Advanced SQL

③

Standard and Non-Standard Data Types

Winter 2025/26

Torsten Grust
Universität Tübingen, Germany

1 ¦ Data Types in (DuckDB's) SQL

- The set of supported **data types** in DuckDB is varied:[1]

```sql
SELECT string_agg(t.type_name) AS "data types"
FROM   duckdb_types AS t
WHERE  t.database_name = 'memory';
```

```
┌──────────────────────────────────────────────────────────┐
│                        data types                        │
│                         varchar                          │
├──────────────────────────────────────────────────────────┤
│ bigint,binary,bit,blob,boolean,char,date,datetime,       │
│ decimal,double,enum,float,hugeint,int,interval,list,     │
│ map,numeric,real,row,struct,text,time,timestamp,uuid,    │
                             ⋮
└──────────────────────────────────────────────────────────┘
```

[1] See https://duckdb.org/docs/sql/data_types/overview

2 ┆ SQL Type Casts

Convert type of value e to τ at *runtime* via a **type cast:**

```
CAST(e AS τ)          -- SQL standard
e :: τ                -- shorthand, cf. FP
TRY_CAST(e AS τ)      -- yield NULL on failure
```

- ⚠️ Type casts can fail at query runtime.

- SQL performs **implicit casts** when the required target type is unambiguous (e.g. on insertion into a table column):

```
INSERT INTO T(a,b,c,d) VALUES (6.2, NULL, 'true', '0')
                                 ↑     ↑       ↑      ↑
      -- implicitly casts to:  int   text   boolean int
```

Literals (Casts From Type text)

SQL supports **literal syntax** for dozens of data types in terms of **casts from type** text:

CAST('<*literal*>' **AS** τ) '<*literal*>' :: τ τ '<*literal*>'	succeeds if <*literal*> has a valid interpretation as τ (without cast ⇏ type text)

- Embed complex literals (e.g., dates/times, JSON, enumerations) in SQL source.

- Casts from text to τ attempted **implicitly** if target type τ known. Also vital when importing data from text or CSV files (*input conversion*).

3 ┆ Text Data Types

```
text       -- UTF-8 string of unlimited length
string     -- ⎫
char       -- ⎬ synonyms for type text
varchar    -- ⎭
```

- Text values are of unlimited length. Specifying a "maximum length" (as in char(*n*), varchar(*n*)) is accepted for SQL compatibility but has no effect on the system or storage.
  - ⚠️ One UTF-8 character may occupy more than one byte.

- **NB.** SQL text literals are enclosed in *single* quotes '…' (or $<*id*>$…$<*id*>$ with matching <*id*>).

4 ¦ NUMERIC:[2] Large Fixed-Point Decimals

numeric(*width*, *scale*)

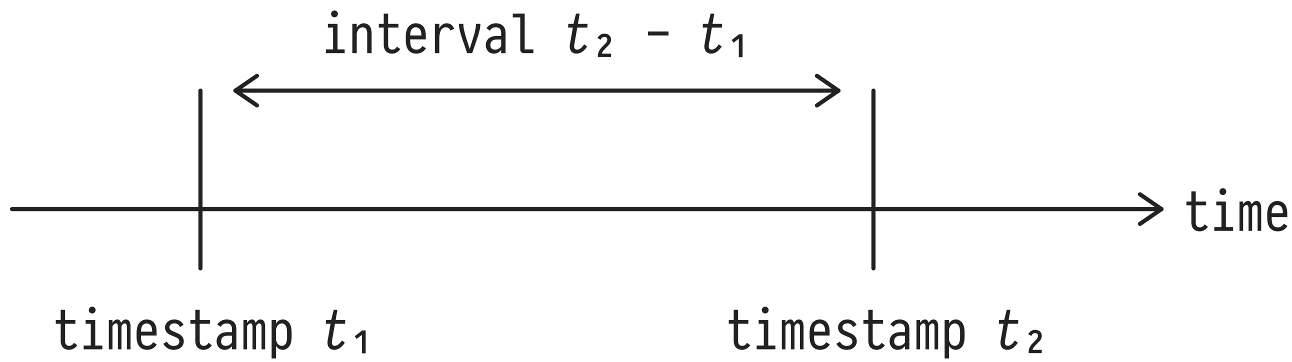
$$\underbrace{1234567.\overbrace{890}^{\textit{scale}}}_{\textit{width}\ (\#\ \text{of digits})}$$

- Shorthands: numeric(*width*,0) ≡ numeric(*width*), numeric ≡ numeric(18,3).
- Exact arithmetics, but may be computationally heavy for large *width* values. Internal representation are integers:

width	internal	size (bytes)
1-4	int16	2
5-9	int32	4
10-18	int64	8
19-38	int128	16

[2] Synonymous: decimal.

5 ¦ Timestamps and Time Intervals



- Types: timestamp ≡ (date, time). Casts between types: timestamp→time/date ✔, date→timestamp assumes 00:00:00. With time zone support: {time,timestamp} with time zone.[3]

- Type interval represents timestamp differences.

- Resolution: timestamp/time/interval: 1 µs, date: 1 day.

[3] Type name shorthands: timetz, timestamptz.

Date/Time/Interval Literals: DuckDB (ISO 8601 Format)

- Date literals (format `YYYY-MM-DD`): `date '1968-08-26'`, `'1904-05-30' :: date`.
- Time literals (`hh:mm:ss[.zzzzzz][+-TT[:tt]]`): `'14:20:33.982+00' :: time`.
- `timestamp` literal ≡ '‹date literal›␣‹time literal›'
- `interval` literal (fields optional, *s* may be fractional) ≡ '‹*n*›years ‹*n*›months ‹*n*›days ‹*n*›hours ‹*n*›mins ‹*s*›secs'
- Built-in date/time/interval literals:
  - `date`: `today()`
  - `timestamp`: `now()`, `today() :: timestamp`
  - `interval`: `interval(‹n›) ‹unit›` (… `interval(6) hours` …)

Computing with Time

- Timestamp arithmetic via +, - (also *, / with intervals):

```sql
SELECT now() :: timestamp - today() :: timestamp;
```

interval
12:58:58.022

- DuckDB: *Extensive* library of date/time/timestamp functions including:
  - strftime(), strptime() (flexible formatting/parsing)
  - age(•), month_name(•), extract(<part> from •)
  - make_date(•,•,•), make_time(…), make_timestamp(…)
  - comparisons (=, <, ...)

6 ┆ Enumerations

Create a *new* type τ, incomparable with any other. Explicitly **enumerate** the literals v_i of τ:

```
CREATE TYPE τ AS ENUM (v₁, …, vₙ);

SELECT vᵢ::τ;
```

- Literals v_i in case-sensitive quoted notation '…'. (Compact storage in $\lceil \log_2(n)/8 \rceil$ bytes, regardless of literal length.)

- Implicit ordering: $v_i < v_{i+1}$ (aggregates MIN, MAX ✓).

7 ¦ Bit Strings

- Data type bit stores strings of binary digits (storage: 1 byte per 8 bits + constant small overhead).

- Literals:

```
SELECT '0'::bit, '00101010'::bit, bitstring('1010',12)
--
--                          ≡ 000000001010 (zero-pad to 12 bits)
```

- Bitwise operations: & (and), | (or), xor, ~ (not), <</>> (shift left/right), get_bit(•,•), set_bit(•,•,•), …
- Operation on bit strings: bit_count(•) (counts 1-bits), bit_length(•), octet_length(•) (number of bytes), …
- Aggregates: bit_and(•), bit_or(•), bitstring_agg(•,•,•), …

Store **binary large object blocks** (BLOBs; 🖼, ♫♬ in column *B* of type blob) inline with alpha-numeric data. BLOBs remain *uninterpreted* by DBMS:

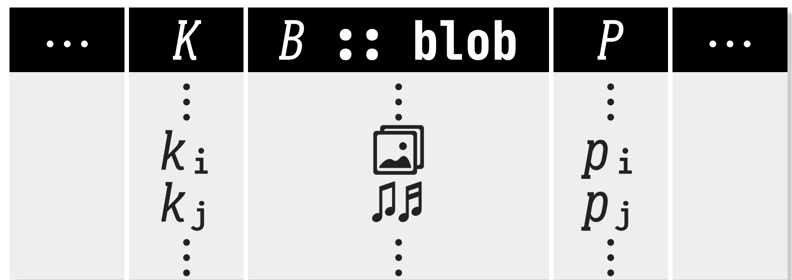
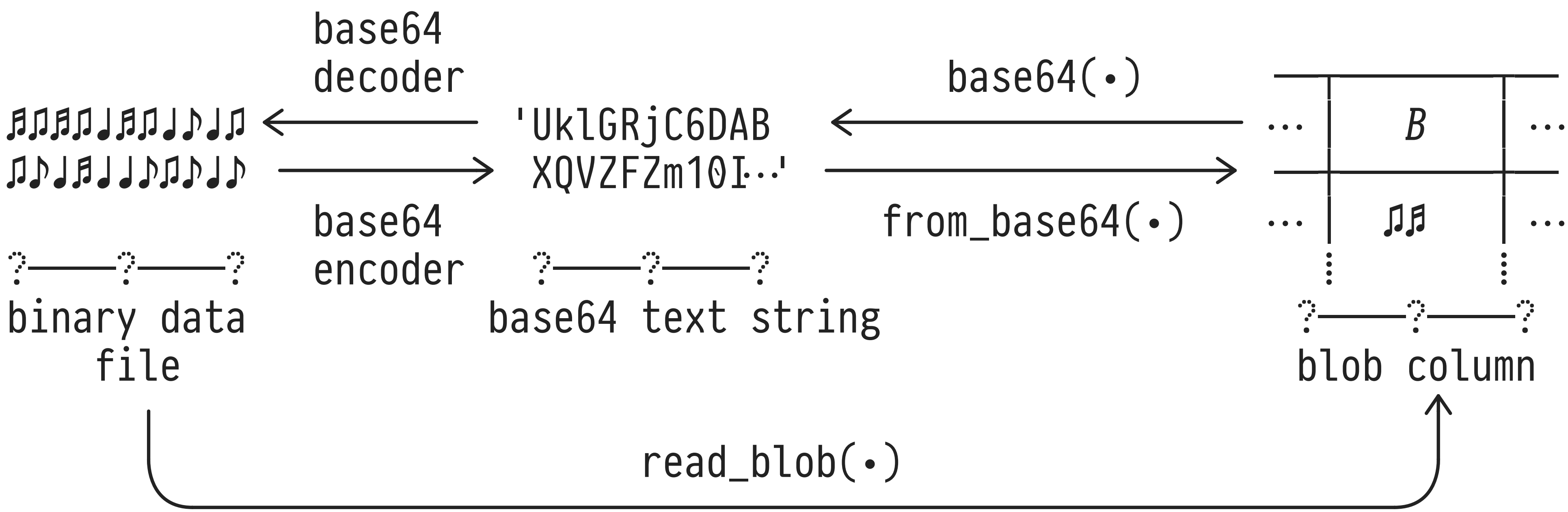


Table T

- Typical setup:
  - BLOBs stored alongside identifying **key** data (column *K*).
  - Additional **properties** (meta data, column(s) *P*) made explicit to filter/group/order BLOBs.

Encoding/Decoding BLOBs

- Import blob data via textual encoding (*e.g.*, base64) or directly from binary files via read_blob(•):



⚠️ Maximum size of a blob column value is 4 GB.

9 ¦ JSON (JavaScript Object Notation)

JSON defines a textual data interchange format. Easy for humans to write and machines to parse (see json.org):

```
‹object›   ::= {} | { ‹members› }
‹members›  ::= ‹pair› | ‹pair› , ‹members›
‹pair›     ::= ‹string› : ‹value›
‹array›    ::= [] | [ ‹elements› ]
‹elements› ::= ‹value› | ‹value› , ‹elements›
‹value›    ::= ‹string› | ‹number› | true | false | null
            |  ‹array› | ‹object›
```

- SQL:2016 defines SQL↔JSON interoperability. JSON ‹*value*›s may be constructed/traversed and held in table cells (we still consider 1NF to be intact).

JSON Sample <*value*>s

```
                          <members>
      { "title":"The Last Jedi", "episode":8 }
      ↑
<object>                                <pair>
```

Table T (see Chapter 02):

```
             [ { "a":1, "b":"x", "c":true,  "d":10   },
               { "a":2, "b":"y", "c":true,  "d":40   },
<elements>     { "a":3, "b":"x", "c":false, "d":30   },
               { "a":4, "b":"y", "c":false, "d":20   },
               { "a":5, "b":"x", "c":true,  "d":null } ]
                      ↑                                ↑
                  <number>              <array> (of <object>s)
```

JSON in DuckDB: Type json

Quoted literal syntax embeds JSON values in SQL queries.
Wrap in function json(·) to validate and minify JSON syntax:

```
VALUES (1, json('{ "b":1, "a":2 }')),
       (2, json('{ "a":1, "b":2, "a":3 }')),
       (3, json('[ 0,   false,null ]'));
```

col0	col1 (:: json)
1	{"b":1,"a":2}
2	{"a":1,"b":2,"a":3}
3	[0,false,null]

Navigating JSON Values

- Given a JSON value *value*, **access** object field *f* (or array element at index *i*) using binary operator `->`:[4]

value`->'`*f*`'` *value*`->`*i*	}	yields a JSON value, navigate further or cast to atomic type for further computation

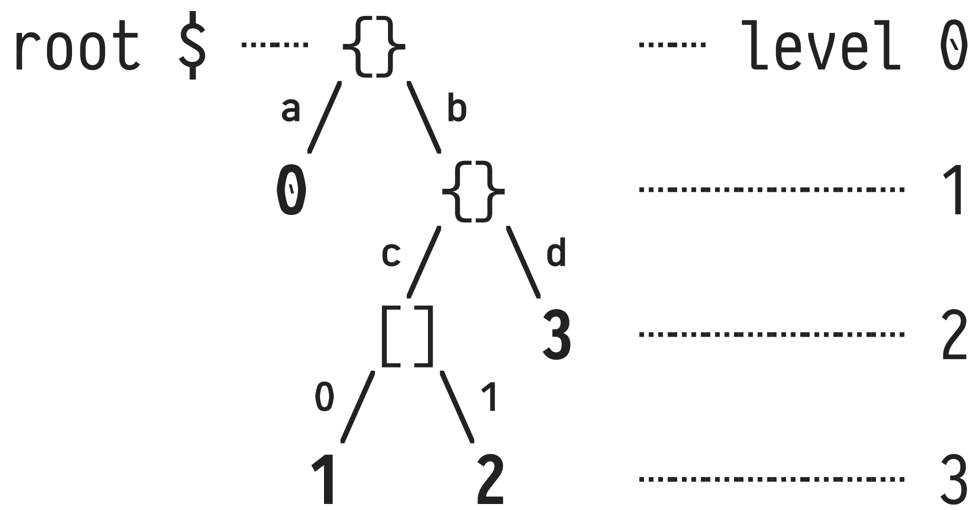
- **Path navigation:**
  1. Chain navigation steps: *value*`->'`f_1`'->'`f_2`'->…->'`f_n`'`.
  2. Use JSONPath expression: *value*`->'`‹*path*›`'`, see next slide.

[4] Accessing non-existent *f*/*i* yields `NULL`. **NB:** Array indexes *i* are 0-based.

Navigating JSON Values: JSONPath Expressions

- JSON values describe **tree-shaped** data:

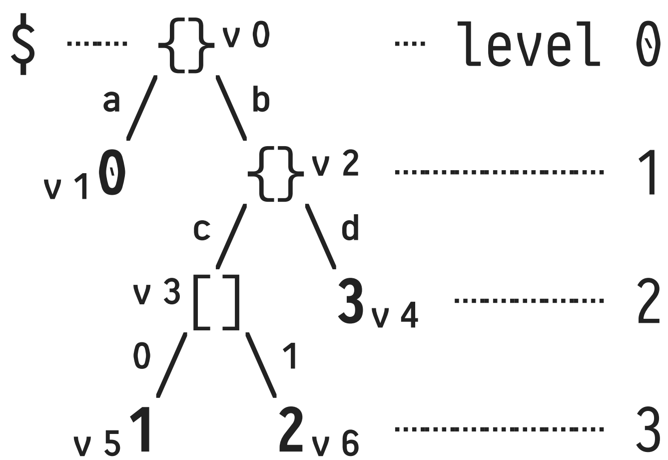
```
o ≡ { "a": 0,
      "b": { "c": [1,2],
             "d": 3
           }
    }
```



- Navigate from root $, return (array of) json values:

```
SELECT o->'<path>';
--
-- embed JSONPath expression as quoted text literal
```

Navigating JSON Values: The JSONPath Language



SELECT *o*->'‹*path*›' **AS** *j*

SELECT json_extract(*o*, '‹*path*›') **AS** *j*

‹*path*›	**result** *j*	
$	v_0	root
$.*	$[v_1,v_2]$	all child values of the root[5]
$.a	v_1	child a of the root
$.b.d	v_4	grandchild d below child b
$.b.c[1]	v_6	2nd array element of array c
$.b.c[*]	$[v_5,v_6]$	all array elements in array c
$.b.c[#-2]	v_5	second to last element in array c
$.b.**	$[v_3,v_4,v_5,v_6]$	all nodes in subtree below b

[5] In this case, -> returns a *SQL array* of values (type json[]). Chapter 04 will cover SQL arrays.

Bridging between JSON Objects and SQL Rows

- A JSON object *o* may be casted into a row value *t* provided that *o*'**s structure is known:**

```
--     o
SELECT json('{"a":1,"b":true}') :: row(a int, b boolean);
--     t
```

- Likewise, json(•) turns row values *t* into JSON objects:

```
SELECT json(t) AS o
FROM   T AS t;
```

o (:: json)
{"a":1,"b":"x","c":true,"d":10}
⋮
{"a":5,"b":"x","c":true,"d":null}

Constructing JSON Values From a Table of Inputs

- json_group_array(v): collect values v in a JSON array:

```
SELECT json_group_array(t.v) AS o
FROM   (VALUES (v₁),
               (v₂),
               (v₃)) AS t(v);
```

$[\ v_1,\ v_2,\ v_3\ ]$

- json_group_object(k,v): construct JSON object o from a table of (k,v) pairs:

```
SELECT json_group_object(t.k, t.v) AS o
FROM   (VALUES (k₁, v₁),
               (k₂, v₂),
               (k₃, v₃)) AS t(k,v);
```

$\{\ k_1\!:\ v_1,\ k_2\!:\ v_2,\ k_3\!:\ v_3\ \}$

Sequences represent counters of type bigint ($-2^{63} \ldots 2^{63}-1$).
Typically used to implement row identity/name generators:

```
CREATE SEQUENCE seq        -- sequence name
  [ INCREMENT inc ]        -- advance by inc (default: 1≡↑)
  [ MINVALUE min ]         -- range of valid counter values
  [ MAXVALUE max ]         --   (defaults: [1…2⁶³-1])
  [ START start ]          -- start (default: min if ↑, max if ↓)
  [ [NO] CYCLE ]           -- wrap around or error(≡ default)?
```

- Columns can be tied to a sequence for key generation:

```
CREATE SEQUENCE T_keys;
CREATE TABLE T (k int DEFAULT nextval('T_keys'), …);
```

Advancing and Inspecting Sequence State

- Counter state can be advanced (nextval(•)) and inspected (currval(•)):

```
-- seq: 41 → 42 → 1 → 2 → … 41 → 42 → 1 → …
CREATE SEQUENCE seq START 41 MAXVALUE 42 CYCLE;
   ⋮
SELECT nextval('seq');        --  ⇛ 41
SELECT nextval('seq');        --  ⇛ 42
SELECT currval('seq');        --  ⇛ 42
SELECT nextval('seq');        --  ⇛ 1   (wrap-around)
                ↑
     ⚠️  sequence/table names are not first class in SQL
```

- *c* τ DEFAULT *e* evaluates epxression *e* (of type τ) whenever row insertion omits a value for column *c*.