



# metis

**The missing  
chapter in your CI  
playbook:  
database  
guardrails**



Adam Furmanek



<https://www.metisdata.io/>

**Be proactive and push to the left!  
Prevent bad code from reaching production.**

**Know the context to find the root cause!  
Monitor and troubleshoot.**

| How do you know your code will work in production?



# | What can go wrong?



**Deployment may go wrong:**

- Windows vs Linux
- Permissions
- Connection strings



**Code may not work well:**

- Different locale
- Bugs
- "Works on my machine"



**Application may not handle the load:**

- Too much data
- Different distribution
- Edge cases

# | CI/CD



# CI/CD

## Continuous Integration:

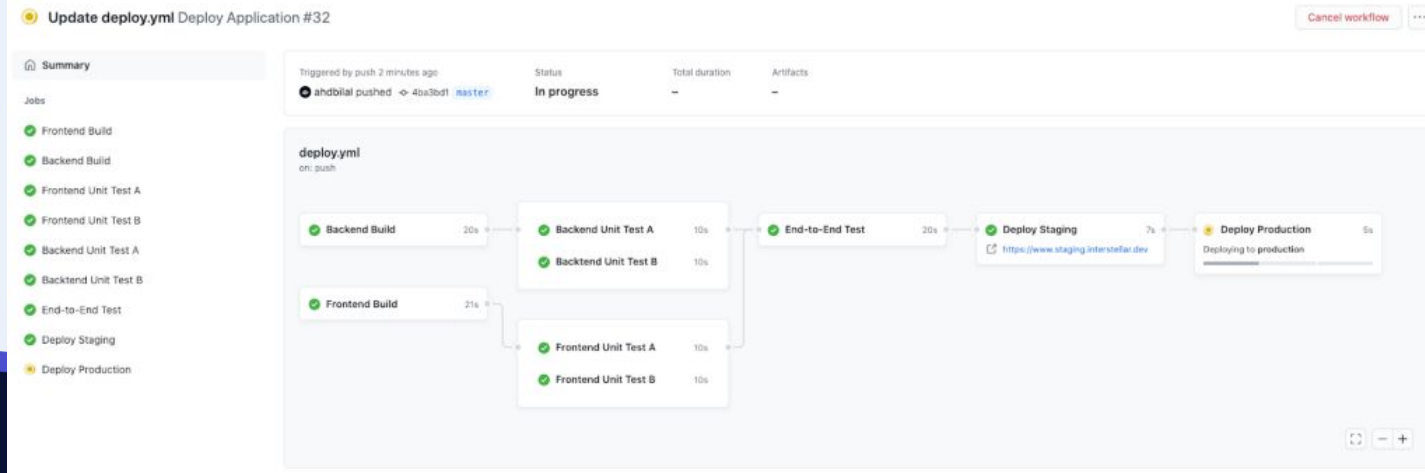
- Changes are frequently merged into the main branch
- Each change is automatically built and tested

## Continuous Delivery:

- Each change is pushed to a non-production environment (staging, test, etc.)
- Change is not deployed to production automatically

## Continuous Deployment:

- Like Continuous Delivery but the change is deployed to production automatically



# So what can go wrong?

We'll cover:

Databases

ORMs

Lack of context



# | Problems with databases

Slow queries.

Inaccurate statistics.

Incompatible changes in schema.

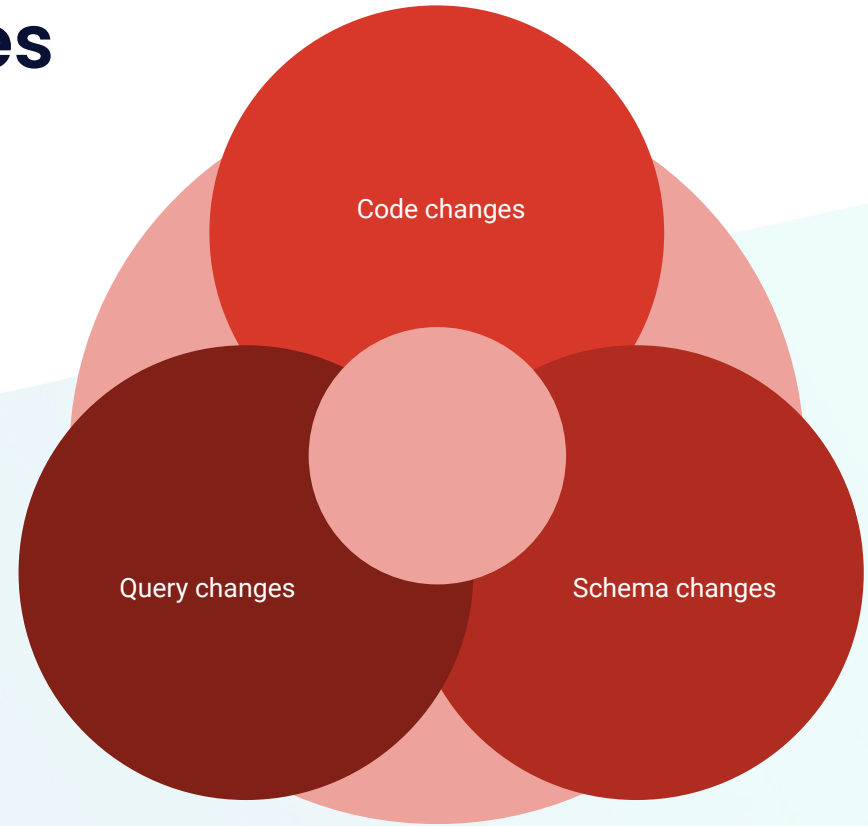
Bugs.

Missing indexes.

Data quality.

Configuration.

Locks.



# | Slow queries

```
const user = repository.get("user")
  .where("user.id = 123")
  .leftJoin("user.details", "user_details_table")
  .leftJoin("user.pages", "pages_table")
  .leftJoin("user.texts", "texts_table")
  .leftJoin("user.questions", "questions_table")
  .leftJoin("user.reports", "reports_table")
  .leftJoin("user.location", "location_table")
  .leftJoin("user.peers", "peers_table")
  .getOne();
return user;
```

```
SELECT *
FROM users AS user
LEFT JOIN user_details_table AS detail ON detail.user_id = user.id
LEFT JOIN pages_table AS page ON page.user_id = user.id
LEFT JOIN texts_table AS text ON text.user_id = user.id
LEFT JOIN questions_table AS question ON question.user_id = user.id
LEFT JOIN reports_table AS report ON report.user_id = user.id
LEFT JOIN location_table AS location ON location.user_id = user.id
LEFT JOIN peers_table AS peer ON peer.user_id = user.id
WHERE user.id = '123'
```

This reads ~300k rows and runs for 25 seconds.

# Slow queries

```
const userQuery = repository.get("user").where("user.id = 123")
const user = userQuery().getOne();
const details = userQuery()
  .leftJoin("user.details", "user_details_table")
  .getOne();
const pages = userQuery()
  .leftJoin("user.pages", "pages_table")
  .getOne();
const texts = userQuery()
  .leftJoin("user.texts", "texts_table")
  .getOne();
const questions = userQuery()
  .leftJoin("user.questions", "questions_table")
  .getOne();
const reports = userQuery()
  .leftJoinAndSelect("user.reports", "reports_table")
  .getOne();
const location = userQuery()
  .leftJoin("user.location", "location_table")
  .getOne();
const peers = userQuery()
  .leftJoin("user.peers", "peers_table")
  .getOne();
return {
  ...user,
  ...details,
  ...pages,
  ...texts,
  ...questions,
  ...reports,
  ...location
  ...peers
};
```

```
SELECT *
FROM users AS user
WHERE user.id = '123'
```

```
SELECT *
FROM users AS user
LEFT JOIN user_details_table AS detail ON detail.user_id=user.id
WHERE user.id = '123'
```

```
SELECT *
FROM users AS user
LEFT JOIN pages_table AS page ON page.user_id=user.id
WHERE user.id = '123'
```

```
SELECT *
FROM users AS user
LEFT JOIN texts_table AS text ON text.user_id=user.id
WHERE user.id = '123'
```

```
SELECT *
FROM users AS user
LEFT JOIN questions_table AS question ON question.user_id=user.id
WHERE user.id = '123'
```

```
SELECT *
FROM users AS user
LEFT JOIN reports_table AS report ON report.user_id=user.id
WHERE user.id = '123'
```

```
SELECT *
FROM users AS user
LEFT JOIN location_table AS location ON location.user_id=user.locationId
WHERE user.id = '123'
```

```
SELECT *
FROM users AS user
LEFT JOIN peers_table AS peer ON peer.user_id=user.clientId
WHERE user.id = '123'
```

# Slow queries

```
-- 7925812 rows  
SELECT COUNT(*)  
FROM boarding_passes
```

```
-- 13 seconds
```

```
WITH cte_performance AS (  
    SELECT *, MD5(MD5(ticket_no)) AS double_hash  
    FROM boarding_passes  
)  
SELECT COUNT(*)  
FROM cte_performance AS C1  
JOIN cte_performance AS C2 ON C2.ticket_no = C1.ticket_no AND C2.flight_id = C1.flight_id AND C2.boarding_no = C1.boarding_no  
JOIN cte_performance AS C3 ON C3.ticket_no = C1.ticket_no AND C3.flight_id = C1.flight_id AND C3.boarding_no = C1.boarding_no  
WHERE  
    C1.double_hash = '525ac610982920ef37b34aa56a45cd06'  
    AND C2.double_hash = '525ac610982920ef37b34aa56a45cd06'  
    AND C3.double_hash = '525ac610982920ef37b34aa56a45cd06'
```

```
-- 8 seconds
```

```
SELECT COUNT(*)  
FROM boarding_passes AS C1  
JOIN boarding_passes AS C2 ON C2.ticket_no = C1.ticket_no AND C2.flight_id = C1.flight_id AND C2.boarding_no = C1.boarding_no  
JOIN boarding_passes AS C3 ON C3.ticket_no = C1.ticket_no AND C3.flight_id = C1.flight_id AND C3.boarding_no = C1.boarding_no  
WHERE  
    MD5(MD5(C1.ticket_no)) = '525ac610982920ef37b34aa56a45cd06'  
    AND MD5(MD5(C2.ticket_no)) = '525ac610982920ef37b34aa56a45cd06'  
    AND MD5(MD5(C3.ticket_no)) = '525ac610982920ef37b34aa56a45cd06'
```

# | Incompatible changes in schema

## Adding a column

- May cause issues when we use `SELECT *`
- May cause table reorganization because of lack of space (and outage in result)

## Dropping a column

- Nearly never safe

## Altering the column type

- May change the representation, this depends on the ORM and the driver
- May require some extensions installed to the database engine
- May cause table reorganization

# | Missing Indexes

May cause scanning whole table instead of getting rows directly.

May cause using inefficient JOIN strategy (nested loop instead of hash join or merge join).

## Index

- Created automatically with a primary key
- May be created on demand
- May store one or more columns
- Stores data in an order, so it's easy to do binary search

## Some index types

- B-Tree
- Hash index
- GIS-based (for geolocation)
- GIN (inverted indexes)

# | Too many indexes

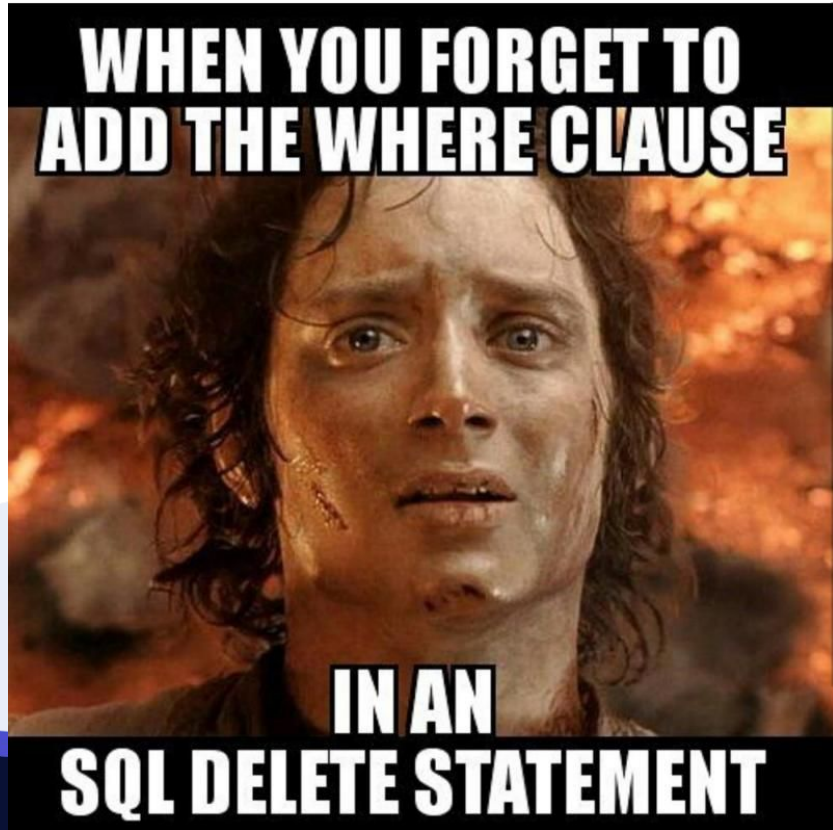
Indexes are not free

- They store data in a specific order that needs to be maintained over time
- They need to copy the data on the side to build additional dictionaries
- Updating one row may cause an update in multiple indexes
- **Do not index blindly!** Evaluate if the performance increases





# Bugs



## Halloween problem

Phenomena when updating the row causes a change in the physical location of the row.

The same row may be modified multiple times.

```
UPDATE employees  
SET salary= salary + (salary * 10 / 100)  
WHERE salary < 10000
```



# ORM challenges - n+1 selects

Problem:

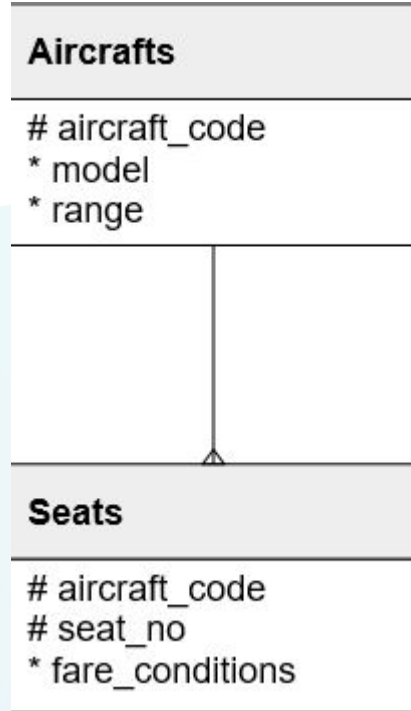
```
aircrafts = aircrafts.load();
for(aircraft in aircrafts) {
  seatsCount = aircraft.seats.size;
}
```

This generates:

```
SELECT * FROM aircrafts;
SELECT * FROM seats WHERE aircraft_code = 1
SELECT * FROM seats WHERE aircraft_code = 2
SELECT * FROM seats WHERE aircraft_code = 3
...
```

However, this could be done in one query:

```
SELECT * FROM aircrafts
LEFT JOIN seats ON seats.aircraft_code = aircrafts.aircraft_code
```



# ORM challenges – joins

Normalization leads to multiple joins that may be slow.

We may need to decompose these queries manually.

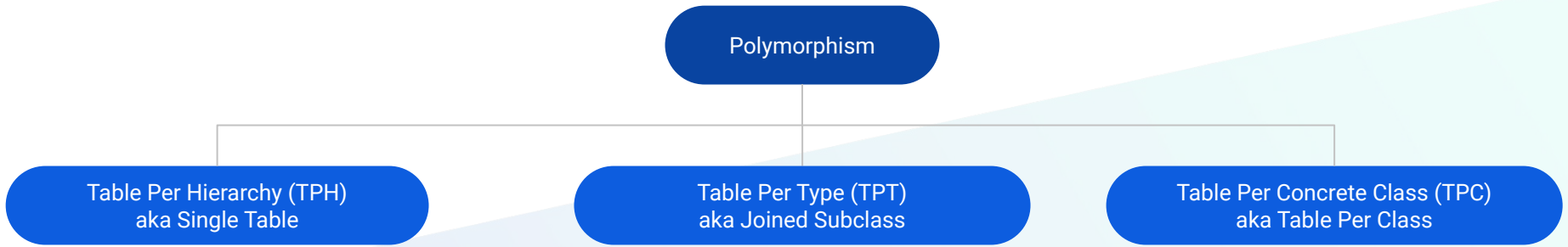
We may need to rework our domain model.

We may need to change bounded contexts.

```
const user = repository.get("user")
  .where("user.id = 123")
  .leftJoin("user.details", "user_details_table")
  .leftJoin("user.pages", "pages_table")
  .leftJoin("user.texts", "texts_table")
  .leftJoin("user.questions", "questions_table")
  .leftJoin("user.reports", "reports_table")
  .leftJoin("user.location", "location_table")
  .leftJoin("user.peers", "peers_table")
  .getOne();
return user;
```

```
SELECT *
FROM users AS user
LEFT JOIN user_details_table AS detail ON detail.user_id=user.id
LEFT JOIN pages_table AS page ON page.user_id=user.id
LEFT JOIN texts_table AS text ON text.user_id=user.id
LEFT JOIN questions_table AS question ON question.user_id=user.id
LEFT JOIN reports_table AS report ON report.user_id=user.id
LEFT JOIN location_table AS location ON location.user_id=user.locationId
LEFT JOIN peers_table AS peer ON peer.user_id=user.clientId
WHERE user.id = '123'
```

# ORM challenges - polymorphism



# ORM challenges - data types

SQL	OOP
Spatial data	Pair of numbers
Binary data	Array of bytes
varchar	String
decimal	float/double

What if your database is used by multiple **heterogeneous** applications?

# ORM challenges – lack of visibility

## Transaction isolation level

- Each transaction has a level (SERIALIZABLE, READ COMMITTED, etc.)
- What's the default?
- Can you change it?

## Transaction scope

- When is transaction started? When does it end?
- Do you have nested transactions?

## Commit/rollback

- Who controls how things are committed and rolled back?
- What happens in case of errors?

## Caching

- Is the data cached?
- Does it work with parallel connections?
- What about sticky sessions/

## Pooling

- Do you have a connection pool?
- Will it scale well?
- How often do you recycle the connection?

## Query hints

- How do you make sure indexes are used?
- How do you configure join strategy?

# | ORM challenges – migrations

How do you define your migrations

- SQL files with CREATE TABLE...
- Code first with ORM model
- Or maybe you already have the database?

How do you track which things were executed

- Keep another table with history
- Make sure changes are idempotent
- You run them manually

How do you roll back

- Up + Down methods

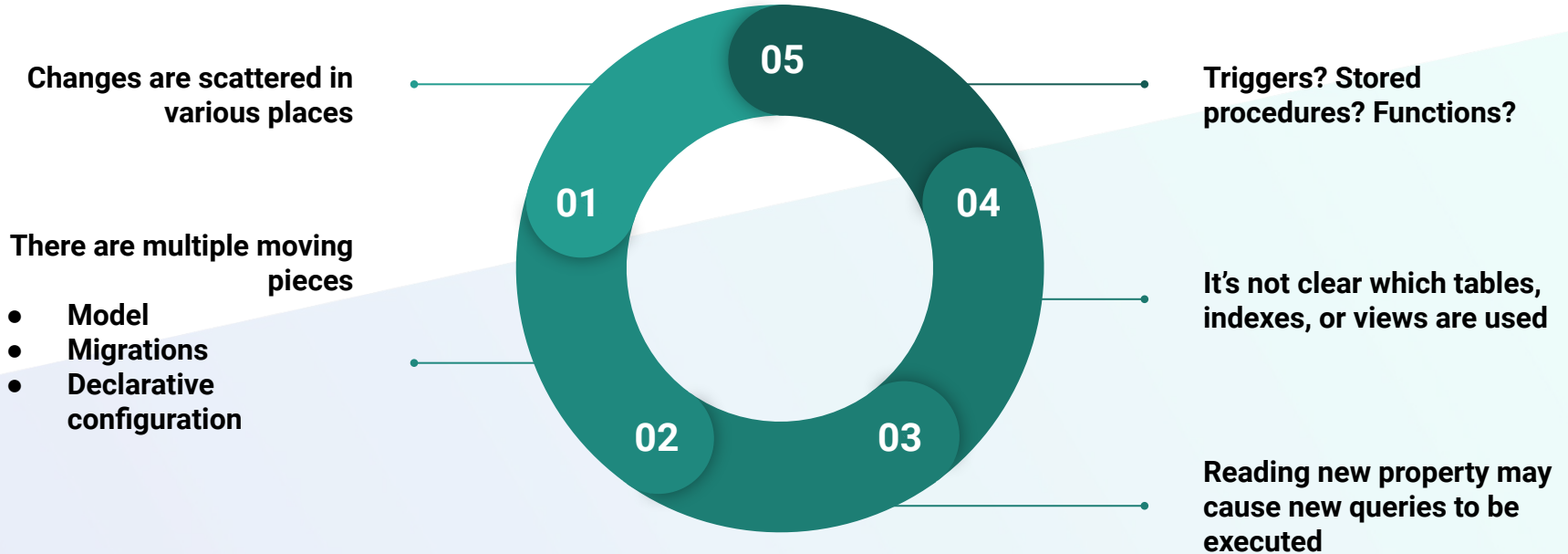
What if there are multiple heterogeneous applications?

What if your ORM creates tables automatically?

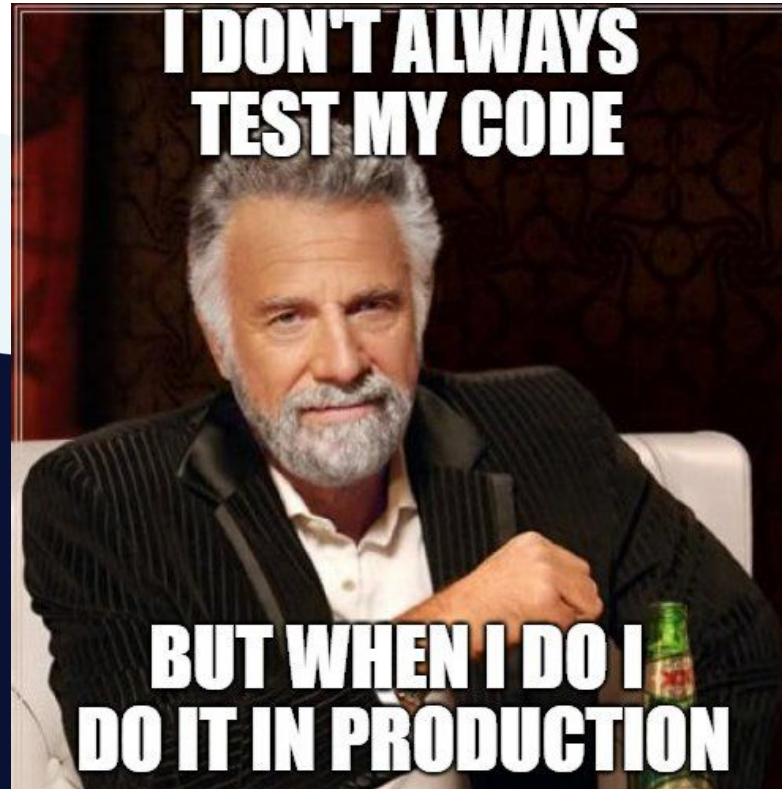
How do you deal with migrations in unit tests?

How do you fix errors which you spot later on?

# ORM challenges - reviews



# | Know the context to find the root cause





# | Executing the query

## Parser

Query is parsed into an Abstract Syntax Tree (AST).  
This allows to manipulate the query mechanically.

## Rewriter

Query is rewritten to a standard form.  
This makes processing the query easier.

## Planner

A plan is prepared. It contains details of how to read data, how to join tables, how to filter rows, etc.

## Executor

Finally, the query is physically executed.

# Anatomy of an SQL query

```
EXPLAIN
SELECT *
FROM flights AS f
LEFT JOIN aircrafts_data AS ad ON ad.aircraft_code = f.aircraft_code
LEFT JOIN seats AS s ON s.aircraft_code = f.aircraft_code
LEFT JOIN ticket_flights AS tf ON tf.flight_id = f.flight_id
LEFT JOIN boarding_passes AS bp ON bp.flight_id = f.flight_id
LEFT JOIN tickets AS t ON t.ticket_no = tf.ticket_no
LEFT JOIN bookings AS b ON b.book_ref = t.book_ref
LEFT JOIN airports AS a ON a.airport_code = f.departure_airport
WHERE f.flight_id = 1676
```

## QUERY PLAN

```
Nested Loop Left Join (cost=6.92..245675.96 rows=12012 width=412)
  Join Filter: (bp.flight_id = f.flight_id)
  -> Nested Loop Left Join (cost=1.28..244888.01 rows=77 width=387)
    Join Filter: (s.aircraft_code = f.aircraft_code)
    -> Nested Loop Left Join (cost=1.28..244849.88 rows=1 width=372)
      Join Filter: (ml.airport_code = f.departure_airport)
      -> Nested Loop Left Join (cost=1.28..244792.02 rows=1 width=273)
        -> Nested Loop Left Join (cost=0.85..244791.55 rows=1 width=252)
          -> Nested Loop Left Join (cost=0.42..244783.10 rows=1 width=148)
            Join Filter: (tf.flight_id = f.flight_id)
            -> Nested Loop Left Join (cost=0.42..9.64 rows=1 width=115)
              Join Filter: (ad.aircraft_code = f.aircraft_code)
              -> Index Scan using flights_pkey on flights f (cost=0.42..8.44 rows=1 width=63)
                Index Cond: (flight_id = 1676)
              -> Seq Scan on aircrafts_data ad (cost=0.00..1.09 rows=9 width=52)
              -> Seq Scan on ticket_flights tf (cost=0.00..244772.15 rows=105 width=33)
                Filter: (flight_id = 1676)
            -> Index Scan using tickets_pkey on tickets t (cost=0.43..8.45 rows=1 width=104)
              Index Cond: (ticket_no = tf.ticket_no)
          -> Index Scan using bookings_pkey on bookings b (cost=0.43..0.47 rows=1 width=21)
            Index Cond: (book_ref = t.book_ref)
        -> Seq Scan on airports_data ml (cost=0.00..56.56 rows=104 width=99)
        -> Seq Scan on seats s (cost=0.00..21.39 rows=1339 width=15)
      -> Materialize (cost=5.64..608.16 rows=156 width=25)
        -> Bitmap Heap Scan on boarding_passes bp (cost=5.64..607.38 rows=156 width=25)
          Recheck Cond: (flight_id = 1676)
          -> Bitmap Index Scan on boarding_passes_flight_id_seat_no_key (cost=0.00..5.60 rows=156 width=0)
            Index Cond: (flight_id = 1676)
```

# Anatomy of an SQL query

Each plan consists of **nodes**.

Nodes have **costs** associated with them.

- Cost is an arbitrary measure of “how hard it is to get the whole dataset”

Most important parts are:

- **Scans** - Sequential Scan Index Scan, index-Only scan
- **Joins** - Nested Loop, Hash, Merge
- Others: Limit, Materialize, Sort

<https://www.pgmustard.com/docs/explain>

QUERY PLAN

```
Nested Loop Left Join (cost=6.89..175713.37 rows=11704 width=411)
Join Filter: (b.flight_id = f.flight_id)
-> Nested Loop Left Join (cost=1.28..174945.21 rows=77 width=386)
Join Filter: (s.aircraft_code = f.aircraft_code)
-> Nested Loop Left Join (cost=1.28..174907.08 rows=1 width=371)
Join Filter: (ml.airport_code = f.departure_airport)
-> Nested Loop Left Join (cost=1.28..174849.22 rows=1 width=272)
-> Nested Loop Left Join (cost=0.85..174848.75 rows=1 width=251)
-> Nested Loop Left Join (cost=0.42..174840.30 rows=1 width=147)
Join Filter: (tf.flight_id = f.flight_id)
-> Nested Loop Left Join (cost=0.42..9.64 rows=1 width=115)
Join Filter: (ad.aircraft_code = f.aircraft_code)
-> Index Scan using flights_pkey on flights f (cost=0.42..8.44 rows=1 width=63)
Index Cond: (flight_id = 1676)
-> Seq Scan on aircrafts_data ad (cost=0.00..1.09 rows=9 width=52)
-> Seq Scan on ticket_flights tf (cost=0.00..174829.35 rows=105 width=32)
Filter: (flight_id = 1676)
-> Index Scan using tickets_pkey on tickets t (cost=0.43..8.45 rows=1 width=104)
Index Cond: (ticket_no = tf.ticket_no)
-> Index Scan using bookings_pkey on bookings b (cost=0.43..0.47 rows=1 width=21)
Index Cond: (book_ref = t.book_ref)
-> Seq Scan on airports_data ml (cost=0.00..56.56 rows=104 width=99)
-> Seq Scan on seats s (cost=0.00..21.39 rows=1339 width=15)
-> Materialize (cost=5.61..592.98 rows=152 width=25)
-> Bitmap Heap Scan on boarding_passes bp (cost=5.61..592.22 rows=152 width=25)
Recheck Cond: (flight_id = 1676)
-> Bitmap Index Scan on boarding_passes_flight_id_seat_no_key (cost=0.00..5.57 rows=152 width=0)
Index Cond: (flight_id = 1676)
```

# | Observability



We need:

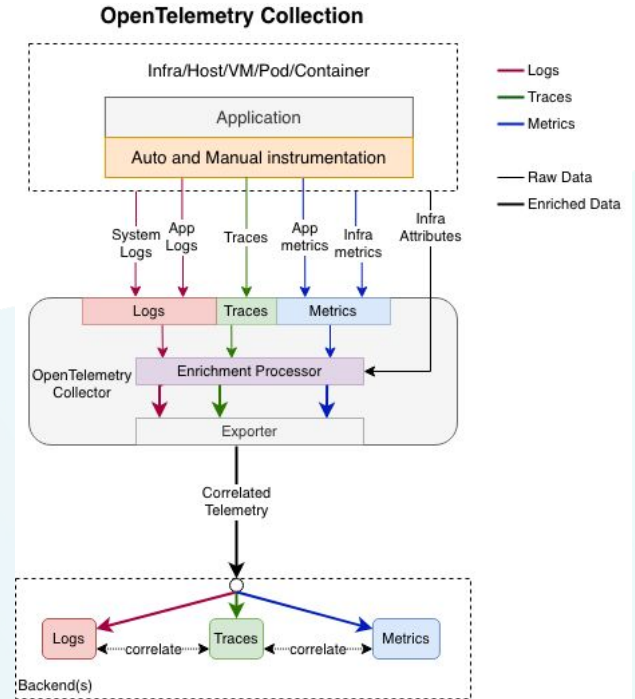
- Logs
- Traces
- Metrics

We face multiple challenges:

- Heterogeneous applications
- Correlations
- Extensibility

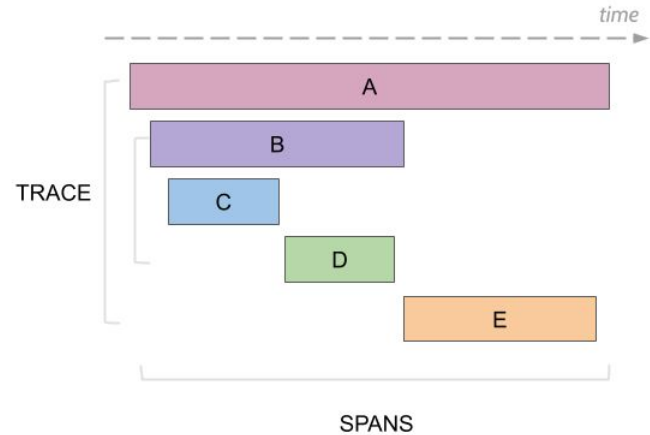
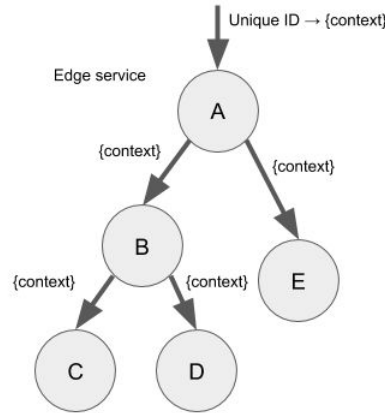
# OpenTelemetry (OTel)

- Set of SDKs for instrumentation
- Supported by Cloud Native Computing Foundation (CNCF)
- This is a standard + a set of libraries for various languages.
- You need some backed as well (i.e. Jaeger or Prometheus)
- Based on **Signals** - information that is categorized and processed
  - Traces
  - Metrics
  - Logs
- Workhorse of modern observability!



# Traces and spans in OTel and Jaeger

```
{
  "name": "Hello-Greetings",
  "context": {
    "trace_id": "0x5b8aa5a2d2c872e8321cf37308d69df2",
    "span_id": "0x5fb397be34d26b51",
  },
  "parent_id": "0x051581bf3cb55c13",
  "start_time": "2022-04-29T18:52:58.114304Z",
  "end_time": "2022-04-29T22:52:58.114561Z",
  "attributes": {
    "http.route": "some_route1"
  },
  "events": [
    {
      "name": "hey there!",
      "timestamp": "2022-04-29T18:52:58.114561Z",
      "attributes": {
        "event_attributes": 1
      }
    },
    {
      "name": "bye now!",
      "timestamp": "2022-04-29T18:52:58.114585Z",
      "attributes": {
        "event_attributes": 1
      }
    }
  ]
},
]
```





# | Load testing?

## Cost

- Load test takes hours to complete (think caching, tiered compilation, etc.)

## Data distribution and cardinality

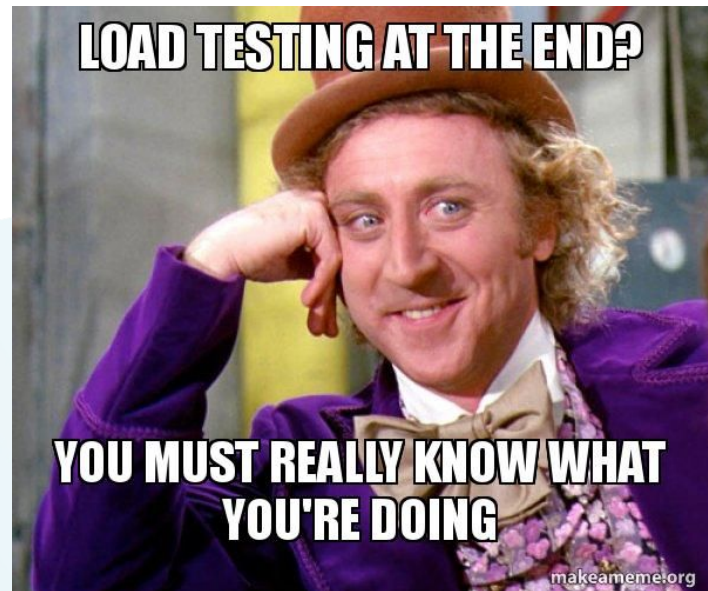
- You can't test your EU stack with the data from the USA
- What about smaller countries?

## Hardware and environment

- GPUs are expensive and not very available
- Edge computing? Custom hardware?
- Do you pay for it 24/7?

## Data anonymity

- What about SSN? How do you anonymize it in pre-production?



# Be proactive and push to the left!

Waiting for tickets from customers is expensive.

Load tests are slow, too late, and too expensive.

Issues need to be identified early and automatically.



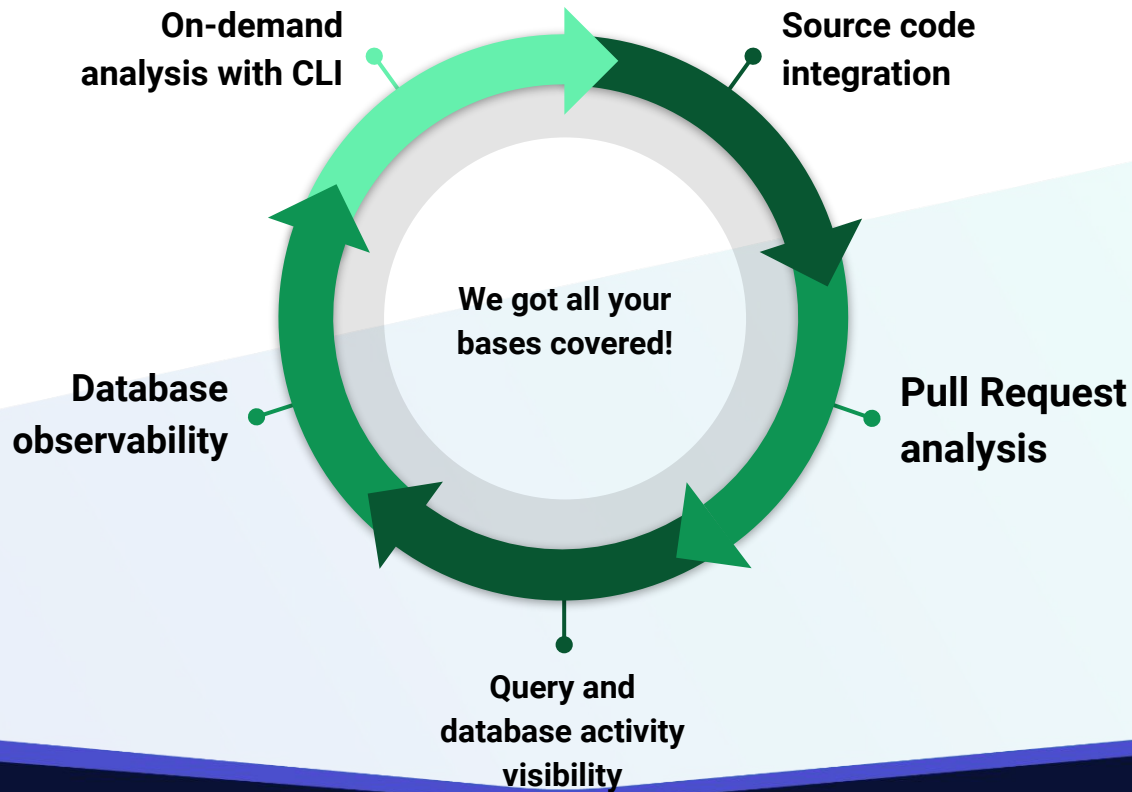




# metis

**Never go blind  
again!**

# Metis



# | Source code integration

## Web server integration

- Open Telemetry
- Capturing trace and REST calls

## ORM integration

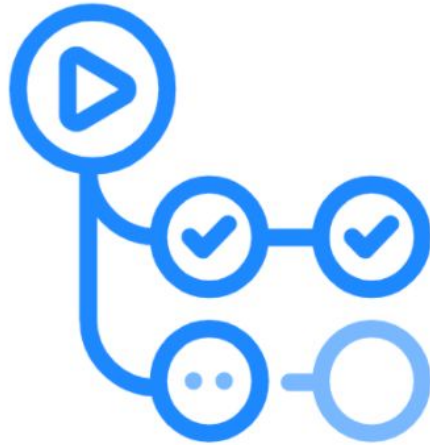
- Instrumenting SQL queries
- Observing all connectivity

## SQL driver integration

- Capturing all database interactions



# | Pull Request analysis



GitHub Actions

# Observability

< 200 GET /flights/departures/AAQ

Feb 25, 2023, 09:02:00 Trace ID: null8afaf2a2538529a494091512524

Send to Slack

0ms	37ms	57ms	83ms	79ms	53ms	Severity
GET /flights/departures/AAQ						

Summary (Estimated)

345,14K	1	931	10
Rows Read	Tables	Rows Returned	Columns

Insights SQL Metrics Query Tale Execution Plan Tables

Rows filtered Critical

Rows read High

Result set size Low

A WHERE clause uses a function Healthy

Rows returned Healthy

Number of table joins Healthy

Optimizer cost prediction Healthy

Columns returned Healthy

Rows sorted Healthy

### Insights Details

The SQL command is not efficient. It read 345,142 rows. However, 342,485 were filtered out (99%)

Table Name	Access Method	Index Name	Rows Read	Rows Filtered	Cost	Cost
bookings.flights	Table Scan	N/A	345,138	342,484	53084	0

### Impact

Reading many rows from a database table and then filtering them out can be inefficient because it can consume a lot of resources, such as memory and CPU time. This can slow down the overall performance of the database and cause issues with scalability.

### Remediation Plan

1. Consider adding an index. Indexes allow the database to quickly locate the specific rows that match a certain condition, without having to scan the entire table.
2. Use the LIMIT clause. It helps return only a subset of the rows.
3. Consider partitioning the table to distribute the data across multiple disks. This can help to reduce the amount of I/O required to read the data, which can improve performance.

< Metis Prod < platform < Observability Reports

Host: metis-prod-v2.cofh7zmy4e-central-1rds.amazonaws.com

Database: platform

Last update: now

Schema Name	Table Name	Index Name	Daily Usage
public	prisma_migrations	prisma_migrations_pkey	
public	api_key	api_key_api_key_key	
public	api_key	api_key_pkey	
public	api_key	api_key_user_id_idx	
public	insights_breakdown	insights_breakdown_pkey	
public	insights_breakdown	insights_breakdown_span_id_idx	
public	insights_breakdown	insights_breakdown_trace_id_idx	

```

SELECT
  pk, metric_name, type, value, description, severity
FROM
  (
    1. sum -- sum --> metrics_aggregate_data -- The fully qualified name (schema.db.table)
    2. table_name -- table_name --> metrics_aggregate_data -- Table Name
    3. table_id -- table_id --> metrics_aggregate_data -- ID of the table
    4. sum-of-columns -- column_number --> metrics_aggregate_data -- Total number of columns
    5. columns -- columns --> metrics_aggregate_data -- List of column names
    6. pk-name -- primary_key_name --> metrics_aggregate_data -- The name of the PK. Empty if the table doesn't have one.
    7. pk-column -- primary_key_column --> metrics_aggregate_data -- A PK entry of the columns of the PK, by their actual order and the data type of each column
    8. sum-of-indices -- num_of_indices --> metrics_aggregate_data -- The number of indices
    9. sum-of-index-expressions -- number --> metrics_aggregate_data -- The number of index expressions
    10. sum-of-index-costs -- index_cost --> metrics_aggregate_data -- The sum of the cost of all indices
    11. sum-of-foreign-keys -- num_of_foreign_keys --> metrics_aggregate_data -- The number of foreign keys
    12. sum-of-foreign-keys -- num_of_foreign_keys --> metrics_aggregate_data -- The number of foreign keys
    13. pk-name -- index_name --> metrics_aggregate_data -- The name of the index
    14. pk-column -- index_name --> metrics_aggregate_data -- The name of the index
    15. pk-column -- index_name --> metrics_aggregate_data -- The name of the index
    16. pk-column -- index_name --> metrics_aggregate_data -- The name of the index
    17. pk-column -- index_name --> metrics_aggregate_data -- The name of the index
    18. pk-column -- index_name --> metrics_aggregate_data -- The name of the index
  )
ORDER BY
  18 DESC, 17 DESC, 16 DESC, 15 DESC, 14 DESC, 13 DESC, 12 DESC, 11 DESC, 10 DESC, 9 DESC, 8 DESC, 7 DESC, 6 DESC, 5 DESC, 4 DESC, 3 DESC, 2 DESC, 1 DESC
  
```

Diagnostic

```

SELECT * FROM
  (
    1. sum -- sum --> metrics_aggregate_data -- The fully qualified name (schema.db.table)
    2. table_name -- table_name --> metrics_aggregate_data -- Table Name
    3. table_id -- table_id --> metrics_aggregate_data -- ID of the table
    4. sum-of-columns -- column_number --> metrics_aggregate_data -- Total number of columns
    5. columns -- columns --> metrics_aggregate_data -- List of column names
    6. pk-name -- primary_key_name --> metrics_aggregate_data -- The name of the PK. Empty if the table doesn't have one.
    7. pk-column -- primary_key_column --> metrics_aggregate_data -- A PK entry of the columns of the PK, by their actual order and the data type of each column
    8. sum-of-indices -- num_of_indices --> metrics_aggregate_data -- The number of indices
    9. sum-of-index-expressions -- number --> metrics_aggregate_data -- The number of index expressions
    10. sum-of-index-costs -- index_cost --> metrics_aggregate_data -- The sum of the cost of all indices
    11. sum-of-foreign-keys -- num_of_foreign_keys --> metrics_aggregate_data -- The number of foreign keys
    12. sum-of-foreign-keys -- num_of_foreign_keys --> metrics_aggregate_data -- The number of foreign keys
    13. pk-name -- index_name --> metrics_aggregate_data -- The name of the index
    14. pk-column -- index_name --> metrics_aggregate_data -- The name of the index
    15. pk-column -- index_name --> metrics_aggregate_data -- The name of the index
    16. pk-column -- index_name --> metrics_aggregate_data -- The name of the index
    17. pk-column -- index_name --> metrics_aggregate_data -- The name of the index
    18. pk-column -- index_name --> metrics_aggregate_data -- The name of the index
  )
ORDER BY
  18 DESC, 17 DESC, 16 DESC, 15 DESC, 14 DESC, 13 DESC, 12 DESC, 11 DESC, 10 DESC, 9 DESC, 8 DESC, 7 DESC, 6 DESC, 5 DESC, 4 DESC, 3 DESC, 2 DESC, 1 DESC
  
```

# | Summary



**Database may break**

- Bugs
- ORM quirks
- Database inefficiency



**You need to be proactive**

- Load tests are too late
- Constant monitoring is needed



**Metis covers all of that**

- App integration
- Pull requests
- Observability
- Safety



**metis**

**Q&A**





# metis

## Thank you!



<https://www.metisdata.io/>