

# Advanced SQL

---

03 — Standard and Non-Standard Data Types

Summer 2024

Torsten Grust  
Universität Tübingen, Germany

## 1 | Data Types in (DuckDB's) SQL

---

- The set of supported **data types** in DuckDB is varied:<sup>1</sup>

```
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, ⋮


<sup>1</sup> See [https://duckdb.org/docs/sql/data\\_types/overview](https://duckdb.org/docs/sql/data_types/overview)

## 2 | SQL Type Casts

---

**Convert type** of value  $e$  to  $\tau$  at *runtime* via a **type cast**:

<code>CAST(<math>e</math> AS <math>\tau</math>)</code>	-- SQL standard
<code><math>e</math> :: <math>\tau</math></code>	-- shorthand, cf. FP
<code>TRY_CAST(<math>e</math> AS <math>\tau</math>)</code>	-- 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):

<code>INSERT INTO T(a,b,c,d) VALUES</code>	<code>(6.2,</code>	<code>NULL,</code>	<code>'true',</code>	<code>'0')</code>
	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$
	-- 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`**:


<code>CAST('⟨literal⟩' AS <math>\tau</math>)</code> <code>'⟨literal⟩' :: <math>\tau</math></code> <code><math>\tau</math> '⟨literal⟩'</code>	$\left. \begin{array}{l} \text{ } \end{array} \right\} \begin{array}{l} \text{succeeds if } \langle \textit{literal} \rangle \text{ has a} \\ \text{valid interpretation as } \tau \\ \text{(without cast } \Rightarrow \text{ type } \underline{\textit{text}}) \end{array}$
--	---

- Embed complex literals (e.g., dates/times, JSON, enumerations) in SQL source.
- Casts from `text` to  $\tau$  attempted **implicitly** if target type  $\tau$  known. Also vital when importing data from text or CSV files (*input conversion*).

### 3 : Text Data Types

---

<u>text</u>	-- UTF-8 string of unlimited length
<u>string</u>	--
<u>char</u>	-- } synonyms for type text
<u>varchar</u>	-- }

- 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:**<sup>2</sup> Large Fixed-Point Decimals

numeric(*width*, *scale*)

$\underbrace{\hspace{1.5cm}}_{\text{scale}}$   
 1234567.890  
 $\underbrace{\hspace{1.5cm}}_{\text{width (\# of digits)}}$

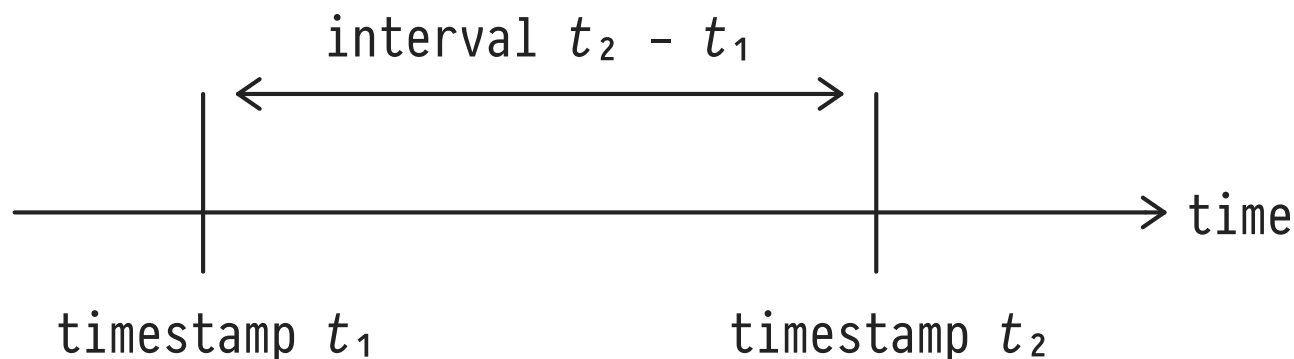
- Shorthands: `numeric(width,0)`  $\equiv$  `numeric(width)`,  
`numeric`  $\equiv$  `numeric(18,3)`.
- Exact arithmetics, but may be computationally heavy for large *width* values. Internal representation are integers:

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

<sup>2</sup> Synonymous: `decimal`.

## 5 : Timestamps and Time Intervals

---



- Types: `timestamp`  $\equiv$  (`date`, `time`). Casts between types: `timestamp`  $\rightarrow$  `time/date`  $\checkmark$ , `date`  $\rightarrow$  `timestamp` assumes 00:00:00. With time zone support: `{time,timestamp}` with time zone.<sup>3</sup>
- Type `interval` represents timestamp differences.
- Resolution: `timestamp/time/interval`: 1  $\mu$ s, `date`: 1 day.

<sup>3</sup> Type name shorthands: `timetz`, `timestampz`.

## 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  $\equiv$  `'<date literal>_<time literal>'`
- `interval` literal (fields optional, `s` may be fractional)  $\equiv$  `'<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):

```
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  $\tau$ , incomparable with any other. Explicitly **enumerate** the literals  $v_i$  of  $\tau$ :

```
CREATE TYPE  $\tau$  AS ENUM ( $v_1$ , ...,  $v_n$ );
```

```
SELECT  $v_i::\tau$ ;
```

- Literals  $v_i$  in case-sensitive quoted notation ' $\dots$ '.  
(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(.,.,.)`, ...

## 8 : Binary Arrays (BLOBs)

---

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



...	<i>K</i>	<i>B</i> :: <i>blob</i>	<i>P</i>	...
	⋮ $k_i$ $k_j$ ⋮	⋮   ⋮	⋮ $p_i$ $p_j$ ⋮	

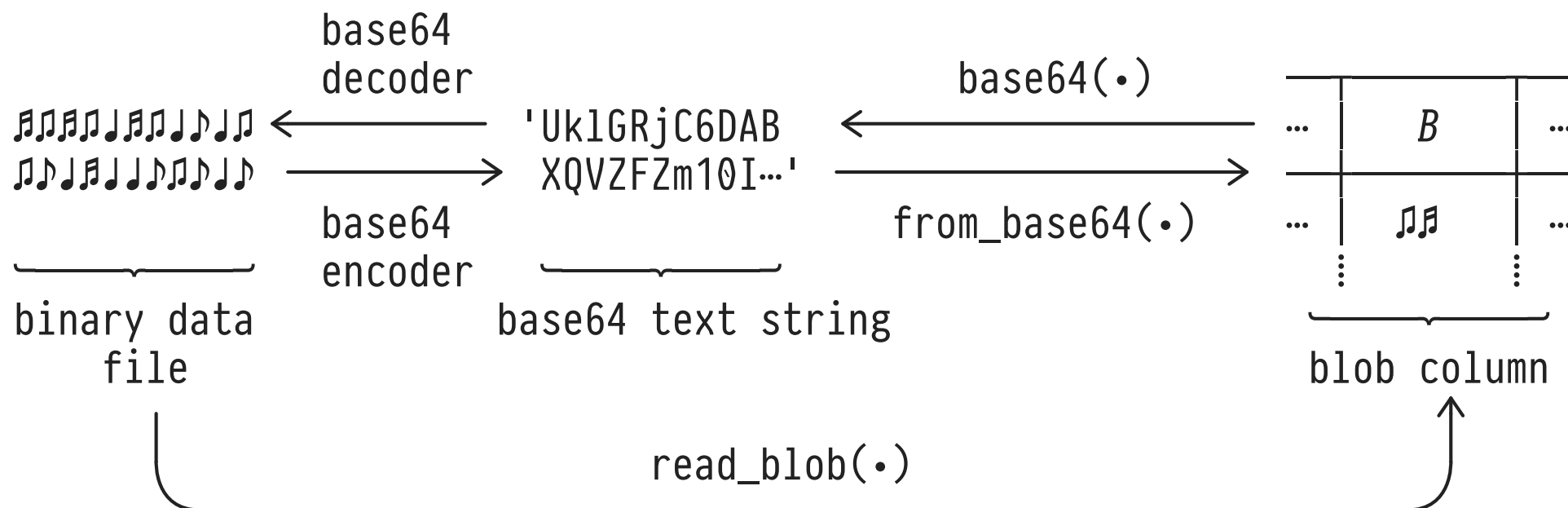
Table *T*

- 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 <http://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>*

$\overbrace{\{ \text{"title": "The Last Jedi", \text{"episode": 8 } \}}^{\text{}}$

$\uparrow$   
*<object>*
 $\underbrace{\text{"episode": 8}}_{\text{}} \text{ } \}$   
*<pair>*

Table T (see Chapter 02):

*<elements>* {
 [
 { "a": 1, "b": "x", "c": true, "d": 10 },
 { "a": 2, "b": "y", "c": true, "d": 40 },
 { "a": 3, "b": "x", "c": false, "d": 30 },
 { "a": 4, "b": "y", "c": false, "d": 20 },
 { "a": 5, "b": "x", "c": true, "d": null } ]
 }

$\uparrow$   
*<number>*
 $\uparrow$   
*<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  $\rightarrow$ :<sup>4</sup>

$value \rightarrow 'f'$	}	yields a JSON value, navigate further or cast to atomic type for further computation
$value \rightarrow 'i'$		

- Path navigation:**

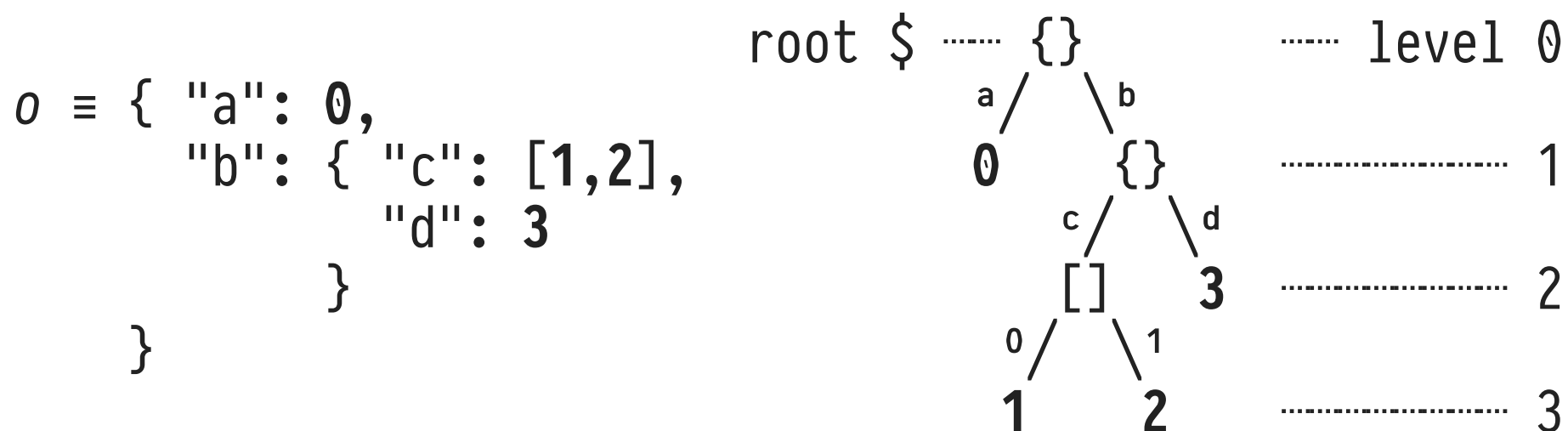
- Chain navigation steps:  $value \rightarrow 'f_1' \rightarrow 'f_2' \rightarrow \dots \rightarrow 'f_n'$ .
- Use JSONPath expression:  $value \rightarrow '\langle path \rangle'$ , see next slide.

<sup>4</sup> 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:



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

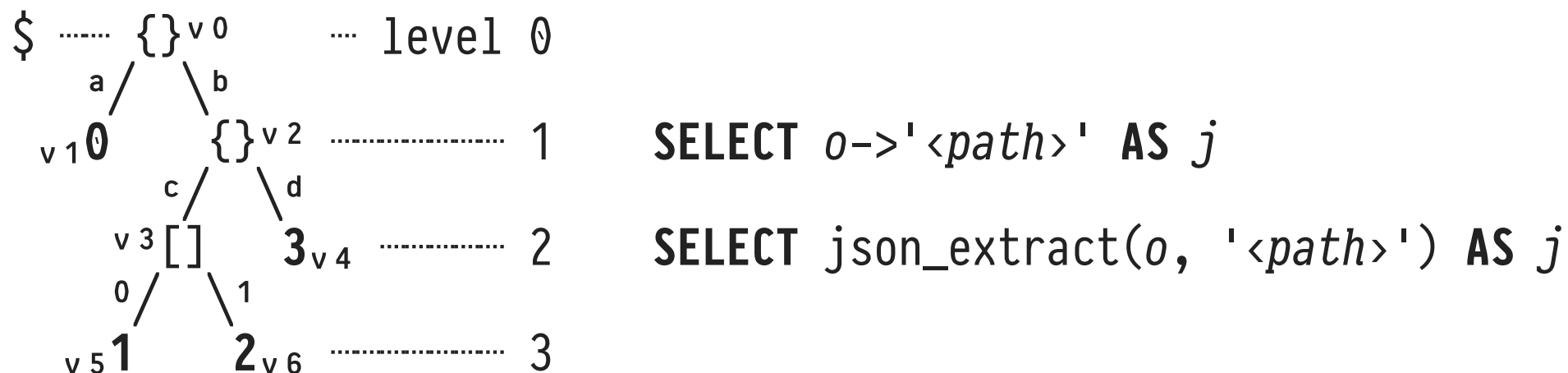
```
SELECT o->'<path>';
```

```
--
```

```
-- embed JSONPath expression as quoted text literal
```

# Navigating JSON Values: The JSONPath Language

---



<path>	result j	
\$	v <sub>0</sub>	root
\$.*	[v <sub>1</sub> , v <sub>2</sub> ]	all child values of the root <sup>5</sup>
\$.a	v <sub>1</sub>	child a of the root
\$.b.d	v <sub>4</sub>	grandchild d below child b
\$.b.c[1]	v <sub>6</sub>	2nd array element of array c
\$.b.c[*]	[v <sub>5</sub> , v <sub>6</sub> ]	all array elements in array c
\$.b.c[#-2]	v <sub>5</sub>	second to last element in array c

<sup>5</sup> 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  { "a":1, "b":true }
SELECT json('{"a":1,"b":true}') :: row(a int, b boolean);
--      t  { "a":1, "b":true }
```

- 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:

<b>SELECT</b> json_group_array(t.v) <b>AS</b> o <b>FROM</b> (VALUES (v <sub>1</sub> ), (v <sub>2</sub> ), (v <sub>3</sub> )) <b>AS</b> t(v);		[ v <sub>1</sub> , v <sub>2</sub> , v <sub>3</sub> ]
---	--	--

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

<b>SELECT</b> json_group_object(t.k, t.v) <b>AS</b> o <b>FROM</b> (VALUES (k <sub>1</sub> , v <sub>1</sub> ), (k <sub>2</sub> , v <sub>2</sub> ), (k <sub>3</sub> , v <sub>3</sub> )) <b>AS</b> t(k,v);		{ k <sub>1</sub> : v <sub>1</sub> , k <sub>2</sub> : v <sub>2</sub> , k <sub>3</sub> : v <sub>3</sub> }
--	--	---

## 10 | Sequences

---

**Sequences** represent counters of type **bigint** ( $-2^{63} \dots 2^{63}-1$ ). 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 100 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 expression `e` (of type `τ`) whenever row insertion omits a value for column `c`.