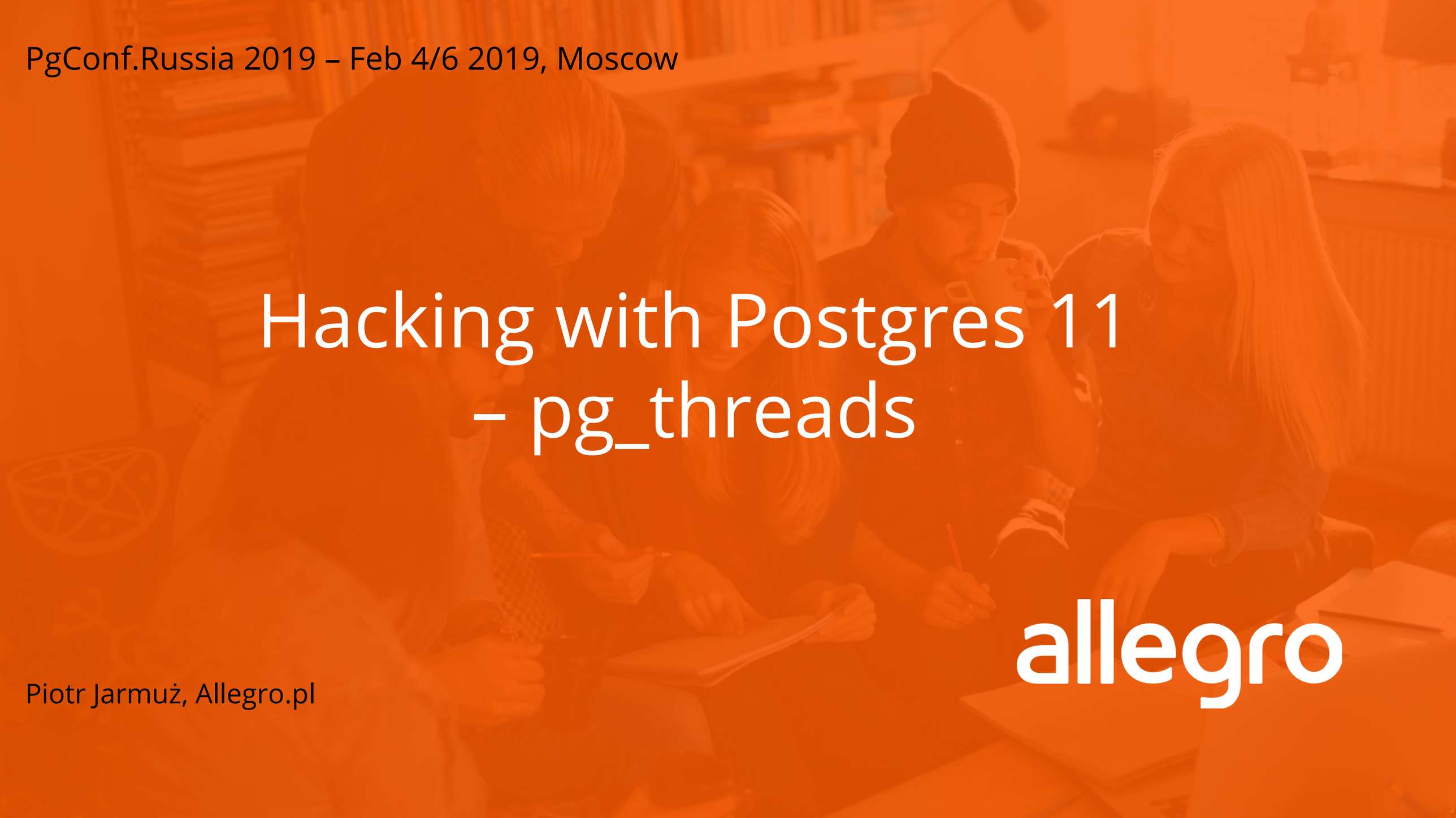
PgConf.Russia 2019 – Feb 4/6 2019, Moscow

Hacking with Postgres 11 – pg_threads



Piotr Jarmuż, Allegro.pl

Agenda

- Introduction about me and my company
- PostgreSQL 11 stored procedures current state and history
- Writing extensions – technical background
- pg_threads – building POSIX thread API using PostgreSQL extension (3)
- How does it work – example usage
- Transactional and nontransactional API – another extension (3)
- How does it work – example usage
- Putting it all together – solving a Wordament game in single thread
- Game solution using pg_threads – scaling up
- Adding another node – scaling out
- Conclusions

PostgreSQL stored procedures current state and history

- How developers see RDBMS vs. what really a modern RDBMS is
- PostgreSQL offers 2D extensibility: create language: create procedure
- PostgreSQL default language is PL/pgSQL
- in PostgreSQL core since 1998 v6.4- loosely based on Oracle PL/SQL
- PL/pgSQL functions, procedures, triggers fully fledged procedural language
- reduced network traffic, encapsulation, security
- low level "C" functions – usually base for PostgreSQL extensions
- resources – official documentation - https://www.postgresql.org/docs/11/static/server-programming.html
- tutorials - http://www.postgresqltutorial.com/postgresql-stored-procedures/

Execution contexts in PostgreSQL

- PostgreSQL is a multiuser, multiprocessing environment
- in the simplest each psql session constitutes an execution context
- in stock version we lack a powerful abstraction of threads
- extension to the rescue – pg_threads
- abstract API borrowed from POSIX threads
  - ✓ create_thread(name,thread_proc,hostname:=NULL);
  - ✓ start_thread(name);
  - ✓ join_thread(name);
  - ✓ destroy_thread(name);

Extension: pg_threads

- using libpq client library
- asynchronous query execution
- exposing thread state via regular table thread_list
- using PostgreSQL backend processes as thread containers
- a thread has a state CREATED,RUNNING,FINISHED
- still in statu nascendi – API may change in future
- data separation – local variables, local temporary tables – private per thread
- data sharing – regular PostgreSQL tables – shared among threads
- time for simple demos: sleepers and idlers

Threads need to communicate – non-transactional API

- By default threads can use regular Postgres tables to communicate
- Using tables is transactional and lacks synchronization primitives
- Need for well defined synchronous/asynchronous communication API
- Non-transactional API – pg_pipe – loosely based on UNIX pipes
- Private and public pipes, blocking and nonblocking mode - timeout
- Non persistent, all unreceived messages lost on instance restart
- Uses dynamic background worker process – pipe server
- Useful for debugging, communication with external service
- Multiplexing large number of users over fewer connections
- Independent transactions

Threads need to communicate – transactional API

- Stock Postgres version has LISTEN, NOTIFY, pg_notify
- Has limitations, no timeout and difficult to pass data programatically
- For complementary purposes – pg_alert – transactional communication
- Transaction based, blocking and nonblocking mode – timeout
- Alerts are sent only sent on COMMIT
- Loosely based on UNIX signals but has idempotency property
- For communication with external service on transaction boundaries
- Uses pg_pipe + Postgres native advisory locks API

Threads need to communicate – tracking thread/session progress

- Threads should also be able to expose its current progress
- In stock version possible writing to log or on a console: raise notice
- Another module pg_app_info implements this feature
- Exposes non-transactionally extra thread info (module, action)
- Info can be updated independently on the transaction boundaries
- Data is visible in a table that can be joined to pg_stats_activity
- Useful for monitoring, tuning and debugging via regular select
- Uses pg_pipe + background process for session tracking

Wordament game

- original from Microsoft
- popular as mobile app
- displays a board 4x4 with random letters
- goal is find as many words as long as possible
- 120 sec for solution
- 3 letter minimum length
- no reuse of board tiles in current run
- great for learning new words :)

CART

Solution - single thread

- data structures
  - ✓ current word being built – local variable
  - ✓ board representing state of the game - temporary table
  - ✓ solution table for found words - temporary table
  - ✓ dictionary for checking valid words - regular PostgreSQL table
- algorithm used – depth first search tree with dynamic decision pruning
- Unicode support for many languages

```
$> (echo "begin;"; aspell -d ru dump master | aspell -l ru expand |
sed 's/ /\n/g' | (sed -r 's/(.*)/\U\1/g'| sort | uniq -i| sed = | sed
'N;s/\n/\t/'";s/'/''/g"| sed -r "s/(.*)\t(.*)/insert into words_ru
values (\1,'\2');/g"; echo "commit;" )) | psql -d wordament
```

Solution - single thread

- let’s play!
- psql> select play(‘xtoe evrc aean ygas’);

Solution - single thread

psql> select play('xtoe evrc aean ygas');

CARNAGE

CAVERNS

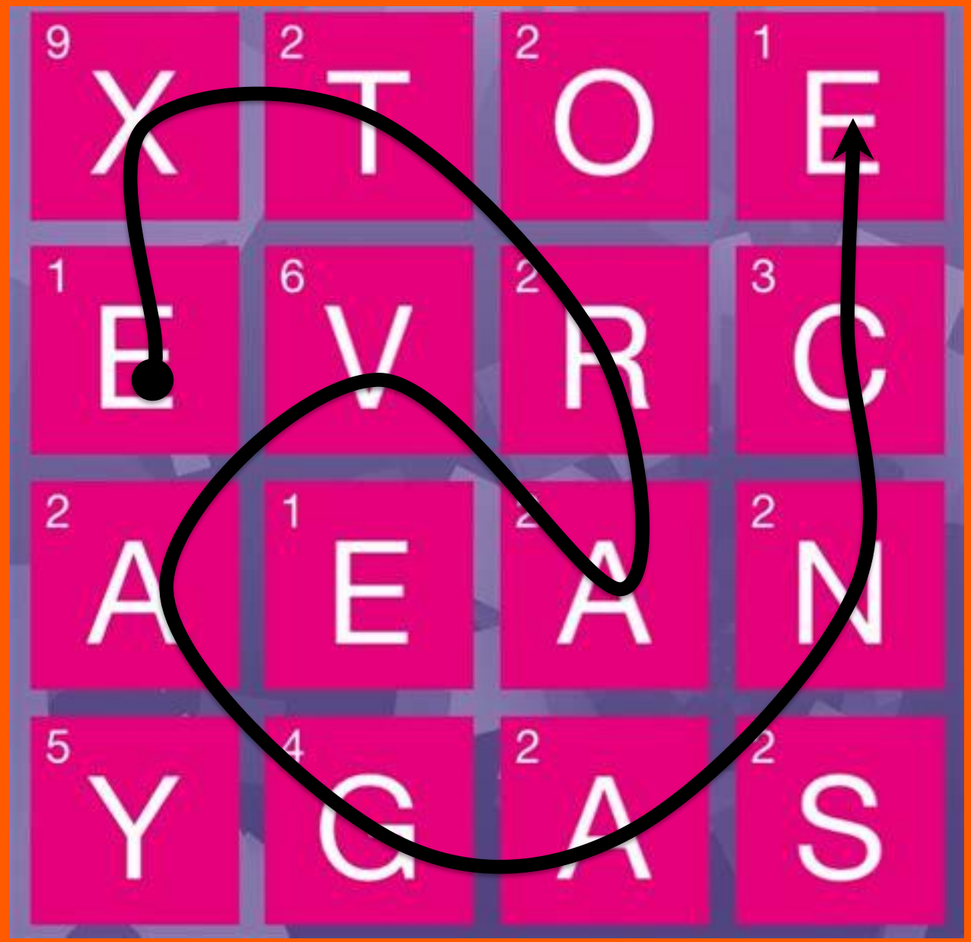
CORNAGE

CRANAGE

EXTRAVAGANCE

(275 rows)

Time: 2413.135 ms

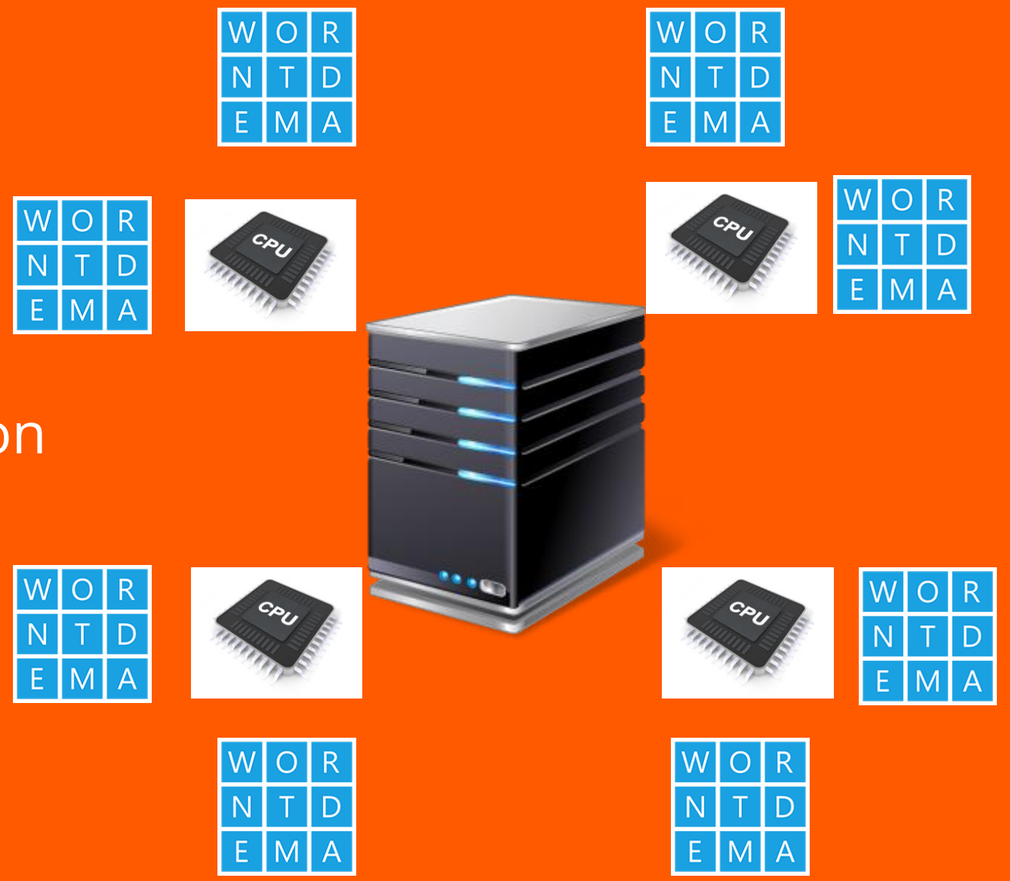


Inherent parallelism in game

- in general game theory is a branch of mathematics
- lots of practical applications in economy, military
- easily parallelizable – “embarrassingly parallel”
- Wordament game is no other than that – inherent parallelism
- up to 16 independent search trees can be run in parallel

Solution - multiple threads

- refactoring code a bit
- partitioning root search for distributing load
- replicating game state
- expanding data structures - new table gsolution
- it scales up!



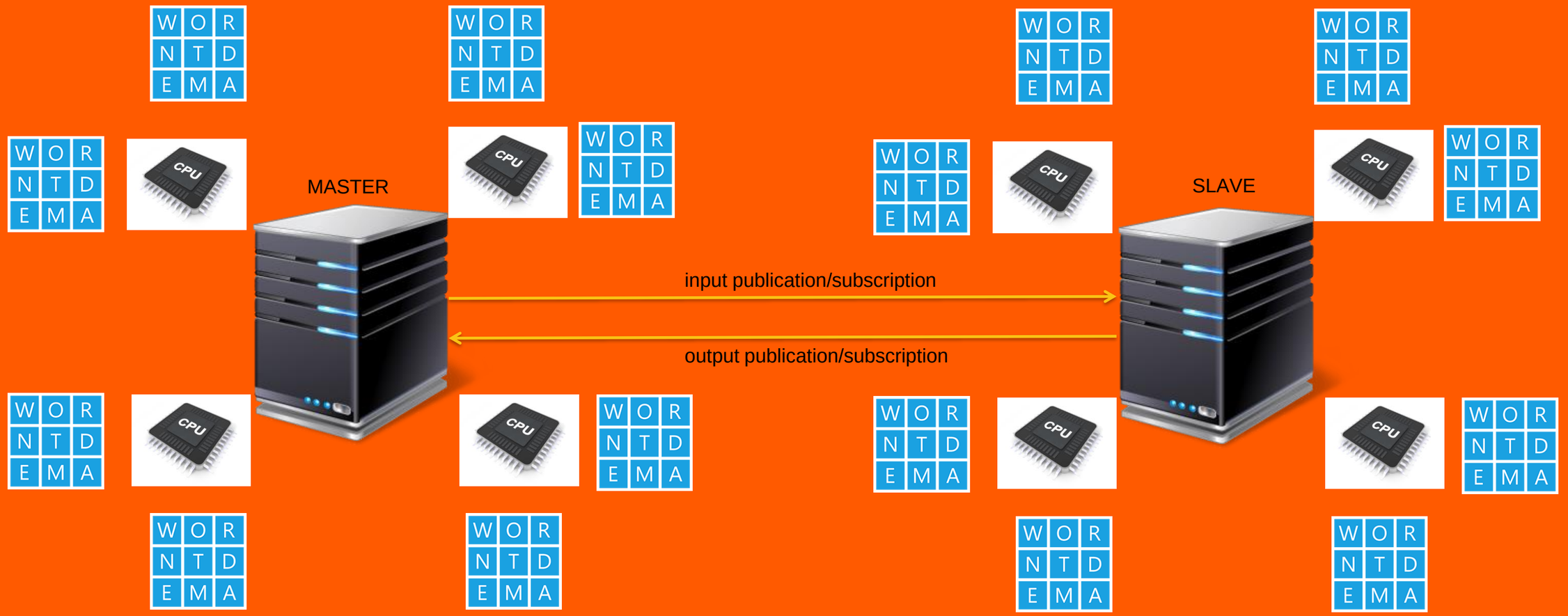
Scaling out – another node, pg_logical

- scaling out > scaling up – use commodity hardware
- using pg_logical publication and subscription
- another node set-up in logical replication – asymmetric – pg_logical is unidirectional
- 2 sets of PUB/SUB: input and output
- input publication pushes input data to slave(s)
- output publication pushes output data to master – optional part
- alternatively master fetches remote data via FDW
- threads extension already support remote threads – execution context distribution
- pg_logical – data distribution/replication
- we can stick to paradigm – process data locally

Solution - multiple threads, multiple nodes

- no refactoring this time - we are already parallel
- enable slave host
- just run the same parallelized version
- let built-in thread scheduler pick up the hosts for running
- language tables converge via input publication -> to slave(s)
- gsolution table converges results via output publication/subscription -> to master
- it scales out!

Solution - multiple threads, multiple nodes



Conclusions

- PostgreSQL is inherently parallel environment
- needs a little user support in parallelization – user assisted
- more and more contexts use parallel workers already – out of the box
- scaling up and out – thread extension + logical replication and/or sharding
- next step look at PostgreSQL dynamic background processes
- try out threads in BDR environment
- PostgreSQL is a powerful computational environment that can be main data hub in your data center

Thank you for your attention.
Questions?