Advanced SQL

02 — Standard and Non-Standard Data Types

Torsten Grust
Universität Tübingen, Germany

1 Data Types in (Postgre)SQL

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

```
SELECT string_agg(t.typname, ' ') AS "data types"
FROM   pg_catalog.pg_type AS t
WHERE  t.typelem = 0 AND t.typrelid = 0;
```

data types
bool bytea char int8 int2 int4 regproc text oid tid oid tid xid cid json xml pg_node_tree pg_ddl_command smgr path polygon float4 float8 abstime reltime tinterval unknown circle money macaddr inet cidr …

[1] See https://www.postgresql.org/docs/9.6/static/datatype.html

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›          ⎬ equivalent   (PostgreSQLism, cf. FP)
‹τ›(‹e›)            ⎭              (if ‹τ› valid func name)
```

- ⚠️ Type cast can fail at 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 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, XML, geometric objects) 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

```
char            -- ≡ char(1)
char(‹n›)       -- fixed length, blank (␣) padded if needed
varchar(‹n›)    -- varying length ≤ n characters
text            -- varying length, unlimited
```

- Length limits measured in characters, *not* bytes. (PostgreSQL: max size ≅ 1 Gb. Large text is “TOASTed.”)

- Length limits are enforced:
  1. Excess characters (other than ␣) yield runtime errors.
  2. Explicit casts truncate to length ‹n›.

- char(‹n›) always *printed/stored* using ‹n› characters: pad with ␣. ⚠️ Trailing blanks removed before computation. ✎

4 ¦ NUMERIC:[2] Large Numeric Values with Exact Arithmetics

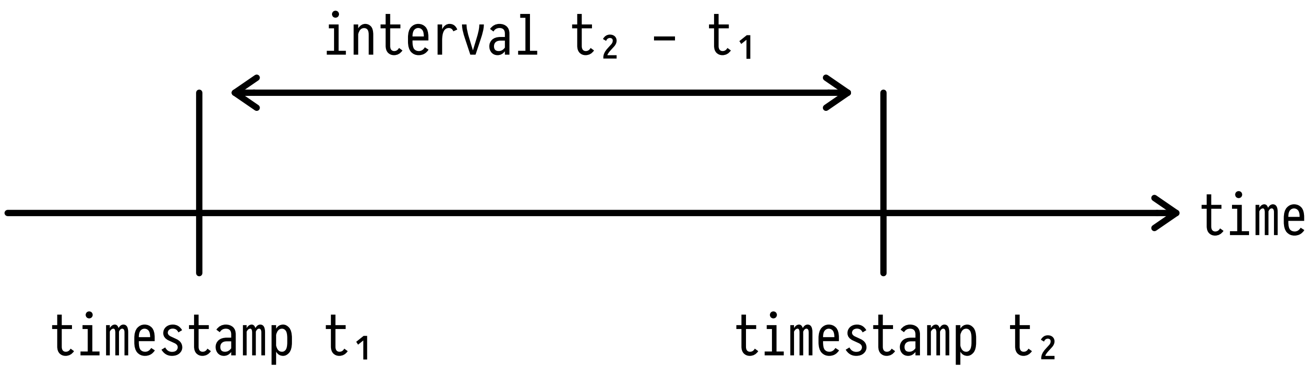
numeric(‹precision›, ‹scale›)

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

- Shorthand: numeric(‹precision›,0) ≡ numeric(‹precision›). numeric ≡ "∞ precision" (PostgreSQL limit: 100000+).
- Exact arithmetics, but computationally heavy. ✎
- Leading/trailing 0s *not* stored ⇒ variable-length data.

[2] Synonymous: decimal.

Timestamps and Time Intervals



- Types: timestamp ≡ (date, time). Casts between types: timestamp→time/date ✔, date→timestamp assumes 00:00:00. Optional timezone support: ‹τ› with time zone or ‹τ›tz.

- Type interval represents timestamp differences.

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

Date/Time Literals: PostgreSQL

- Literal input and output: flexible/human-readable ✎, affected by `SET datestyle='{German,ISO},{MDY,DMY,YMD}'`

$$\texttt{SET datestyle='}\underbrace{\texttt{\{German,ISO\}}}_{\text{output}}\texttt{,}\underbrace{\texttt{\{MDY,DMY,YMD\}}}_{\text{input}}\texttt{'}$$

- `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'

- Special literals:
  - timestamp: 'epoch', '[-]infinity', 'now'
  - date: 'today', 'yesterday', 'tomorrow'

Computing with Time

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

```sql
SELECT ('now'::timestamp - 'yesterday'::date)::interval
```

interval
1 day 17:27:47.454803

- PostgreSQL: *Extensive* library of date/time functions including: ✎

  - timeofday() (⚠️ yields text)
  - extract(<field> from •)
  - make_date(•,•,•), make_time(…), make_timestamp(…)
  - comparisons (=, <, ...), (•,•) overlaps (•,•)

6 ┊ Enumerations

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

```
CREATE TYPE ‹τ› AS ENUM (‹v_1›, ..., ‹v_n›);

SELECT ‹v_i›::‹τ›;
```

- Literals ‹v_i› in case-sensitive string notation '…'.
  (Storage: 4 bytes, regardless of literal length.)

- Implicit ordering: ‹v_i› < ‹v_j› (aggregates MIN, MAX ✓).

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

- Literals:

```
SELECT B'00101010', X'2A', '00101010'::bit(8), 42::bit(8)
```

$\underbrace{\texttt{X'2A'}}_{2 \times 4 \text{ bits}}$

- Bitwise operations: `&` (and), `|` (or), `#` (xor), `~` (not), `<</>>` (shift left/right), `get_bit(•,•)`, `set_bit(•,•)`

- String-like operations: `||` (concatenation), `length(•)`, `bit_length(•)`, `position(• in •)`, ...

Binary Arrays (BLOBs)

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

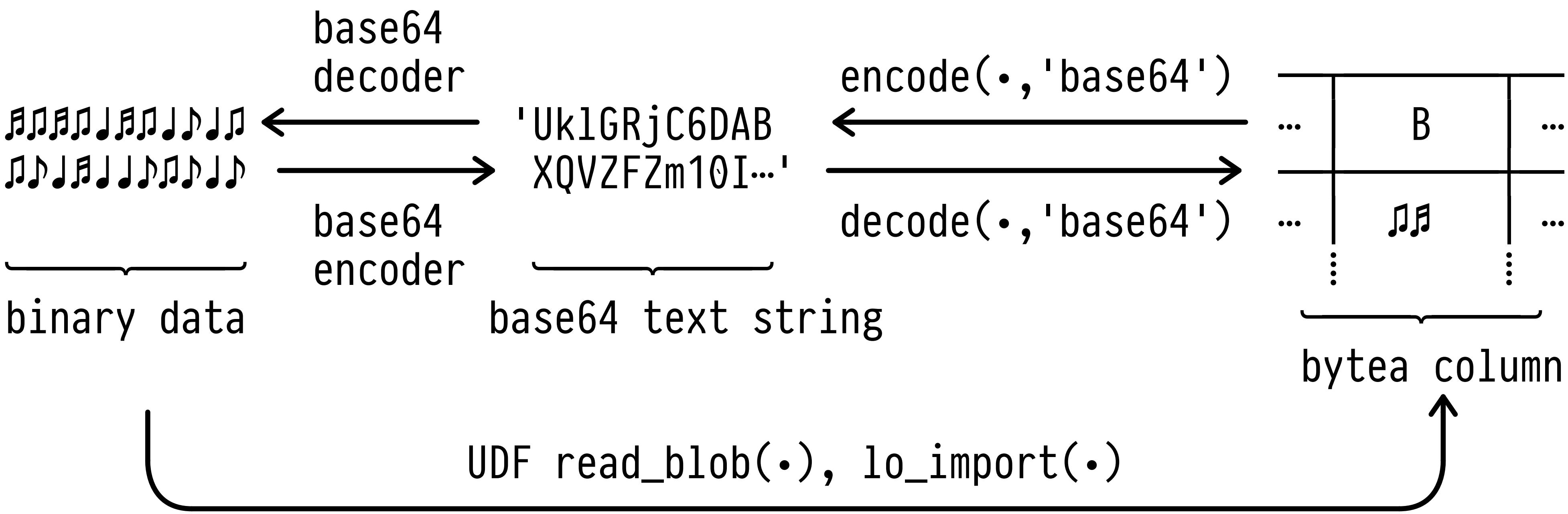
Table T

…	K	B :: bytea	P	…
	⋮	⋮	⋮	
	k_i	👁	p_i	
	k_j	♫♬	p_j	
	⋮	⋮	⋮	

- 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/export bytea data via textual encoding (e.g., base64) or directly from/to binary files:



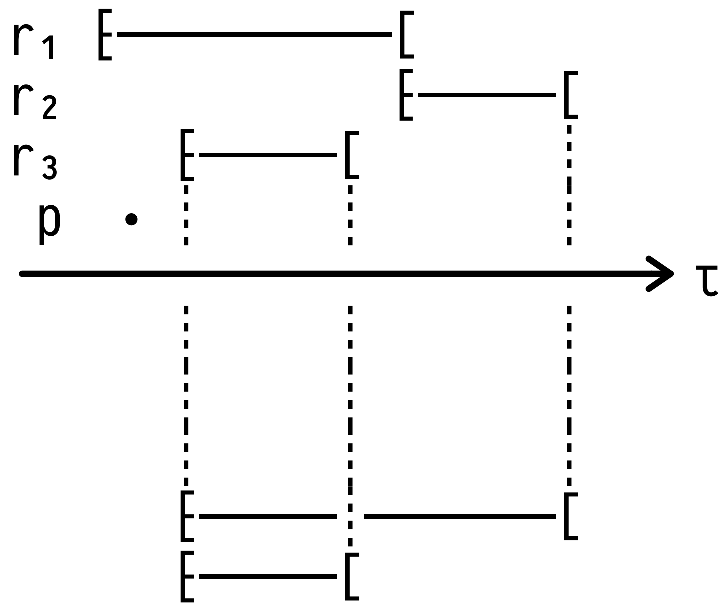
⚠️ File I/O performed by DBMS server (paths, permissions).

Given lower and/or upper bounds ‹ℓ›, ‹u› of an ordered type ‹τ› ∈ {int4,int8,num(eric),timestamp,date}, construct **range** literals of type ‹τ›range via

[‹ℓ›,‹u›]	‹ℓ› ≤ x ≤ ‹u›	[————]
[‹ℓ›,‹u›)	‹ℓ› ≤ x < ‹u›	[————[
(,‹u›]	x ≤ ‹u›	··————]
(‹ℓ›,)	‹ℓ› < x	]————··
empty	ϕ	

- Alternatively use function ‹τ›range(‹ℓ›,‹u›,'[)'), NULL represents no bound (∞).

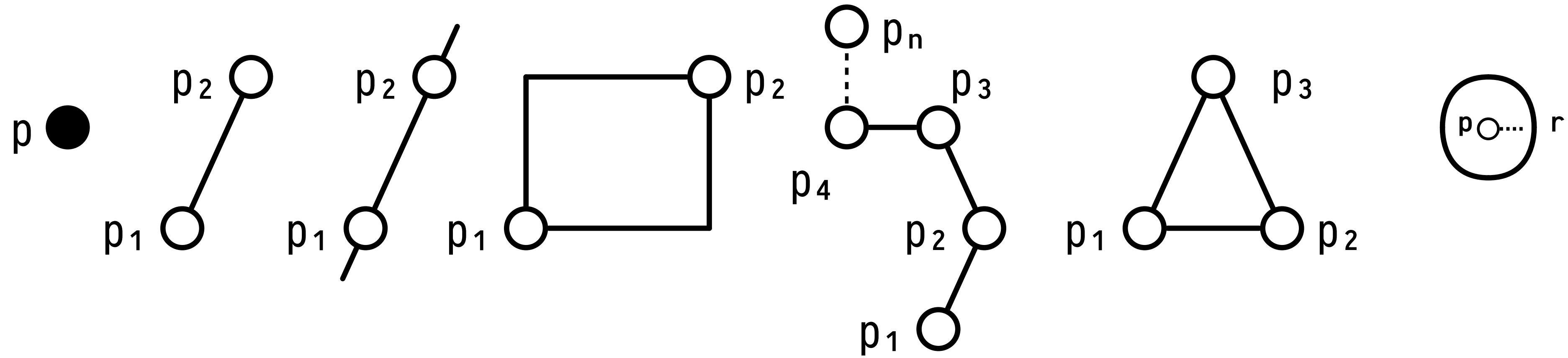
Range Operations



r_1 @> p	r_3 <@ r_1	contains, contained by
r_1 -|- r_2		is adjacent to
r_3 << r_2	r_1 << r_2	strictly left of
r_2 + r_3		union
r_1 * r_3		intersection
r_1 && r_3		overlaps

- Additional family of range-supporting functions:
  - lower(·), upper(·) (bound extraction)
  - lower_inc(·) (bound closed?), lower_inf(·) (unbounded?)
  - isempty(·)

Constructing **geometric objects** in PostgreSQL:



`'(x,y)'`
`point(x,y)` · `line(`p_1`,`p_2`)` · `box(`p_1`,`p_2`)` · `'[`p_1`,...,`p_n`]'` (open path) · `'(`p_1`,...,`p_n`)'` (polygon) · `circle(p,r)`

`lseg(`p_1`,`p_2`)`

- Alternative string literal syntax (see PostgreSQL docs):
  - `'((`x_1`,`y_1`),(`x_2`,`y_2`))'::lseg`, `'<(x,y),r>'::circle`, ...

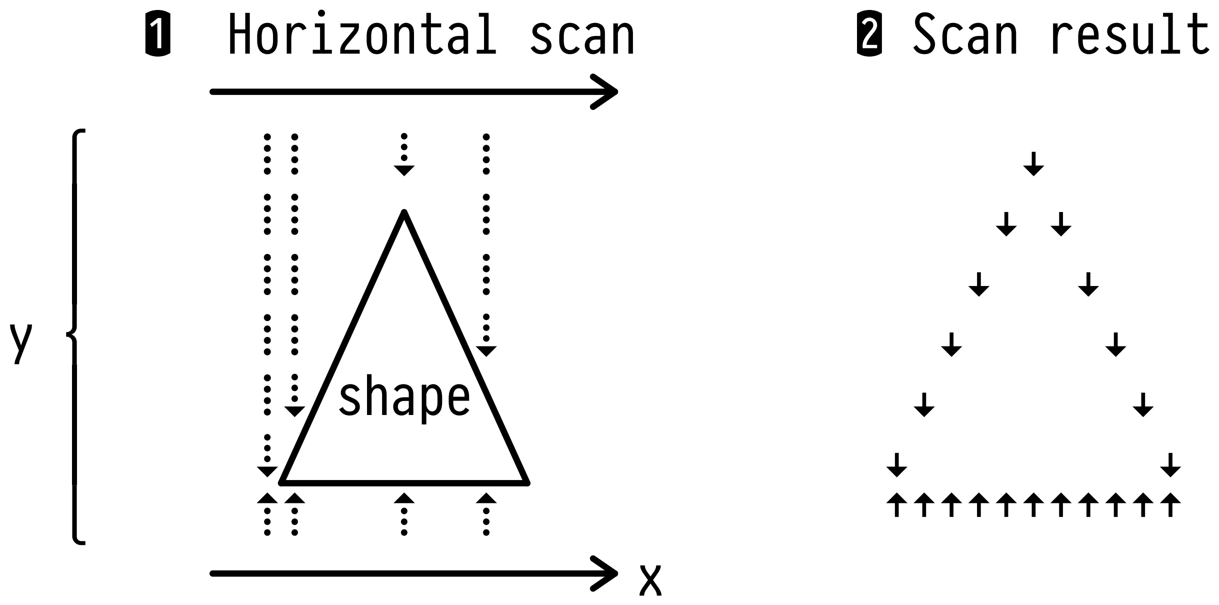
Querying Geometric Objects ✎

- A vast library of geometric operations (excerpt):

	Operation		Operation
+, -	translate	area(•)	area
*	scale/rotate	height(•)	height of box
@-@	length/circumference	width(•)	width of box
@@	center	bound_box(•,•)	bounding box
<->	distance between	diameter(•)	diameter of circle
&&	overlaps?	center(•)	center
<<	strictly left of?	isclosed(•)	path closed?
?-|	is perpendicular?	npoints(•)	# of points in path
@>	contains?	pclose(•)	close an open path

- (p)[0], (p)[1] to access x/y coordinate of point p.

Use Case: Shape Scanner



- Given an unknown shape (a polygon geometric object):
  1. Perform horizontal “scan” to trace minimum/maximum (i.e., bottom/top) y values for each x.
  2. Use bottom/top traces to render the shape.

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

Table T (see Chapter 01):

```
            [ { "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 PostgreSQL: Type jsonb[3]

Literal string syntax embeds JSON <value>s in SQL queries.
Casting to type jsonb validates and encodes JSON syntax:

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

column1	column2
1	{"a": 2, "b": 1}
2	{"a": 3, "b": 2}
3	[0, false, null]

[3] Alternative type json preserves member order, duplicate fields, and whitespace.
⚠️ Reparses JSON values on each access, no index support.

Navigating JSON <value>s

- **Access** field <f> / element at index <i> in array <value> via -> or ->>:[4]

```
<value>-><f>  ⎱  yields a jsonb value, permits further
<value>-><i>  ⎰  navigation steps via ->, ->>

<value>->><f> ⎱  yields a text value (cast to atomic type
<value>->><i> ⎰  for further computation)
```

- **Path navigation:** chain multiple navigation steps via #> or #>>: <value> #> '{<f or i>,...,<f or i>}'.

[4] Extracting non-existing fields yields NULL. JSON arrays are 0-based.

Bridging between JSON and SQL

Turn the fields and/or nested values inside JSON object ‹o› ≡ { ‹f₁›:‹v₁›,...,‹fₙ›:‹vₙ› } or array ‹a› ≡ [‹v₁›,...,‹vₙ›] **into tables** which we can query:[5]

```
SELECT *
FROM   jsonb_each(‹o›)
```

key	value
$‹f_1›$	$‹v_1›$
⋮	⋮
$‹f_n›$	$‹v_n›$

```
SELECT *
FROM   jsonb_array_elements(‹a›)
```

value
$‹v_1›$
⋮
$‹v_n›$

[5] Re jsonb_each(•): jsonb_to_record(•) or jsonb_populate_record(τ,•) help to create typed records.

Constructing JSON ‹value›s ✎

- row_to_json(•)::jsonb

  Convert a single **SQL row into a JSON ‹object›.** Column names turn into field names:

```
SELECT row_to_json(t)::jsonb -- yields objects of the form
FROM   T AS t;               -- {"a":•,"b":•,"c":•,"d":•}
```

- array_to_json(array_agg(•))::jsonb

  Aggregate **JSON ‹object›s into a JSON ‹array›:**

```
--          a unity for now
--
SELECT array_to_json(array_agg(row_to_json(t)))::jsonb
FROM   T AS t;
```

XML defines textual format to describe ordered *n*-ary trees:

```
<movie>
  <release>Dec 18, 2017</release>
  <title episode="8">The Last Jedi</title>
</movie>
```

```
                 movie
                /     \
          release      title
             |        /     \
      "Dec 18" @epsiode "The Last Jedi"
                  :
                 "8"
```

- XML support in SQL predates JSON support. Both are similar in nature. XML not discussed further here.[6]

[6] See the course *Database-Supported XML Processors*.

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>, ↓<max>)
  [ [NO] CYCLE ]          -- wrap around or error(≡ default)?
```

- Declaring a column of type serial creates a sequence:

```
CREATE TABLE <T> (..., <c> serial, ...) -- implies NOT NULL
                  ⇩
CREATE SEQUENCE <T>_<c>_seq;
```

Advancing and Inspecting Sequence State ✎

- Counter state can be (automatically) advanced and inspected:

```
CREATE SEQUENCE <seq> START 41 MAXVALUE 100 CYCLE;
   ⋮
SELECT nextval('<seq>');      -- ⇛ 41
SELECT nextval('<seq>');      -- ⇛ 42
SELECT currval('<seq>');      -- ⇛ 42
SELECT setval ('<seq>',100);  -- ⇛ 100 (+ side effect)
SELECT nextval('<seq>');      -- ⇛ 1   (wrap-around)
               ⬆
    ⚠  sequence/table names are not 1st class in SQL
```

- Columns of type `serial` automatically populate with (and advance) their current counter value when set to `DEFAULT`.