Prof. Dr. Florian Heinz
florian.heinz@sysv.de

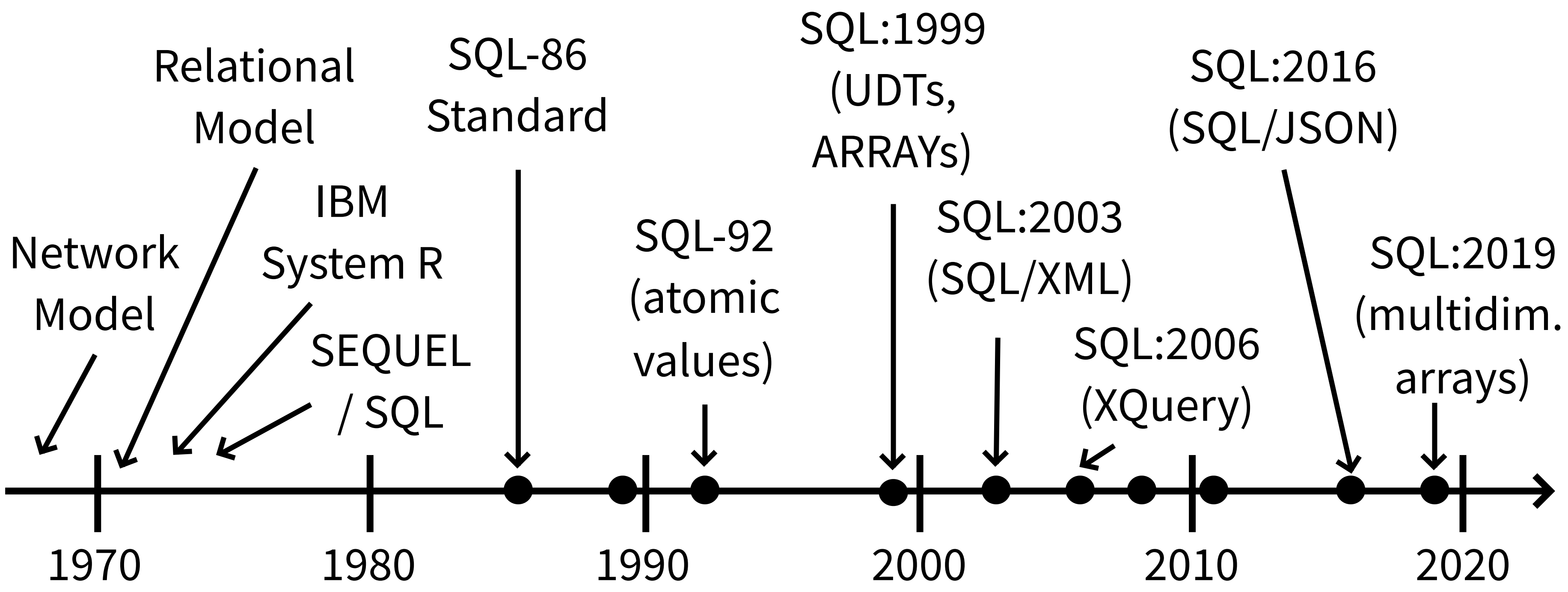
Modern Database Concepts

Chapter 3: SQL Features for Semi-Structured Data

CC BY

OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

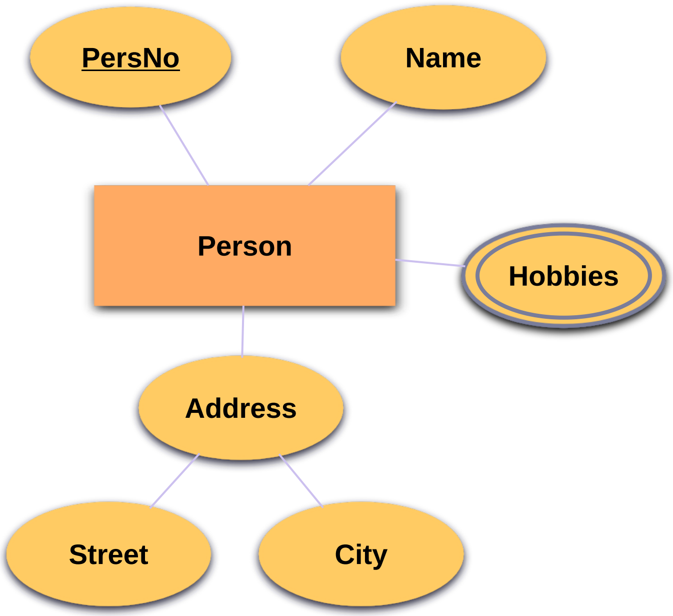
History of SQL



Before 1970, hierarchical databases and network databases were the most prominent ones. They did not have a query language, you need to develop a program to query it. The relational data model supports relational-algebra operators, and later the query language SQL. This makes it possible for non-developers to query the database.

Each dot in this chart is a revision of the SQL Standard. Up to SQL-92, each column value had an atomic type. Nonscalar types were firstly introduces in SQL:1999.

Normalization of Nonscalar Types



(NF)² - Non-first Normal Form

<u>PersNo</u>	**Name**	**Address**		**Hobbies**
		Street	**City**	
5	Peter	Highway 5	Berlin	piano, yoga

1NF - First Normal Form

<u>PersNo</u>	**Name**	**Address_Street**	**Address_City**
5	Peter	Highway 5	Berlin

<u>PersNo</u>	**<u>Hobby</u>**
5	piano
5	yoga

There are multiple normal forms in databases to avoid redundancies and anomalies, improve data integrity, and allows easily querying the data. In 1NF, each column consists of only one atomic value.

UDT - User-Defined Data Types

SQL:1999: `CREATE TYPE ...` (Structured Types, Subtyping, ...)

```
CREATE TYPE ADDRESS AS (street VARCHAR(200), city VARCHAR(80));
CREATE TABLE people (name VARCHAR(200), address ADDRESS);
INSERT INTO people VALUES ('Peter', NEW address('Highway 5','Berlin'));
SELECT address.city FROM people;
```

In PostgreSQL:

```
CREATE TYPE ADDRESS AS (street VARCHAR(200), city VARCHAR(80));
CREATE TABLE people (name VARCHAR(200), address ADDRESS);
INSERT INTO people VALUES ('Peter', ('Highway 5', 'Berlin'));
SELECT (address).city FROM people;
```

User-Defined Data Types can be used as column types for tables and views, for parameters and return types of functions, and more. The PostgreSQL dialect is very close to the SQL Standard.

Composite Types

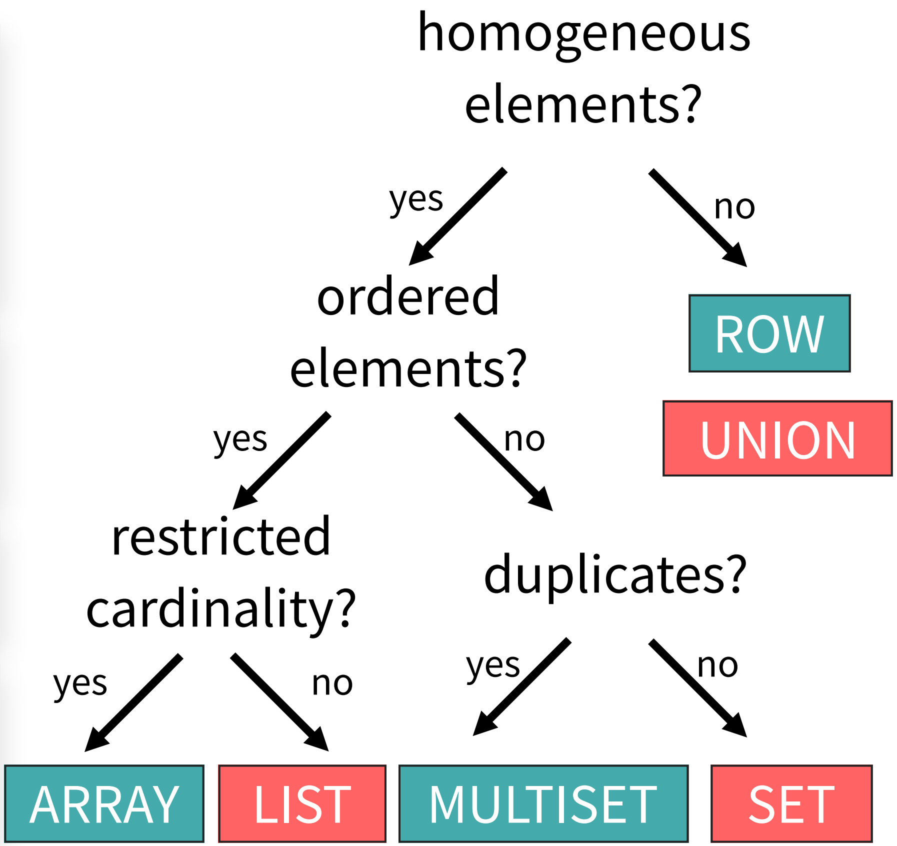
SQL:1999: ARRAY, MULTISET, ROW, … data types

```
CREATE TABLE people (
 name VARCHAR(200),
 address ROW(street VARCHAR(200),
             city VARCHAR(80)),
 hobbies VARCHAR(100) ARRAY[3]);
```

```
INSERT INTO people VALUES ('Peter',
 ROW('Highway 5', 'Berlin'),
 ARRAY['piano', 'yoga']);
```

```
SELECT hobbies[2] FROM people
 WHERE address.city = 'Berlin';
```

```
SELECT p.name, h.hobby
 FROM people p,
 UNNEST(p.hobbies) AS h(hobby);
```



The SQL Standard introduced three composite types for storing and working with collections of data values: ARRAY, MULTISET and ROW. UNION (a unification of two data-type domains), LIST, and SET were not introduced in the standard. Most RBDMSs do not supported these types. Instead, they recommend creating user-defined data types or using XML or JSON.

Composite Types

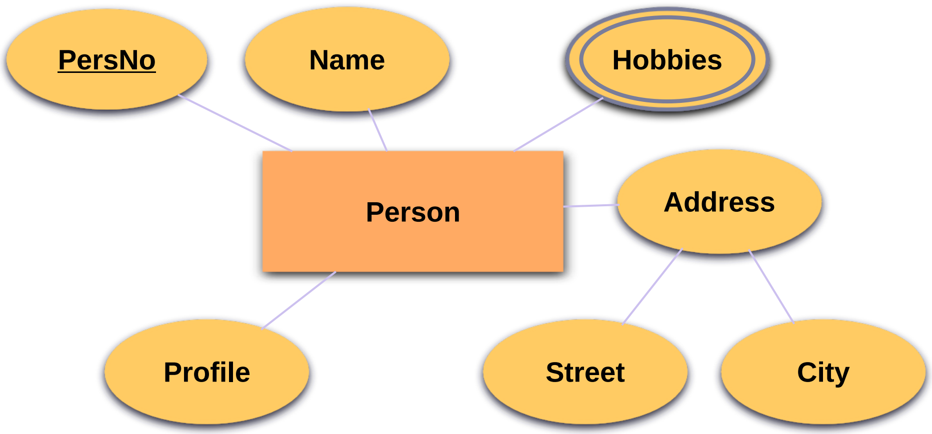
ARRAY type in PostgreSQL:

```
CREATE TABLE people (name VARCHAR(200), hobbies VARCHAR(100) ARRAY);
INSERT INTO people VALUES ('Peter', '{ piano, yoga }');
SELECT hobbies[2] FROM people;
SELECT name FROM people WHERE 'piano' = ANY(hobbies);
SELECT p.name, h.hobby FROM people p, UNNEST(p.hobbies) AS h(hobby);
```

name	hobby
Peter	piano
Peter	yoga

PostgreSQL follows supports SQL Standard `INT ARRAY[3]`, but also `INT[3]`, and `INT[]`. In any case, the number of elements in the array is not limited, so PostgreSQL does not support size restrictions. The syntax for creating and working with array is similar to the SQL Standard.
Other composite types (ROW, MULTISET) are not supported by PostgreSQL.

SQL/XML and SQL/JSON



<u>**PersNo**</u>	**Name**	**Address**	**Hobbies**	**Profile**

5	Peter	`<address>`
		`<street>Highway 5</street>`
		`<city zip="10123">Berlin</city>`
		`</address>`
			`["piano", "yoga"]`	`{ "job": "programmer" }`

The SQL datatypes `XML` and `JSON` can be used to model complex data structure. Furthermore, these types are often used for storing schemaless data. The attribute "profile" is used in this example to store arbitrary further information about a person in a flexible way.

SQL/XML

SQL:2003: XML data type, mappings, predicates, functions

```
-- try out at http://bit.ly/sqlxml
CREATE TABLE people (name VARCHAR(200), address XML);
```

```
INSERT INTO people VALUES ('Peter', '<address>
<street>Highway 5</street><city zip="10123">Berlin</city></address>');
```

```
SELECT XMLQUERY('$addr//city/text()' PASSING BY REF address AS "addr")
FROM people;
```

```
SELECT * FROM people WHERE XMLEXISTS('$addr//city[@zip=10123]'
                           PASSING BY REF address AS "addr");
```

```
SELECT p.name, a.zip, a.city
FROM people p, XMLTABLE('$addr/address'
PASSING BY REF address AS "addr" COLUMNS zip CHAR(5) PATH 'city/@zip'
                                         city VARCHAR(100) PATH 'city') a;
```

SQL/XML

XML type in PostgreSQL:

```
CREATE TABLE people (name VARCHAR(200), address XML);
```

```
INSERT INTO people VALUES ('Peter', '<address>
<street>Highway 5</street><city zip="10123">Berlin</city></address>');
```

name	address
Peter	<address><street>Highway 5</street> <city zip="10123">Berlin</city></address>

```
SELECT xpath('//city/text()', address) AS c FROM people;
```

```
SELECT * FROM people WHERE xpath_exists('//city[@zip=10123]', address);
```

The PostgreSQL syntax for working with XML data is similar to the SQL standard. PostgreSQL only supports XPath, the SQL standard also allows XQuery FLWOR expressions. Mind that the return type of PostgreSQL's `xpath` function is `XML ARRAY`. So, to get the city as a `VARCHAR`, the following expression would be needed: `CAST((xpath('//city/text()', address))[1] AS VARCHAR(100))`

SQL/JSON

SQL:2016: JSON data type, mappings, predicates, functions

```
-- try out at http://bit.ly/sqljson
CREATE TABLE people (name VARCHAR(200), hobbies JSON);
INSERT INTO people VALUES ('Peter', '["piano","yoga"]');
```

name	hobbies
Peter	["piano","yoga"]

```
SELECT JSON_VALUE(hobbies, '$[1]') FROM people;
```

```
SELECT * FROM people WHERE JSON_EXISTS(hobbies, '$?(@=="yoga")');
```

Similar to SQL/XML, the SQL Standard introduced a native JSON data type for storing JSON data. Within the functions, e.g. JSON_VALUE, JSONPath expressions are used (see previous chapter). JSON_VALUE returns a scalar value (here: VARCHAR), another function JSON_QUERY returns a value of type JSON.

SQL/JSON

`JSON` data type

Implementation alternatives:

- native `JSON` type
  - PostgreSQL (`JSONB` type), MySQL
- `JSON` as an alias for a string type
  - PostgreSQL (`JSON` type), MariaDB (alias for `LONGTEXT`), SQLite
- no `JSON` type; instead use `CLOB`, `VARCHAR`, `BLOB`
  - Microsoft SQL Server, Oracle: store strings
  - DB2: BSON storage

SQL/JSON was introduces in the SQL:2016 standard. Different vendors implement their JSON functionality in different ways. Some offer a native JSON datatype, some store the JSON data as text. The SQL/Standard defines multiple functions on the JSON datatype.

`JSON` / `JSONB` types in PostgreSQL

`JSON`

- introduced in PostgreSQL 9.2 (2012)
- JSON data stored as a string (fast storage)
- check for well-formedness
- preserves the original formatting (whitespaces, duplicate keys, key ordering)

`JSONB`

- introduced in PostgreSQL 9.4 (2014)
- "JSON Binary" (or "JSON better" ;-))
- efficient querying
- index support

For most use cases, it is recommended to use the `JSONB` data type. Using this data type, JSON data is internally stored in a binary format which supports much faster queries than querying `JSON` data stored as a string.

Well-formedness

The native JSON datatype only accepts well-formed JSON documents.

```
-- PostgreSQL (also checks well-formedness for JSON data type)
CREATE TABLE people (name VARCHAR(200), hobbies JSONB);
INSERT INTO people VALUES ('Peter', '["piano","yoga"');
```

ERROR: invalid input syntax for type json
Detail: The input string ended unexpectedly.

CHECK constraint for well-formedness:

```
-- MariaDB
CREATE TABLE people (name VARCHAR(200),
                     hobbies JSON CHECK (JSON_VALID(hobbies)));
-- Oracle
CREATE TABLE people (name VARCHAR(200),
                     hobbies CLOB CHECK (hobbies IS JSON));
```

If a text column is used to store JSON data (`VARCHAR`, `CLOB`, in MariaDB also `JSON`, ...), is is possible to store non-well-formed JSON documents. It is recommended to use a `CHECK` constraint to avoid this.

Building JSON

cities

city	population
Regensburg	153094
Berlin	3669491

districts

city	district
Regensburg	Galgenberg
Regensburg	Kumpfmühl
Berlin	Wedding

```
SELECT JSONB_BUILD_OBJECT('city', c.city, 'population', c.population,
                          'districts', JSONB_AGG(d.district))
FROM cities c JOIN districts d ON c.city=d.city
GROUP BY c.city, c.population;
```

{"city": "Berlin", "districts": ["Wedding"], "population": 3669491}
{"city": "Regensburg", "districts": ["Galgenberg", "Kumpfmühl"], "population": 153094}

Relational databases support many functions to build JSON data. PostgreSQL's `JSONB_BUILD_OBJECT` takes a list of field names and their values to create an JSON object. The values can be of any type, also `JSON` or `JSONB`. `JSONB_AGG` is an aggregation function (like `SUM`, ...). It creates a JSON array with all elements of a group of values.

SQL/JSON in PostgreSQL

```
CREATE TABLE people (name VARCHAR(200), profile JSONB);
INSERT INTO people VALUES ('Peter', '{"city":"Berlin",
                                      "hobbies":["piano","yoga"]}');
SELECT profile->'hobbies'->>0 FROM people;
SELECT * FROM people WHERE profile->'hobbies' ? 'yoga';
```

operator	return type	description
`->n`	`JSON`	n-th array element
`->'x'`	`JSON`	value of field x
`->>n`	`text`	n-th array element
`->>'x'`	`text`	value of field x
`? 'x'`	`boolean`	does the array contain the value `'x'`? does the object contain the field x?

PostgreSQL's JSON operators `->>`, `?` and so on are not standard-compliant.

@> Containment Operator

operator	return type	description
`'json' @> 'json'`	`boolean`	Does the left JSON contain the entries of the right JSON?

```
SELECT * FROM people WHERE profile @> '{"hobbies" : ["yoga"]}';
```

SQL/JSON Path Expressions

operator	return type	description
`@? 'path'`	`boolean`	Does the path expression return anything?

```
SELECT profile @? '$.hobbies?(@=="yoga")' FROM people;
```

≙

```
SELECT jsonb_path_exists(profile, '$.hobbies?(@=="yoga")') FROM people;
```

Path Queries

```
SELECT jsonb_path_query(profile, '$.hobbies[0]') FROM people;
```

The query at the bottom of this slide finds the first entry in the hobbies array. The query is equivalent to the first `SELECT` query on the previous slide. The `@?` operator and the functions `jsonb_path_exists` and `jsonb_path_query` only work for `JSONB` values (not `JSON`).

Unnesting

```
SELECT name, hobby
FROM people,
     jsonb_array_elements_text(profile->'hobbies') hobbies(hobby);
```

```
SELECT name, hobby->>0
FROM people,
     jsonb_path_query(profile, '$.hobbies[*]') hobbies(hobby);
```

name	hobby
Peter	piano
Peter	yoga

`jsonb_array_elements` and `jsonb_path_query` return a set of `JSONB` values. When `jsonb_path_query` finds multiple matching items, each of them is returned as a separate row in the query result. In the first query on this slide, `jsonb_array_elements_text` is used, it directly returns a set of text. In the second query, a set of JSONB is returned so that we need to convert each value into text with `->>0`.

Indexes on JSON

In PostgreSQL, **g**eneralized **i**nverted i**n**dexes (GIN) can be created on `JSONB`:

```
CREATE INDEX people_profile_idx ON people USING GIN (profile);
```

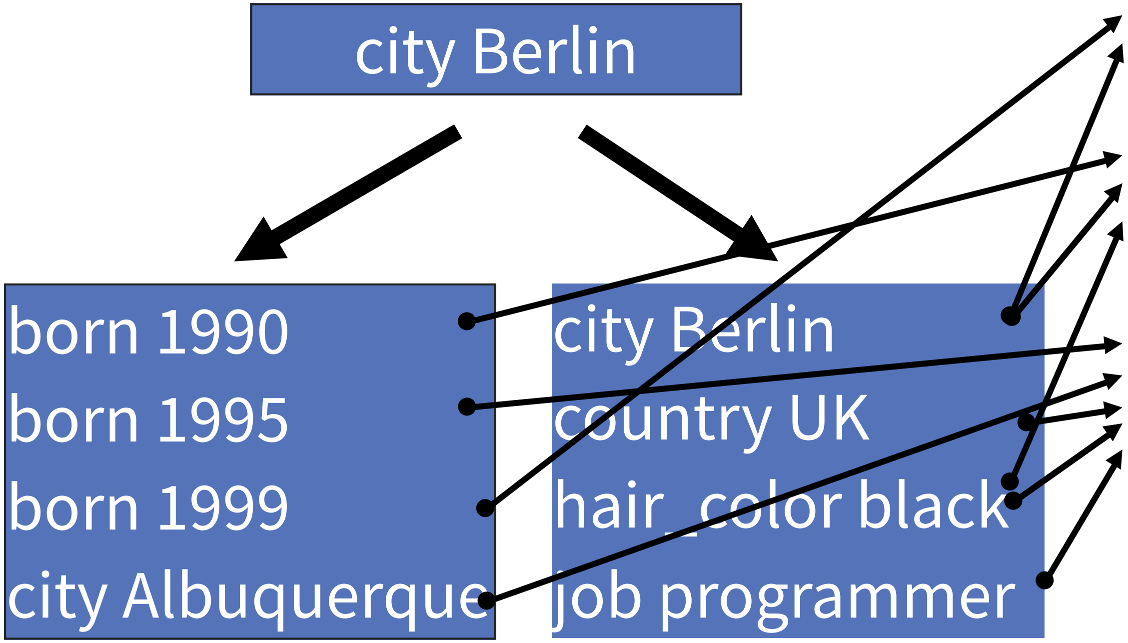
- index each key and value at the top level of the JSONB document (see next slide)
- supports existence (`?, ?|, ?&`) and path operators (`@>, @@, @?`)

```
-- this query can make use of the index
SELECT * FROM people WHERE profile @> '{"city" : "Berlin"}';  -- ✓

-- for this query, the index cannot be used:
SELECT * FROM people WHERE profile->>'city' = 'Berlin';       -- ✗
```

After creating a GIN on a JSONB column, in PostgreSQL, each attribute-value pair of the top level of the stored JSON document in that column is inserted into the index. This way, the index can be used in SQL/JSON queries that filter on these top-level attributes.

GIN - **G**eneralized **I**nverted **I**ndex



name	**profile**
Jane	{ "born" : 1999, "city": "Berlin" }
Peter	{ "born" : 1990, "city": "Berlin", "hair_c
Susan	{ "born" : 1995, "city": "Albuquerque", "country": "USA", "hair_color": "black", "job": "programmer" }

This GIN is a tree structure that contains all attribute-value pairs of the JSON documents in the column "profile" together with a pointer to the corresponding row. We can use the index to find all people that live in the city Berlin. Mind that only the top-level attributes are in the index.

Expression Indexes

B+Tree Index (no GIN) on the result of an expression:

```
CREATE INDEX people_profile_city_idx ON people ((profile->>'city'));
```

```
SELECT * FROM people WHERE profile->>'city' = 'Berlin';  -- ✓
```

GIN:

```
CREATE INDEX people_profile_hobby_idx ON people
USING GIN ((profile->'hobbies'));
```

```
SELECT * FROM people WHERE profile->'hobbies' ? 'yoga';  -- ✓
```

Expression indexes can be used to index only parts of the JSON data (e.g., only specific sub-fields). An expression index does not directly index the values of a column but the result of an expression. In the first query, we simply create a normal B+tree index on all city values that appear in the JSON column. In the second query, a GIN is built just on the JSON field "hobbies".

Indexes on JSON in other RDBMSs

Oracle: expression indexes

```
-- try out at http://bit.ly/ojsonidx
CREATE INDEX people_profile_city_idx ON people
(json_value(profile, '$.city'));
```

MariaDB: index on virtual columns

```
-- try out at http://bit.ly/mjsonidx
ALTER TABLE people
ADD profile_city VARCHAR(80) AS (JSON_VALUE(profile, '$.city'));
CREATE INDEX people_profile_city_idx ON people(profile_city);
```

Oracle, MariaDB, and others do not have a native `JSON` type (they use `VARCHAR`) and no native JSON index support. But there are workarounds which allow efficient queries on JSON data. Expression indexes as shown on the previous slide can be used in Oracle as well. Here, an index of all city values is created. Oracle will automatically use the index on queries with the given expression in the WHERE clause. In MariaDB, indexes on JSON can be achieved by creating virtual columns. A virtual column is a generated column which is computed when it is queried. It is not persisted, but MySQL supports indexes on virtual columns. Mind that only queries on the virtual column (`WHERE profile_city = ...`) use the index, not on the expression behind that column (`WHERE JSON_VALUE(profile, '$.city') = ...`).

Summary

- User-Defined Data Types and Composite Types in SQL
- SQL/XML: `XML` data type, `xpath`, `xpath_exists`, …
- SQL/JSON: `JSON` data type vs. string storage
- JSON in PostgreSQL: `JSONB`, `jsonb_build_object`, `jsonb_agg`, `->`, `->>`, `?`, `@>`, path expressions, …
- Indexes on JSON data: GIN, Expression Indexes, Virtual Columns