

# General

- [Sql Recursive Loop Check](#)
- [Database Transaction Helper](#)

# Sql Recursive Loop Check

## SqlRecursiveCheckTask

`SqlRecursiveCheckTask` helps you find **loops** in data where one item points to another — for example, parent-child or dependency relationships.

It checks for **cycles**, like when  $A \rightarrow B \rightarrow C \rightarrow A$ . These can break trees, graphs, or dependency systems.

---

## What You Give It

You can give it data in two ways:

### 1. Query Builder or Eloquent Builder

Pass a query where you select exactly two fields:

- `from_id`: the starting point
- `to_id`: where it points to

#### Example:

```
Model::select('id as from_id', 'parent_id as to_id')
```

### 2. Array or Collection

You can pass an array or a Laravel collection of pairs:

- As named keys:

```
[
    ['from_id' => 1, 'to_id' => 2],
    ['from_id' => 2, 'to_id' => 3],
    ['from_id' => 3, 'to_id' => 1],
]
```

- Or just plain arrays:

```
[
  [1, 2],
  [2, 3],
  [3, 1],
]
```

---

## What It Does

It runs a recursive SQL query to find **all loops** — meaning chains where something eventually points back to itself.

This is useful if you want to prevent circular references that could cause bugs or infinite loops in your system.

---

## What It Returns

It returns a list of found loops. Each loop is shown from the perspective of each node inside the loop.

**Output format (same as `DB::select()`):**

```
[
  {
    "root_id": 3,
    "path": "{5, 8, 3}"
  },
  {
    "root_id": 5,
    "path": "{8, 3, 5}"
  },
  {
    "root_id": 8,
    "path": "{3, 5, 8}"
  },
  ...
]
```

```
]

```

Each item means:

- `root_id`: the node where the search started
- `path`: the full loop path it found (as a string like a PostgreSQL array)

# Database Transaction Helper

## RunDbTransaction

`RunDbTransaction` is a fluent helper for running **safe and clean database transactions** in Laravel.

It handles edge cases around **rollback levels**, especially when using custom exceptions like `DefaultException` (common in our codebase).

## Why This Matters

When a `DefaultException` is thrown (one that already performs a rollback), using `DB::beginTransaction()` with a try-catch around it can lead to **double rollback errors** or broken transaction levels.

That's the main problem `RunDbTransaction` solves:

- It **detects if rollback has already happened** (e.g. inside `DefaultException`)
- Prevents **Laravel's transaction level** from going out of sync
- Lets you **catch and handle exceptions** cleanly — even continue without rollback if needed

It also works great with regular exceptions that don't trigger rollback on their own.

## ⚠ try - catch - rollback issue

```
public function run()
{
    DB::beginTransaction();
    try {
        $this->transactionBody();
        DB::commit();
    } catch (\Exception $e) {
        DB::rollback(); //ISSUE: in case of DefaultException this causes over rollback!!!
        ...
    }
}
```

```
private function transactionBody($value)
{
    if (!$value) {
        throw new DefaultException("Missing value!");
    }
    return $value
}
```

When `DefaultException` is thrown rollback is triggered by itself and `DB::rollback()` inside `catch() { ... }` over rollback the transaction and rolling back all your current work! While rollback is still needed when normal exceptions are thrown!

`RunDbTransaction` is here to solve that !

---

## □ Basic Usage

```
$successful = RunDbTransaction::create()
    ->transaction(fn() => $this->mainTransaction())
    ->run();
```

- Always call `run()` at the end.
  - Returns `true` if transaction was **committed**.
  - Returns `false` if it was **rolled back**.
- 

## ⚙️ Methods

→ `transaction(callable $callback)`

Defines the main code block that runs inside the transaction.

```
->transaction(fn() => $this->doSomething())
```

---

→ `intercept(callable $handler)`

Optional. Runs **only if an exception was thrown and rollback hasn't yet happened**.

This is where you can handle something like `DefaultException`, and still choose to **commit** the transaction.

```
->intercept(fn($e) => $this->handleBeforeRollback($e))
```

- Return **truthy** value → **Commit happens**
- Return **falsey** value → Rollback proceeds

→ `catch(callable $handler)`

Optional. Runs **after rollback**. Even if `intercept` ran but returned false.

```
->catch(fn($e) => $this->handleAfterRollback($e))
```

## Full Example with All Handlers

```
$successful = RunDbTransaction::create()  
    ->transaction(fn() => $this->mainTransaction())  
    ->intercept(fn($e) => $this->interceptException($e))  
    ->catch(fn($e) => $this->catchException($e))  
    ->run();
```

## Force Rollback Manually

You can force a rollback by returning a special flag:

```
$committed = RunDbTransaction::create()  
    ->transaction(function () {  
        $this->doSomething();  
  
        return RunDbTransaction::DO_ROLLBACK;  
    })  
    ->run();
```

- Returns `false`
- No exception handlers are triggered
- Transaction is rolled back cleanly

# ☐☐ Shortcuts

```
RunDbTransaction::runTransaction( fn() => ... )
```

Basic one-liner with safe rollback handling:

```
$success = RunDbTransaction::runTransaction( fn() => $this->mainTransaction() );
```

shorthand for:

```
$success = RunDbTransaction::create()  
☐->transaction( fn() => $this->mainTransaction() )  
    ->run();
```

Where no handler is needed just transaction and boolean `Success / Fail` result.

---

```
RunDbTransaction::runOrFail( fn() => ... )
```

Same as above, but rethrows the exception if one happens and allows it to propagate further.  
When you only need managed transaction block:

```
RunDbTransaction::runOrFail( fn() => $this->mainTransaction() );
```

Also works with `DO_ROLLBACK` – in that case it rolls back without throwing.

Also shorthand for:

```
$successful = RunDbTransaction::create()  
    ->transaction( fn() => $this->mainTransaction() )  
    //This shape is required when intercept handler is needed!  
    ->intercept( fn( $e ) => $this->handleBeforeRollback( $e ) )  
    ->catch( function ( $e ) {  
        ☐//handle catch here  
        ☐throw $e; //rethrow exception to propagate further  
    } )  
    ->run();
```

Use this shorthand when you want error to propagate further and only need save transaction block.

---

# ☐☐ Summary Table



Method	Description
<code>transaction()</code>	Required. Code block to wrap in a transaction
<code>intercept()</code>	Optional. Runs if exception is thrown <b>before rollback</b>
<code>catch()</code>	Optional. Runs <b>after rollback</b> , always gets the exception
<code>run()</code>	Executes the transaction logic
<code>runTransaction()</code>	Shorthand with rollback detection
<code>runOrFail()</code>	Shorthand that rethrows exception
<code>DO_ROLLBACK</code>	Special return value to force rollback without exception

## ☐ What About Nested Transactions?

`RunDbTransaction` also handles **multiple levels of transactions** safely — something that can easily go wrong if you're using raw `DB::beginTransaction()` and `DB::commit()` in deeply nested code.

Laravel supports **nested transactions** using savepoints, but if you're not careful, breaking out early or not committing the right number of levels can mess up the whole DB state.

```
function transactionBody() {
    $startLevel = DB::transactionLevel(); // e.g., 1
    DB::beginTransaction(); // level 2

    if ($something) {
        DB::beginTransaction(); // level 3

        // do some work...

        return; // who commits both levels? RunDbTransaction DOES !!!
    }
    ...
}
```

If you would find yourself in such scenario in the first place.

It's smarter to nest `RunDbTransaction` blocks