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:

- Atomic
- Consistent
- Independent
- Durable

We’ll call them ACID for short.
ACID

Transactions should be . . .

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

• **Consistent**
  All transactions leave the database in a consistent state.
  Constraints that were true before the transactions are true after.

• **Independent or Isolated**
  Transactions running concurrently act as if they were running sequentially, isolated and independent of each other.

• **Durable**
  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.
Deadlock
  • Can be non-obvious.

Deadlock Example
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;

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)

```sql
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)

What can go wrong?

<Transaction T1 (update transaction)>
Update A set balance = balance - $400.00
  where A.aid = 'A1';
  (Now balance = $500.00.)

<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;

(from O’Neil and O’Neil’s awesome Database book)
Why Bother?

Consider bank account transactions for user “A”.
Two accounts: \( A_1 = $900 \) \( A_2 = $100 \) ($1000 total)

Inconsistent!
Why Bother? Because it preserves consistency.

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

Consistent
What about deadlock?

<table>
<thead>
<tr>
<th>Transaction T1</th>
<th>Transaction T2</th>
</tr>
</thead>
</table>
| Update A set balance = balance - $400.00  
where A = 'A1';  
(Lock achieved.) | int bal, sum = 0.00; |
| | 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!

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

<table>
<thead>
<tr>
<th></th>
<th>Write locks on rows of a table are long-term</th>
<th>Read locks on rows of a table are long-term</th>
<th>Read and write locks on predicates are long-term</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Uncommitted</td>
<td>No (but it’s read-only)</td>
<td>No Read locks at all</td>
<td>No predicate locks at all</td>
</tr>
<tr>
<td>Read Committed</td>
<td>Yes</td>
<td>No</td>
<td>Short-term Read predicate locks</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Long-term Write predicate locks</td>
</tr>
<tr>
<td>Repeateable Read</td>
<td>Yes</td>
<td>Yes</td>
<td>Short-term Read predicate locks</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Long-term Write predicate locks</td>
</tr>
<tr>
<td>Serializable</td>
<td>Yes</td>
<td>Yes</td>
<td>Long-term Read and Write predicate locks</td>
</tr>
</tbody>
</table>

**Figure 10.9** Long-Term Locking Behavior of SQL-99 Isolation Levels

(from O’Neil and O’Neil’s awesome Database book)
### Locking Isolation Levels

<table>
<thead>
<tr>
<th></th>
<th>Write locks on rows of a table are long-term</th>
<th>Read locks on rows of a table are long-term</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Uncommitted (dirty reads)</td>
<td>No (but it’s read-only)</td>
<td>No Read locks at all</td>
</tr>
<tr>
<td>Read Committed</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Repeatable Read</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Serializable</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Figure 10.9** Long-Term Locking Behavior of SQL-99 Isolations

---

Accuracy?
### Locking Isolation Levels

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

#### PostgreSQL Locking Modes and Uses

<table>
<thead>
<tr>
<th>Mode</th>
<th>Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Share Lock</td>
<td>SELECT</td>
</tr>
<tr>
<td>Row Share Lock</td>
<td>SELECT FOR UPDATE</td>
</tr>
<tr>
<td>Row Exclusive Lock</td>
<td>INSERT, UPDATE, DELETE</td>
</tr>
<tr>
<td>Share Lock</td>
<td>CREATE INDEX</td>
</tr>
<tr>
<td>Share Row Exclusive Lock</td>
<td>EXCLUSIVE MODE but allows ROW SHARE LOCK</td>
</tr>
<tr>
<td>Exclusive Lock</td>
<td>Blocks ROW SHARE LOCK and SELECT...FOR UPDATE</td>
</tr>
<tr>
<td>Access Exclusive Lock</td>
<td>ALTER TABLE, DROP TABLE, VACUUM</td>
</tr>
</tbody>
</table>

(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.

T1 before
A1 = $900
A2 = $100
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.

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

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 Durability.