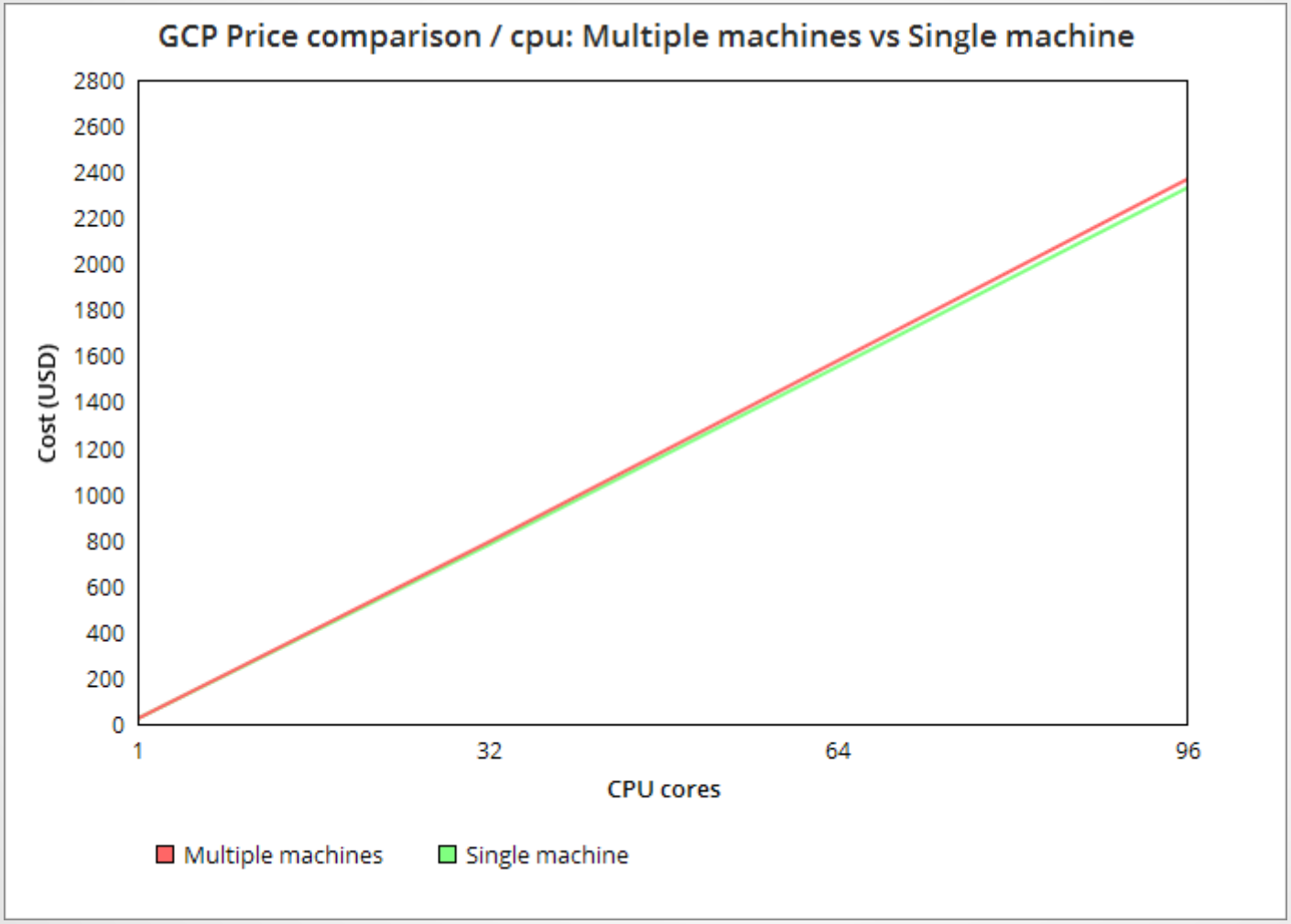
Pushing the limits on a single machine

A case study on squeezing the most out of a single PostgreSQL instance

By:
Ashu Pachauri

Why single instance?

Why single instance?



Why single instance?

- Multiple machines have:
  - Higher operational complexity
  - Higher fixed “tax” (OS, monitoring etc)
  - Higher application complexity

About me?

About me?

- 6+ years of building and managing DBs
- Scaling from scratch to 10s of Petabytes
- CTO @ Clarisights
- Apache HBase Committer
- Data Infra @ Facebook and Rocket Fuel

What is Clarisights?

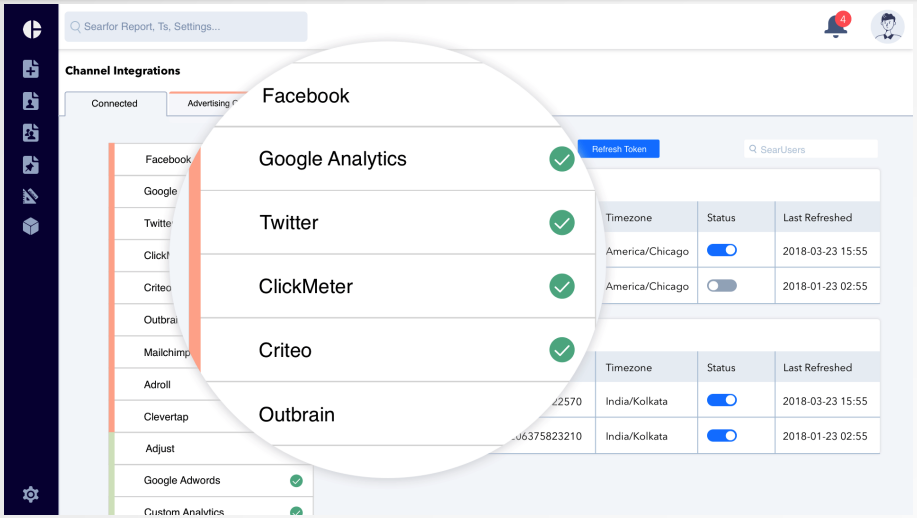
What is Clarisights?

- A SaaS platform
- Realtime, interactive and contextual reporting
- Targeted to high performance “marketing teams”
- Derives maximum value for medium to large enterprises

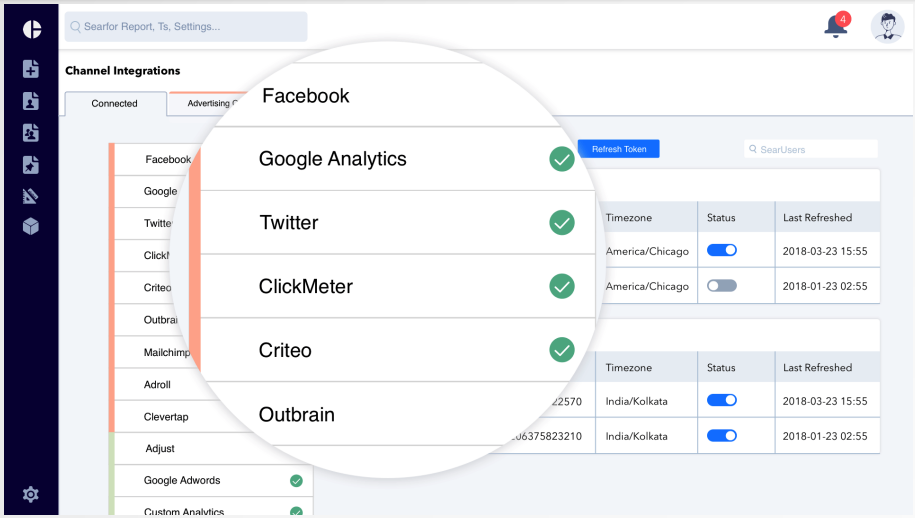
What is Clarisights?

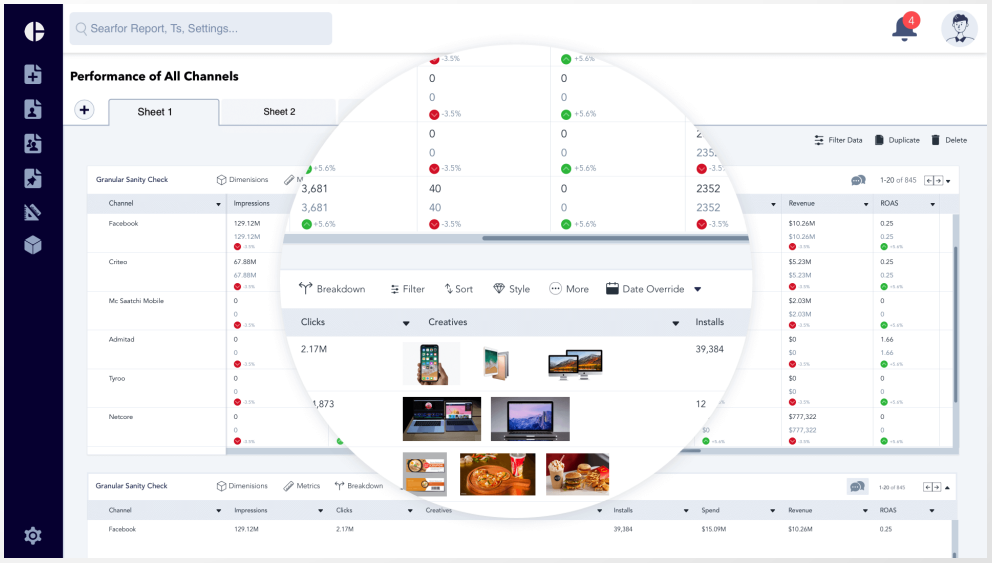
- A SaaS platform
- Realtime, interactive and contextual reporting
- Targeted to high performance “marketing teams”
- Derives maximum value for medium to large enterprises
- A single vertical solution for 3 problems:
  - API integration
  - Data transformation and enrichment
  - Interactive analytics

Clarisights



Clarisights





Data engg challenge @ Clarisights

Data engg challenge @ Clarisights

- Aggregate timeseries data for each Ad/Keyword etc
- 100s of dynamic user defined dimensions
  - Across different sources
  - E.g “City” can be defined for both FB and Google
  - Data enrichment after ingest: Historical updates to dimensions
- Reports (facts) with wide rows (10k+ metrics)
- Historical updates to facts
- High dimensional analytics on “mutating” data

Data engg challenge @ Clarisights

Data engg challenge @ Clarisights

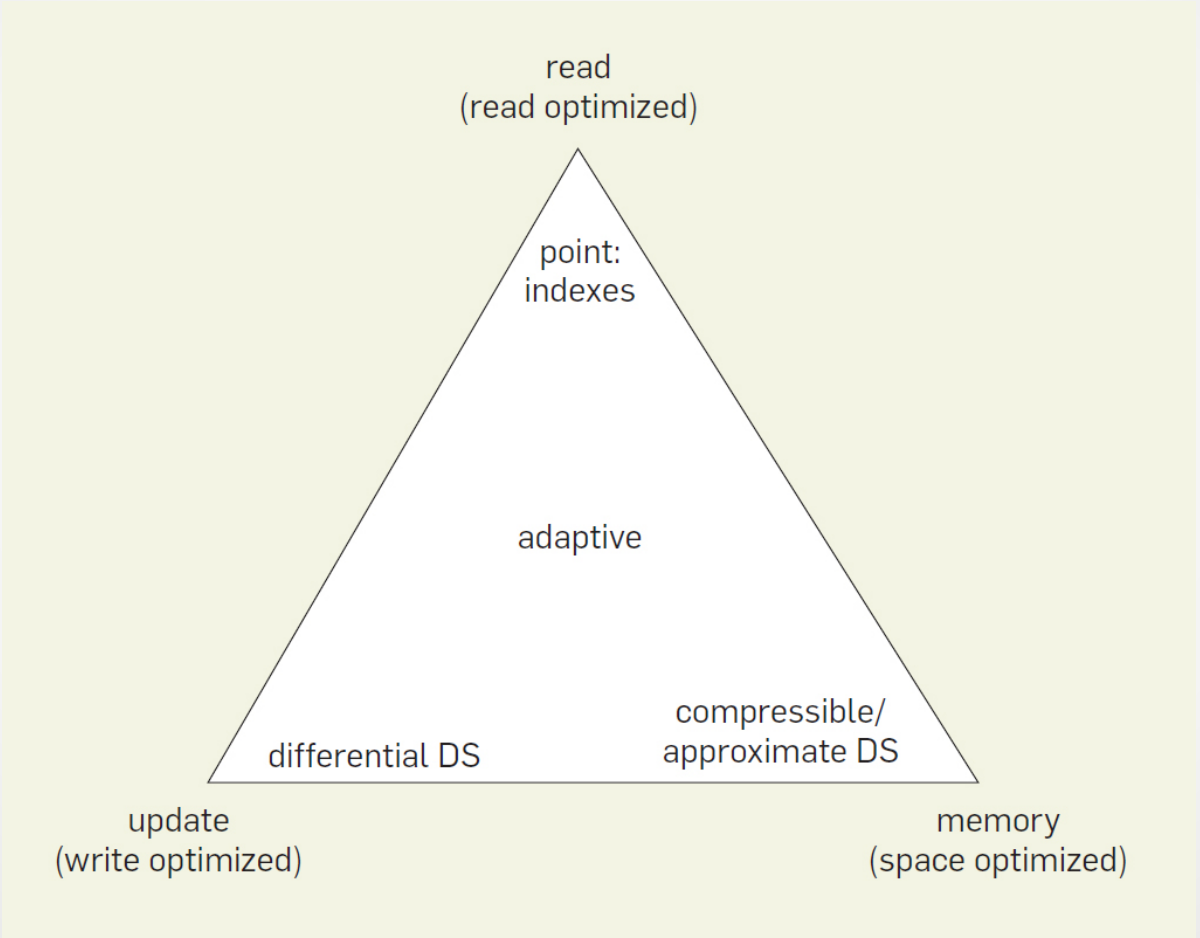
- An HTAP system that works for
  - 5 billion+ updates per day
  - 400 billion+ of reads per day
  - High dimensional space (500+ dimensions)
  - Constantly changing dimensions

Data engg challenge @ Clarisights

- An HTAP system that works for
  - 5 billion+ updates per day
  - 400 billion+ of reads per day
  - High dimensional space (500+ dimensions)
  - Constantly changing dimensions
  - *Still ongoing research in industry*

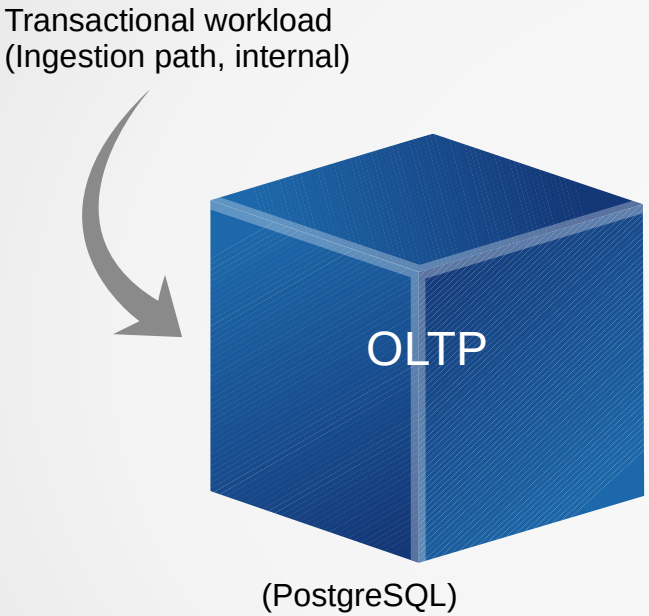
Problem with HTAP: RUM conjecture

Problem with HTAP: RUM conjecture

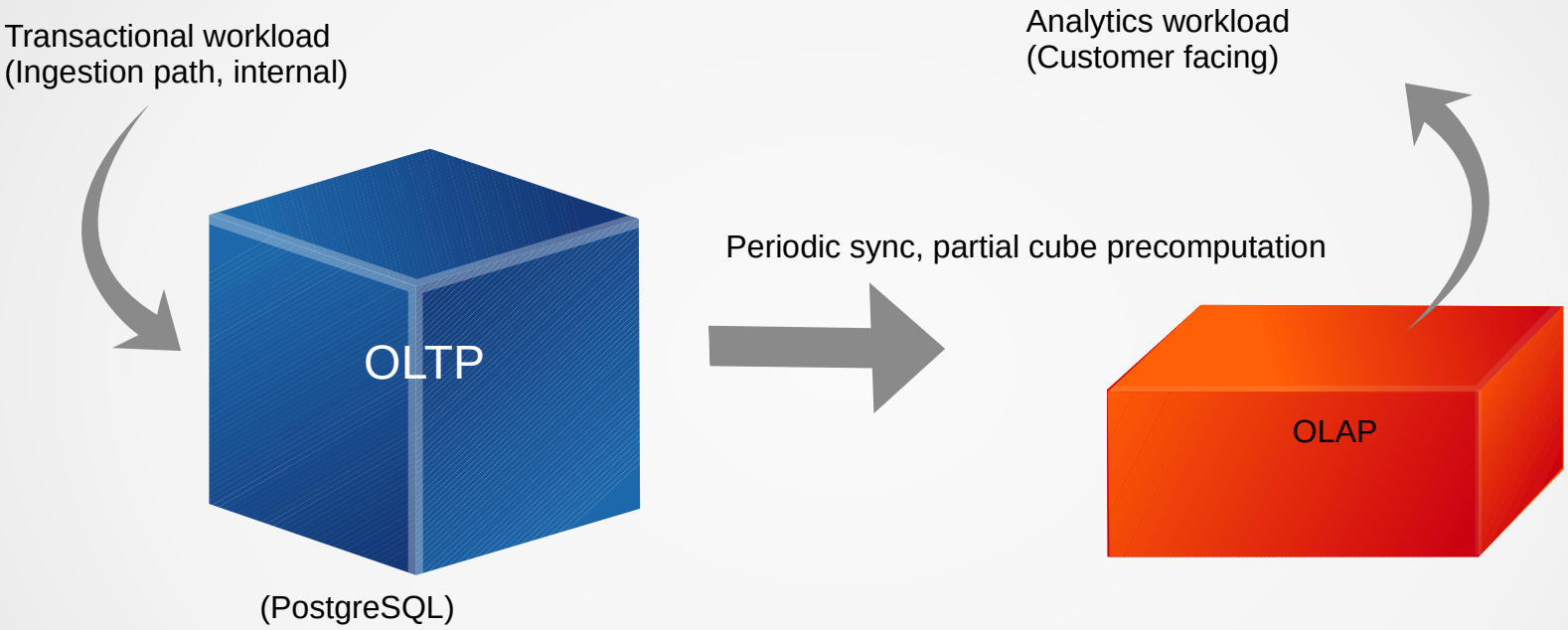


Two systems

Two systems


Transactional workload
(Ingestion path, internal)
OLTP
(PostgreSQL)

Two systems


Transactional workload
(Ingestion path, internal)
OLTP
(PostgreSQL)
Periodic sync, partial cube precomputation
Analytics workload
(Customer facing)
OLAP

General approach to scaling

General approach to scaling

- Scaling PostgreSQL as a service
  - Config/Systems tuning
  - Intelligent data storage layout

General approach to scaling

- Scaling PostgreSQL as a service
  - Config/Systems tuning
  - Intelligent data storage layout
- RUM aware access patterns

Optimizing for throughput

Optimizing for throughput

- Write latency is harder to optimize in PostgreSQL
  - Synchronous access patterns from client
  - Backed by asynchronous design of server
  - Design is more throughput driven than latency
- Read latency and throughput can both be optimized

Optimizing for throughput

- Write latency is harder to optimize in PostgreSQL
  - Synchronous access patterns from client
  - Backed by asynchronous design of server
  - Design is more throughput driven than latency
- Read latency and throughput can both be optimized
- Our use case is throughput driven

Optimizing for throughput: Concurrency

Approach

- Latency is not a problem
- Throw massive number of clients
- 10k+ clients connected simultaneously per instance

Optimizing for throughput: Concurrency

Approach

- Latency is not a problem
- Throw massive number of clients
- 10k+ clients connected simultaneously per instance

Problems

- Memory overhead for connections
- 1 GB allocation limit in PostgreSQL ~ 10K max connections

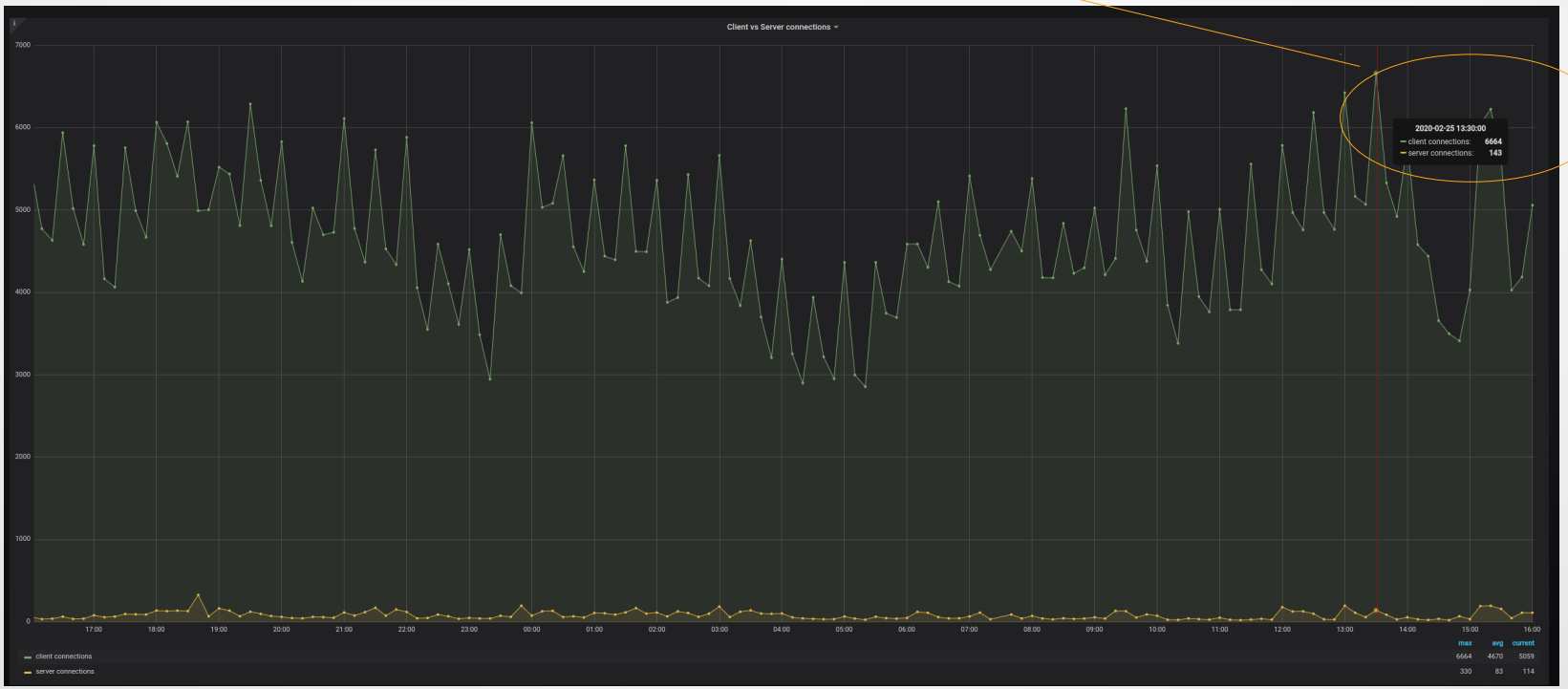
Concurrency: PGBouncer for the save

- Session, Transaction and Statement pooling modes

Concurrency: PGBouncer for the save

- Session, Transaction and Statement pooling modes
- Transaction pooling mode provides good enough protection

Server conns: 163
Client conns: 6664



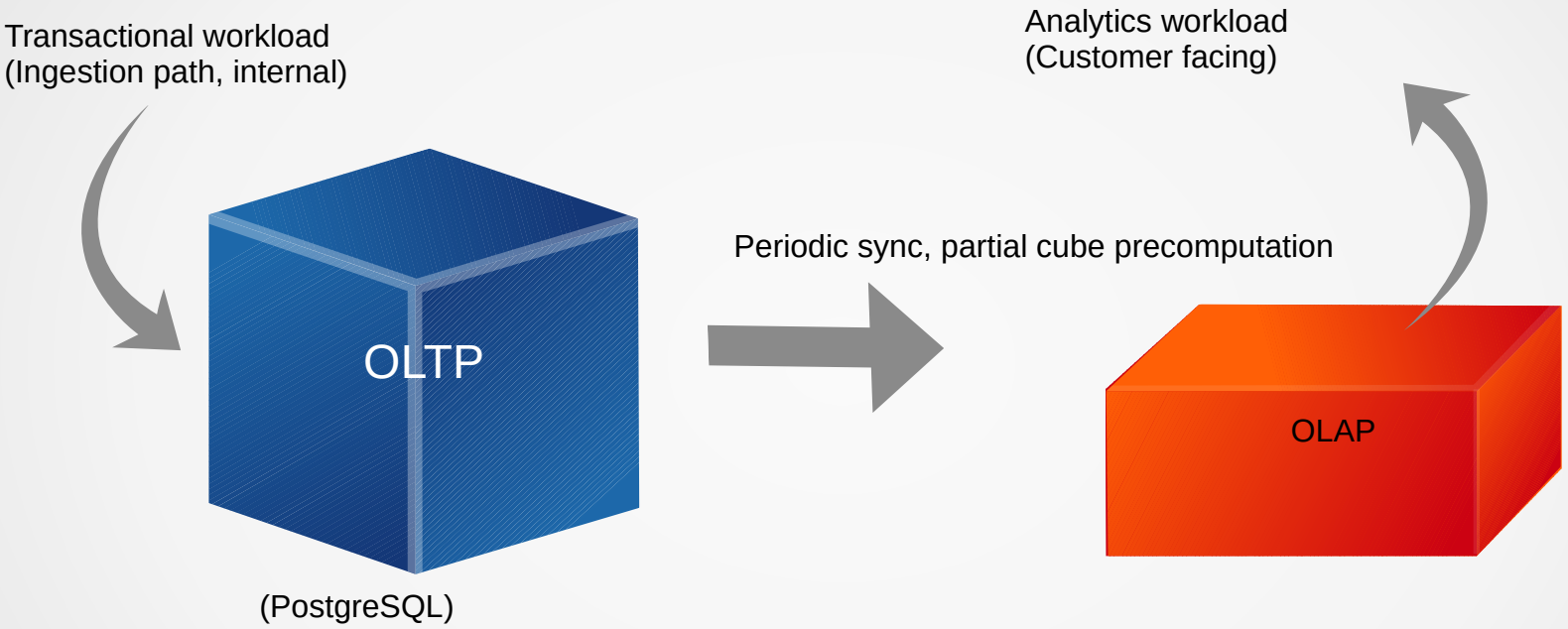
Transaction Pooling: What do we lose?

Transaction Pooling: What do we lose?

- Session specific features:
  - Prepared statements
  - Cursors with Hold
  - Advisory locks (Rails migrations)

Scaling reads: OLAP Sync

Scaling reads: OLAP Sync



OLAP sync: precomputation

- Queries over several days worth of data.
- Queries involving aggregations:

```
select sum(installs)::bigint as aj_installs,
sum(sessions)::bigint as aj_sessions, .....,
to_char(date_trunc('YYYY-MM-DD',
timestamp_utc::timestamp at time zone 'UTC')) as day
from adjust_revenues
where account_id = 1
group by internal_id, day
```

OLAP sync: Batch reads

- Need to read in batches of 1000 results

OLAP sync: Batch reads

- Need to read in batches of 1000 results

  Initial approach

- Rails Active Record *find_in_batches*

  select sum(installs)::bigint as aj_installs, ...
  group by account_id, internal_id, day ***ORDER BY id ASC OFFSET 1001 LIMIT 1000***

OLAP sync: Batch reads

- Need to read in batches of 1000 results

  Initial approach

- Rails Active Record *find_in_batches*

  select sum(installs)::bigint as aj_installs, ...
  group by account_id, internal_id, day ***ORDER BY id ASC OFFSET 1001 LIMIT 1000***

- Too much redundant work
  - Filter supported by indexes
  - But still need to do group by and order by

OLAP sync: Removing redundant work

OLAP sync: Removing redundant work

First optimization: Use cursors

- Transaction open for too long
  - Memory overhead
  - Impacts other operations

OLAP sync: Removing redundant work

First optimization: Use cursors

- Transaction open for too long
  - Memory overhead
  - Impacts other operations

Second optimization: Use cursors with hold

- Session specific feature
- Does not work with transaction pooling

OLAP sync: Removing redundant work

OLAP sync: Removing redundant work

Third optimization: Materialized views

- Do not automatically refresh
- Need to clean up old data from the view: expensive

OLAP sync: Removing redundant work

Third optimization: Materialized views

- Do not automatically refresh
- Need to clean up old data from the view: expensive

Fourth optimization: Temporary tables

- We got it right this time

OLAP sync: Removing redundant work

Third optimization: Materialized views

- Do not automatically refresh
- Need to clean up old data from the view: expensive

Fourth optimization: Temporary tables

- We got it right this time
- Did we?

OLAP sync: Temporary tables

OLAP sync: Temporary tables

- Replication stopped working
- Temporary tables' writes are also logged to WAL
- Too many writes to the WAL
  - Replication can't run fast enough to keep up

OLAP sync: Final solution

- Normal PostgreSQL tables
- Unlogged tables
  - Not written to WAL
  - No need to replicate or recreate
- Important: Separate tablespace and disk

Unlogged tables: What did we lose?

- Need to cleanup the tables
- Need to adjust monitoring scripts, automated workflows
- pg_class fragmentation

Scaling reads: Indexing “impactful” queries

Scaling reads: Indexing “impactful” queries

- Attempt 1: Index suggestions using external tools
  - Dexter, PGHero
  - Problem: Targetted only towards slow queries

Scaling reads: Indexing “impactful” queries

- Attempt 1: Index suggestions using external tools
  - Dexter, PGHero
  - Problem: Targetted only towards slow queries
- Attempt 2: Using application sampling profiler to pinpoint bottlenecks
  - Problem: Higher turnaround time to pinpoint PG queries of interest

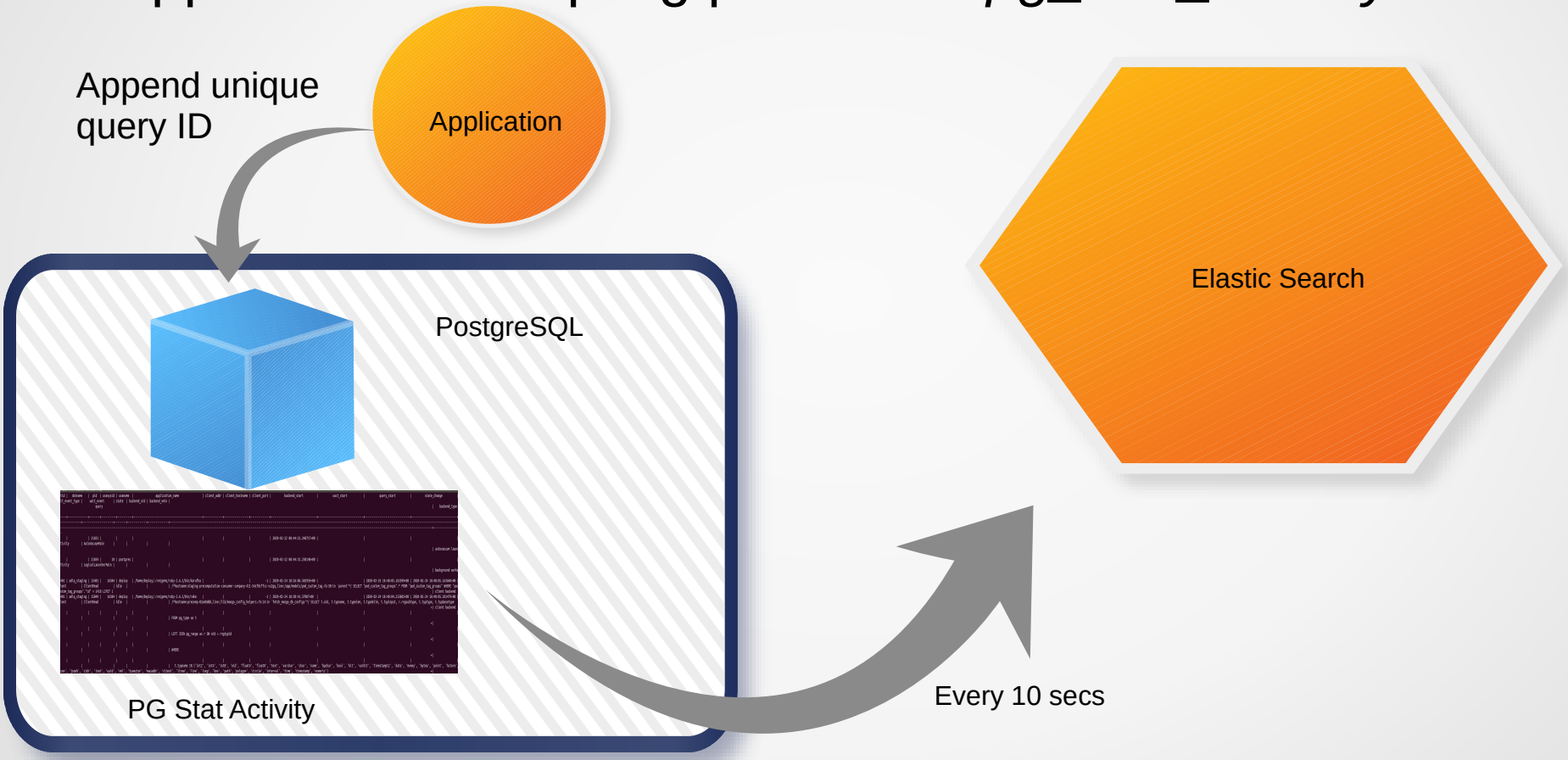
Scaling reads: Finding “impactful” queries

Scaling reads: Finding “impactful” queries

- Approach 3: Sampling profiler on *pg_stat_activity*

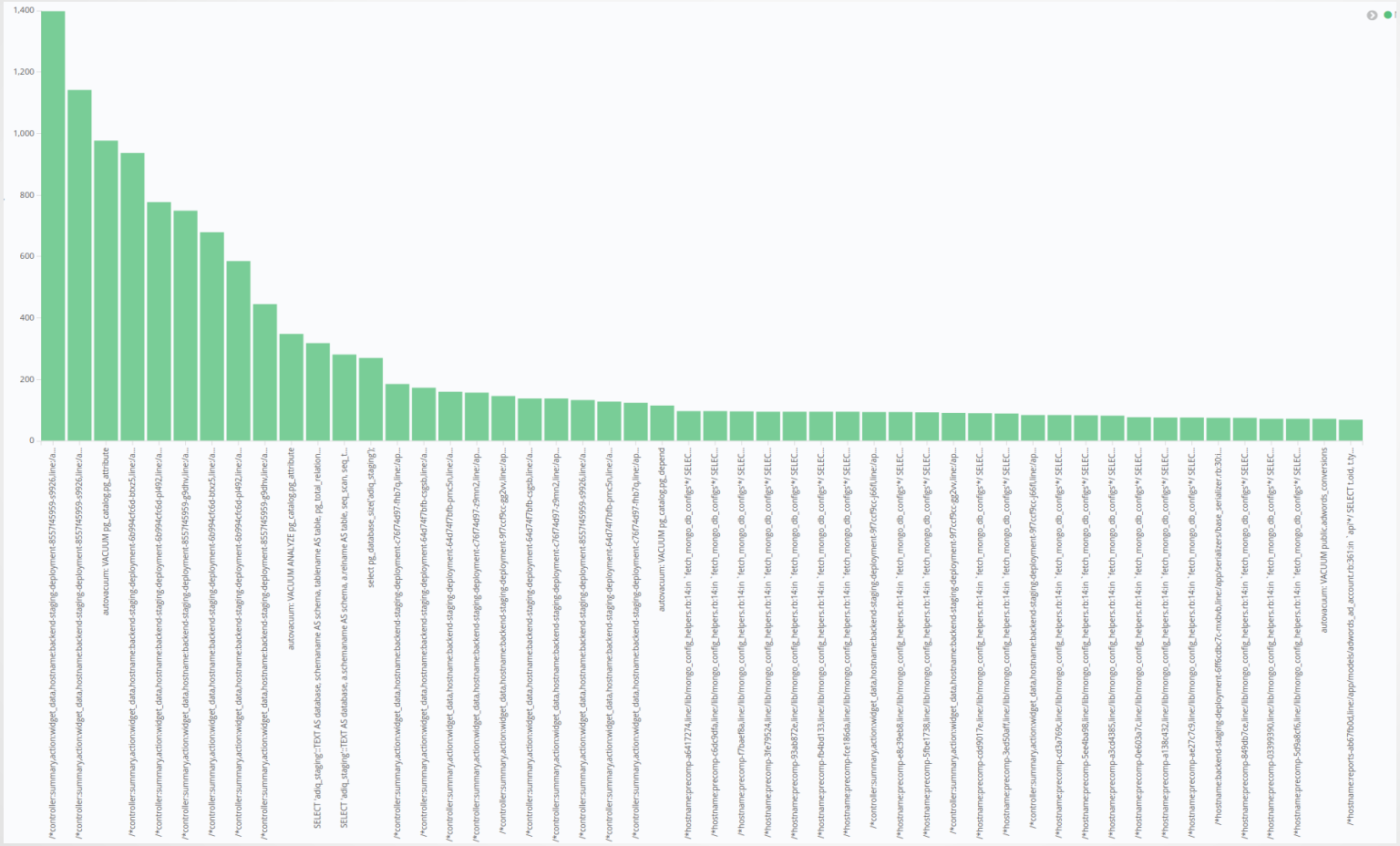
Scaling reads: Finding “impactful” queries

- Approach 3: Sampling profiler on *pg_stat_activity*



Scaling reads: Finding “impactful” queries

Fig: Count of queries by ID



No. of times query was active (Last X minutes)

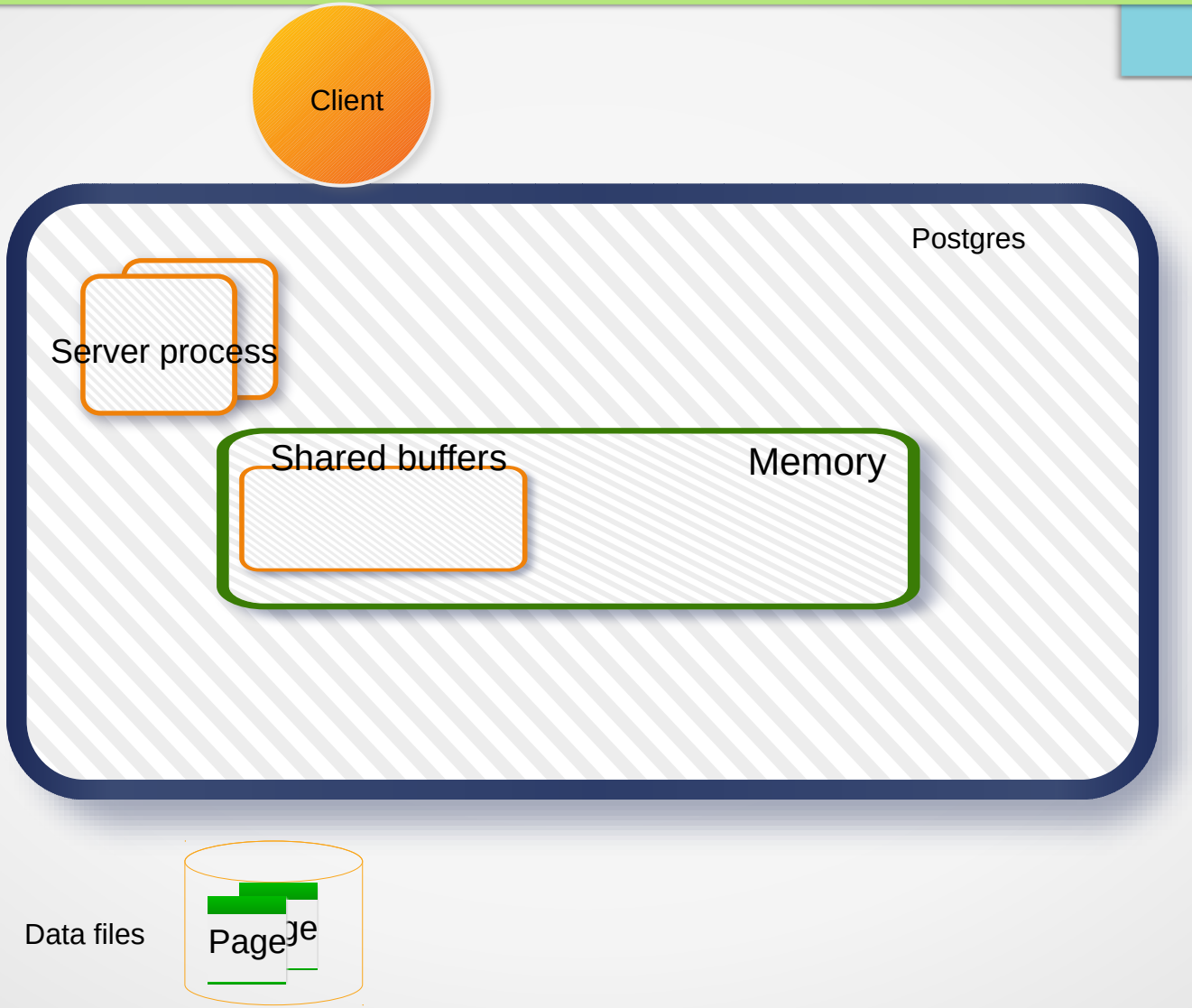
clarisights

Scaling writes: Write amplification

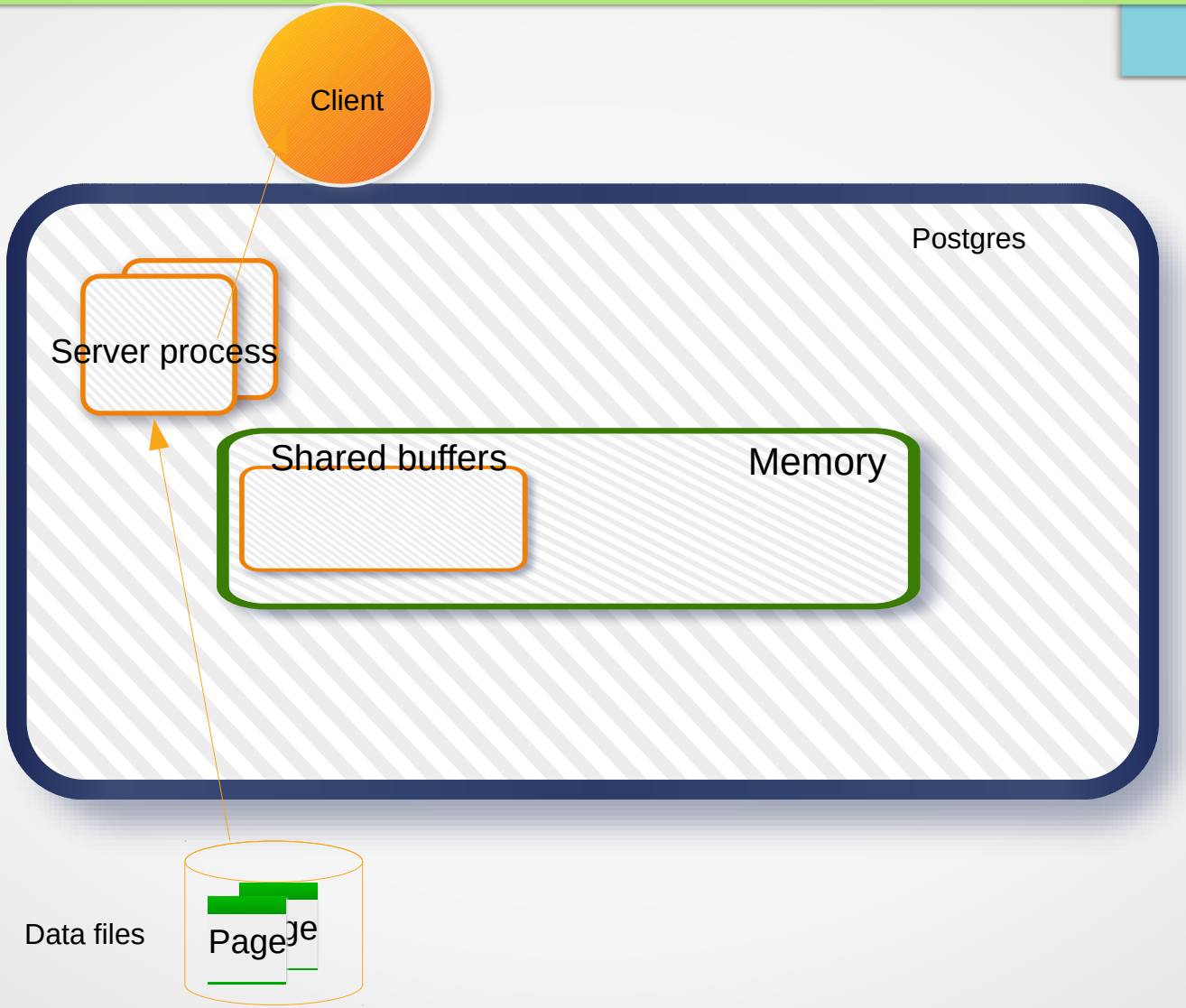
- PostgreSQL is not “optimized” for point updates
- Any update is a Copy-on-Write (CoW) operation
  - PostgreSQL deals only in pages (8KB)
- The CoW happens with a delay

Scaling writes: Delayed write amplification

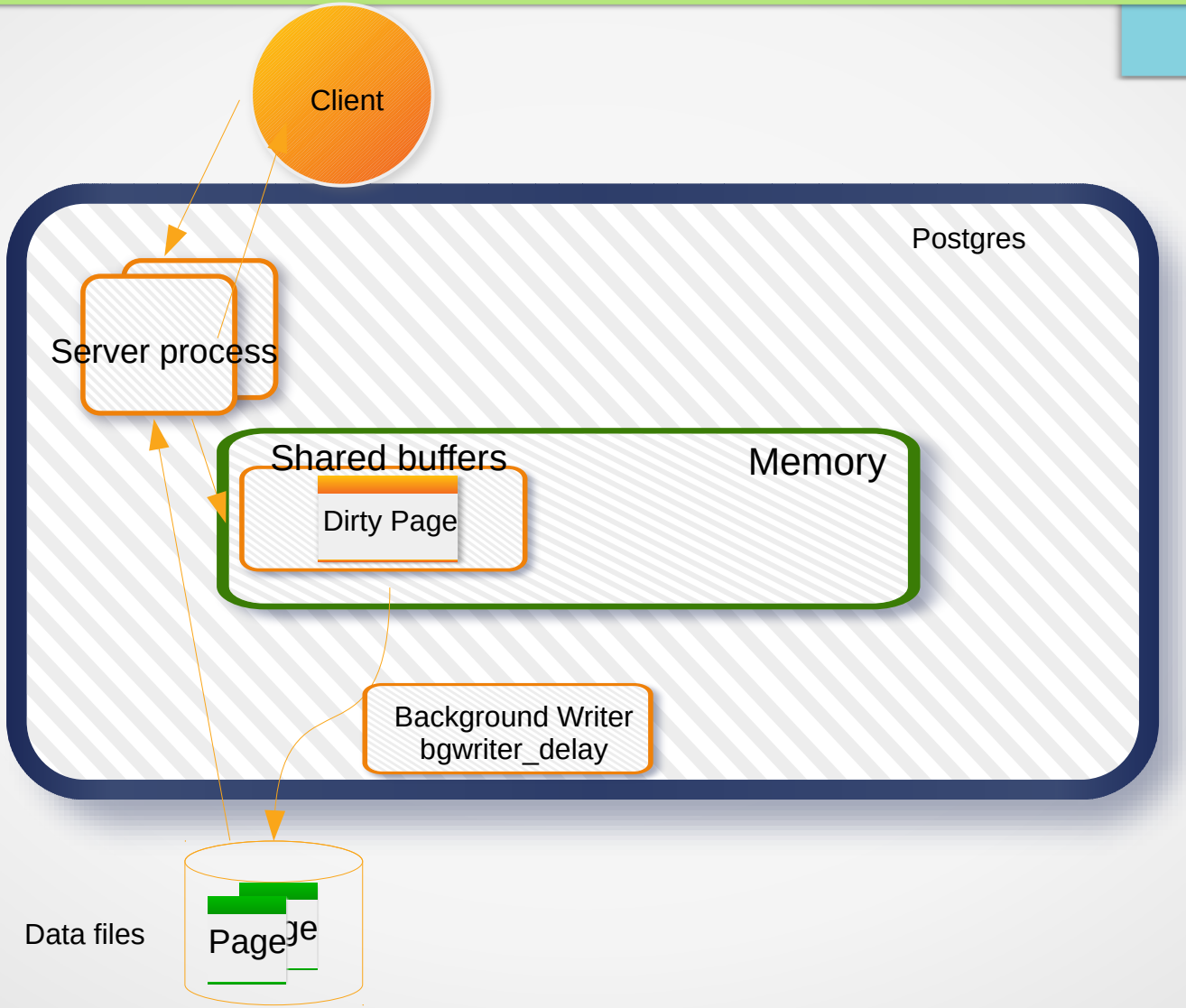
Scaling writes: Delayed write amplification



Scaling writes: Delayed write amplification



Scaling writes: Delayed write amplification



Scaling Writes: Traffic shaping

Scaling Writes: Traffic shaping

- Make use of the delayed flush
  - Group writes on application side
  - Bulk writes to the same table / partition

Scaling Writes: Traffic shaping

- Make use of the delayed flush
  - Group writes on application side
  - Bulk writes to the same table / partition
  - BG writer writes less dirty pages less often
  - More than 400% gain in write throughput

Scaling writes: Traffic seggregation

Scaling writes: Traffic seggregation

- Optimizing for reducing random iops (reducing iowaits)
- IO schedulers for block devices in Linux are intelligent
  - IO scheduler per block device
  - Merge IO access to similar locations on disk

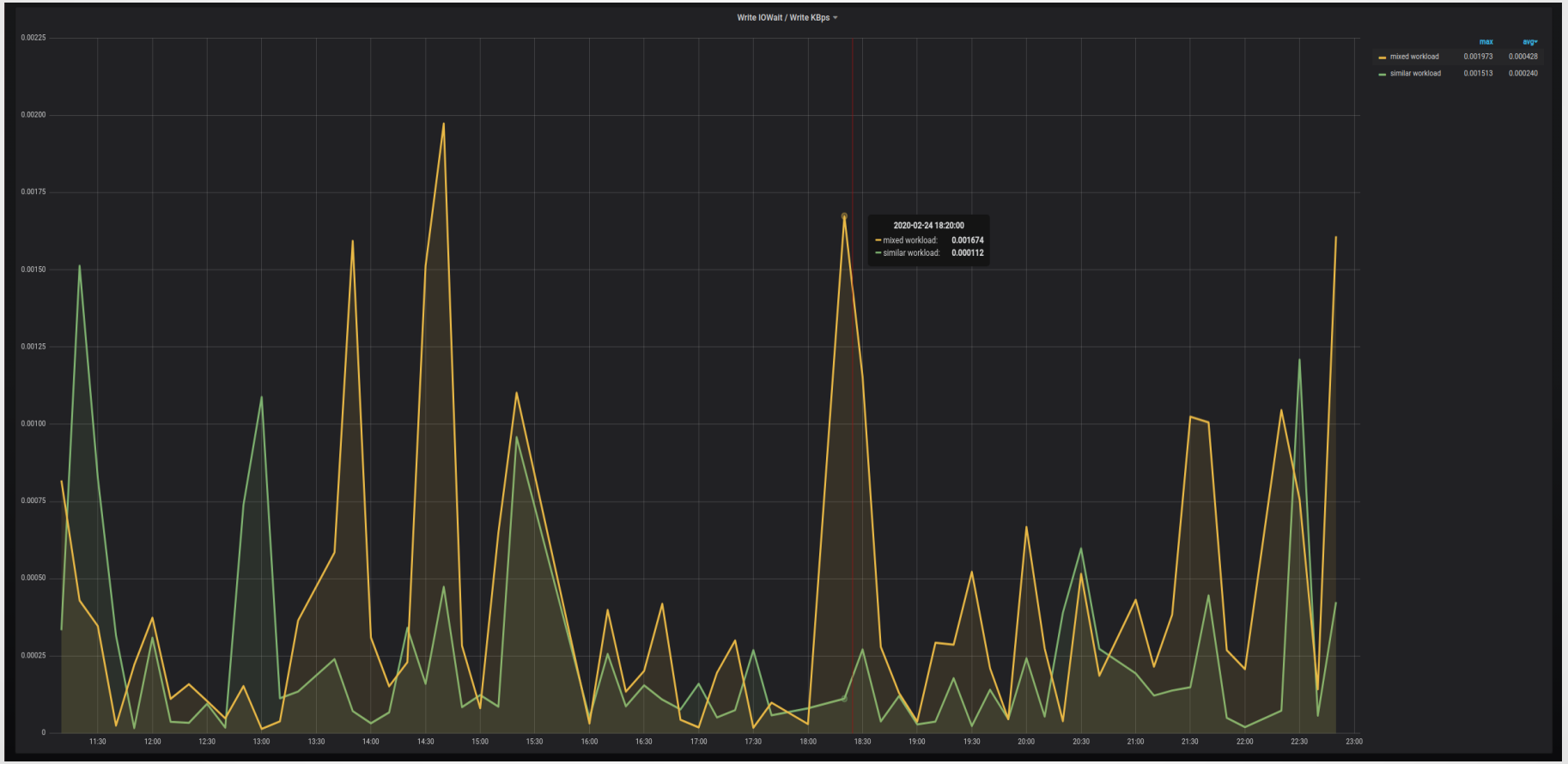
Scaling writes: Traffic seggregation

- Optimizing for reducing random iops (reducing iowaits)
- IO schedulers for block devices in Linux are intelligent
  - IO scheduler per block device
  - Merge IO access to similar locations on disk

  Approach: Segregate access patterns on disks
- Put data for different types of data sources on different disks
  - Logically different workloads
  - Reducing concurrent writes to different tables on same disk
  - Use tablespaces

Scaling writes: Traffic segregation

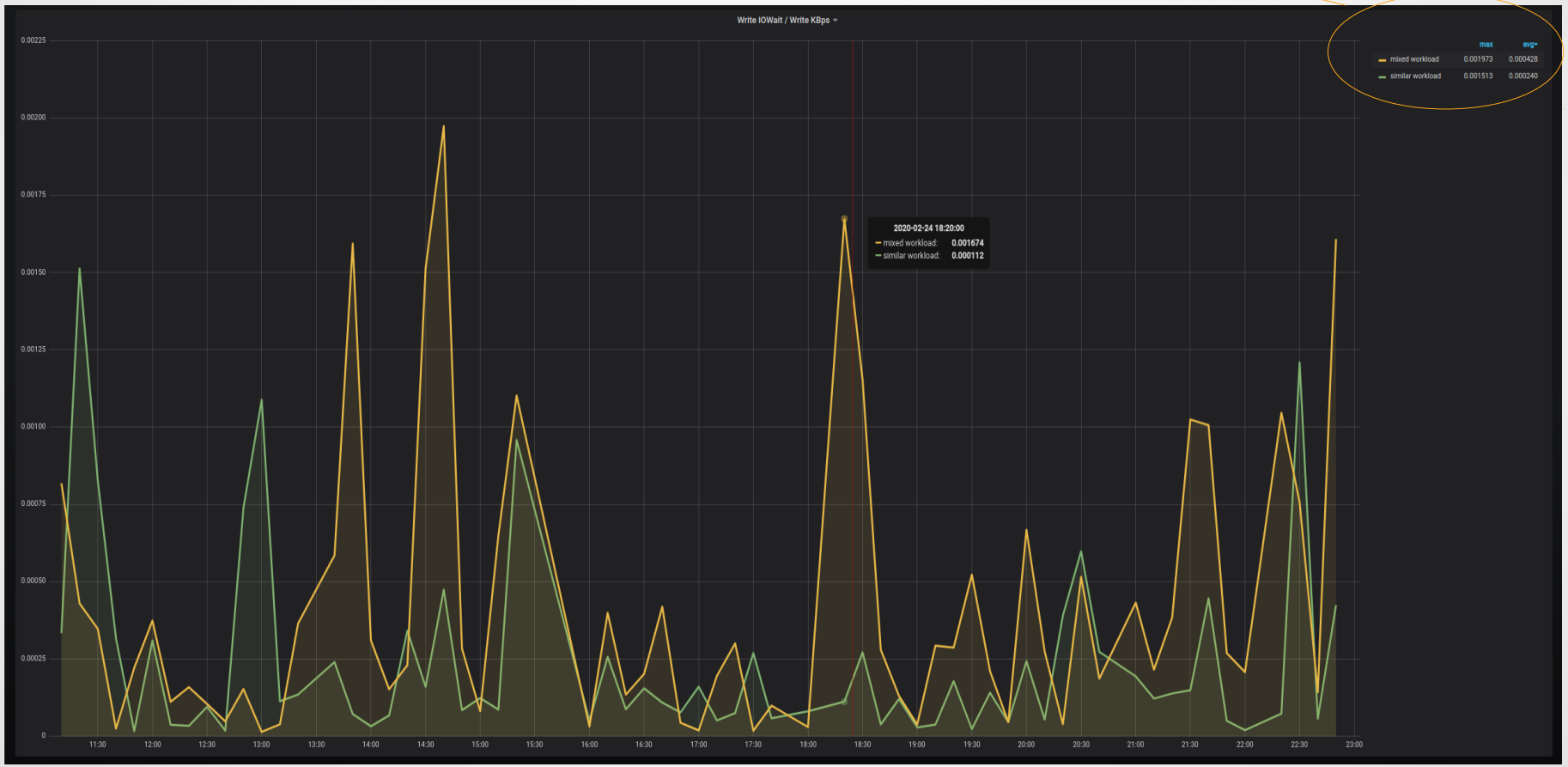
wawait / WkBps



Scaling writes: Traffic segregation

wawait / WkBps

Mixed workload: 0.000428
Similar workload: 0.000240



clarisights

Scaling backups

Approach 1

- pg_dump
  - Custom scripts and external parallel tools
  - Too slow, > 1 hour per TB

Scaling backups

Approach 2

- Differential disk snapshots on slave
  - Ability to take consistent snapshot by stopping process
  - Zero impact on master
  - Fast backup and restore < 5 mins per TB
  - Combined with WAL archiving on master for Point in Time Recovery (PITR)

Scaling ops: Autovacuum

Scaling ops: Autovacuum

- Billions of updates lead to billions of stale versions of data
- Increase load on autovacuum
- 1 GB memory limit causes multiple full index scans

Scaling ops: Autovacuum

- Billions of updates lead to billions of stale versions of data
- Increase load on autovacuum
- 1 GB memory limit causes multiple full index scans

Approach

- More aggressive autovacuum
  - Lower thresholds
  - Increased number of autovacuum workers
  - *1 GB mem limit is still there*
  - *Affects production workload*

Scaling ops: Indexing

- Larger tables mean
  - Longer time for index creation
  - Higher lock contention while indexing thus impacting workload
  - Starvation from long running transactions

Scaling ops: Indexing

- Larger tables mean
  - Longer time for index creation
  - Higher lock contention while indexing thus impacting workload
  - Starvation from long running transactions

  Approach
- Concurrent indexing
- Getting rid of long running transactions

Scaling ops: Indexing

- Larger tables mean
  - Longer time for index creation
  - Higher lock contention while indexing thus impacting workload
  - Starvation from long running transactions

  Approach
- Concurrent indexing
- Getting rid of long running transactions
- *Still affects production workload*

Scaling ops: Partitioning

- Helps across the board

Scaling ops: Partitioning

- Helps across the board
- Autovacuum
  - Smaller tables mean smaller working set
  - 1 GB limit is mitigated

Scaling ops: Partitioning

- Helps across the board
- Autovacuum
  - Smaller tables mean smaller working set
  - 1 GB limit is mitigated
- Indexing
  - Partial indexes for the save
  - Easily drop data / indexes on older partitions

Other knobs to turn

Other knobs to turn

- Parallel scans
  - PG 9.6+
  - Increased scan throughput
  - *max_parallel_workers, max_parallel_workers_per_gather*

Other knobs to turn

- Parallel scans
  - PG 9.6+
  - Increased scan throughput
  - *max_parallel_workers, max_parallel_workers_per_gather*
- Understand your block device
  - *seq_page_cost* vs *random_page_cost*
  - Equal for SSDs, vastly different for spinning disks

Other knobs to turn

Other knobs to turn

- Understand your access pattern
  - Separate tables with different patterns on diff disks
  - Set block device *readahead* for sequential vs random access patterns

Other knobs to turn

- Understand your access pattern
  - Separate tables with different patterns on diff disks
  - Set block device *readahead* for sequential vs random access patterns
- Understand your filesystem
  - Use XFS
  - Configure *noatime*

Other knobs to turn

- Understand your database
  - PostgreSQL is row oriented
  - Huge read amplification with small number of tables

Questions?

We are hiring

clarisights