YugabyteDB Intro

Štěpán Kroupa

MFF UK

4. května 2022

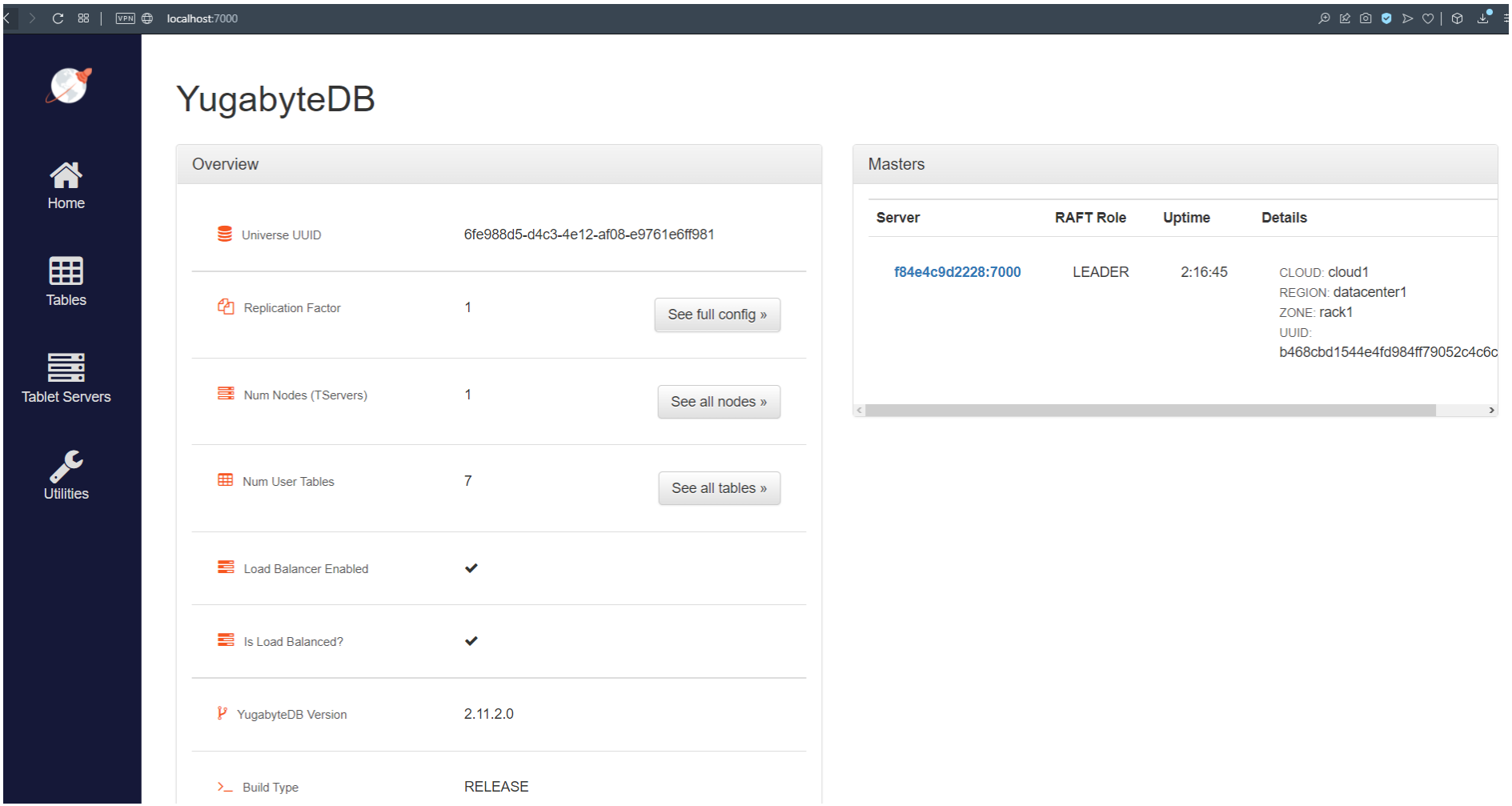
Basic description

- Distributed SQL multi-model database
- Motivation ?
  - Relational DBMS but cloud-native, scalable, resilient to failure
  - Designed for OLTP applications
  - Data geo-distribution - latency & local regulations
- Open-source
- 2016 Release
- PSQL, CQL compatible
  - "Reuses PostgreSQL's query layer" supposedly

Features

- Distributed ACID transactions, but also high availability
  - Sharding, replication
- Supports json, doesn't support XML
- SQL proccessing layer of the database (YSQL) directly uses PostGreSQL code
  - Doesn't implement everything from PSQL
  - Supports JSON but not XML
  - Generally doesn't support features hard to implement in a distributed setting
- Keyspaces - Internally each row is a document with a key
- Support for Cassandra API
- YCQL doesn't support triggers, procedures
  - Intended more for low latency, large scale
- Doesn't support MapReduce
- Decent documentation

Overview Client



- Runs on Linux, MacOS, Kubernetes & Docker

YSQL DDL

- Supports referential integrity, constraints
- Supports all basic types (varchar, integer etc.)
- A few differences from PSQL
- Supports JSON but not XML
- Documentation lists DEFAULT constraint as unsupported
  - Not true ?

```
yb_test=# ALTER TABLE test_table ADD COLUMN Pocet_Veci INT DEFAULT 1000 CHECK (Pocet_Veci < 500);
ALTER TABLE test_table ADD COLUMN Pocet_Veci INT DEFAULT 1000 CHECK (Pocet_Veci < 500);
ALTER TABLE
yb_test=# INSERT INTO test_table (json_test) VALUES ('{}');
INSERT INTO test_table (json_test) VALUES ('{}');
ERROR:  new row for relation "test_table" violates check constraint "test_table_pocet_veci_check"
DETAIL:  Failing row contains (2022-05-04 09:06:12.762404, {}, null, 1000).
yb_test=# INSERT INTO test_table (json_test, Pocet_Veci) VALUES ('{}', 10);
INSERT INTO test_table (json_test, Pocet_Veci) VALUES ('{}', 10);
INSERT 0 1
yb_test=#
```

YSQL DDL

- Doesn't support deferrable unique/primary key constraints
  - Handling of constraints after commit
  - Only checks constraints immidiately
- Doesn't support DROP CONSTRAINT primary_key
  - Tables are sharded into smaller tablets according to PK
- Doesn't support GiST indexes
- Supports indexes on columns, JSON attributes
  - Unique indexes, partial indexes
- "EXCLUDE" not supported
  - Worst case - requires checking all existing rows in a table

YSQL DML

- Promises nearly identical JSON support to PostgreSQL
- Supports functions for json & jsonb types
  - e.g. to_jsonb(), to_json(), jsonb_each()
- Also supports JSON operators
  - e.g. ->, -», ?>, @>

YSQL DML examples

- jsonb_each()

```
yb_test=# INSERT INTO test_table (jsonb_test, pocet_veci) VALUES ('{"count": 3, "kitchen": { "size": 200, "items": ["knife", "pan", "chai
r"]}, "garage": { "size": 500, "items": ["Auto1", "lawnmower", "bike"]}, "living_room": { "size": 500, "items": ["chair", "television", "
couch", "table"]}}', 314);
INSERT INTO test_table (jsonb_test, pocet_veci) VALUES ('{"count": 3, "kitchen": { "size": 200, "items": ["knife", "pan", "chair"]}, "gar
age": { "size": 500, "items": ["Auto1", "lawnmower", "bike"]}, "living_room": { "size": 500, "items": ["chair", "television", "couch", "t
able"]}}', 314);
INSERT 0 1
yb_test=# SELECT jsonb_each(jsonb_test) FROM test_table WHERE pocet_veci=314;
SELECT jsonb_each(jsonb_test) FROM test_table WHERE pocet_veci=314;
                                        jsonb_each
-------------------------------------------------------------------------------------------
 (count,3)
 (garage,"{""size"": 500, ""items"": [""Auto1"", ""lawnmower"", ""bike""]}")
 (kitchen,"{""size"": 200, ""items"": [""knife"", ""pan"", ""chair""]}")
 (living_room,"{""size"": 500, ""items"": [""chair"", ""television"", ""couch"", ""table""]}")
(4 rows)

yb_test=#
```

- <@ operator - checks for a field with specified value at top level of a jsonb type

```
yb_test=# SELECT jsonb_test->>'count' AS Pocet_Mistnosti FROM test_table WHERE '{"count": 3}' <@ jsonb_test;
SELECT jsonb_test->>'count' AS Pocet_Mistnosti FROM test_table WHERE '{"count": 3}' <@ jsonb_test;
 pocet_mistnosti
-----------------
 3
(1 row)
```

YSQL DML examples

- Indexes - Standard, partial, unique

```
SELECT jsonb_test FROM test_table;
                                                                                                  jsonb_test
------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------

 {"count": 3, "garage": {"size": 500, "items": ["Auto1", "lawnmower", "bike"]}, "kitchen": {"size": 200, "items": ["knife", "pan", "c
500, "items": ["chair", "television", "couch", "table"]}}
 {}
(3 rows)

yb_test=# CREATE INDEX jsonb_index ON test_table (((jsonb_test->>'count')::int) ASC);
CREATE INDEX jsonb_index ON test_table (((jsonb_test->>'count')::int) ASC);
CREATE INDEX
yb_test=# CREATE INDEX jsonb_index_partial ON test_table (((jsonb_test->>'count')::int) ASC) WHERE jsonb_test->>'count' IS NOT NULL;
CREATE INDEX jsonb_index_partial ON test_table (((jsonb_test->>'count')::int) ASC) WHERE jsonb_test->>'count' IS NOT NULL;
CREATE INDEX
yb_test=# \d test_table
                       Table "public.test_table"
   Column   |            Type             | Collation | Nullable | Default
------------+-----------------------------+-----------+----------+---------
 cas        | timestamp without time zone |           | not null | now()
 json_test  | json                        |           |          |
 jsonb_test | jsonb                       |           |          |
 pocet_veci | integer                     |           |          | 1000
Indexes:
    "test_table_pk" PRIMARY KEY, lsm (cas HASH)
    "jsonb_index" lsm (((jsonb_test ->> 'count'::text)::integer) ASC)
    "jsonb_index_partial" lsm (((jsonb_test ->> 'count'::text)::integer) ASC) WHERE (jsonb_test ->> 'count'::text) IS NOT NULL
    "time_index" lsm (cas HASH)
Check constraints:
    "test_table_pocet_veci_check" CHECK (pocet_veci < 500)
```

YCQL

- More lightweight, no procedures, triggers
- Can specify replication & partition columns

```
ycqlsh> CREATE KEYSPACE test WITH REPLICATION = {'class': 'SimpleStrategy', 'replication_factor': '3'} AND DURABLE_WRITES = true;
ycqlsh> USE test;
ycqlsh:test> CREATE TABLE lide(student_id INT, vek INT, jmeno TEXT, obor TEXT, PRIMARY KEY((student_id, jmeno), obor));
ycqlsh:test> INSERT INTO lide(student_id, vek, jmeno, obor) VALUES (21642, 22, 'Jmeno123', 'Informatika');
ycqlsh:test> INSERT INTO lide(student_id, vek, jmeno, obor) VALUES (21612, 24, 'Jmeno456456', 'Informatika');
ycqlsh:test> INSERT INTO lide(student_id, vek, jmeno, obor) VALUES (21612, 24, 'Jmeno456456', 'Informatika');
ycqlsh:test> SELECT * FROM lide;

 student_id | jmeno       | obor        | vek
------------+-------------+-------------+-----
      21612 | Jmeno456456 | Informatika |  24
      21642 |    Jmeno123 | Informatika |  22

(2 rows)
```

YCQL

- Also handles JSON types

```
ycqlsh> CREATE TABLE IF NOT EXISTS test.domy (id INT PRIMARY KEY, popis jsonb);
ycqlsh> SELECT * FROM test.domy;

 id | popis
----+--------------------------------------------------------------------
  1 | {"garaz":{"plocha":150},"kuchyn":{"plocha":99},"pocet_mistnosti":3}

(1 rows)
ycqlsh> INSERT INTO test.domy (id, popis) VALUES (1, '{"pocet_mistnosti": 2, "kuchyn": {"plocha": 100}, "garaz": {"plocha": 150}}');
ycqlsh> SELECT * FROM test.domy;

 id | popis
----+--------------------------------------------------------------------
  1 | {"garaz":{"plocha":150},"kuchyn":{"plocha":100},"pocet_mistnosti":2}

(1 rows)
ycqlsh> INSERT INTO test.domy (id, popis) VALUES (2, '{"pocet_mistnosti": 3, "kuchyn": {"plocha": 99}, "garaz": {"plocha": 150}}');
ycqlsh> INSERT INTO test.domy (id, popis) VALUES (3, '{"pocet_mistnosti": 3, "kuchyn": {"plocha": 1000}, "garaz": {"plocha": 1000}}');
ycqlsh> SELECT * FROM test.domy;

 id | popis
----+--------------------------------------------------------------------
  1 |   {"garaz":{"plocha":150},"kuchyn":{"plocha":100},"pocet_mistnosti":2}
  2 |    {"garaz":{"plocha":150},"kuchyn":{"plocha":99},"pocet_mistnosti":3}
  3 | {"garaz":{"plocha":1000},"kuchyn":{"plocha":1000},"pocet_mistnosti":3}

(3 rows)
ycqlsh> SELECT * FROM test.domy WHERE id < 3;
SyntaxException: Invalid CQL Statement. Partition column cannot be used in this expression
SELECT * FROM test.domy WHERE id < 3;
                              ^^
 (ql error -12)
ycqlsh> SELECT * FROM test.domy WHERE CAST(popis->>'pocet_mistnosti' AS integer) = 3;

 id | popis
----+--------------------------------------------------------------------
  2 |    {"garaz":{"plocha":150},"kuchyn":{"plocha":99},"pocet_mistnosti":3}
  3 | {"garaz":{"plocha":1000},"kuchyn":{"plocha":1000},"pocet_mistnosti":3}

(2 rows)
```

Distributed transactions

- Two-phase commit
- Changes are stored in "provisional records" until commit.
- Optimistic locking
- No pessimistic locking (unlike PostgresQL), currently in development
- No blocking, lower priority transaction transaction aborted
  - Explicit locking - user can give arbitrarily high priority to transactions

In conclusion

- Easy to transition from PSQL / CQL like DMBS's
- Ideal for processing of large numbers of local transactions
  - both geo-distribution wise and locking wise
- Modern, tries to address problems that consistent R-DBMS's have at larger scales
- Doesn't implement some more advanced features
  - for better availability & performance

End

Thank you for your attention