

# Postgresql and The Strong Anthropic Principle of Cosmology

**Mayuresh**  
mayur555b@gmail.com

# POSTGRESQL AND THE STRONG ANTHROPIC PRINCIPLE.

**A database comedy or philosophy – Mayur, DBA@WISE**

COGITO ERGO MUNDUS TALIS EST. – DESCARTES

**UNIVERSE IS DESIGNED FOR US**



"I THINK, THEREFORE THE WORLD IS AS IT IS."

PostgreSQL reminded me of the strong anthropic principle in cosmology. The strong anthropic principle says the Universe has these conditions because it *must* have them in order to have intelligent life (us). Hence, our existence is the end goal of a plan. The strong form of the anthropic principle insists that we are special, an intellectual center of the Universe (all intelligent species would be at their "center"), because we exist and think.

# THOU SHALT NOT HINT

Yes, rest of the industry should bend the knee. Data pattern, data velocity, data usage and statistics accuracy should always behave nice to avoid those naughty plan fluctuations.

This dogma against hints in PostgreSQL community fascinated me while transitioning from Oracle world. However, PostgreSQL triumphantly saves so much money compared to Oracle databases that those concerns fade into the background.

# "THE ORACLE, SHE TOLD ME THAT..." -- NEO

My previous job was with a large company that had thousands of Oracle databases and ancient code maintained by dozens of vendors. Whenever DBAs suggest code changes, developers ask why we are so hostile to them. Hints were the generally accepted way to fix query performance issues because code/design changes would require several months of test cycles and bureaucratic approvals.

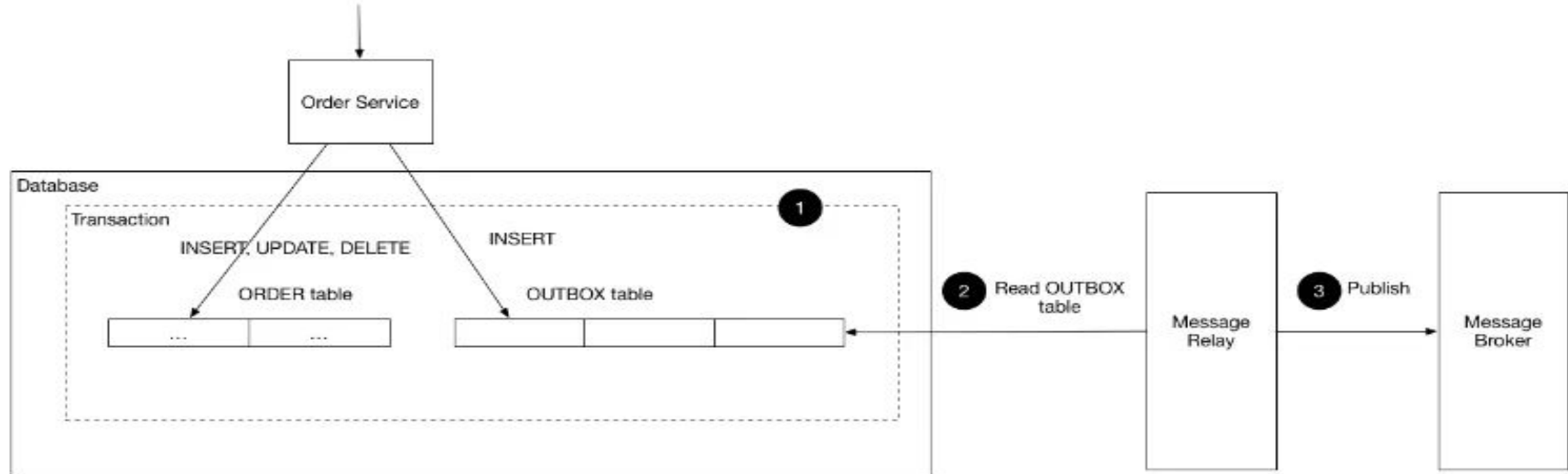
We had a policy to only provide hints that made business sense, for example in an OLTP with large concurrency you would provide nested loop hint along with index hint cause if only index hint was provided and planner used it but still picked hash join causing full scan of leading table then it would lead to an incident.

# BRAVE NEW WORLD

Wise started with a web page and a single database backend, like many companies. As we grew and the product became more advanced, it became obvious that coping with our growth would require a shift towards a microservice architecture. Microservice architecture provides agility for rolling out innovative products or capturing new geographies. To maintain autonomy of application development teams and achieve business domain level sharding our architects decided to provision a separate database for every micro-service. As a result, we have PostgreSQL and MariaDB spread out among our 500+ AWS RDS and a few dozen self-hosted databases.

A micro-service typically needs to update the database and send messages/events consumed by other micro-services. This can be achieved with the Transactional outbox pattern which is discussed in detail [here](#).

A service that uses a relational database inserts messages/events into an *outbox* table (e.g. **MESSAGE** ) as part of the local transaction. An service that uses a NoSQL database appends the messages/events to attribute of the record (e.g. document or item) being updated. A separate *Message Relay* process publishes the events inserted into database to a message broker.





# MICRO-SERVICE BUT MACRO PROBLEMS

At first glance, it's obvious that this will trouble PostgreSQL MVCC. Every transaction has a corresponding insert/update in the outbox, and the data volume continuously changes due to deletes based on polling frequency. Also table is continuously being read by a message relay. A perfect recipe for disaster as it generates many dead tuples and autoanalyze may collect stats when the outbox table is empty which results in subsequent full scans. Tiny RDS instances provisioned for micro-services hit Io/throughput/cpu limits quickly on the cloud. So you would have autovacuum playing catch up with DML rate on outbox tables hence accumulating dead tuples. You can make autovacuum more aggressive but that comes at the cost of instance throughput and CPU capacity. Even then, with many such full scan queries running on a bloated outbox table, the database would be overloaded. A simple index hint via `pg_hint_plan` on the outbox table's queries saves on-call DBAs sleep hours.

# DATABASE STATISTICS BIAS

- Table has 10 million records.  
It really does.
- However, database thinks it is empty.
- Database executes **sequential scan** for any query done against that table.
- Database I/O meanwhile:



outgoing\_message\_0\_0

---

---

---

---

---

---

---

---

## POSTGRES AND DEAD TUPLES

- Table has 1 record.
- We query all records from that table.
- Somehow IO equivalent 10 million records happens. And finally, 1 minute later, 1 record is returned.
- Database CPU and I/O, meanwhile:



outgoing\_message\_0\_0

[illegible]

# SAVIOR OF DBA SLEEP CYCLE

**PG\_HINT\_PLAN**



```
postgres=> -- create sample tables
CREATE TABLE table1 (
    id serial primary key,
    col1 integer,
    col2 text
);

CREATE TABLE table2 (
    id serial primary key,
    col3 integer,
    col4 text
);

-- insert sample data into the tables
INSERT INTO table1 (col1, col2) VALUES
    (1, 'A'),
    (2, 'B'),
    (3, 'C'),
    (4, 'D'),
    (5, 'E');

INSERT INTO table2 (col3, col4) VALUES
    (1, 'X'),
    (2, 'Y'),
    (3, 'Z');

CREATE TABLE
CREATE TABLE
INSERT 0 5
```

```
postgres=>
postgres=> -- Use pg_hint_plan to enforce a plan opposite of currently taken by planner
EXPLAIN
SELECT *
FROM table1 JOIN table2
ON table1.col1 = table2.col3;

               QUERY PLAN
-----
Hash Join  (cost=1.07..2.17 rows=3 width=20)
  Hash Cond: (table1.col1 = table2.col3)
    -> Seq Scan on table1  (cost=0.00..1.05 rows=5 width=10)
    -> Hash  (cost=1.03..1.03 rows=3 width=10)
        -> Seq Scan on table2  (cost=0.00..1.03 rows=3 width=10)

(5 rows)
```

```
postgres=> -- with leading hint
EXPLAIN /*+ leading( (table2 table1)) */
SELECT *
FROM table1 JOIN table2
ON table1.col1 = table2.col3;

               QUERY PLAN
-----
Hash Join  (cost=1.11..2.18 rows=3 width=20)
  Hash Cond: (table2.col3 = table1.col1)
    -> Seq Scan on table2  (cost=0.00..1.03 rows=3 width=10)
    -> Hash  (cost=1.05..1.05 rows=5 width=10)
        -> Seq Scan on table1  (cost=0.00..1.05 rows=5 width=10)

(5 rows)
```

```
postgres=> -- Test nested loop join
EXPLAIN /*+ NestLoop(table1 table2) */
SELECT * FROM table1 JOIN table2 ON table1.col1 = table2.col3;
          QUERY PLAN
-----
Nested Loop (cost=0.00..2.31 rows=3 width=20)
  Join Filter: (table1.col1 = table2.col3)
    -> Seq Scan on table1 (cost=0.00..1.05 rows=5 width=10)
    -> Materialize (cost=0.00..1.04 rows=3 width=10)
        -> Seq Scan on table2 (cost=0.00..1.03 rows=3 width=10)
(5 rows)
```

```
postgres=> -- Test hash join
EXPLAIN /*+ HashJoin(table1 table2) */
SELECT * FROM table1 JOIN table2 ON table1.col1 = table2.col3;
          QUERY PLAN
-----
Hash Join (cost=1.07..2.17 rows=3 width=20)
  Hash Cond: (table1.col1 = table2.col3)
    -> Seq Scan on table1 (cost=0.00..1.05 rows=5 width=10)
    -> Hash (cost=1.03..1.03 rows=3 width=10)
        -> Seq Scan on table2 (cost=0.00..1.03 rows=3 width=10)
(5 rows)
```

# EVOLVE

`pg_hint_plan` works better than MariaDB/MySQL hints on some complex queries and delete queries. A simple index hint can save on-call DBAs sleep hours. We have already seen a real-life illustration of how modern architectural changes in application development are driving database engineers to evolve and grow beyond the strict dogmas.



# REFERENCES

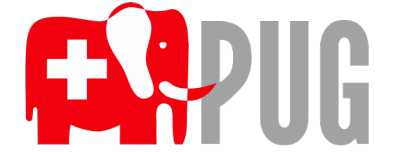
Pg\_hint\_plan [https://github.com/ossc-db/pg\\_hint\\_plan/tree/master](https://github.com/ossc-db/pg_hint_plan/tree/master)

Micro-services

<https://microservices.io/patterns/data/transactional-outbox.html>

The Anthropic principle

[https://ned.ipac.caltech.edu/level5/Peacock/Peacock3\\_5.html](https://ned.ipac.caltech.edu/level5/Peacock/Peacock3_5.html)



# Common pitfalls when demoralizing in PostgreSQL

Gilad Kleinman

[gilad@epsio.io](mailto:gilad@epsio.io)

# Common Pitfalls When Demoralizing



# Pitfall #1 – Missing new data



# Pitfall #2 – performance issues due to UPDATE locks

```
CREATE OR REPLACE FUNCTION new_salary_trigger()  
| RETURNS trigger AS  
$$  
BEGIN  
| update salaries_sum set total=total + new.salary;  
| RETURN NEW;  
END IF;  
END;  
$$  
LANGUAGE 'plpgsql';
```

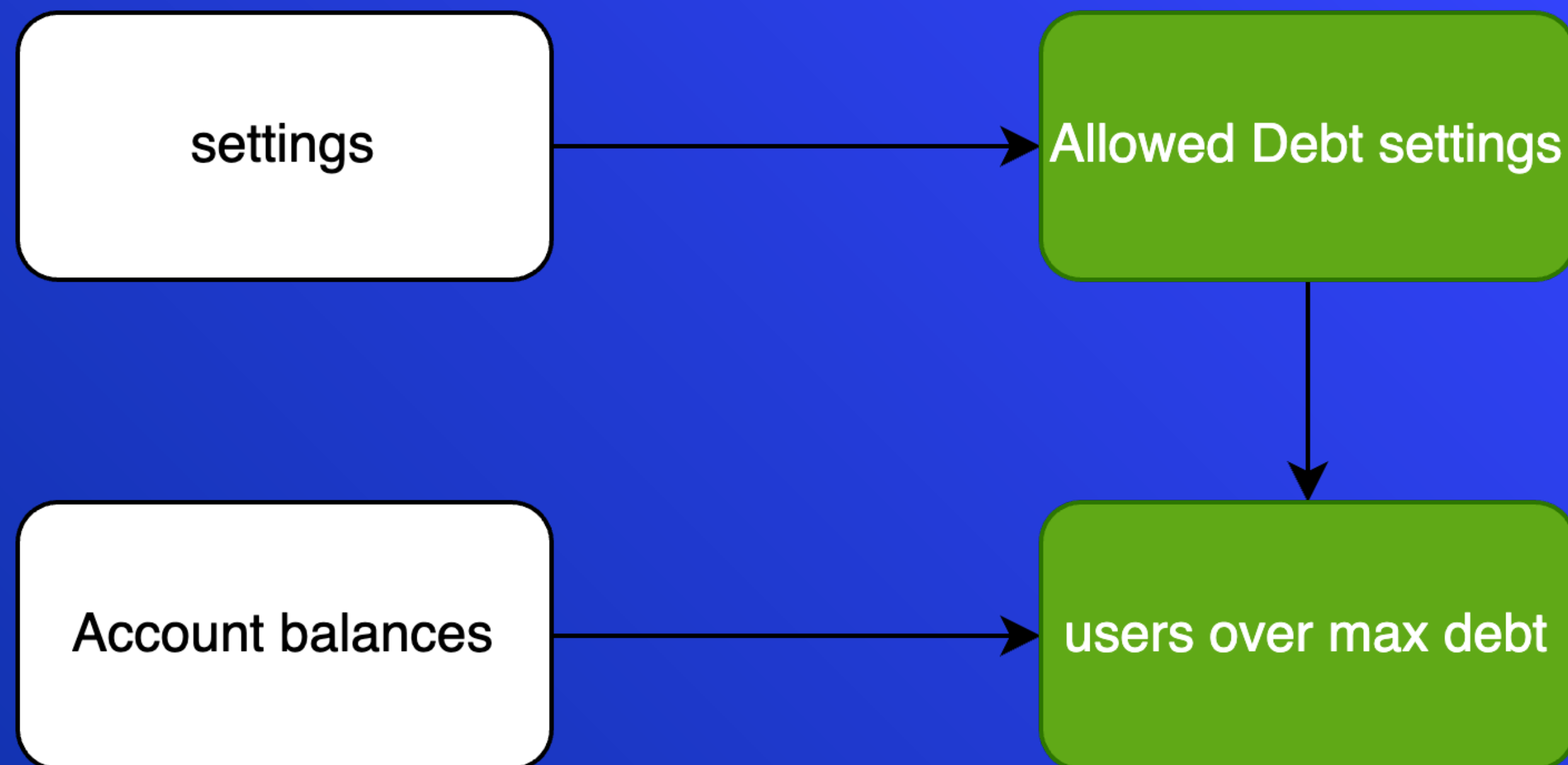
# Pitfall #3 – deadlocks due to UPDATE locks

```
CREATE OR REPLACE FUNCTION new_salary_trigger()  
| RETURNS trigger AS  
$$  
BEGIN  
| update salaries_sum set total=total + new.salary;  
| RETURN NEW;  
END IF;  
END;  
$$  
LANGUAGE 'plpgsql';
```

```
CREATE OR REPLACE FUNCTION new_employee_trigger()  
| RETURNS trigger AS  
$$  
BEGIN  
| update employees_count set count=count + 1;  
| RETURN NEW;  
END IF;  
END;  
$$  
LANGUAGE 'plpgsql';
```



# Pitfall #4 – Missing internal consistency



Transaction 1:  
increase allowed  
balance from 900 to  
1000

Transaction 2:  
change account  
balance from 850 to  
950



# Ein Märchen

Hans Schönig

[hs@cybertec.at](mailto:hs@cybertec.at)

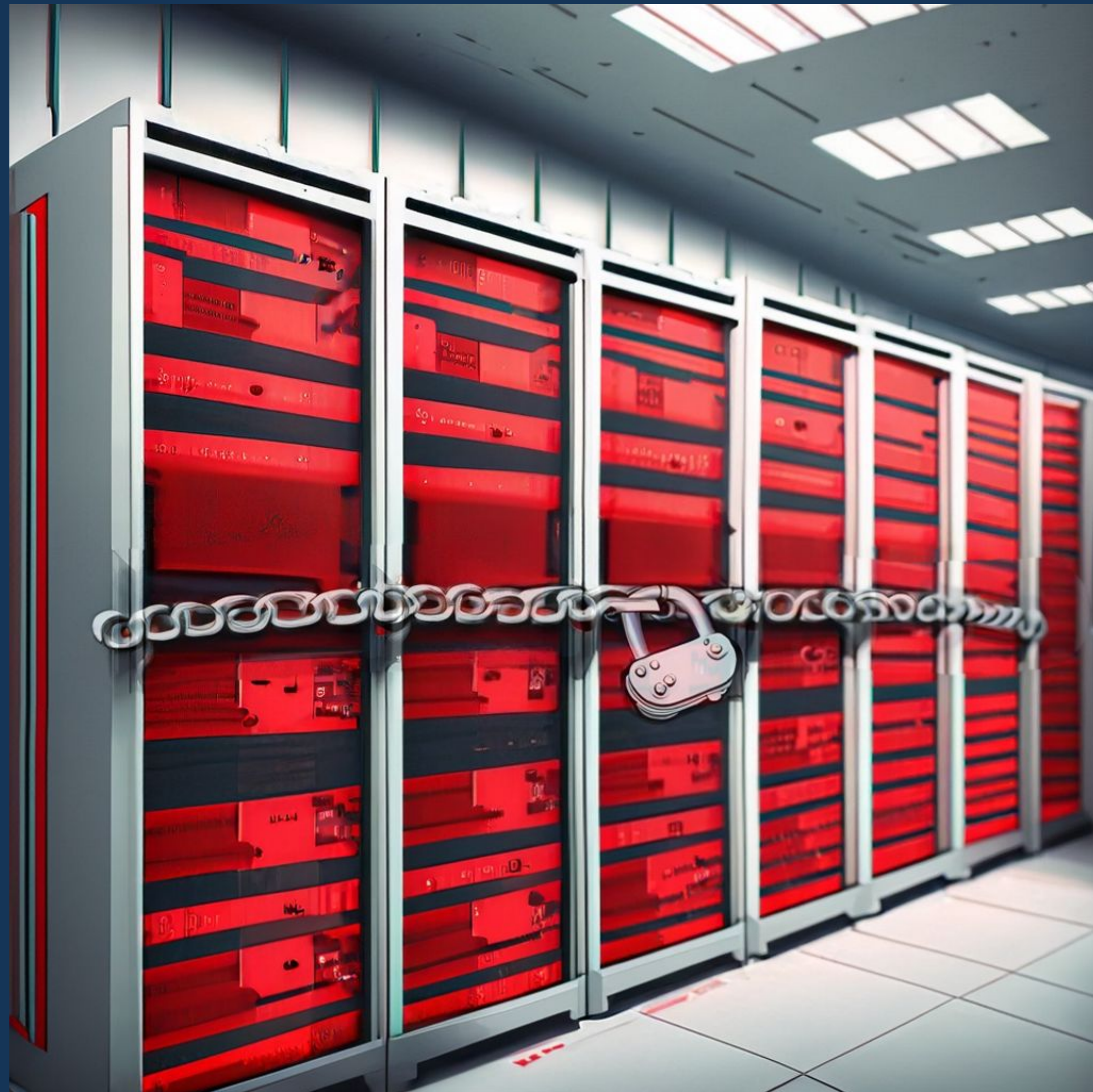
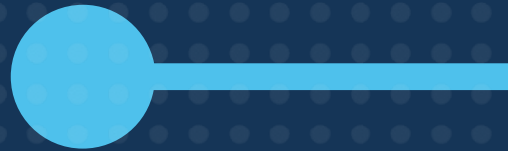




EIN MÄRCHEN

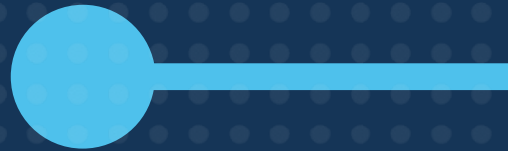
# “Database-itchen und die 7 Server”





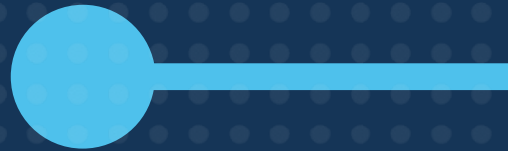
**Es waren einmal 7 Server  
in einem Data Center**





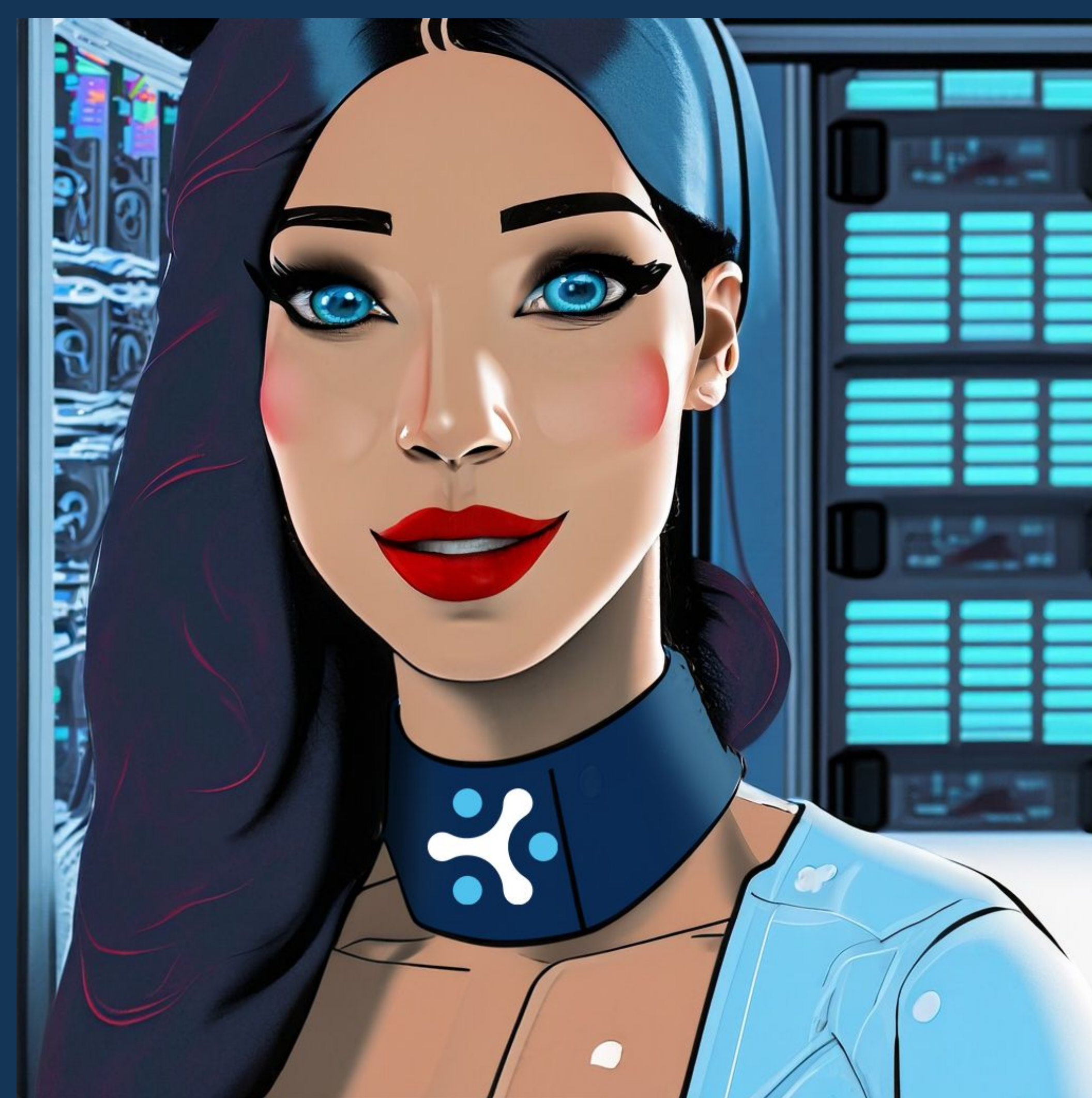
**Die 7 Server waren sehr  
unglücklich, weil der  
böse Wolf Ihr  
Geld stehlen wollte**





Und so saßen die 7 Server  
in Ihrem Rechenzentrum  
und waren sehr sehr  
traurig,  
weil Sie in Ihrem  
goldenen Rack  
eingesperrt waren






Eines Tages kam  
Databaseitchen in das Data  
Center  
und sah die traurigen  
Server





**“Wieso seid Ihr so traurig”  
fragte Databaseitchen die 7 Server**



**“Weil der böse Wolf dauernd an unser Gold will”  
jammerten die 7 kleinen Server**






**Databaseitchen war sehr traurig und sagte ...**



**“Warum macht ihr nicht mal einen Tag blau?”**




**Die 7 kleinen Server schauten sich an und sagten zu sich**  
**“Wieso nicht?”**





**Schnell riefen Sie Ihren  
Freund  
etcd  
an und schon hatten Sie  
eine Mehrheit**

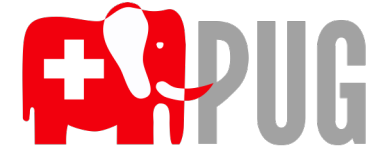




**Und schwups kopierten die 7 kleinen Server  
die Daten mit dem  
graphischen Migrator**



**Und wenn sie nicht  
rebootet haben, so  
laufen Sie heute noch**



# Maxing out max\_parallel\_workers

James Guthrie

[james@guthrie.ch](mailto:james@guthrie.ch)

```
SET max_parallel_workers = 8;  
SET max_parallel_workers_per_gather = 8;  
SET parallel_leader_participation = off;  
SET parallel_tuple_cost = 0;  
SET parallel_setup_cost = 0;  
SET min_parallel_table_scan_size = 0;
```



```
CREATE TABLE data(val INTEGER);  
INSERT INTO data (val) SELECT * FROM generate_series(1, 1000000
```

```
EXPLAIN (ANALYZE) SELECT sum(val) FROM data;
```

QU

---

```
Finalize Aggregate (cost=5987.53..5987.54 rows=1 width=8) (act
```

```
-> Gather (cost=5987.50..5987.51 rows=8 width=8) (actual t
```

```
Workers Planned: 8
```

```
Workers Launched: 8
```

```
-> Partial Aggregate (cost=5987.50..5987.51 rows=1 w
```

```
-> Parallel Seq Scan on data (cost=0.00..5675.0
```

```
Planning Time: 0.055 ms
```

```
Execution Time: 24.413 ms
```

```
(8 rows)
```

```
SET max_parallel_workers = 16;
```

```
SET max_parallel_workers_per_gather = 16;
```

```
EXPLAIN (ANALYZE) SELECT sum(val) FROM data;
```

QU

-----  
Finalize Aggregate (cost=5987.53..5987.54 rows=1 width=8) (actual time=24.535 ms)

-> Gather (cost=5987.50..5987.51 rows=8 width=8) (actual time=24.535 ms)

Workers Planned: 8

Workers Launched: 8

-> Partial Aggregate (cost=5987.50..5987.51 rows=1 width=8) (actual time=24.535 ms)

-> Parallel Seq Scan on data (cost=0.00..5675.00 rows=8 width=8) (actual time=0.000 ms)

Planning Time: 0.051 ms

Execution Time: 24.535 ms

(8 rows)

```
/*
 * Select the number of workers based on the log of the size of
 * the relation. This probably needs to be a good deal more
 * sophisticated, but we need something here for now. Note that
 * the upper limit of the min_parallel_table_scan_size GUC is
 * chosen to prevent overflow here.
 */
heap_parallel_threshold = Max(min_parallel_table_scan_size, 1);
while (heap_pages >= (BlockNumber) (heap_parallel_threshold * 3))
{
    heap_parallel_workers++;
    heap_parallel_threshold *= 3;
    if (heap_parallel_threshold > INT_MAX / 3)
        break;    /* avoid overflow */
}

parallel_workers = heap_parallel_workers;
```

```
SELECT ceil(log(3, relpages)) FROM pg_class WHERE relname = 'data'
```

```
ceil
```

```
-----
```

```
8
```

```
(1 row)
```

```
ALTER TABLE data SET (parallel_workers = 64);
```

```
SET max_parallel_workers = 64;
```

```
SET max_parallel_workers_per_gather = 64;
```

```
EXPLAIN (ANALYZE) SELECT sum(val) FROM data;
```

QUI

-----  
Finalize Aggregate (cost=4620.48..4620.49 rows=1 width=8) (actual t

-> Gather (cost=4620.31..4620.32 rows=64 width=8) (actual t

Workers Planned: 64

Workers Launched: 64

-> Partial Aggregate (cost=4620.31..4620.32 rows=1 wi

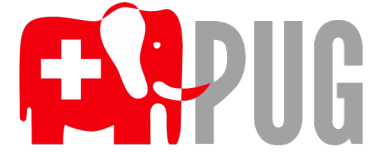
-> Parallel Seq Scan on data (cost=0.00..4581.2

Planning Time: 0.051 ms

Execution Time: 49.251 ms

(8 rows)

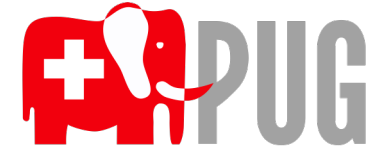




# pg\_replication\_slot\_advance with 'skip'

Patrick Stählin

me@packi.ch



# Cloud and I/O: going full circle

Frits Hoogland

@fritshoogland