.16 Transaction Locks Fix Inconsistent View of Data of Figure 5.15

Consistent

What about deadlock?

Transaction T1	Transaction T2
	`int bal, sum = 0.00;`
`Update A set balance = balance - $400.00 where A = 'A1';` (Lock achieved.)	
	`select A.balance into :bal from A where A = 'A2'; sum = sum + bal;` (Lock achieved.)
	`select A.balance into :bal from A where A = 'A1';` (Conflict with T1: WAIT.)
	. . .
`Update A set balance = balance + $400.00 where A = 'A2';` (Conflict with T2: WAIT).	. . .

(from O'Neil and O'Neil's awesome Database book)

What about deadlock!

Transaction T1	Transaction T2
	`int bal, sum = 0.00;`

`Update A set balance = balance - $400.00`
`where A = 'A1';`
(Lock achieved.)	

	`select A.balance into :bal from A`
	`where A = 'A2';`
	`sum = sum + bal;` (Lock achieved.)

	`select A.balance into :bal from A`
	`where A = 'A1';`
	(Conflict with T1: WAIT.)

	. . .

`Update A set balance = balance + $400.00`
`where A = 'A2';`
(Conflict with T2: WAIT).
	. . .

. . .	. . .
. . .	. . .

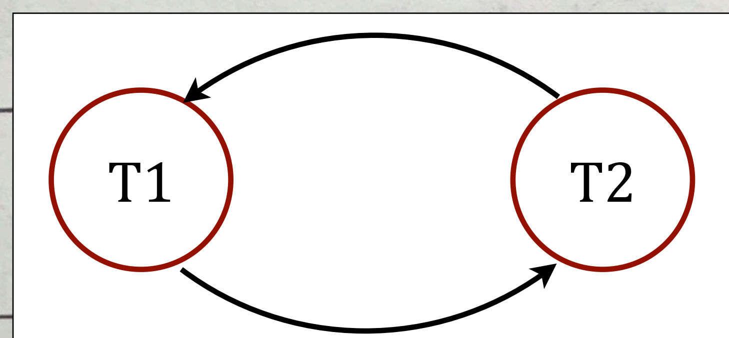


Figure 5.17 Transaction Deadlock

Two-phase Locking

Locking Rules

- If a transactions wants to **read** an object, it must first acquire a read/shared lock.
- If a transactions wants to **write** an object, it must first acquire a write/exclusive lock.
- The database engine will permit many shared locks per object, but only one exclusive lock per object.

Two-phase Locking

- **Growing** phase — Transaction acquires all the locks it's ever going to need before doing anything else. It never asks for more.
- **Shrinking** phase — Transaction releases it's locks, typically on commit or rollback.

- Does not prevent deadlock in all cases, but helps reduce it.

Locking Recipes

Great. So what lock types, duration, and granularity should we use?

Lock Types:	read/shared, write/exclusive
Lock Terms:	short, long
Lock Granularity:	database, table, page, row, field

Rather than get lost in the many combinations of lock **type**, **duration**, and **granularity**, let's look at some pre-defined locking schemes, called "isolation levels".

Locking Isolation Levels

	Write locks on rows of a table are long-term	Read locks on rows of a table are long-term	Read and write locks on predicates are long-term
Read Uncommitted (dirty reads)	No (but it's read-only)	No Read locks at all	No predicate locks at all
Read Committed	Yes	No	Short-term Read predicate locks Long-term Write predicate locks
Repeatable Read	Yes	Yes	Short-term Read predicate locks Long-term Write predicate locks
Serializable	Yes	Yes	Long-term Read and Write predicate locks

Figure 10.9 Long-Term Locking Behavior of SQL-99 Isolation Levels[2]

(from O'Neil and O'Neil's awesome Database book)

Locking Isolation Levels

performance

contention

	Write locks on rows of a table are long-term	Read locks on rows of a table are long-term
Read Uncommitted (dirty reads)	No (but it's read-only)	No Read locks at all
Read Committed	Yes	No
Repeatable Read	Yes	Yes
Serializable	Yes	Yes

Figure 10.9 Long-Term Locking Behavior of SQL-99 Isolat

Accuracy?

Locking Isolation Levels

The "A", "C", and "I" in ACID are achieved with locking and blocking.

Mode	Used
Access Share Lock	SELECT
Row Share Lock	SELECT FOR UPDATE
Row Exclusive Lock	INSERT, UPDATE, DELETE
Share Lock	CREATE INDEX
Share Row Exclusive Lock	EXCLUSIVE MODE but allows ROW SHARE LOCK
Exclusive Lock	Blocks ROW SHARE LOCK and SELECT...FOR UPDATE
Access Exclusive Lock	ALTER TABLE, DROP TABLE, VACUUM

PostgreSQL Locking Modes and Uses

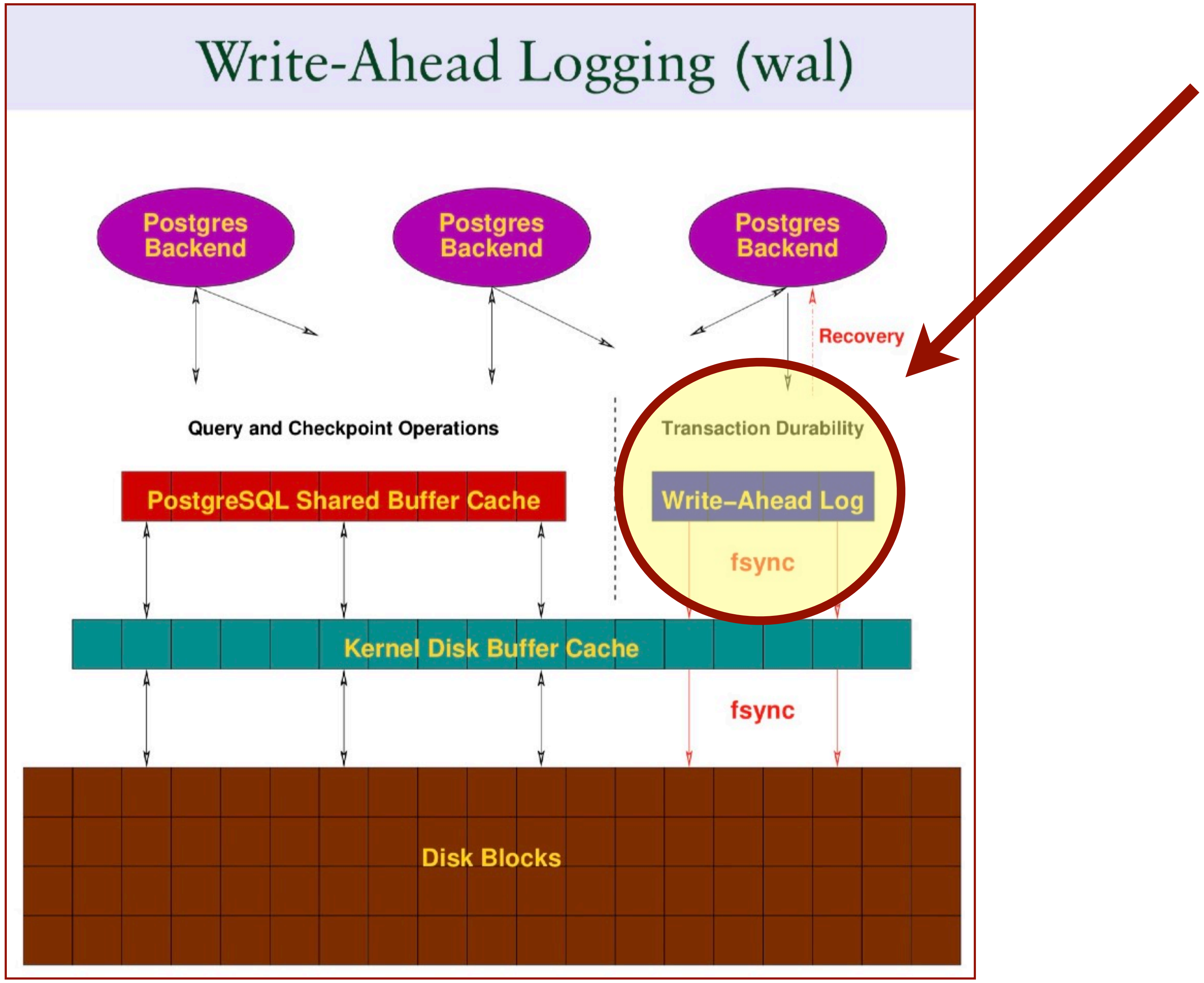
(from the great Bruce Momjian)

What about "D" — durability?

The Log File

The "D" in ACID comes from the log file.

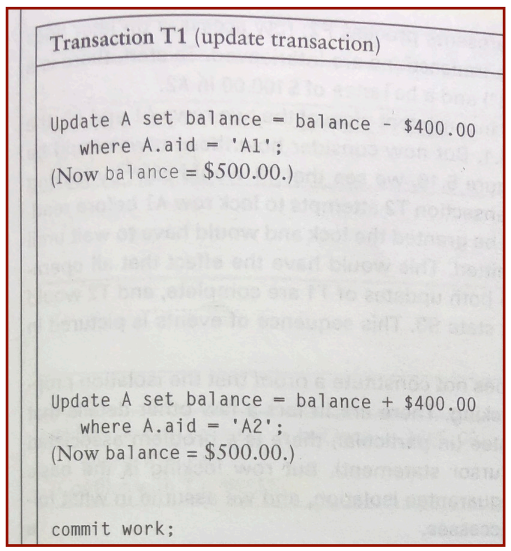
Specifically the Write-Ahead Log file



(from the great Bruce Momjian)

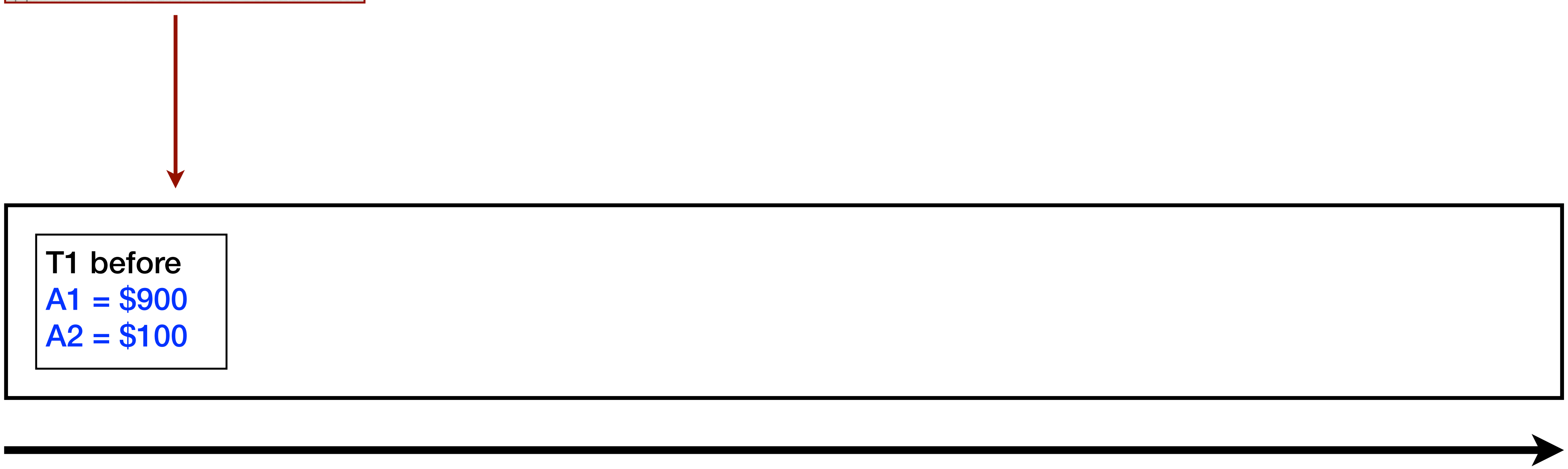
The Write-Ahead Log

When a transaction begins, the plan is written to a log file, along with the values of the data elements involved as they exist at that time, called a “before” image.



Transaction T1 (update transaction)

```
Update A set balance = balance - $400.00
   where A.aid = 'A1';
(Now balance = $500.00.)

Update A set balance = balance + $400.00
   where A.aid = 'A2';
(Now balance = $500.00.)

commit work;
```



The Write-Ahead Log

When a transaction begins, the plan is written to a log file, along with the values of the data elements involved as they exist at that time, called a “before” image.

Transaction T1 (update transaction)

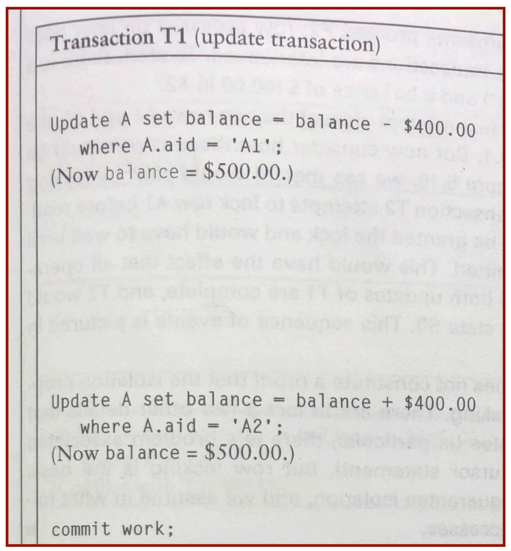
```
Update A set balance = balance - $400.00
   where A.aid = 'A1';
(Now balance = $500.00.)

Update A set balance = balance + $400.00
   where A.aid = 'A2';
(Now balance = $500.00.)

commit work;
```

When that same transaction commits, the new values are written to the log file as an “after image” and then saved in the database.

T1 before	T1 after
A1 = $900	A1 = $500
A2 = $100	A2 = $500

The Write-Ahead Log

When a transaction begins, the plan is written to a log file, along with the values of the data elements involved as they exist at that time, called a “before” image.


Transaction T1 (update transaction)

```
Update A set balance = balance - $400.00
    where A.aid = 'A1';
```
(Now balance = $500.00.)

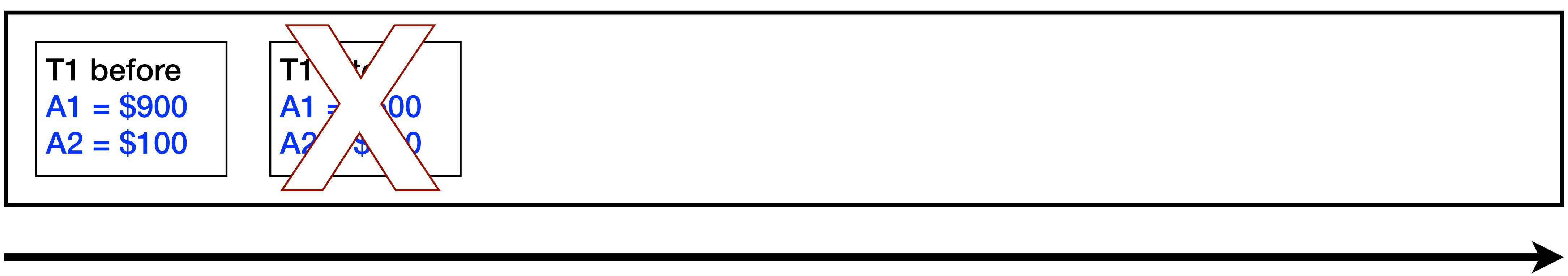
```
Update A set balance = balance + $400.00
    where A.aid = 'A2';
```
(Now balance = $500.00.)

```
commit work;
```

When that same transaction commits, the new values are written to the log file as an “after image” and then saved in the database.

But if there is a rollback instead, the before values are restored.

Everything or Nothing.

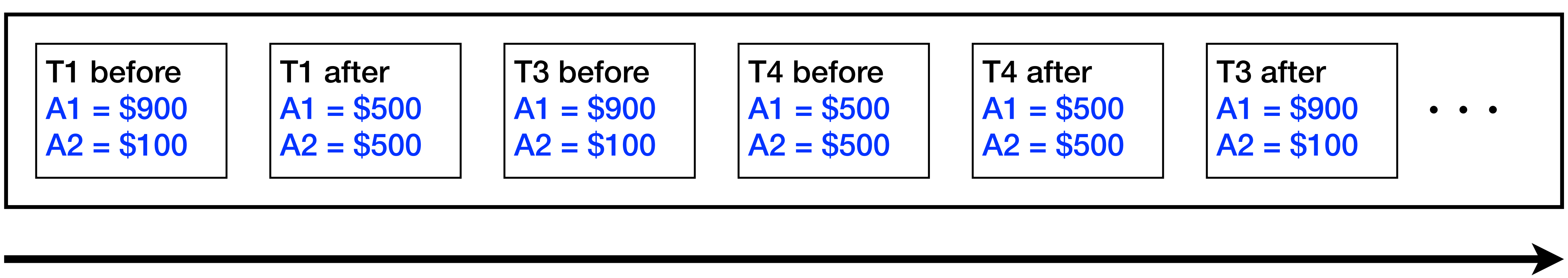


The Write-Ahead Log

When the database brought back online after a server failure, the recovery process looks at WAL for transactions that were running at the time of the failure. Those transactions…

- can be completed if there is enough data in the WAL.
  This is REDO.
- can be rolled back by restoring their before images.
  This is UNDO.
- In either case, the EON guarantee holds.

That's **D**urability.



The Write-Ahead Log

Another Example

(from Andy Pavlo's CMU Database class)

LSN	WAL Record
1	<T1 BEGIN>
2	<T1, X, 1, 2>
3	<T2 BEGIN>
4	<T2, Y, 1, 2>
5	<T1 COMMIT>
6	<T2, Y, 2, 3>
7	<T3 BEGIN>
8	<T3, Z, 1, 2>
9	<T2, X, 2, 3>
10	**<CHECKPOINT>**
11	<T2, Y, 3, 4>
12	<T3, Z, 2, 3>
13	<T3 COMMIT>
14	<T2, Z, 3, 4>

CRASH!

The Write-Ahead Log

Another Example

(from Andy Pavlo's CMU Database class)

LSN	WAL Record
1	<T1 BEGIN>
2	<T1, X, 1, 2>
3	<T2 BEGIN>
4	<T2, Y, 1, 2>
5	<T1 COMMIT>
6	<T2, Y, 2, 3>
7	<T3 BEGIN>
8	<T3, Z, 1, 2>
9	<T2, X, 2, 3>
10	**<CHECKPOINT>**
11	<T2, Y, 3, 4>
12	<T3, Z, 2, 3>
13	<T3 COMMIT>
14	<T2, Z, 3, 4>

< transaction id, object id, before value, after value>

CRASH!

Log Sequence Number

The Write-Ahead Log

Another Example

(from Andy Pavlo's CMU Database class)

LSN	WAL Record
1	<T1 BEGIN>
2	<T1, X, 1, 2>
3	<T2 BEGIN>
4	<T2, Y, 1, 2>
5	<T1 COMMIT>
6	<T2, Y, 2, 3>
7	<T3 BEGIN>
8	<T3, Z, 1, 2>
9	<T2, X, 2, 3>
10	**<CHECKPOINT>**
11	<T2, Y, 3, 4>
12	<T3, Z, 2, 3>
13	<T3 COMMIT>
14	<T2, Z, 3, 4>

Recovering

What should the recovery manager do about transaction T1?

The Write-Ahead Log

Another Example

(from Andy Pavlo's CMU Database class)

LSN	WAL Record
1	<T1 BEGIN>
2	<T1, X, 1, 2>
3	<T2 BEGIN>
4	<T2, Y, 1, 2>
5	<T1 COMMIT>
6	<T2, Y, 2, 3>
7	<T3 BEGIN>
8	<T3, Z, 1, 2>
9	<T2, X, 2, 3>
10	**<CHECKPOINT>**
11	<T2, Y, 3, 4>
12	<T3, Z, 2, 3>
13	<T3 COMMIT>
14	<T2, Z, 3, 4>

Recovering

What should the recovery manager do about transaction T1?

Nothing. T1 was committed before the checkpoint (all changes written to the disk) so it's fine. There's nothing to do.

The Write-Ahead Log

Another Example

(from Andy Pavlo's CMU Database class)

LSN	WAL Record
1	`<T1 BEGIN>`
2	`<T1, X, 1, 2>`
3	`<T2 BEGIN>`
4	`<T2, Y, 1, 2>`
5	`<T1 COMMIT>`
6	`<T2, Y, 2, 3>`
7	`<T3 BEGIN>`
8	`<T3, Z, 1, 2>`
9	`<T2, X, 2, 3>`
10	**`<CHECKPOINT>`**
11	`<T2, Y, 3, 4>`
12	`<T3, Z, 2, 3>`
13	`<T3 COMMIT>`
14	`<T2, Z, 3, 4>`

Recovering

What should the recovery manager do about transaction T2?

The Write-Ahead Log

Another Example

(from Andy Pavlo's CMU Database class)

LSN	WAL Record
1	<T1 BEGIN>
2	<T1, X, 1, 2>
3	<T2 BEGIN>
4	<T2, Y, 1, 2>
5	<T1 COMMIT>
6	<T2, Y, 2, 3>
7	<T3 BEGIN>
8	<T3, Z, 1, 2>
9	<T2, X, 2, 3>
10	**<CHECKPOINT>**
11	<T2, Y, 3, 4>
12	<T3, Z, 2, 3>
13	<T3 COMMIT>
14	<T2, Z, 3, 4>

Recovering

What should the recovery manager do about transaction T2?

UNDO. T2 never committed so all its changes need to be undone.

The Write-Ahead Log

Another Example

(from Andy Pavlo's CMU Database class)

LSN	WAL Record
1	<T1 BEGIN>
2	<T1, X, 1, 2>
3	<T2 BEGIN>
4	<T2, Y, 1, 2>
5	<T1 COMMIT>
6	<T2, Y, 2, 3>
7	<T3 BEGIN>
8	<T3, Z, 1, 2>
9	<T2, X, 2, 3>
10	**<CHECKPOINT>**
11	<T2, Y, 3, 4>
12	<T3, Z, 2, 3>
13	<T3 COMMIT>
14	<T2, Z, 3, 4>

Recovering

What should the recovery manager do about transaction T3?

The Write-Ahead Log

Another Example

(from Andy Pavlo's CMU Database class)

LSN	WAL Record
1	<T1 BEGIN>
2	<T1, X, 1, 2>
3	<T2 BEGIN>
4	<T2, Y, 1, 2>
5	<T1 COMMIT>
6	<T2, Y, 2, 3>
7	<T3 BEGIN>
8	<T3, Z, 1, 2>
9	<T2, X, 2, 3>
10	**<CHECKPOINT>**
11	<T2, Y, 3, 4>
12	<T3, Z, 2, 3>
13	<T3 COMMIT>
14	<T2, Z, 3, 4>

Recovering

What should the recovery manager do about transaction T3?

REDO. T3 committed after the checkpoint so the recovery manager needs to redo all of its changes.

The Write-Ahead Log

Another Example

(from Andy Pavlo's CMU Database class)

LSN	WAL Record
1	<T1 BEGIN>
2	<T1, X, 1, 2>
3	<T2 BEGIN>
4	<T2, Y, 1, 2>
5	<T1 COMMIT>
6	<T2, Y, 2, 3>
7	<T3 BEGIN>
8	<T3, Z, 1, 2>
9	<T2, X, 2, 3>
10	**<CHECKPOINT>**
11	<T2, Y, 3, 4>
12	<T3, Z, 2, 3>
13	<T3 COMMIT>
14	<T2, Z, 3, 4>

Recovery complete

Values after recovery is complete:

X = 2 (T1 committed before checkpoint)

Y = 1 (T2 changes undone in roll back)

Z = 3 (T3 changes redone in roll forward)