Zürcher Kantonalbank

# Tabellen vs Objekte: warum nicht einfach beides (in Postgres)?

Dr. Andreas Geppert
Zürcher Kantonalbank

Swiss PGDay 2025

# About Me

- DevOps Solution Engineer at ZKB (since July 2022)
- Long-term Postgres User (e.g. in teaching)
- Vice president of the Swiss Postgres User Group
- https://www.linkedin.com/in/andreas-geppert-6964a0178/

# Introduction

- The criticism against relational DBMS is almost as old as the ttechnology itself
- Impendance mismatch
  - Object-oriented DBMS
- Rigidness of the relational model
  - NoSQL
  - Key/value pairs, documents, big table
- Scalability
  - Big Data

# Then, why are RDBMS still here?

- SQL & Query Processing
- SQL as “inter-galactic dataspeak”
- Query optimization
- Parallelization
- Transaction management
- Extensible and object-relational DBMS
- Multi-model DBMS

# Content

- Intro
- JSON
- Use Cases and Examples
- Working with Documents in Postgres

# JSON & Postgres

- JSON: JavaScript Object Notation
  - Objects
    - Key/value pairs
    - Value must be JSON values again
  - Arrays
    - Elements must be JSON values again
  - Primitive types
- Postgres supports JSON since many major releases (staring with 9.2)
- Keeps adding new functionality with new releases (as recent as 17)

# JSON & JSONB

- Postgres supports JSON in two variants
  - JSON and JSONB
  - both are represented through their respective data types
  - JSON and JSONB thus can be used as column data types
  - JSON respects the original structure (including white space, field order)
  - JSONB document storage may change structure (re-order fields, remove white space, no duplicate field names)
- JSONB takes longer to ingest, but is faster and more efficient to process
- Examples

```
create table trains (
  id      int,
  day     date,
  train   json
);
```

```
create table trains (
  id      int,
  day     date,
  trains  jsonb
);
```

# Content

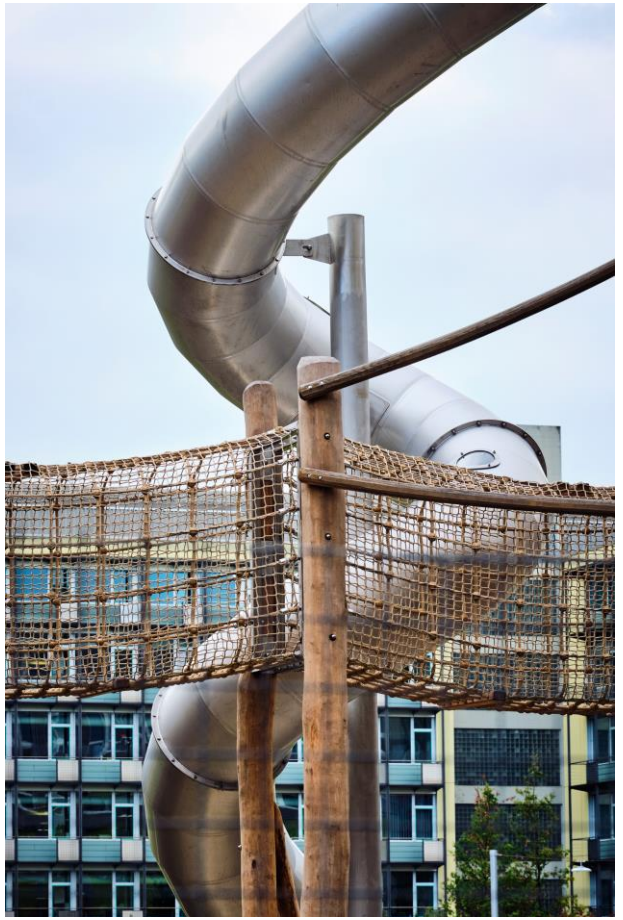
Intro

JSON

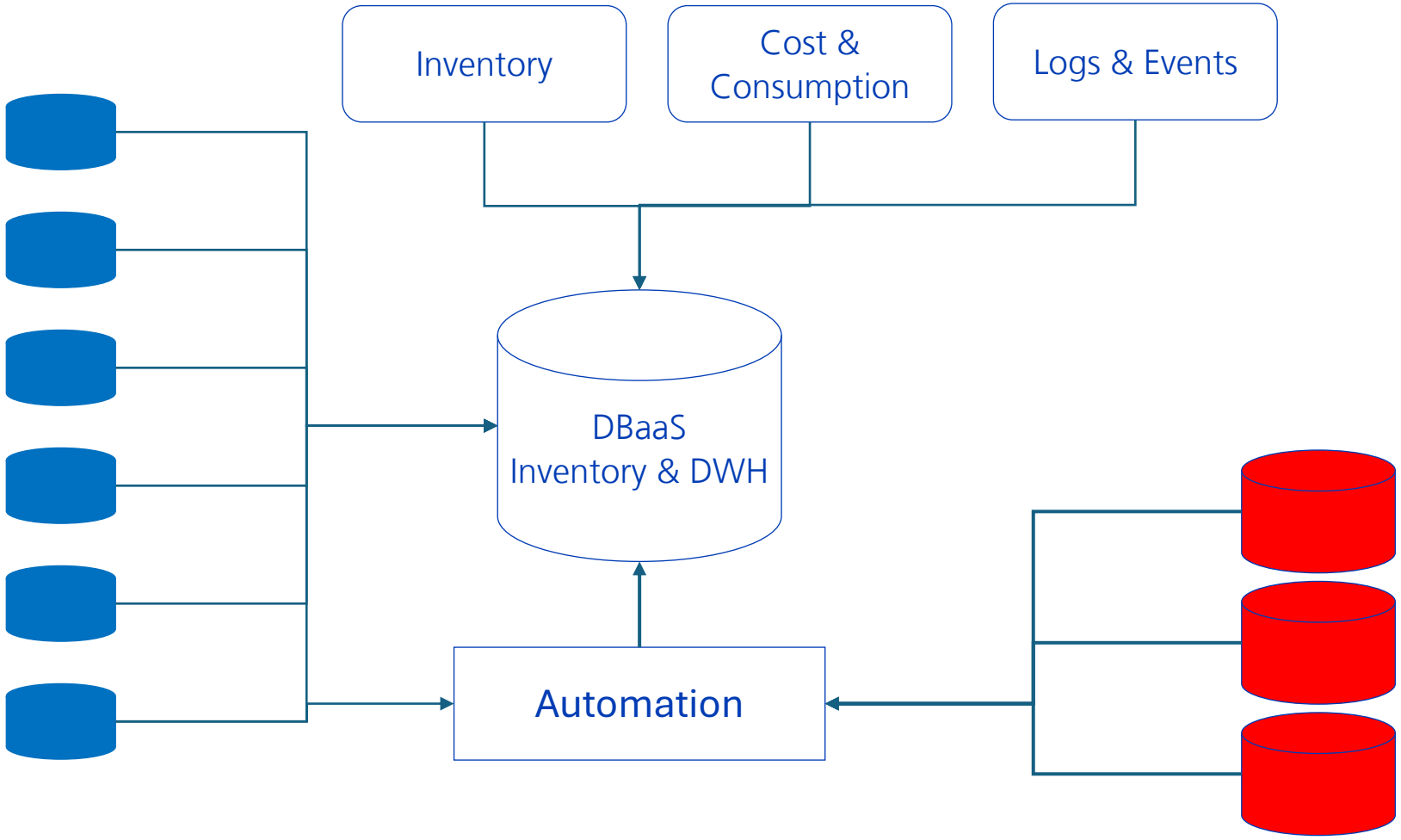
Use Cases and Examples

- Data Exchange
- Semi-structured data
- Data with flexible, varying structure
- Hierarchical, structured data

Working with Documents in Postgres

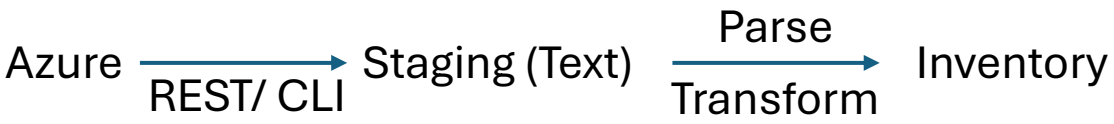


# JSON in a Large DBaaS Environment



# Data Exchange: Receiving Data

- Data provider supports sending of data in JSON format
- Examples:
  - Cost and consumption data
  - Reconciliation between inventories




```
select jsonb_pretty(payload::jsonb)
from staging.containerjson ;
 [
     {
         "id": "/subscriptions/...",
         "sku": {
             "name": "Standard_D2ds_v5",
             "tier": "GeneralPurpose"
         },
         "name": "pg-dbaas-eng-1-khjmhnum",
         "type":
"Microsoft.DBforPostgreSQL/flexibleServers",
         "state": "Ready",
         "network": {
             "publicNetworkAccess": "Disabled",
         ...
     }
 ]
```

# Data Exchange: Sending Data

- Send data as JSON object instead of serialized or csv string
- Example: Calculate Server configuration based on profile, #cpus and work_mem

```
select getcontainerconfiguration
       ('GP_Standard_D', 4, 16);


-- Return function result in json:
select row_to_json(t) from tmp t;

-- Possible result:
{
     "work_mem": 16384,
     "shared_buffers": 524288,
     "max_connections": 256,
     …
     "max_connections_alert": 243
}
```

# Data with flexible, varying structure

- Differently structured data can be homogenized
- Varieties are hidden within JSON documents
- Example: pgwatch2_metrics tables
- Monitoring tables have the same top-level structure
- Very simple data exchange

```
\dt
log_replication_standby,
replication_slots,
log_replication_stream
```

```
  Column   |           Type
-----------+--------------------------+
 time      | timestamp with time zone
 dbname    | text
 data      | jsonb
 tag_data | jsonb
```

# Storing Semi-structured Data

- Storing documents consisting of (fully) structured attributes and text
- Present attributes can vary between documents
- Example: Ora2Pg assessment reports

```
select jsonb_pretty(report) from assessment;


{ "Size": "7998177.50 MB",
  "Schema": "…",
  "Version": "Oracle Database 19c …",  …
  {"object":"TABLE", "number":71, "invalid":
0, "cost value":7.10, "details":"Total
number of rows: 427226311. Top 10 of tables
sorted by number of rows:
   …}
  "human days cost": "7 person-day(s)",
  "migration level": "C-5"
}
```

# Hierarchical Structured Data

- Maintaining the structure of complex objects/documents
- Instead of normalizing them
- Complex structure can be visualized without re-building it
- Example: Zugmonitor

```
{
    "13229": {
        "stations": [ {
                "delay": 5,
                "arrival": 1987,
                "station_id": 394,
                "delay_cause": "Verzögerung
beim Ein-/Ausstieg" }
     …  ],
        "train_nr": "CNL 1246"
    },
    "13230": {
        "stations": [ {
                "arrival": 1887,
                "station_id": 63 }
        ],
        "train_nr": "CNL 1247"
. . . }
```

# Content

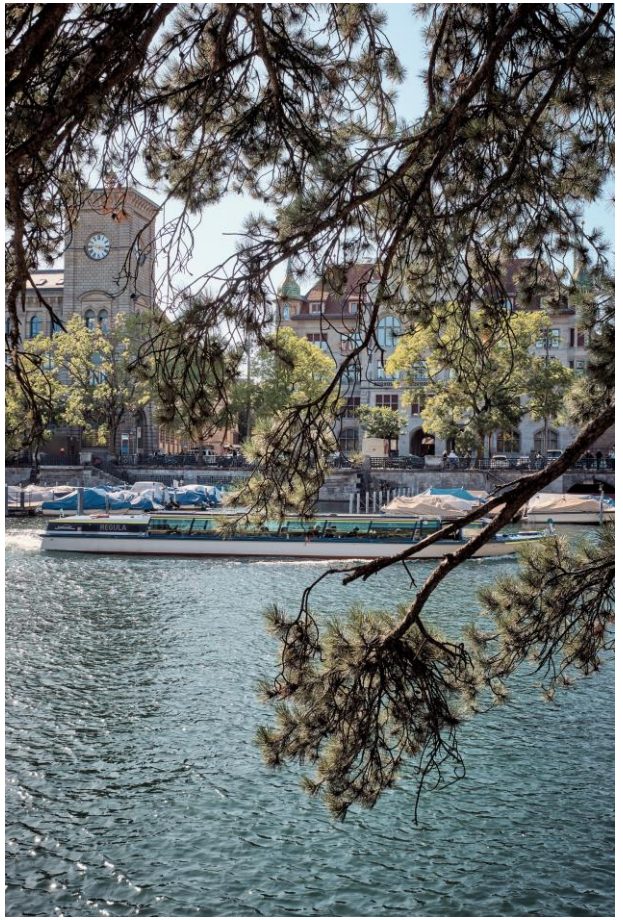
Intro

JSON

Use Cases and Examples

Working with Documents in Postgres

- Exploration / Validation
- Processing & querying
  - Processing hierarchies
- Indexing
- Modification
- Display



# JSON(B) in Postgres' SQL

- JSON functions and operators
  - Starting with Postgres 9.3
- SQL/JSON Path
  - Since Postgres 12
- JSONB-Subscripting
  - Since Postgres 14
- json_table
  - Postgres 17

- Integrated into SQL
- Process JSON and include into processing, analytics, reporting, etc

# Exploration & Validation

- Which types do JSON objects have?
- What are the keys of documents?
- What's the length of arrays?
- Is a certain path valid?

```
select json_typeof(data),
       json_typeof(data->0),
       json_array_length(data)
from staging.servers ;

select json_object_keys(data->0)
from staging.servers;

select jsonb_path_exists(datab,
'$[*].name')
from staging.servers;
```

# Processing (Structured) JSON

- How do we process and query JSON?
- Flattening (Arrays)
- Removing Nesting (Objects)
- Turning JSON collections into tables
- Filtering JSON

# Processing: Flattening

- JSON arrays can be converted into sets of JSON values, which then can be processed further like tables
- Multiple levels of nesting can be processed
- `json_array_elements` typically used in the from-clause of queries

```
select time, a->>'name'
from staging.servers,
  jsonb_array_elements(datab) a;
```

# Processing: Un-Nesting

- Document collections can be transformed into sets of key/value pairs
- JSON(B)_each typically occurs in the from-clause
- Helpful (e.g.) if keys are random or unique

```
{
    "13229": {
        "stations": [ {
           "delay": 5,
           "arrival": 1987,
           "station_id": 394,
           "delay_cause": "Verzögerung beim Ein-/Ausstieg"
            } …  ],
        "train_nr": "CNL 1246"
    },
    "13230": {
. . . } }
```

```
select v.value->>'train_nr'
  from trains t,
    jsonb_each(t.trainsb) v;
```

# Processing: Turning JSON into Tables

- Relational views can be constructed (in particular) with the two aforementioned functions (and others)
- However, a more declarative approach would be appreciated …
- Enter json_table in Postgres 17
- A function to define relational views over JSON collections
- Specify columns using JSON Path expressions

# json_table: Example

- Views showing station id, delay and cause

```
select t.day,
       train->>'train_nr' as zugnummer,
       station_id, departure, arrival, delay, cause
from trainsb t,
     json_table(train, '$.stations[*]' columns (id for ordinality,
                                                departure text path '$.departure',
                                                arrival text path '$.arrival',
                                                delay int path '$.delay',
                                                station_id int path '$.station_id',
                                                cause text path '$.delay_cause')) as jt
  where …;
```

# json_table : Examples

- But we would also like to have the train_nr!

```
select t.day,
       zugnummer, station_id, departure, arrival, delay, cause
from trainsb t,
     json_table(train, '$' COLUMNS (id for ordinality,
                                    zugnummer text path '$.train_nr',
                                    nested path '$.stations[*]' columns (
                                                departure text path '$.departure',
                                                arrival text path '$.arrival',
                                                delay int path '$.delay',
                                                station_id int path '$.station_id',
                                                cause text path '$.delay_cause')))
as jt
  where delay is not null and train->>'train_nr' like 'ICE%';
```

# Processing: Filtering

- Many possibilities for filtering data
- Containment (@>)

- Example: all train runs of ICE 5 that did not make it to Zurich

```
select day,
jsonb_path_query(train, '$.stations[last-1]')
 from trainsb t
where train->>'train_nr' = 'ICE 75’
      and
      train->'stations' @>
             '[{"station_id":252,
             "delay_cause":"Halt entfällt"}]'
```

# Processing: Filtering

- Many possibilities for filtering data

- Example: all train runs that had a delay of more than 5 minutes at any station

```
select day,
       train->>'train_nr'
  from trainsb
 where train->'stations'
          @? '$.delay ? (@ > 5)' ;
```

# Indexing

- Sample query (without index)

```
explain  select day, jsonb_array_length(train->'stations')
           from trainsb
          where train @> '{"train_nr": "ICE 75"}';
                                QUERY PLAN
-----------------------------------------------------------------------
 Gather  (cost=1000.00..67608.44 rows=43 width=8)
   Workers Planned: 2
   ->  Parallel Seq Scan on trainsb  (cost=0.00..66604.14 rows=18 width=8)
         Filter: (train @> '{"train_nr": "ICE 75"}'::jsonb)
```

# Indexing

- GIN indexes can be defined over JSONB columns
- Some operators (such as containment @> above) can make use of such indexes

- Example

```
create index idxgin
    on trainsb
 using gin (train);
```

# Indexing: Example

- Number of stations for ICE 75

```
explain  select day, jsonb_array_length(train->'stations')
           from trainsb
          where train @> '{"train_nr": "ICE 75"}';

                                  QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on trainsb  (cost=38.82..208.23 rows=43 width=8)
   Recheck Cond: (train @> '{"train_nr": "ICE 75"}'::jsonb)
   ->  Bitmap Index Scan on idxgin  (cost=0.00..38.81 rows=43 width=0)
         Index Cond: (train @> '{"train_nr": "ICE 75"}'::jsonb)
```

# Display

- jsonb_pretty: “pretty-print” JSONB-values

```
select jsonb_pretty(train)
from trainsb;
```

```
{
    "stations": [
        {
            "arrival": 2047,
            "scraped": 1,
            "station_id": 490,
            "delay_cause": "Halt entfällt"
        } ...
    ],
    "train_nr": "CNL 40456"
}
```

# Modification

- JSON modifications are probably necessary rather rarely
- But are possible, e.g. removal of keys or array elements

- Example

```
select jsonb_pretty( train -
        '{status,nextrun,started,finished}'::text[])
from trainsb limit 1;
```

# Conclusion

- Powerful support for JSON in Postgres
- Improvements keep being made with new major releases
- No need for separate, parallel DBMS infrastructure

?!?