ShareLock

ShareLock is the row-level locking mechanism used internally by PostgreSQL.

Deadlocks and prevention

When several DB transactions are acting on multiple rows of the same table, it's possible to incur in a deadlock and so into an error. This can be prevented by enforcing the same consistent order of lock acquisition onall the transactions performingINSERT ,UPDATE orDELETE on a given table.

On top of this, if multiple DB transactions act on multiple tables a deadlock will occur, even if they follow the order on each table described above, if they acquire locks on said tables in a different order. This can also be prevented by using a consistent order of lock acquisitionbetween different tables.

Imposing the lock acquisition order on a table with Ecto

WhenINSERT ing a list of rows Postgres will respect the order in which they appear in the query, so the reordering can happen beforehand.

For example, this will work:

```

Copy entries=[...]

ordered_entries=Enum.sort_by(entries,& &1.id)

Repo.insert_all(**MODULE**,ordered_entries)

```

PerformingUPDATE s is trickier because there is noORDER BY clause. The solution to this is toJOIN on a subquery thatSELECT s with the optionFOR UPDATE .

Using Ecto this can be done, for example, like this:

```

Copy query= from( entryinEntry, where:notis_nil(entry.value), order_by: entry.id, lock:"FOR UPDATE" )

Repo.update_all( from(einEntry,join: sinsubquery(query),on: e.id==s.id), [set: [value:nil]], timeout: timeout)

```

DELETE has the same quirks asUPDATE and it is too solved in the same way.

For example:

```

Copy query= from( entryinEntry, where:is_nil(entry.value), order_by: entry.id, lock:"FOR UPDATE" )

Repo.delete_all(from(einEntry,join: sinsubquery(query),on: e.id==s.id))

```

Imposing the lock acquisition order between tables with Ecto

When using anEcto.Multi to performINSERT ,UPDATE orDELETE on multiple tables the order to keep is between different operation. For example, supposingEntryA was established to be modified beforeEntryB , this is not correct:

```

Copy Multi.new() |>Multi.run(:update_b,fnrepo,_->

# operations with ordered locks on EntryB

end) |>Multi.run(:update_a,fnrepo,_->
```

operations with ordered locks on `EntryA`

end) |>Repo.transaction()

```

When possible, the simple solution is to move:update_a to be before:update_b . When not possible, for instance if:update_a depends on the result of:update_b , this can be solved by acquiring the locks in a separate operation.

For example:

```

Copy Multi.new() |>Multi.run(:acquire_a,fnrepo,_->

acquire locks in order on `EntryA`

end) |>Multi.run(:update_b,fnrepo,_->

operations with ordered locks on `EntryB`

end) |>Multi.run(:update_a,fnrepo,%{acquire_a: values}->

operations (no need to enforce order again) on `EntryA`

end) |>Repo.transaction()

```

Note also that for the same reasons multiple operations on the same table in the same transaction are not safe to perform if they each acquire locks in order, because locks are not released until the transaction is committed.

Order used for Explorer's tables

This is a complete list of the ordering currently in use on each table. It also specifies the order between tables in the same transaction: locks for a table on top need to be acquired before those from a table on the bottom.

Note that this should always be enforced because as long as there is one DB transaction performing in a different order there is the possibility of a deadlock.

schema module table name ordered by Explorer.Chain.Address addresses asc: :hash Explorer.Chain.Address.Name address_names [asc: :address_hash, asc: :name] Explorer.Chain.Address.CoinBalance address_coin_balances [asc: :address_hash, asc: :block_number] Explorer.Chain.Block blocks asc: :hash Explorer.Chain.Block.SecondDegreeRelation block_second_degree_relations [asc: :nephew_hash, asc: :uncle_hash] Explorer.Chain.Block.Reward block_rewards [asc: :address_hash, asc: :address_type, asc: :block_hash] Explorer.Chain.Block.EmissionReward emission_rewards asc: :block_range Explorer.Chain.Transaction transactions asc: :hash Explorer.Chain.Transaction.Fork transaction_forks [asc: :uncle_hash, asc: :index] Explorer.Chain.Log logs [asc: :transaction_hash, asc: :index] Explorer.Chain.InternalTransaction internal_transactions [asc: :transaction_hash, asc: :index] Explorer.Chain.Token tokens asc: :contract_address_hash Explorer.Chain.TokenTransfer token_transfers [asc: :transaction_hash, asc: :log_index] Explorer.Chain.TransactionAction transaction_actions [asc: :hash, asc: :log_index] Explorer.Chain.PolygonEdge.Deposit polygon_edge_deposits [asc: :msg_id] Explorer.Chain.PolygonEdge.DepositExecute polygon_edge_deposit_executes [asc: :msg_id] Explorer.Chain.PolygonEdge.Withdrawal polygon_edge_withdrawals [asc: :msg_id] Explorer.Chain.PolygonEdge.WithdrawalExit polygon_edge_withdrawal_exits [asc: :msg_id] Explorer.Chain.Optimism.OutputRoot op_output_roots [asc: :l2_output_index] Explorer.Chain.Optimism.TxnBatch op_transaction_batches [asc: :l2_block_number] Explorer.Chain.Optimism.Deposit op_deposits [asc: :l2_transaction_hash] Explorer.Chain.Optimism.DisputeGame op_dispute_games [asc: :index] Explorer.Chain.Optimism.FrameSequence op_frame_sequences [asc: :id] Explorer.Chain.Optimism.FrameSequenceBlob op_frame_sequence_blobs [asc: :id] Explorer.Chain.Optimism.WithdrawalEvent op_withdrawal_events [asc: :withdrawal_hash, asc: :l1_event_type] Explorer.Chain.Optimism.Withdrawal op_withdrawals [asc: :msg_nonce] Explorer.Chain.Address.TokenBalance address_token_balances [asc: :address_hash, asc: :token_contract_address_hash, asc: :block_number] Explorer.Chain.Address.CurrentTokenBalance address_current_token_balances [asc: :address_hash, asc: :token_contract_address_hash] Explorer.Chain.Scroll.Batch scroll_batches [asc: number] Explorer.Chain.Scroll.BatchBundle scroll_batch_bundles [asc: final_batch_number] Explorer.Chain.Scroll.Bridge scroll_bridge [asc: :type, asc: message_hash] Explorer.Chain.Scroll.L1FeeParam scroll_l1_fee_params [asc: :block_number, asc: tx_index, asc: name] Explorer.Chain.Shibarium.Bridge shibarium_bridge [asc: :operation_hash, asc:
```

l1_transaction_hash, asc: l2_transaction_hash] Explorer.Chain.StakingPool staking_pools :staking_address_hash Explorer.Chain.StakingPoolsDelegator staking_pools_delegators [asc: :delegator_address_hash, asc: :pool_address_hash] Explorer.Chain.ContractMethod contract_methods [asc: :identified, asc: :abi] Explorer.Market.MarketHistory market_history asc: :date Explorer.Chain.Withdrawal withdrawals asc: :index Explorer.Chain.Zkevm.TransactionBatch zkevm_transaction_batches [asc: :number] Explorer.Chain.Zkevm.BatchTransaction zkevm_batch_l2_transactions [asc: :hash] Explorer.Chain.Zkevm.LifecycleTransaction zkevm_lifecycle_l1_transactions [asc: :id] Explorer.Chain.Zkevm.Bridge zkevm_bridge [asc: :type, asc: :index] Explorer.Chain.Zkevm.BridgeL1Token zkevm_bridge_l1_tokens [asc: :address] Explorer.Chain.ZkSync.TransactionBatch zksync_transaction_batches [asc: :number] Explorer.Chain.ZkSync.BatchBlock zksync_batch_blocks [asc: :hash] Explorer.Chain.ZkSync.BatchTransaction zksync_batch_transactions [asc: :hash] Explorer.Chain.ZkSync.LifecycleTransaction zksync_lifecycle_transactions [asc: :id] Explorer.Chain.Celo.EpochReward celo_epoch_rewards [asc: :block_hash] Explorer.Chain.Celo.PendingEpochBlockOperation celo_pending_epoch_block_operations [asc: :block_hash] Explorer.Chain.Celo.ValidatorGroupVote celo_epoch_validator_group_votes [asc: :transaction_hash, asc: :account_address_hash, asc: :group_address_hash] Explorer.Chain.Celo.ElectionReward celo_election_rewards [asc: :block_hash, asc: type, asc: :account_address_hash, asc: :associated_account_address_hash]

Last updated15 days ago