DROP TABLE IF EXISTS public.employee;

CREATE TABLE public.employee (

id SERIAL PRIMARY KEY,

name VARCHAR(255) NOT NULL,

status VARCHAR(50) NOT NULL

);

|  |  |
| --- | --- |
| Step | Transaction Behavior |
| First parallel execution (insert Alice) | Transactions that read before T1 commits don’t see Alice. Transactions that read after see Alice. |
| Second parallel execution (delete Alice & reinsert) | Transactions that read before T1 commits still see Alice. After commit, the original Alice row is gone, and a new row is created. |
| Third parallel execution (update Alice) | Transactions that read before T1 commits see old Alice. After commit, they see updated Alice ("Fired"). |

1. SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
2. SHOW TRANSACTION ISOLATION LEVEL;
3. What Changed After Adding cmin and cmax?

|  |  |  |
| --- | --- | --- |
| **Column** | **Original Version** | **Modified Version** |
| xmin | Tracked row creation transaction | No change |
| xmax | Tracked row deletion/updation transaction | No change |
| cmin | Not included | Shows the command ID within a transaction that created/updated a row |
| cmax | Not included | Shows the command ID that deleted a row within a transaction |

1. \* Try to cause a serialization anomaly on the employee table (add more data if necessary).

Change your isolation level to serializable and try to cause serialization anomaly one more

time. What happened?

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

INSERT INTO public.employee (name, status)

VALUES

('Alice', 'Active'),

('Bob', 'Active');

|  |  |
| --- | --- |
| BEGIN;  SELECT \* FROM public.employee WHERE name = 'Alice'; -- Assume Alice is 'Active'  UPDATE public.employee SET status = 'Promoted' WHERE name = 'Alice'; | BEGIN; SELECT \* FROM public.employee WHERE name = 'Alice'; -- Also sees Alice as 'Active' UPDATE public.employee SET status = 'Terminated' WHERE name = 'Alice'; COMMIT; |
| COMMIT; |  |

What Happens in REPEATABLE READ Mode?

No serialization anomaly occurs, but one update overwrites the other because PostgreSQL does not allow transaction conflicts in REPEATABLE READ. The second commit succeeds, but the first transaction was not aware of the other transaction’s changes.

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

What Happens in SERIALIZABLE Mode?

PostgreSQL prevents serialization anomalies by aborting one of the transactions. When Session 1 tries to commit, it fails with an error. The database prevents inconsistencies, but forces a rollback, requiring the failed transaction to retry.

1. \* Set your isolation level to read committed. Try to cause a lost update database anomaly on the employee table (add more data if necessary). What happened? What do you think are the downsides of the approach that Postgres took to handle this anomaly?

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

|  |  |
| --- | --- |
| BEGIN;  SELECT \* FROM public.employee WHERE name = 'Bob'; -- Assume Bob is 'Active' UPDATE public.employee SET status = 'On Leave' WHERE name = 'Bob'; | BEGIN;  SELECT \* FROM public.employee WHERE name = 'Bob'; -- Also sees Bob as 'Active' UPDATE public.employee SET status = 'Fired' WHERE name = 'Bob';  COMMIT; |
| COMMIT; |  |

What Happened? Lost Update Occurred:

* Session 2's update (Fired) was committed first.
* Session 1’s update (On Leave) was then committed, overwriting Fired.

The database did not detect a conflict, leading to one update being lost.

Downsides of PostgreSQL's Approach to Handling Lost Updates in READ COMMITTED Mode:

* Overwrites Data Silently: PostgreSQL does not detect conflicting updates in READ COMMITTED, so the second transaction’s update is lost without error.
* No Automatic Retry: Unlike SERIALIZABLE, PostgreSQL does not retry the failed transaction.
* Requires Application-Level Handling: The application must manually check whether another transaction modified the row before committing.

|  |  |  |
| --- | --- | --- |
| **Isolation Level** | **Serialization Anomaly** | **Lost Update** |
| *Read Committed* | ✅ Possible | ✅ Possible |
| *Repeatable Read* | 🚫 Prevented | ✅ Possible |
| *Serializable* | 🚫 Prevented | 🚫 Prevented |

**💡 Takeaways:**

* Use SERIALIZABLE for strict consistency but expect transaction rollbacks.
* Use REPEATABLE READ if you want stability, but be careful of lost updates.
* READ COMMITTED is faster but risks data anomalies.