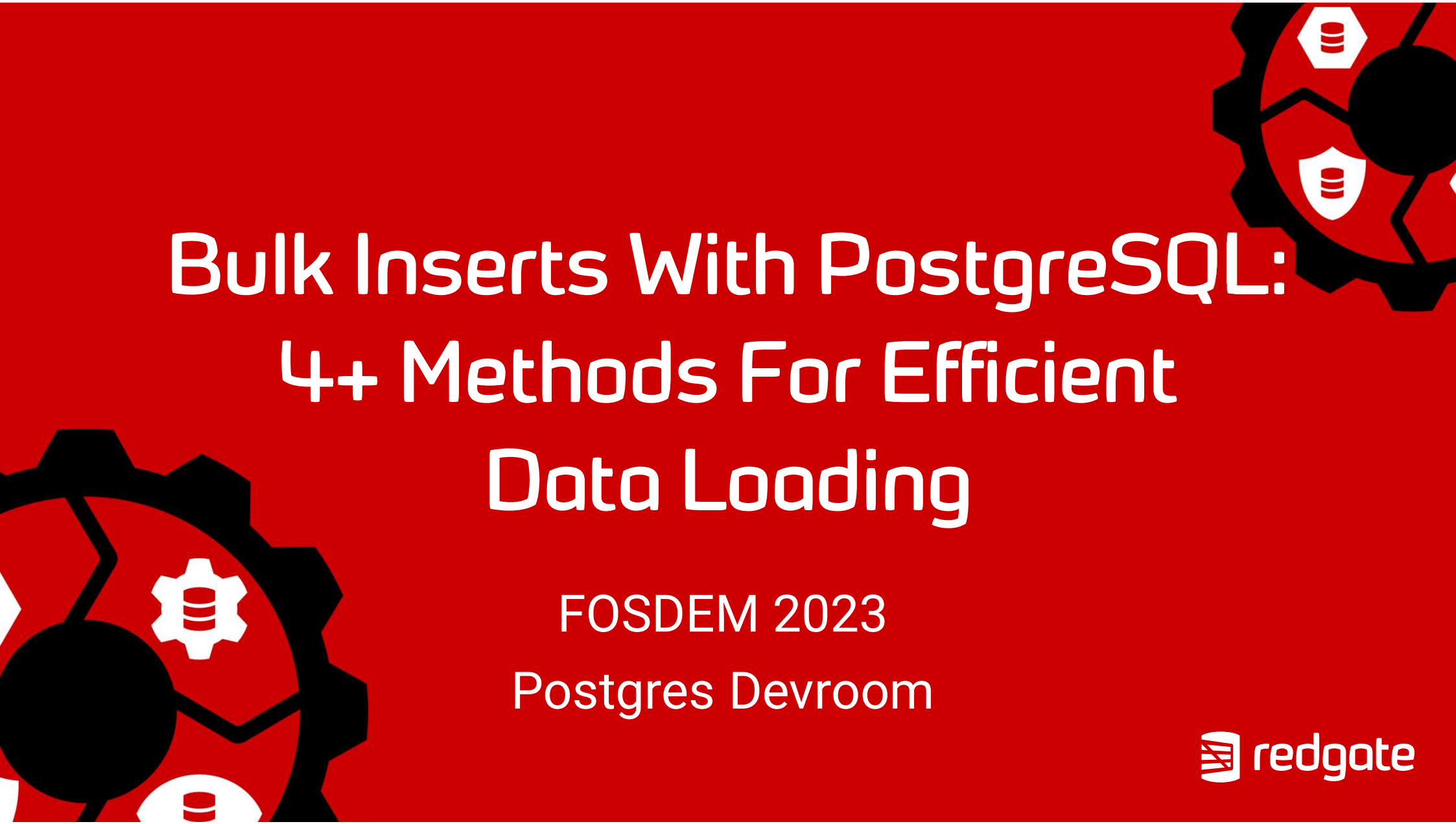

Bulk Inserts With PostgreSQL:
4+ Methods For Efficient
Data Loading
FOSDEM 2023
Postgres Devroom
redgate

redgate | About me

Ryan Booz

PostgreSQL & DevOps Advocate

@ryanbooz

/in/ryanbooz

www.softwareandbooz.com

github.com/ryanbooz/presentations

redgate

Agenda

01 Death By 1,000 INSERT's

02 4+ methods for bulk loading data

03 Demos

04 Database DevOps Goal

redgate

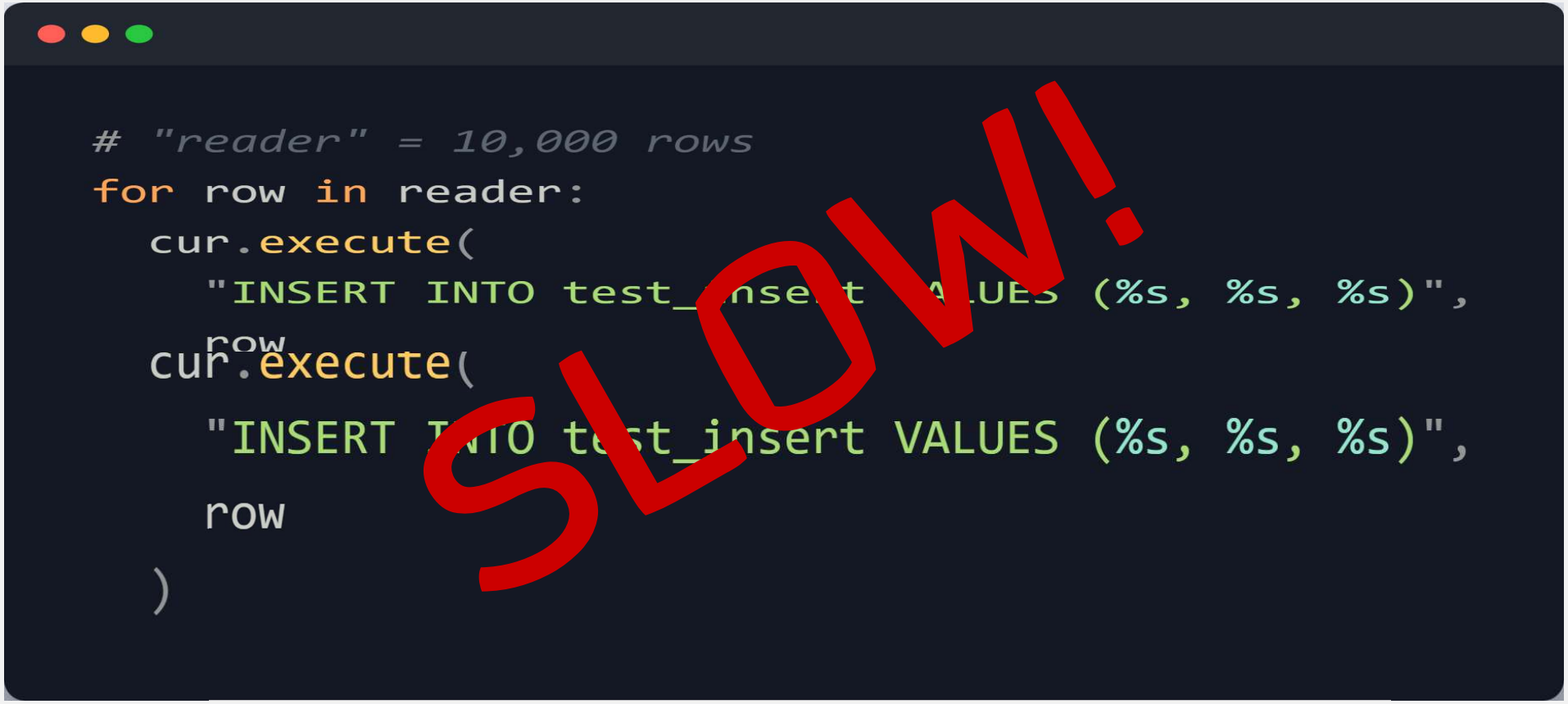
01/04

Death By 1,000 INSERT's

redgate

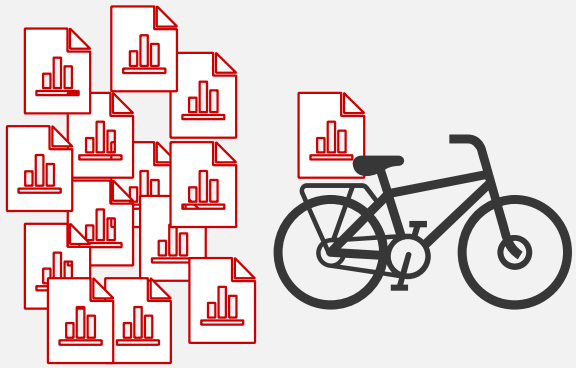
How often do you load a lot of data into PostgreSQL?

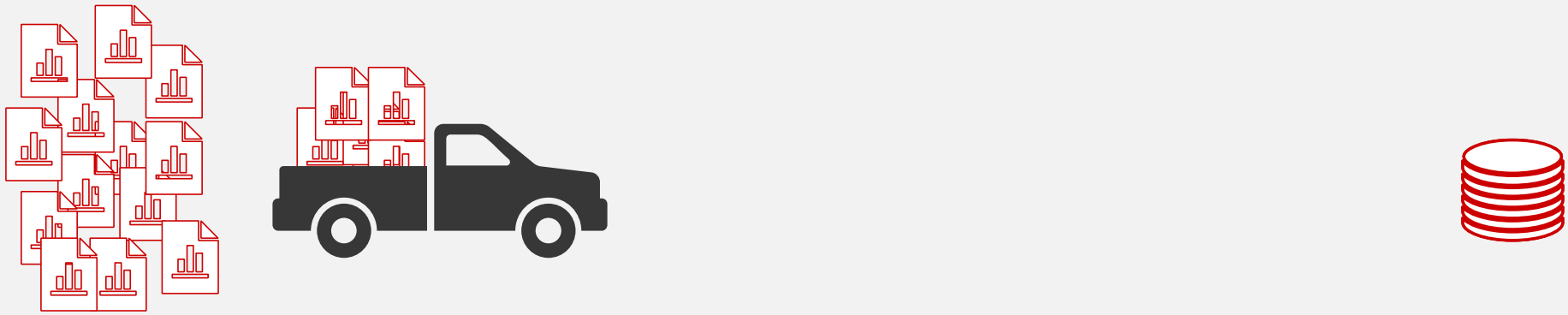

01011
10011
00001
11001
10110
{JSON}
C,S,V

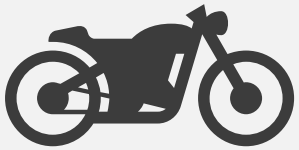

SLOW!

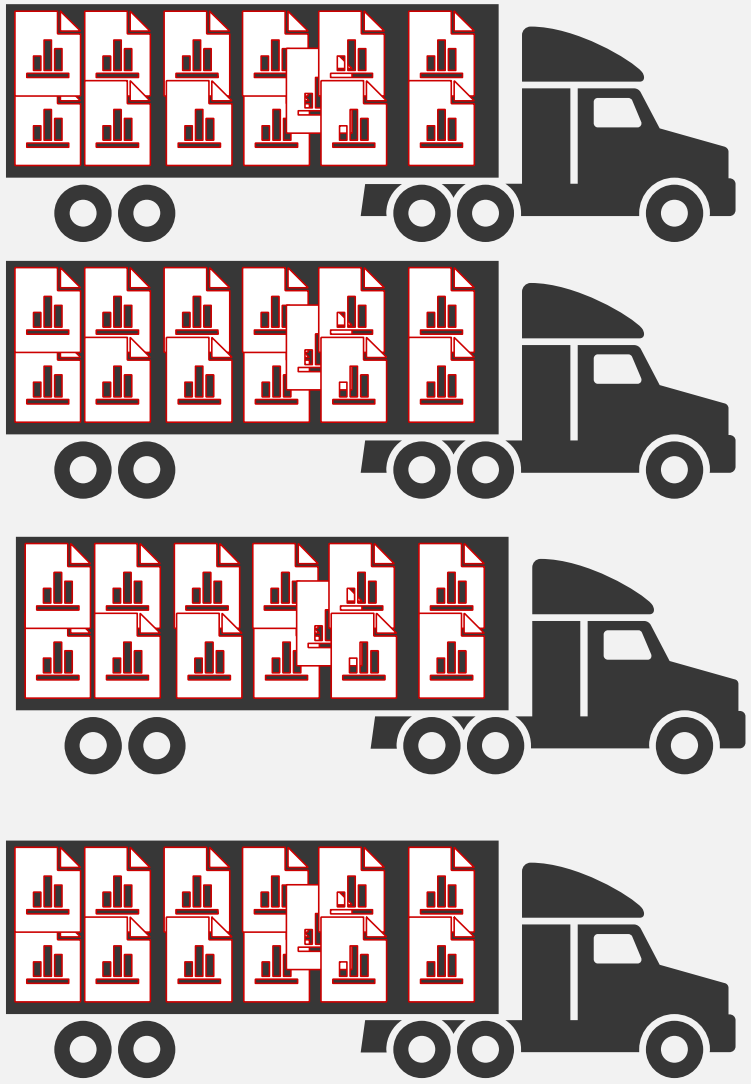
INSERTs are slow because...

- ...every statement incurs overhead
  - Network
  - Parsing
  - Planning
  - Locks
  - Execution
  - Response
  - Indexes & Constraints
- Even local application --> DB has overhead









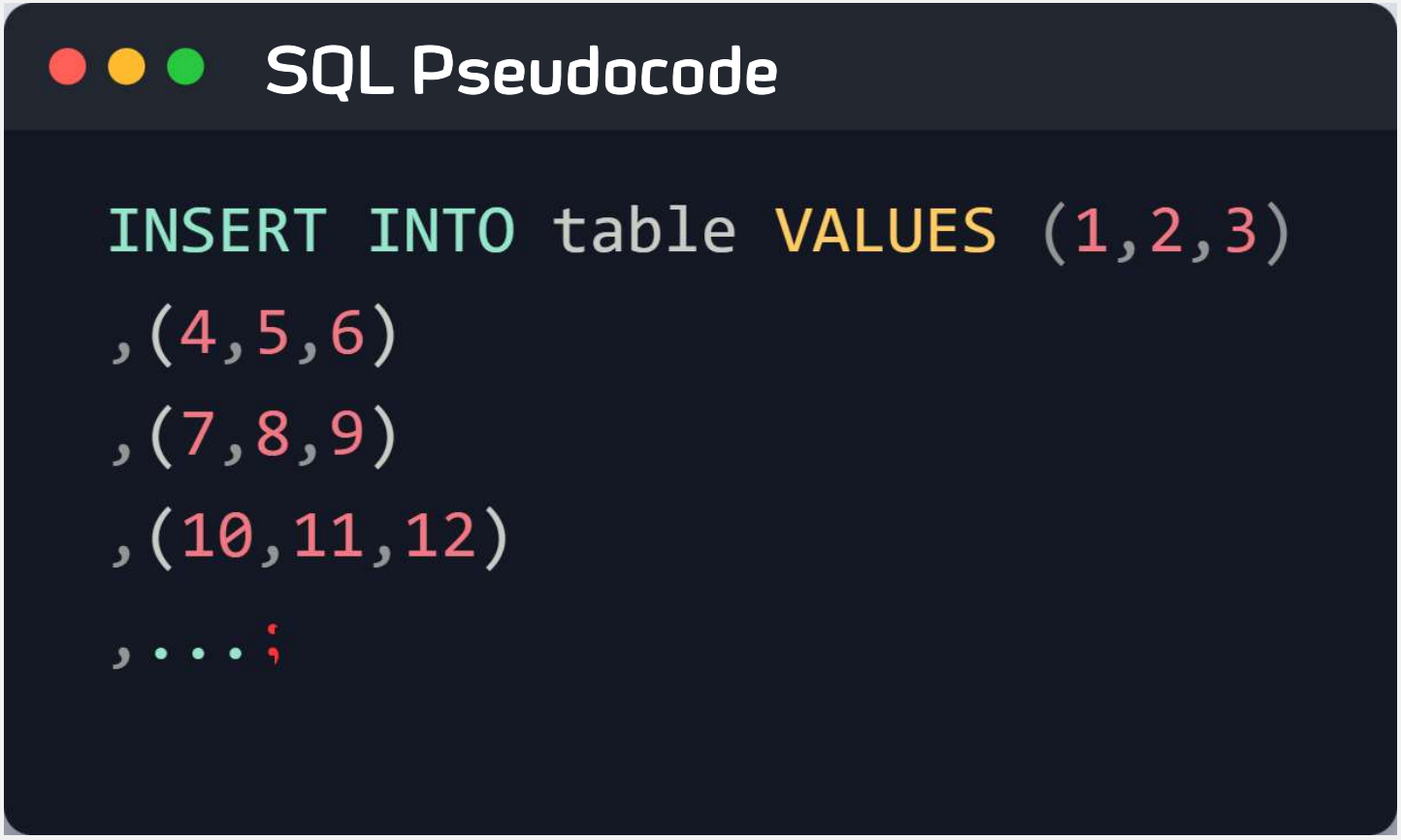
02/04

4+ Methods for Bulk Loading Data

redgate

Multi-valued INSERT

redgate


SQL Pseudocode
INSERT INTO table VALUES (1,2,3)
,(4,5,6)
,(7,8,9)
,(10,11,12)
,...;

Python Pseudocode

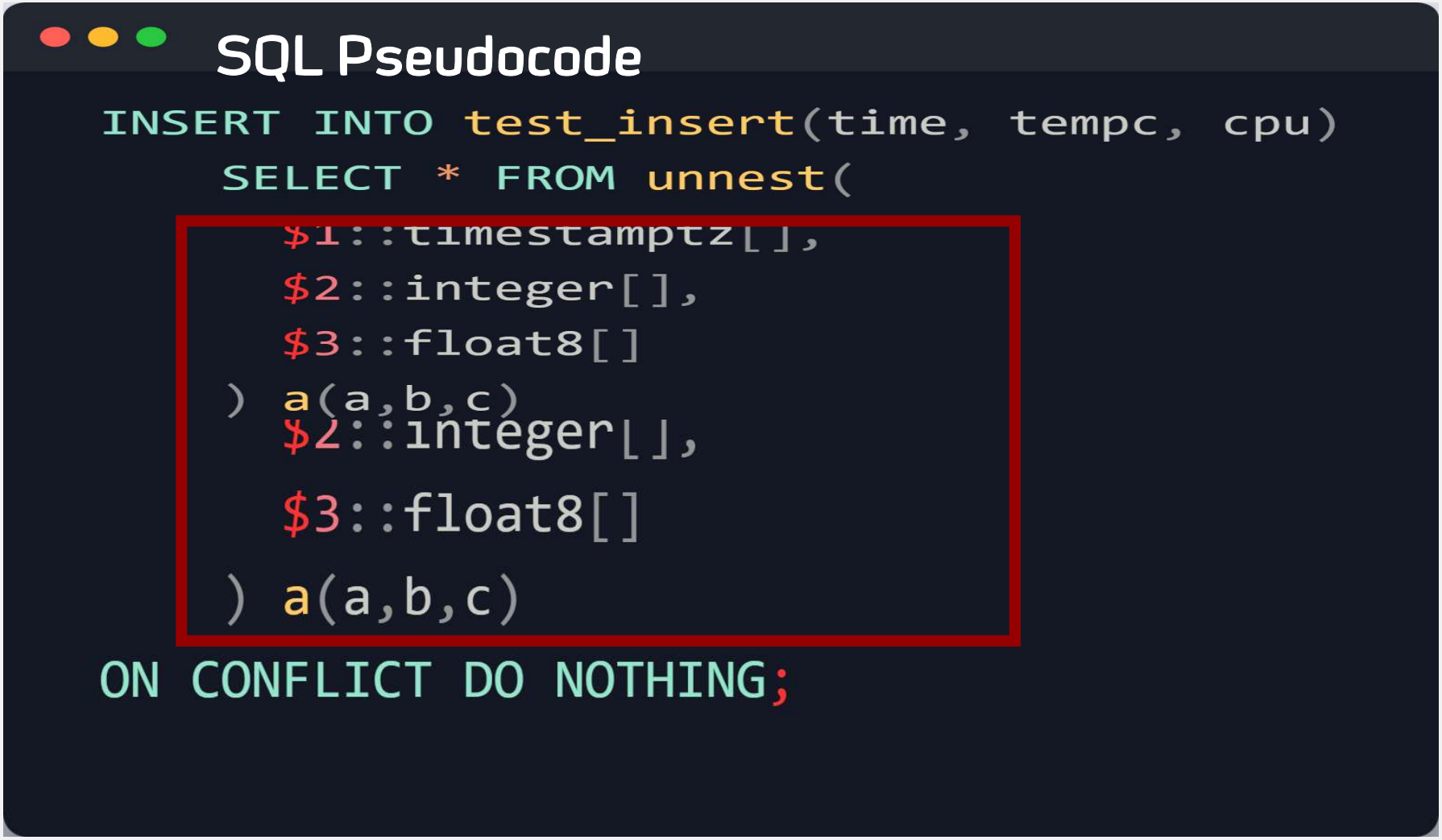
```
# "reader" = 10,000 rows
sql = "INSERT INTO test_insert VALUES "

next(reader) # Skip the header row.
  for row in reader:
      batch_count += 1
      sql += "('{}', {}, {}),".format(*row)

      if batch_count == 500:
         cur.execute(sql[:-1])
         batch_count = 0
         sql = "INSERT INTO test_insert VALUES "
```

Multi-valued INSERT

- Little extra programming effort
- Supported regardless of the driver or language (even dynamic pl/pgsql)
- Moderately faster
- Generally requires batching subsets of rows

ARRAY values INSERT

redgate


SQL Pseudocode
INSERT INTO test_insert(time, tempc, cpu)
SELECT * FROM unnest(
$1::timestamptz[],
$2::integer[],
$3::float8[]
) a(a,b,c)
$2::integer[],
$3::float8[]
) a(a,b,c)
ON CONFLICT DO NOTHING;

Python Pseudocode

```
# converting column data to list
date = data['time'].tolist()
tempc = data['tempc'].tolist()
cpu = data['cpu'].tolist()

i=0
batch_size=2000
batch_end=batch_size
total_length = len(date)
while batch_end < total_length:
    cur.execute(
     "INSERT INTO test_insert SELECT * FROM
          unnest(%s::timestamptz[],%s::int[],%s::double precision[])
          a(t,v,s)
        ON CONFLICT DO nothing;",(date[i:batch_end],tempc[i:batch_end],cpu[i:batch_end]))
    i=batch_end
    batch_end+=batch_size
    conn.commit()
```

ARRAY values INSERT

- Can be faster than multi-valued INSERT
  - …in some cases
- Avoids the 65,535 parameter limit 😀
- YMMV with language support for PostgreSQL arrays
- Caution: May not handle custom types correctly

COPY

redgate

COPY*

- Preferred, optimized tool for PostgreSQL bulk load
- Reads from files or STDIN
- Paths are local to the PostgreSQL server
- Can also pull data out to a file
- Not in the SQL standard

*NOT psql \COPY… but closely related

redgate

COPY Limitations

- Single transaction
- Single threaded
- No progress feedback prior to PG14
  - **pg_stat_progress_copy** view
  - In **psql:**

```
SELECT * FROM pg_stat_progress_copy \watch 1
```

COPY Limitations

- Minimal format configuration
- No failure tolerance - stops on first error
  - Failed import takes space and leaves rows inaccessible ☹

redgate

`COPY` stops operation at the first error. This should not lead to problems in the event of a `COPY TO`, but the target table will already have received earlier rows in a `COPY FROM`. These rows will not be visible or accessible, but they still occupy disk space. This might amount to a considerable amount of wasted disk space if the failure happened well into a large copy operation. You might wish to invoke `VACUUM` to recover the wasted space.

COPY does one job really well:

import/export data fast!

redgate

pgloader.io

PGLOADER

- Initially created by Dimitri Fontaine
- CLI application
- Many migration formats
  - CSV, DBF, IXF
  - SQLite, MySQL, SQL Server
  - PostgreSQL to PostgreSQL
  - Limited Redshift support
- Before/After scripts
- Logfile support
- Data casting
- Error support
- Continuous migrations
- ...and more

redgate

Timescale Parallel Copy

- Created by Timescale to assist in time-ordered inserts
- Written in Golang
- Multi-threading through multiple COPY commands
- Progress output
- Configurable rows per batch
- Significantly faster for high-latency (remote) connections

https://github.com/timescale/timescaledb-parallel-copy

redgate

Unlogged Tables

redgate

UNLOGGED Tables - Caution!

- Data inserted into UNLOGGED tables is not written to the Write-Ahead Log (WAL)
- Eliminates some of that INSERT overhead
- Data is not crash safe
- Not accessible on replication servers (requires WAL)
- Available with CREATE and ALTER table

Why use UNLOGGED tables?

- Your data process can accept the risk of loss for increased INSERT
- Staging tables for ETL processes
- Intermittent, repeatable work (easy to redo)

03/04
Demos

redgate

04/04

Final Thoughts

Index & Constraints

- Constraints and Indexes cannot be disabled
- Constraints are always checked
- Indexes are always updated
  - Heap Only Tuples aside
- Dropping before insert can significantly improve performance… at your own risk 😱

redgate

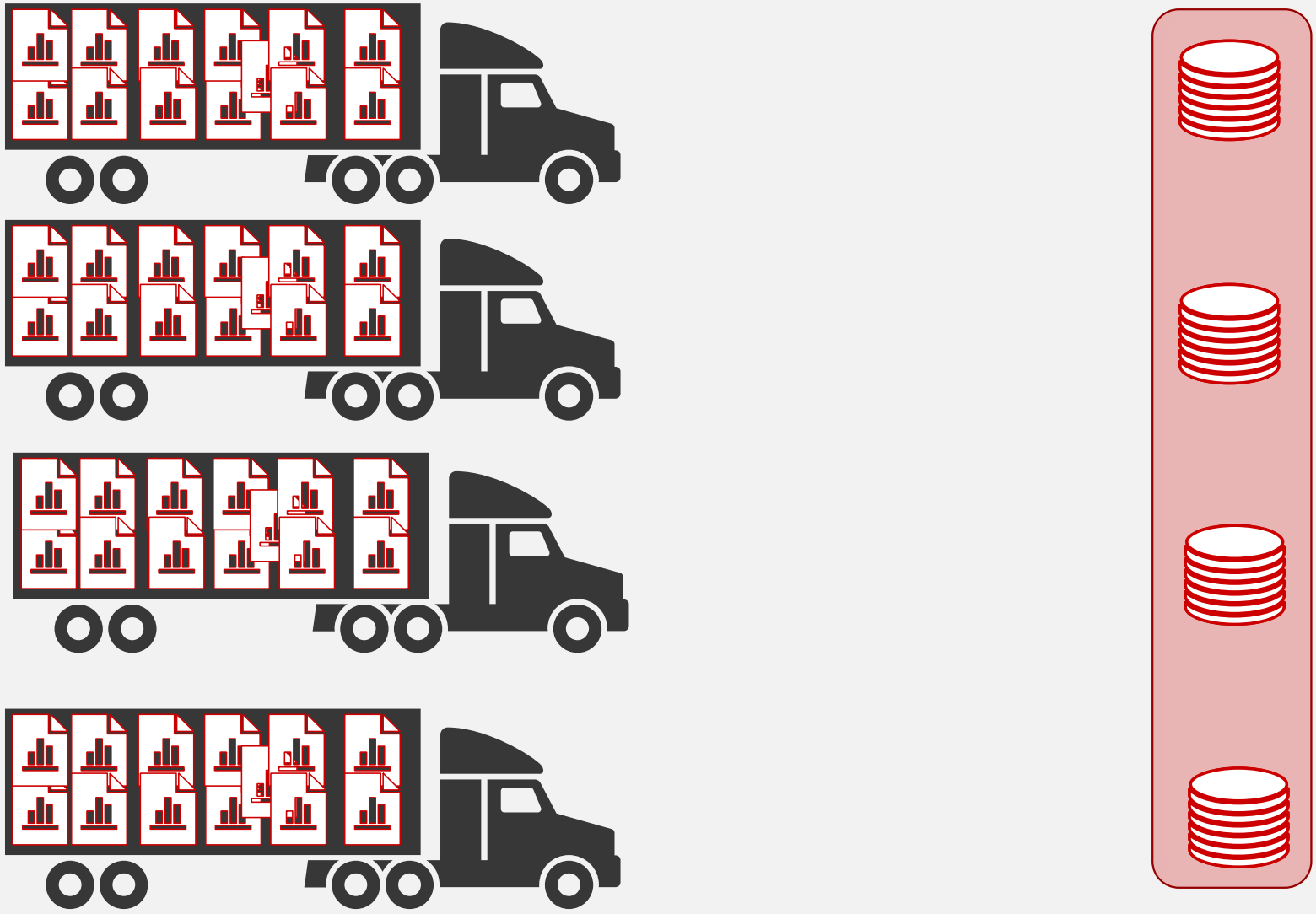
Surprise Demo!

redgate

Partitioning for long-term growth

- PG11+ includes native partitioning
- Particularly good for time-series data
- Indexes are local to the partition

redgate

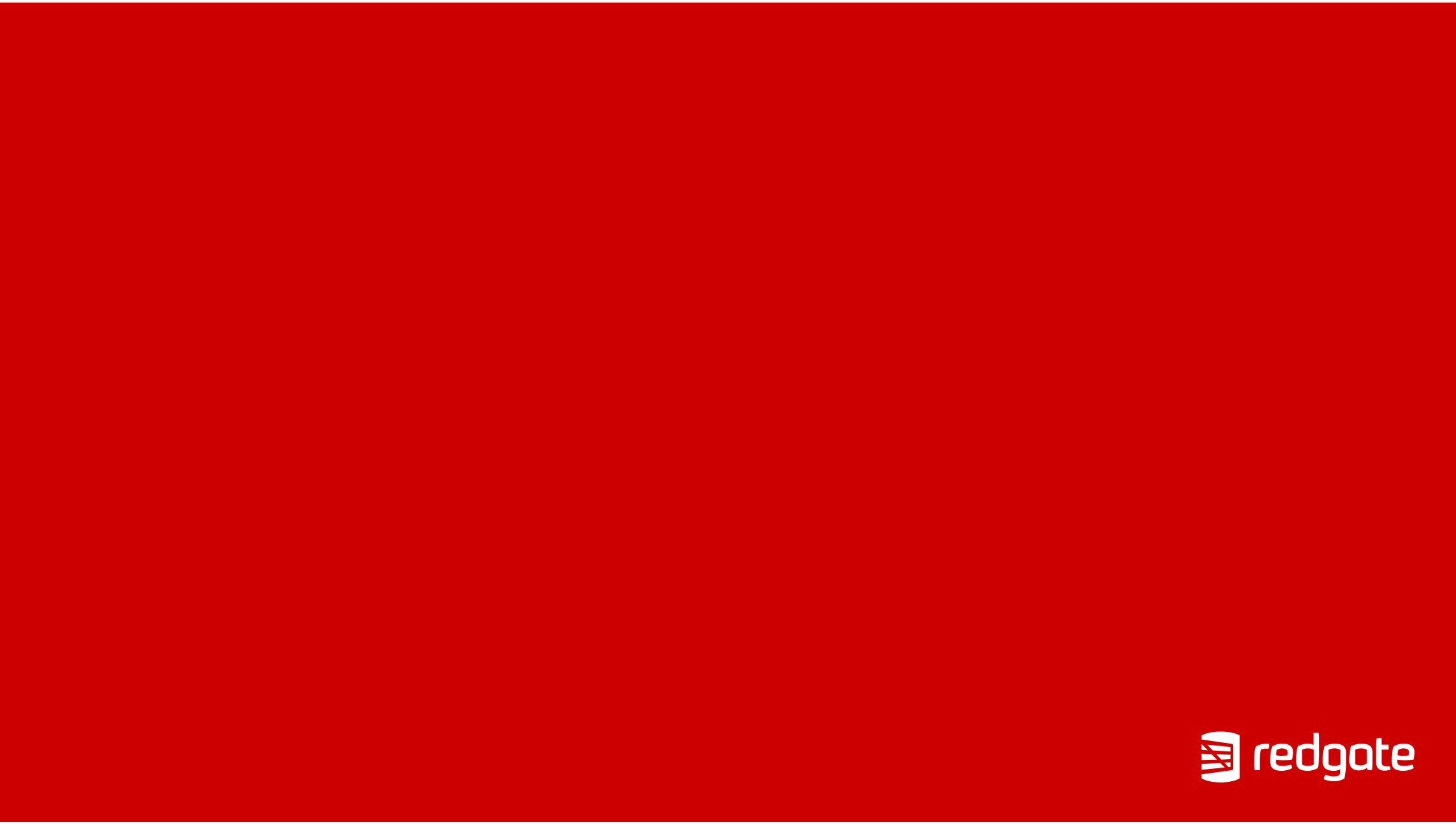


What to look for in language SDKs

- Specific COPY/BINARY COPY support
- Multi-valued and batching functions
- How is auto-commit handed?
- Avoid parameterized query formatter
  - Use the ARRAY trick if the SDK only uses parameterized queries

What questions do you have for me?

redgate


redgate