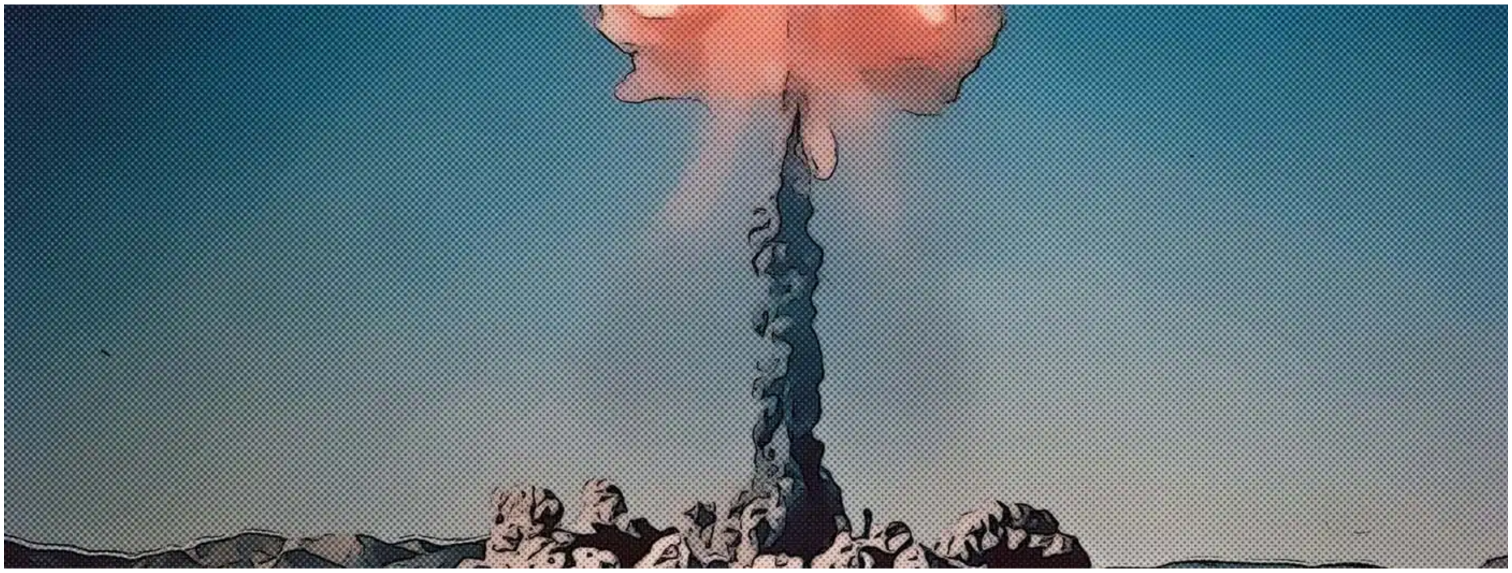
# The Case of the Exploding COALESCE Statements



James Robb (https://yellowbrick.com/author/james-robb/)

April 10, 2023 (https://yellowbrick.com/2023/04/)

**James Robb**
(https://yellowbrick.com/author/james-robb/)
Senior Software Engineer

## Introduction

I'd like to take you through an interesting bug I dug into and fixed in my first months here at Yellowbrick. I joined the planning and optimization team about four months ago, which focuses most of its time on the query-planning component of our core product, the Yellowbrick Data Warehouse (https://yellowbrick.com/yellowbrick-data-warehouse/), a MPP (Massively Parallel Processing) SQL database. The query planner makes sure queries run quickly, efficiently, and that the results of a query are correct.

The bug in question comes from a query that a customer sent to us. The query, when ran, very quickly caused the database to run out of memory. This was an interesting bug to dive into as a new team member because it gave me a lot of exposure to the different parts of the stack.

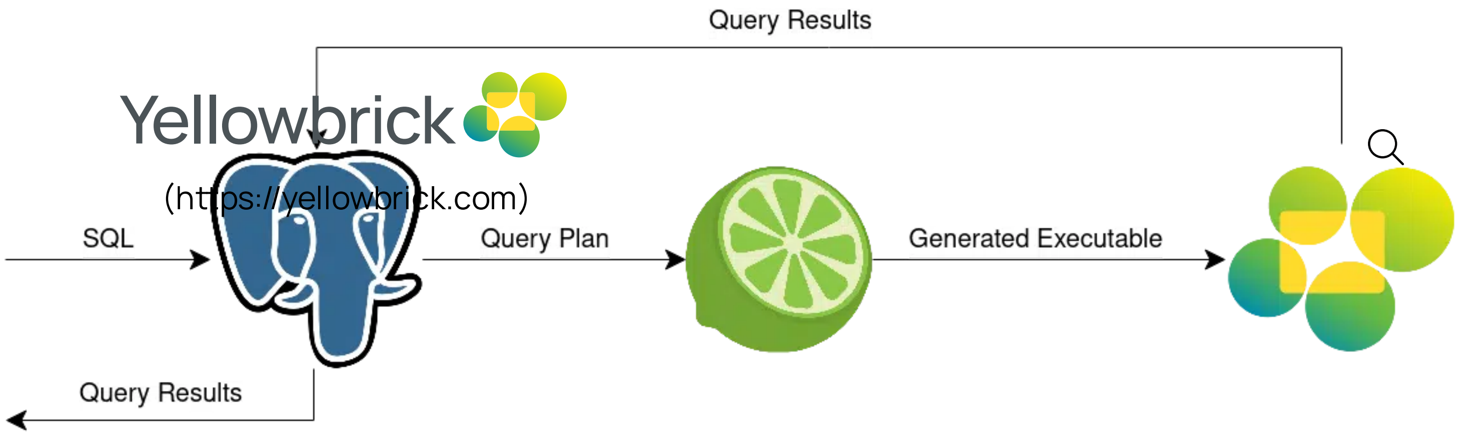
I call this bug the "Case of the Exploding COALESCE" for reasons which I hope will become obvious. In this blog post, you'll get a glimpse of the structure of the Yellowbrick stack, and an example of the abstract problems computer science students learn about manifesting in the real world.

## The Yellowbrick Stack

Before diving into the query that sparked the "Case of the Exploding COALESCE" I'd like to give you a brief overview of the Yellowbrick (https://yellowbrick.com/yellowbrick-data-warehouse/technical-overview/) stack. There are three main components to consider:

1. Postgres front-end (forked and heavily modified)
2. Yellowbrick Lime Query Compiler
3. Yellowbrick Workers

When a user submits a query, it first lands in Postgres front-end which parses the query and produces a query plan. That query plan is then serialized and sent off to Lime. Lime has several jobs, but in this context, its job is to take the serialized query plan and generate query-specific C++ code that is then compiled and sent to the workers – this all happens on-the-fly. To finally execute the query, the workers execute the compiled C++ code and send the results back to the user via Postgres.



There are definitely many more moving pieces required to realize the Yellowbrick stack in its entirety, but we now have the basic prerequisite knowledge of the stack to make sense of the bug and the rest of this blog post.

## The Explosions Begin

Now let's dive into the query that sparked the investigation. Of course, I can't share the exact query submitted to us, which was kind of large and difficult to understand without staring at it for a bit (okay more than a bit), but I can share a query that captures the exact essence of the problem.

The query contains a lot of **COALESCE** functions. As a quick reminder, a **COALESCE** function takes any number of arguments and returns the first non-null argument, and if all arguments are null then it returns null; it's a convenient way to encode basic if/else logic where nullness is concerned.

Here is the version of the query I'd like to share with you:

```
SELECT
    COALESCE(column1, 'default') as field1,
    COALESCE(field1, column2) as field2,
    ...
    COALESCE(field24, column25) as field25
FROM
    some_table;
```

It is important to point out something here that is not seen in many other databases. We can see in the query that **field2** expression references **field1** as part of its definition. The only other database I am aware of that has this convenient syntactic sugar is Teradata (https://yellowbrick.com/resources/comparisons/stacking-up-versus-teradata). For clarity, if we had to write out the definition for **field2** without this syntactic sugar, it would look like this:

```
COALESCE(COALESCE(column1, 'default'), column2)' );
```

It then follows that in the query **field25** would have **COALESCE** functions nested 25 layers deep.

When I ran the query the memory the query planner (i.e., Postgres) consumed quickly ballooned and an OOM (out of memory) event occurred – it didn't really matter how much memory I made available. Even if I made 256GB of memory available Postgres would still eat through all of it. I honestly didn't quite know where to begin at first, so I attached a debugger and hit pause while I watched Postgres' memory consumption swell. This wound up being a reasonably effective way to get going.

The debugger landed in a function that calculated the common typemod (type modifier) of the arguments in a **COALESCE** function – let's call it **select_common_typemod**. If you've worked with SQL, you've seen a typemod before. It's not common terminology though. As a basic example, when declaring a column as **varchar(10)** the typemod is 10. The reason we need to calculate the common typemod is that the arguments to a **COALESCE** function can have different typemods and we need to know the largest typemod across them to accurately estimate the memory required to execute a query. The following is the core of what **select_common_typemod** does (in pseudo-code):

```
def select_common_typemod(coalesce_expr):
        typemod = expression_typemod(coalesce_expr.first_arg())

        for arg in coalesce_expr.args():
            typemod = max(typemod, expression_typemod(arg))

        return typemod
```

And **expression_typemod** is defined as:

```
def expression_typemod(expr):
    if type_of_expression(expr) == "coalesce":
      return select_common_typemod(expr)

    # code to calculate the typemod of other expressions
    # for example, a varchar expression

    return -1
```

What a place to land because this is where the issue was, though admittedly I had to stare at it for a bit. Consider the case where we are evaluating the outer **COALESE** function in **field2**. When we pass a **COALESCE** function to **select_common_typemod** the first thing it does is call **expression_typemod** on **coalesce_expr**'s first argument, which is itself another **COALESCE** function and results in another call to **select_common_typemod**. This is a totally legit way to traverse through the nested **COALESCE** functions, but the problem is that we call **expression_typemod** a second time on **coalesce_expr**'s first argument in the loop in **select_common_typemod**. This means that for each "layer" of the nested **COALESCE** functions we duplicate the number of calls to **select_common_typemod** (via **expression_typemod**) which results in an exponential explosion of function calls.

If we now want to evaluate **field25**, we will need on the order of $2^{25}$ = 33,554,432 calls to **select_common_typemod**! This is a lot of function calls, but the stack doesn't get too deep at any given point. The OOM is a result of this large number of function calls and how Postgres handles memory and garbage collection (https://jnidzwetzki.github.io/2022/05/28/postgres-memory-context.html). In short, memory allocations are associated with "contexts" in Postgres, and all allocations associated with a context are automatically cleaned up when a context is freed. The benefit here is that a programmer doesn't need to worry about freeing each allocation as it will be cleaned up later. In our case there was a single context active for the duration of these roughly 33 million function calls, so even if one allocates just a small amount of memory in each call (which does happen, but I have excluded it from the pseudo-code) the total memory allocated quickly swells to whatever the system has available.

## The Fix (Part 1)

The fix here was simple. What I did was simply cache the common typemod of a **COALESCE** function on its associated data structure. Here's the modified pseudo-code:

```
def select_common_typemod(coalesce_expr):
    if coalesce_expr.typemod is not None:
        return coalesce.typemod

    typemod = expression_typemod(coalesce_expr.first_arg())

    for arg in coalesce_expr.args():
        typemod = max(typemod, expression_typemod(arg))

    coalesce_expr.typemod = typemod
    return typemod


def expression_typemod(expr):
    if type_of_expression(expr) == "coalesce":
        if expr.typemod is not None:
            return expr.typemod
        return select_common_typemod(expr)

    # code to calculate the typemod of other expressions
    # for example, a varchar expression
    return -1
```

With the fix in place, I was ready for happy days. The number of function calls to **select_common_typemod** was now linear in the number of coalesce statements. I ran the query again and… the query failed to execute. Postgres didn't run out of memory – that issue was fixed. I began digging into the logs of the rest of the stack and saw that Lime was generating OOM events now.

## The Fix (Part 2)

The Lime logs told me that it was running out of memory during the templating phase of the code generation. That is, we hadn't even gotten to the point where we had started to compile the code; we were generating so much code to compile that we were eating up all of the system's available memory.

This was maybe the second or third time where I needed to poke around in Lime. I figured this was still related to the deep-nesting of **COALESCE** functions so I took a look at how Lime generates code to represent **COALESCE** functions. It turns out they were represented as a series of nested if-else statements. As an example, the following is the pseudo-code for how **COALESCE(column1, column2)** was represented:

```
IF column1.not_null() THEN
    return column1
ELSE
    IF column2.not_null() THEN
       return column2
    ELSE
       return NULL
```

This seemed fine at first glance, but things got a bit hairier when we needed to represent nested **COALESCE** functions. To fully appreciate where it gets weird, let's look at the same pseudo-code as before, but in its more generic template-like form:

```
IF (coalesce_argument_1).not_null() THEN
    RETURN (coalesge_argument_1)
ELSE
    (include IF template for coalesge_argument_2)
```

That means the representation of **COALESCE(COALESCE(column1, 'default'), column2)** was:

```
IF
    (
    IF column1.not_null() THEN
        RETURN column1
    ELSE
        RETURN 'default'
    ).not_null()
THEN
    IF column1.not_null() THEN
        RETURN column1
    ELSE
        RETURN 'default'
ELSE
    IF column2.not_null THEN
        RETURN column2
    ELSE
        RETURN NULL
```

```
x = (
    IF column1.not_null() THEN
        RETURN column1
    ELSE
        RETURN 'default'
)

IF x.not_null() THEN
    RETURN x

IF column2.not_null() THEN
    RETURN column2

RETURN NULL
```

Another case of duplication! By outputting the same text in the condition and body of the if statement, we see yet another exponential explosion, except this time in the amount of text produced instead of the number of function calls.

The solution was again straightforward once I could see the issue. I decided to do two things to remedy the situation: removed the duplication and unwound the nested if-statements. The following is what the same COALESCE function looked like after my changes:

With this code fix in place, I ran the query again and… success! I got back the right results quickly and without generating any OOM events (or any noticeable increase in memory for that matter).

Happy days had arrived!

## Concluding Thoughts

In solving the "Case of the Exploding COALESCE" we dove into a real-world example of how things that are hard to see when looking at the source code result in big issues for the "right" input. We found two instances of logic that led to an exponential blow-up in memory consumption and solved them with memoization and deduplication, respectively. It turns out the problems we learned about in our algorithms classes really do pop up in all sorts of places in the wild.

(https://yellowbrick.com/blog/modernization/th-to-analytics-cloud-native-data-warehousing/)

There is one last thing I would also like to talk about. When reading this, one might have asked themselves, "Who would actually write a query like this?" Who indeed – as it is a bit unusual, at least to a human. In many instances like this no one did. All sorts of engineers, data scientists, and others use a variety of tools to generate SQL queries. In previous roles, I've worked on projects that use ORMs

ar

qu

sy

th

ca

Th

## Related Posts

Engineering (https://yellowbrick.com/category/blog/engineering/)

(https://yellowbrick.com/blog/engineering/lldb-extension-for-structure-visualization/)

March 2, 2023 (https://yellowbrick.com/2023/03/02/)

**Low-level Debugger (LLDB) Extension for Structure Visualization (https://yellowbrick.com/blog/engineering/lldb-extension-for-structure-visualization/)**

(https://yellowbrick.com/blog/engineering/methods-and-techniques-for-fast-and-efficient-logic-simulation/)

February 27, 2023 (https://yellowbrick.com/2023/02/27/)

**Methods and Techniques for Fast and Efficient Logic Simulation (https://yellowbrick.com/blog/engineering/methods-and-techniques-for-fast-and-efficient-logic-simulation/)**

(https://yellowbrick.com/blog/engineering/stories-from-the-field-the-devil-is-in-the-to_number-details/)

February 1, 2023 (https://yellowbrick.com/2023/02/01/)

**Stories From the Field: The Devil Is In the to_number Details (https://yellowbrick.com/blog/engineering/stories-from-the-field-the-devil-is-in-the-to_number-details/)**

## Meet Our Authors (/our-authors/)

(https://yellowbrick.com)

(https://yellowbrick.com/author/alice-russell/profile/)

(https://yellowbrick.com/author/alice-russell/profile/) Alice Russell

(https://yellowbrick.com/author/allen-holmes/profile/)

(https://yellowbrick.com/author/allen-holmes/profile/) Allen Holmes

(https://yellowbrick.com/author/bob-rumsby/profile/)

(https://yellowbrick.com/author/bob-rumsby/profile/) Bob Rumsby

(https://yellowbrick.com/author/bryson-boatwright/profile/)

(https://yellowbrick.com/author/bryson-boatwright/profile/) Bryson Boatwright

Yellowbrick
(https://yellowbrick.com)

(https://yellowbrick.com/author/christopher-edgar/profile/)

(https://yellowbrick.com/author/christopher-edgar/profile/) Christopher Edgar

(https://yellowbrick.com/author/container-environments/profile/)

(https://yellowbrick.com/author/container-environments/profile/) Ryan Atkinson

(https://yellowbrick.com/author/david-tran/profile/)

(https://yellowbrick.com/author/david-tran/profile/) David Tran

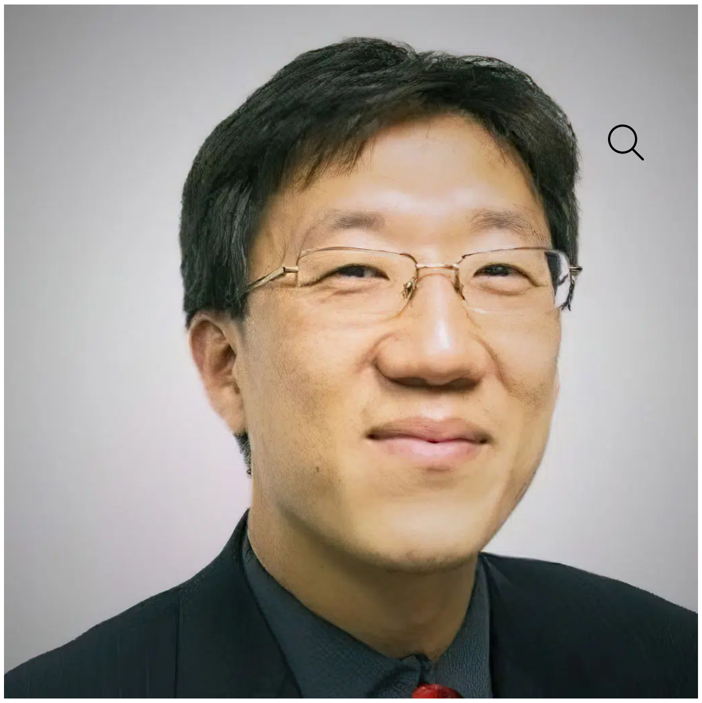
(https://yellowbrick.com/author/drew-gillies/profile/)

(https://yellowbrick.com/author/drew-gillies/profile/) Drew Gillies

(https://yellowbrick.com)

(https://yellowbrick.com/author/gary-west/profile/)

(https://yellowbrick.com/author/gary-west/profile/) Gary West

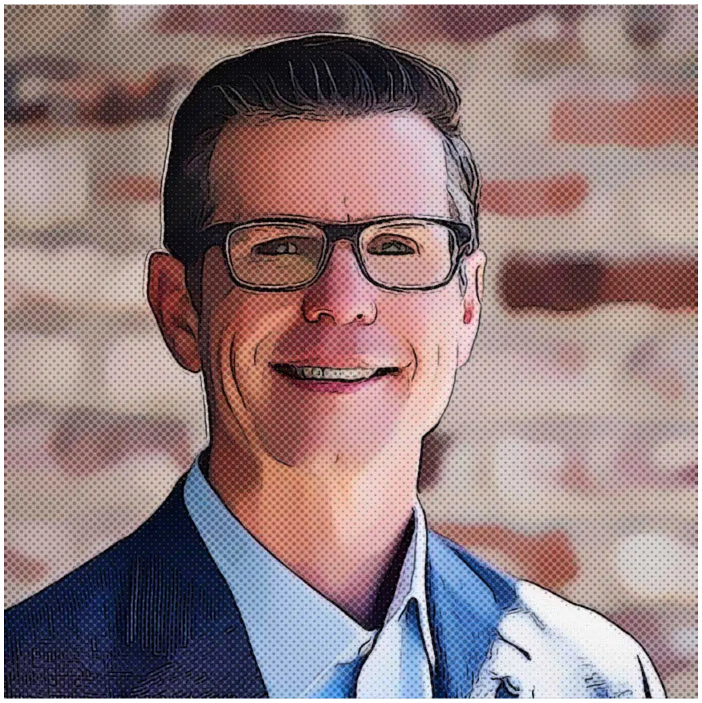


(https://yellowbrick.com/author/hyoun-park/profile/)

(https://yellowbrick.com/author/hyoun-park/profile/) Hyoun Park

(https://yellowbrick.com/author/james-robb/profile/)

(https://yellowbrick.com/author/james-robb/profile/) James Robb



(https://yellowbrick.com/author/jason-snodgress/profile/)

(https://yellowbrick.com/author/jason-snodgress/profile/) Jason Snodgress

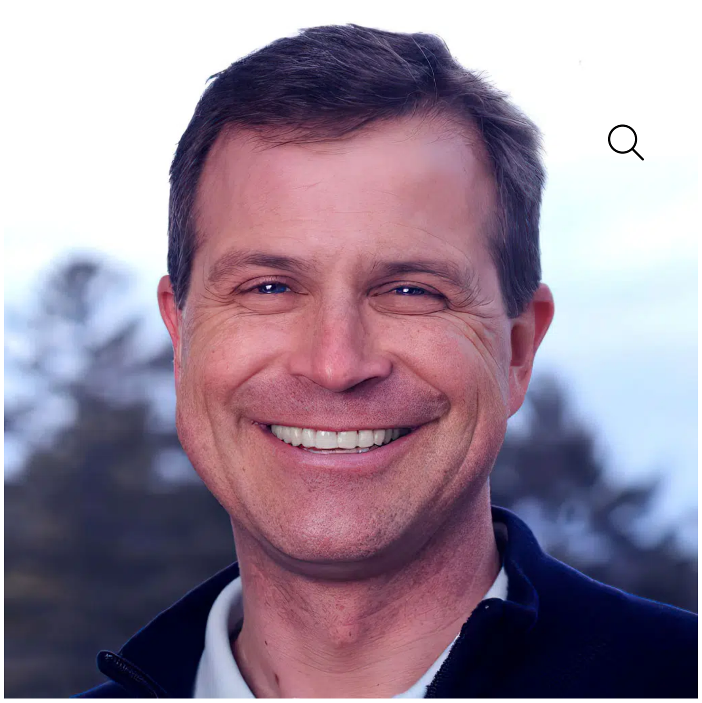
Yellowbrick

(https://yellowbrick.com)

(https://yellowbrick.com/author/keerti-

garg/profile/)

(https://yellowbrick.com/author/keerti-garg/profile/) Keerti Garg



(https://yellowbrick.com/author/kevin-

petrie/profile/)

(https://yellowbrick.com/author/kevin-petrie/profile/) Kevin Petrie

(https://yellowbrick.com/author/mark-

cusack/profile/)

(https://yellowbrick.com/author/mark-cusack/profile/) Mark Cusack

(https://yellowbrick.com/author/matt-

aslett/profile/)

(https://yellowbrick.com/author/matt-aslett/profile/) Matt Aslett

(https://yellowbrick.com)

(https://yellowbrick.com/author/maxence-menager/profile/)

(https://yellowbrick.com/author/maxence-menager/profile/) Maxence Menage

(https://yellowbrick.com/author/mike-ferguson/profile/)

(https://yellowbrick.com/author/mike-ferguson/profile/) Mike Ferguson

(https://yellowbrick.com/author/neil-carson/profile/)

(https://yellowbrick.com/author/neil-carson/profile/) Neil Carson

(https://yellowbrick.com/author/nick-cox/profile/)

(https://yellowbrick.com/author/nick-cox/profile/) Nick Cox

(https://yellowbrick.com/author/paritosh-kulkarni/profile/)

(https://yellowbrick.com/author/paritosh-kulkarni/profile/) Paritosh Kulkarni

(https://yellowbrick.com/author/quality-engineer/profile/)

(https://yellowbrick.com/author/quality-engineer/profile/) Henry Cate

(https://yellowbrick.com/author/ramnath-sai-sagar/profile/)

(https://yellowbrick.com/author/ramnath-sai-sagar/profile/) Ramnath Sai Sagar

(https://yellowbrick.com/author/richard-soundy/profile/)

(https://yellowbrick.com/author/richard-soundy/profile/) Richard Soundy

(https://yellowbrick.com)

(https://yellowbrick.com/author/software-engineer/profile/)

(https://yellowbrick.com/author/software-engineer/profile/) Morgane Rouvroy

(https://yellowbrick.com/author/tim-young/profile/)

(https://yellowbrick.com/author/tim-young/profile/) Tim Young

(https://yellowbrick.com/author/torben-mathiasen/profile/)

(https://yellowbrick.com/author/torben-mathiasen/profile/) Torben Mathiasen

(https://yellowbrick.com/author/umair/profile/)

(https://yellowbrick.com/author/umair/profile/) Umair Wahee

## Get the latest Yellowbrick News & Insights

Enter Business Email*

This reCAPTCHA is for testing purposes only. Please report to the site admin if you are seeing this.
protected by reCAPTCHA
Privacy - Terms

Yellowbrick (https://yellowbrick.com)

Subscribe

(https://yellowbrick.com/blog/cloud/a-direct-path-to-analytics-cloud-native-data-warehousing/)
Cloud (https://yellowbrick.com/category/blog/cloud/)

Yellowbrick April 12, 2023

## A "Direct" Path to Analytics: Cloud-Native Data Warehousing

https://youtu.be/1fBkyl1fmlI In today's data-driven world, businesses are grappling with vast...

(https://yellowbrick.com/blog/engineering/the-case-of-the-exploding-coalesce/)
Engineering (https://yellowbrick.com/category/blog/engineering/)

(https://yellowbrick.com/author/workloadmanagement/profile/) (https://yellowbrick.com/author/yellowbrick/profile/)
James Robb April 10, 2023
(https://yellowbrick.com/author/workloadmanagement/profile/) (https://yellowbrick.com/author/yellowbrick/profile/) Kurt Westerfeld

## The Case of the Exploding COALESCE Statements

IntroductionI'd like to take you through an interesting bug I...

(https://yellowbrick.com/blog/modernization/still-doing-battle-with-your-monolithic-multi-terabyte-data-warehouse/)
Modernization (https://yellowbrick.com/category/blog/modernization/)

Umair Waheed April 7, 2023

## Still Doing Battle With Your Monolithic Multi-terabyte Data...

The Challenges of Running a Monolithic Multi-terabyte Data Warehouse Are...

## Product (/yellowbrick-data-warehouse/)

Why Yellowbrick(https://yellowbrick.com/why-yellowbrick/)

New

Yellowbrick Data Warehouse (https://yellowbrick.com/yellowbrick-data-warehouse/)

Technical Overview

(https://yellowbrick.com/yellowbrick-data-warehouse/technical-overview/)

(https://yellowbrick.com)

Product Advantages

(https://yellowbrick.com/yellowbrick-data-warehouse/product-advantages/)

Pricing(https://yellowbrick.com/yellowbrick-data-warehouse/pricing/)

Customers(https://yellowbrick.com/yellowbrick-customers/)

## Solutions (/modernize-your-database/)

Modernize Your Databases(https://yellowbrick.com/modernize-your-database/)

Embrace the Cloud(https://yellowbrick.com/embrace-the-cloud/)

Control Spend(https://yellowbrick.com/reduce-cloud-data-warehouse-costs/)

Financial Services(https://yellowbrick.com/financial-services/)

Hedge Funds(https://yellowbrick.com/hedge-funds/)

Insurance(https://yellowbrick.com/insurance/)

Telecommunications(https://yellowbrick.com/telecommunications/)

## Resources (/resources/)

Resource Library(/resources/)

Blog(https://yellowbrick.com/resources/blog/)

Events(https://yellowbrick.com/resources/events/)

Product Documentation(https://yellowbrick.com/resources/product-documentation/)

Customer Center

(https://support.yellowbrick.com/hc/en-us/restricted?

return_to=https%3A%2F%2Fsupport.yellowbrick.com%2Fhc%2Fen-us)

Brand Standards(https://yellowbrick.com/brand-standards/)

## Partners (/yellowbrick-partners)

Partners(https://yellowbrick.com/yellowbrick-partners/)

## Company (/about-yellowbrick/)

About Yellowbrick(https://yellowbrick.com/about-yellowbrick/)

Press Releases(https://yellowbrick.com/about-yellowbrick/press-releases/)

In the News(https://yellowbrick.com/about-yellowbrick/in-the-news/)

Careers(https://yellowbrick.com/careers/)

Contact Us(https://yellowbrick.com/about-yellowbrick/contact-us-at-yellowbrick/)

(https://yellowbrick.com)

**US**

Phone 877.492.3282 (tel:8774923282)

info@yellowbrick.com (mailto:info@yellowbrick.com?subject=Yellowbrick Website Inquiry)

sales@yellowbrick.com (mailto:sales@yellowbrick.com?subject=Yellowbrick Website Inquiry)

**US Headquarters**

660 W. Dana Street

Mountain View, CA 94041

**International**

Phone +1.650.687.0896 (tel:16506870896)

info@yellowbrick.com (mailto:info@yellowbrick.com?subject=Yellowbrick Website Inquiry)

sales@yellowbrick.com (mailto:sales@yellowbrick.com?subject=Yellowbrick Website Inquiry)

**Europe Headquarters**

60 Trafalgar Square

London, WC2N 5DS UK

**Available On:**

Yellowbrick

(https://yellowbrick.com)

(https://aws.amazon.com/marketplace/pp/prodview-qovhbunvu3q4y?sr=0-1&ref_=beagle&applicationId=AWSMPContessa)

(https://twitter.com/yellowbrickdata)

(https://www.linkedin.com/company/yellowbrickdata/)