PostgreSQL @ TOMTOM – lessons learned
About us

We are from Łódź, Poland!

Michał Gutkowski
michal.gutkowski@tomtom.com
Software engineer
solving problems with Java, SQL, Python, Bash

Rafał Hawrylak
rafal.hawrylak@tomtom.com
Software hacker and database expert
TomTom – What do we do?

The New TomTom Go

Motorcycle
Camper & Caravan

The New TomTom VIO

Mobile
Business to Business

GET FIT
TOUCH
FITNESS TRACKERS
SPARK 3
FITNESS WATCHES
GOLFER 2
GOLF WATCHES
RUNNER 3
RUNNING WATCHES
ADVENTURER
OUTDOOR WATCHES
BANDIT
ACTION CAMS

GET BETTER
GET OUT THERE

3
TomTom - What do we do?

- Database with spatial features
- Massive automated tools
- 2000+ of manual editors
- Billions of map objects
Map Making Platform

- Database Machines – 200+ machines (40 cores, 256GB RAM, RAID 10 ssd drives)
- Queries count – over 600k per second
- Inserted rows count 15k per second
- Storage – 30TB
- Daily db size increase – 200GB
- Reads : Writes ~ 100 : 1
PostgreSQL + PostGIS

- Out-of-the-box extension for geometry types
- Processing and analytic functions
- Spatial predicates: intersects, covers, covered by, inside
- Spatial GIST index on geometry (based on bounding boxes)
- Hint: spatial queries faster on simplified geometry
Query optimization: it is all about indexes

- Run analyze to update pg_statistics
  
  “ALTER TABLE SET STATISTICS to 1000” for large tables

- Benchmark queries on production data using “explain analyze”

- Changing parameters may completely change query plan

- Indexes are not for free - increases disk size and row insert time

- Multicolumn Indexes - order of columns in B-tree index definition does matter for query performance

check out:
https://explain.depesz.com
http://use-the-index-luke.com/

<table>
<thead>
<tr>
<th>name</th>
<th>order_date</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>2016-09-01</td>
<td>Paid</td>
</tr>
<tr>
<td>Alan</td>
<td>2016-09-02</td>
<td>Unpaid</td>
</tr>
</tbody>
</table>

idx_customer (name, order_date)

idx_type (type, order_date)
Query optimization: index bloat

- Check bloat on indexes: updates, inserts, deletes causes increased latency of query execution time and increases size of indexes
- Remove not used indexes (pg_stat_all_indexes view shows usage)
- Requires “reindex” (locks entire table for writes) or re-creating an index
Replication: streaming replication

- Streaming vs logical
- Replication tree – failover
- Application is prechecking if data is available on standby
- Standby in physical replication is read-only
- Initial copy: rsync vs pg_basebackup.

Check out: https://wiki.postgresql.org/wiki/Streaming_Replication
Replication: replication lag

- High CPU/IO load
- Check with query: "SELECT now() - pg_last_xact_replay_timestamp()" or in pg_stat_replication view
- Graphite feed from nagios plugin
- Separate volumes for data, xlog, logs, pg_stats
- Use WALs compression for slow network – since 9.5 (increased CPU usage both on master and standby)
- Configuration tuning:
  
  ```
  wal_keep_segments = 330000 # 5TB of WALs

  max_wal_senders = 10
  ```
Replication: replication vs checkpoints

- Checkpoint configuration tuning:
  - checkpoint_completion_target = 0.9
  - checkpoint_timeout = 1h
  - checkpoint_warning = 30s
  - max_wal_size = 100GB
  - min_wal_size = 1GB
  - bgwriter_delay = 50ms
  - bgwriter_lru_maxpages = 2000
Database High Availability

What we learned you cannot do in live production system:

✓ Vacuum full -> well tuned autovacuum instead (more WALs)
✓ Create index -> Create index concurrently (takes longer)
✓ Reindex -> Create copy of index and switch with original (needs extra disk space)
DDL under control

- Versioning and tracking database layout with Liquibase
- Changes tested on pre-production environments – it lowers risk of human error and inconsistencies
- Rollback logic

```xml
<changeSet id="journaltransactions_index" author="TomTom">
  <createIndex tableName="journaltransactions"
    indexName="journal_version_index" schemaName="${schemaName}"
  >
    <column name="txn_version" />
  </createIndex>

  <sql>
    CREATE INDEX CONCURRENTLY journal_geometry_extremes_index
    ON
    ${schemaName}.journaltransactions
    USING gist(txn_geometry_extremes)
  </sql>

  <rollback>
    <sql>
      DROP INDEX CONCURRENTLY ${schemaName}.journal_geometry_extremes_index;
    </sql>
  </rollback>
</changeSet>
```
Query optimization - monitoring

- Use statistics collector
  - `pg_stat_activity, pg_stats_statements`
- Configuration in `postgresql.conf`
  - `log_min_duration_statement = 20000`
  - `shared_preload_libraries = 'pg_stat_statements,auto_explain'`
  - `auto_explain.log_min_duration = 20000`
  - `auto_explain.log_analyze = true;` --- can be expensive to run query twice
Monitoring: pg_stat_statements

- gathers a bunch of useful statistics of query execution
- the best way to track lots of short queries
- one cumulative sack
- not usable if you need track query behavior changes

ElastAlert [https://github.com/Yelp/elastalert](https://github.com/Yelp/elastalert)
In Kibana, we can easily observe for each particular statement on each and every machine separately (if we want to):

- total execution time
- cpu execution time
- io execution time
- number of calls
- number of rows returned / affected
- average execution time
- average cpu execution time
- average io execution time
- average number of calls
- average number of rows returned / affected

In terms of:
- historical data
- trends
- behaviour changes
- the heaviest query
- distribution
Monitoring: stat_statements in Kibana
Monitoring: other tools we use

- Munin
- System/postgresql statistics
- AWS CloudWatch
- AppDynamics for performance
- Unix tools: htop / iotop
- pg_view
Questions?

We are hiring!