CYBERTEC

DATA SCIENCE & POSTGRESQL

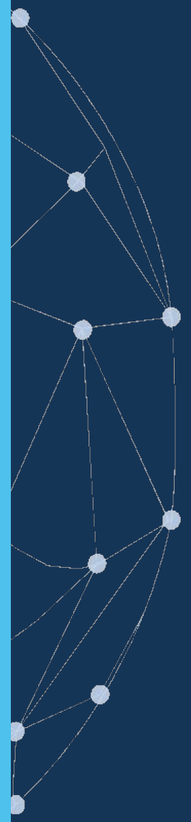
How to handle 1000 application users

Laurenz Albe

Laurenz Albe

- PostgreSQL contributor since 2006
- author of oracle_fdw and other PostgreSQL related software
- consultant
- trainer

DATABASE SERVICES



DATA Science

- Artificial Intelligence
- Machine Learning
- Deep Learning
- Big Data
- Business Intelligence
- Data Mining
- Etc.

PostgreSQL Services

- 24/7 Support
- High Availability
- Consulting
- Performance Tuning
- Clustering
- Migration
- Etc.

CYBERTEC
DATA SCIENCE & POSTGRESQL
www.cybertec-postgresql.com

AUSTRIA (HQ)

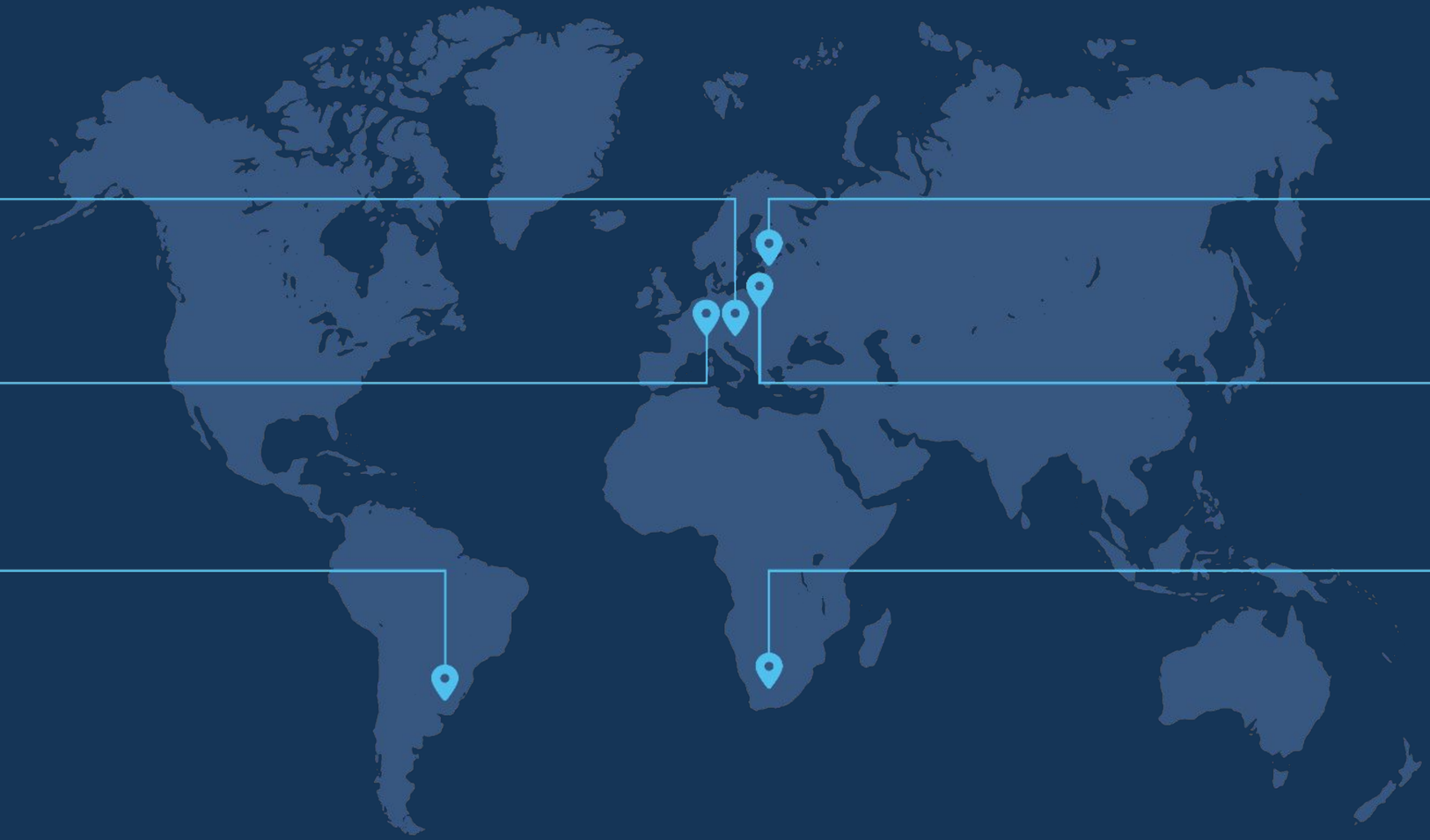
CYBERTEC POSTGRESQL
INTERNATIONAL (HQ)

SWITZERLAND

CYBERTEC POSTGRESQL
SWITZERLAND

URUGUAY

CYBERTEC POSTGRESQL
SOUTH AMERICA



ESTONIA

CYBERTEC POSTGRESQL
NORDIC

POLAND

CYBERTEC POSTGRESQL
POLAND

SOUTH AFRICA

CYBERTEC POSTGRESQL
SOUTH AFRICA

CYBERTEC
DATA SCIENCE & POSTGRESQL
www.cybertec-postgresql.com

DATABASE PRODUCTS

scalefield

CYPEX

CYBERTEC MIGRATOR

DATA MASKING

CYBERTEC PGEE

POSTGRESQL TDE

KUNDEN BRANCHEN

- IT
- universities
- government
- automotive
- industry
- retail
- financial
- and many more

What is this talk about?

- Management of database connections
- connection pooling
- sizing of connection pools
- pitfalls you should avoid

Problem statement

Requirements of the applikation

- we have to handle 1000 concurrent application users
- we need many processes/threads to handle that load
- we need several instances of the application server, potentially on different machines
- each application thread accesses the database frequently

The naïve approach

The naïve approach

- we open a database connection whenever an application thread needs to access the database and close the connection when we are done
- very bad idea, because opening database connections is expensive:
  - starts a process on the database server
  - loads catalog tables into a cache
  - authenticates the database user
- if database requests are short, this can consume more than half of the database server's resources just for establishing connections

Testing the naïve approach

pgbench with persistent database connections:

```
$ pgbench -c 5 -T 60 test
tps = 4163.234087
```

pgbench opens a new connection for each request:

```
$ pgbench --connect -c 5 -T 60 test
tps = 446.301527
```

Even with local connections performance is reduced by almost 90%!

The “less naïve” approach

Keep database connections open

- each application thread keeps its database connections open rather than closing them
- reuse database connections rather than close them
- idea: idle connections don't use many resources
- can lead to thousands of open database connections

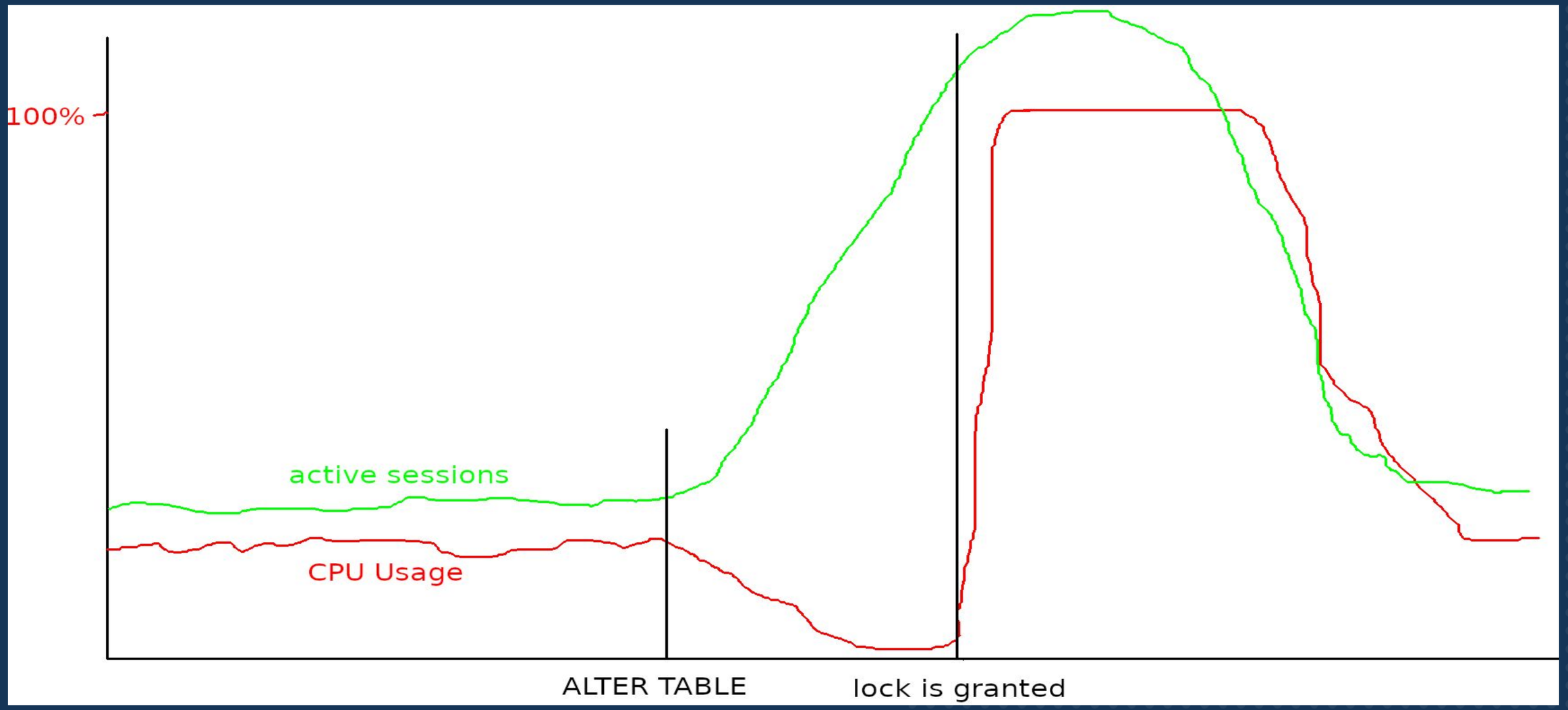
Problems with many open sessions

- idle database connections also affect the performance:
  - slow down the “snapshot” created for each SQL statement
  - this was improved in PostgreSQL v14, but is still relevant
- the number of active (working) database sessions cannot be limited in PostgreSQL
  - this can overload the database
  - the risk of overload grows with the number of database connections
- if there are many connections, `work_mem` has to be kept small to avoid going “out of memory”
  - that is bad for the performance of SQL statements

Causes of database overload

- CPU is at capacity
  - all SQL statements execute much slower
- disk is at capacity
  - all SQL statements execute much slower
  - I/O-“wait events” like `WALSync`, `DataFileRead`, ...
- internal resource conflicts in the database
  - “wait events” for internal “light-weight locks” like `LockManager`, `BufferContent`, ...
- `wait_events` can be monitored with `pg_stat_activity`

An example of database overload



(any similarity to existing customer systems is coincidental)

The example explained

- Statements get stuck behind an `ACCESS EXCLUSIVE` lock
- “active” connections pile up
- at some point the lock goes away
- the avalanche is turned loose ⇒ CPU spike
- intuition advises:
  there are many active connections, so we need to increase the limit
- however, more connections exacerbate the problem

Fighting database overload by using a connection pool

Limiting the connection count

- `max_connections` is the cluster-wide limit
- `ALTER DATABASE|ROLE ... CONNECTION LIMIT n`
- error message if the limit is exceeded
  ⇒ no user-friendly solution
- we would like to limit the number of **active** database connections
  - not possible in PostgreSQL

What is a connection pool?

- a piece of software that keeps a number of database connections open
- client processes grab a connection from the pool and return it to the pool after use
- if all connections are busy, the client application is suspended
- typically built into the application server, but can also be a standalone program (pgBouncer)

Advantages of a connection pool

- connections are kept open
  ⇒ no waste of resources by opening new connections
- ideal configuration: minimal pool size = maximal pool size
- the number of active database connections is limited
  ⇒ prevents database overload
- fewer database processes
  ⇒ more resources for each process
  ⇒ fewer “context switches” in the CPU

Pooling strategy “session pooling”

- application thread holds a database connection for its whole life time
- only useful if application threads are short-lived
- no restriction on the available SQL constructs (the state of a session gets reset when it is returned into the pool)

Pooling strategy “statement pooling”

- connection is returned to the pool after each statement
- most effective strategy that uses few connections (connections that are held by an application thread are never idle)
- no multi-statement transactions possible (except when using database functions)
  ⇒ limited usefulness

pooling strategy “transaction pooling”

- connections are returned to the pool after each database transaction
- the best compromise between efficiency and usability
- cannot be used with SQL constructs whose life time exceeds a database transaction

Connection pooling: limitations

- database connections can only be reused for the same database and user
  ⇒ use only a single database user
- except with session pooling, constructs that live longer than a transaction cannot be used:
  - temporary tables (workaround: `UNLOGGED` tables)
  - `WITH HOLD` cursors (workaround: `UNLOGGED` tables)
  - prepared statements
- PL/pgSQL functions can be used instead of prepared statements, because they also cache the execution plans of SQL statements

Pooling in the application server

- most efficient
- easy to use (most application servers and ORMs have built-in support)
- effective pooling if there is a single application servers (or very few)
- with many application servers you will end up with many pools
  ⇒ number of active database connections is not limited effectively

Connection pooling with pgBouncer

- https://www.pgbouncer.org/
- simple software, proxy between client and database
- usually installed on the database machine (local connections)
- fewer features than pgPool, but simpler and more robust
- use it if there is no built-in connection pool or the built-in pool is not effective (e.g. with many application servers)

Authentication and pgBouncer

- pgBouncer is a proxy = “man in the middle”
  ⇒ application authenticates with pgBouncer,
  pgBouncer authenticates with PostgreSQL
- password authentication is easy to set up
- TLS certificate authentication is also straightforward (particularly if pgBouncer runs on the database machine)
- other authentication techniques are complicated
  ⇒ potential problem with high security requirements

Sizing the connection pool

The problem of sizing

- if the pool is too small, application performance will be bad
- if the pool is too large, the database will be overloaded, and application performance will also be bad
- pool size usually determined by trial and error
  (but we will try to do better!)
- good load tests are a big help

Limits for active connection count

- the limits are usually determined by the CPU and disk capacity
  - not more active connections than CPU cores
  - not more active connections than parallel I/O requests that the disk can handle
- database-internal resource conflicts are hard to estimate and will be ignored here (experience tells that they mostly happen with high numbers of connections, which we will avoid anyway)

Accounting for “idle in transaction”

- connections in use by the application are not always “active”
- in-use connections that are idling don't use CPU or disk resources
- we have to figure out the following ratio:

$$\texttt{active\_factor} = \frac{\text{connection active time}}{\text{connection in use time}}$$

Determining `active_factor`

- easy with the new session statistics in PostgreSQL v14:

```
SELECT active_time / (active_time + idle_in_transaction_time)
FROM pg_stat_database
WHERE datname = 'mydb'
AND active_time > 0;
```

- for a coarse estimate, query `pg_stat_activity` several times and use an average:

```
SELECT (count(*) FILTER (WHERE state = 'active'))::float8
       / count(*) FILTER (WHERE state in
                                ('active', 'idle in transaction'))
FROM pg_stat_activity
WHERE datname = 'mydb';
```

Determining the ideal pool size

$$\text{pool size} = \frac{\min(\text{num\_cores}, \text{max\_parallel\_ios})}{\text{active\_factor} \times \text{parallelism}}$$

- “num_cores” is the number of CPU cores
- “max_parallel_ios” is the upper limit of concurrent I/O requests that the disk can handle
- “parallelism” is the average number of server processes used for a single SQL statement

Impact of query duration

- the shorter the statements, the more you can run
- the more statements you can run, the more concurrent application users your system can handle
  ⇒ **tune queries as good as possible**
- keep “idle in transaction” time as short as you can
  (otherwise the connection pool is not used effectively)
- avoid long transactions as much as you can (always commendable)

The most important points

Conclusion

- you need a single connection pool
- if you cannot get that from the application server ⇒ pgBouncer
- don't configure the pool size too large
- you can calculate the ideal connection pool size
- short statements and transactions ⇒ many concurrent users

Questions