# PostgreSQL Native Data Types

- When designing a database table, you should take care to pick the appropriate data type.
- When the database goes to production, changing the data type of a column can become a very costly operation, especially for heavily-loaded tables.
- The cost often comes from locking the table, and in some cases, rewriting it.

- When picking a data type, consider a balance between the following factors:
  - **Extensibility**: Can the maximum length of a type be increased or decreased without a full table rewrite and a full table scan?
  - **Data type size**: Going for a safe option, such as choosing big integers instead of integers, will cause more storage consumption.
  - **Support**: This factor is important for rich data types, such as XML, JSON, and hstore. If the drivers, such as JDBC drivers, don't support rich types, you need to write your own code to serialize and deserialize the data.

- PostgreSQL provides a very extensive set of data types.
- Some of the native data type categories are as follows:
  - Numeric type
  - Character type
  - Date and time types
- These data types are common for most relational databases.
- Moreover, they are often sufficient for modeling traditional applications.

# Numeric Types

| Name | Comments | Size | Range |
|---|---|---|---|
| `smallint` | SQL equivalent: `Int2` | 2 bytes | -32,768 to +32,767. |
| `integer` | SQL equivalent: `Int4`<br><br>Integer is an alias for `INT`. | 4 bytes | -2,147,483,648 to +2,147,483,647. |
| `bigint` | SQL equivalent: `Int8`<br>8 bytes | 8 bytes | -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807. |

| | | | |
|---|---|---|---|
| `numeric` or `decimal` | No difference in PostgreSQL | Variable | Up to 131,072 digits before the decimal point; up to 16,383 digits after the decimal point. |
| `real` | Special values: Infinity, Infinity, `NaN` | 4 bytes | Platform-dependent, at least six-digit precision. Often, the range is 1E-37 to 1E+37. |
| `double precision` | Special values: Infinity, Infinity, `NaN` | 8 bytes | Platform dependent, at least 15-digit precision. Often, the range is 1E-307 to 1E+308. |

- PostgreSQL supports various mathematical operators and functions, such as geometric functions and bitwise operations.
- The `smallint` data type can be used to save disk space, while `bigint` can be used if the `integer` range isn't sufficient.

- Similar to the C language, the result of an integer expression is also an integer.
  - So, the results of the mathematical operations `3/2` and `1/3` are `1` and `0`, respectively.
  - Thus, the fractional part is always truncated.

```
postgres=# SELECT 2/3 AS "2/3", 1/3 AS "1/3", 3/2 AS "3/2";
 2/3 | 1/3 | 3/2
-----+-----+-----
   0 |   0 |   1
(1 row)
```

- The `numeric` and `decimal` types are recommended for storing monetary and other amounts where precision is required.
- There are three forms of definition for a numeric or decimal value:
  - Numeric (precision, scale)
  - Numeric (precision)
  - Numeric
- Precision is the total number of digits, while scale is the number of digits of the fraction part. For example, the number `12.344` has a precision of `5` and a scale of `3`.
- If a `numeric` type is used to define a column type without precision or scale, the column can store any number with any precision and scale.
- If precision isn't required, don't use the `numeric` and `decimal` types.
  - Operations on numeric types are slower than floats and double precision.

- Floating-point and double precision are inexact; this means that the values in some cases can't be represented in the internal binary format, and are stored as approximations.

- The full documentation about numeric data types can be found at https://www.postgresql.org/docs/current/static/datatype-numeric.html.

- Serial types, namely `smallserial`, `serial`, and `bigserial`, are wrappers on top of `smallint`, `integer`, and `bigint`, respectively.
- `serial` types aren't true data types.
- They're often used as surrogate keys, and by default, they aren't allowed to have a `null` value.
- The `serial` type utilizes the `sequence`s behind the scene.
- A `sequence` is a database object that's used to generate sequences by specifying the minimum, maximum, and increment values.

```
postgres=# CREATE TABLE customer (
postgres(#   customer_id SERIAL
postgres(# );
CREATE TABLE
postgres=# \d customer
                                Table "public.customer"
   Column    |  Type   | Collation | Nullable |                   Default
-------------+---------+-----------+----------+--------------------------------------------
 customer_id | integer |           | not null | nextval('customer_customer_id_seq'::regclass)

postgres=# \d customer_customer_id_seq
              Sequence "public.customer_customer_id_seq"
  Type   | Start | Minimum |  Maximum   | Increment | Cycles? | Cache
---------+-------+---------+------------+-----------+---------+-------
 integer |     1 |       1 | 2147483647 |         1 | no      |     1
Owned by: public.customer.customer_id
```

- When creating a column with the serial type, remember the following things:
  - A sequence will be created with the name `tableName_columnName_seq`. In the preceding example, the sequence name is `customer_customer_id_seq`.
  - The column will have a `NOT NULL` constraint.
  - The column will have a default value generated by the `nextval()` function.
  - The sequence will be owned by the column, which means that the sequence will be dropped automatically if the column is dropped.

# Serial Types and Identity Columns

- Serial types and identity columns are used to define surrogate keys.
- The synopsis for defining a column as an identity column is as follows:
  - `GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ ( sequence_options ) ]`

- It's better to use identity columns, which were introduced in PostgreSQL 10, rather than serial, because this overcomes some of the serial type's limitations:
  - **Compatibility**: The identity column adheres to SQL standards, this makes it easier to migrate PostgreSQL to other relational databases and vice versa.
  - **Permissions**: The permissions of the sequence object that was created by using the serial column is managed separately. Often, developers tend to forget to assign proper permissions to the sequence object when changing the permissions of the table that contains the defined serial type.
  - **Sequence value and user data precedence**: The serial type uses default constrain to assign the value of the column. That means you can override the default values. The identity column can control this behavior. The `BY DEFAULT` option allows the user to insert data into the column. If `ALWAYS` is specified, the user value won't be accepted unless the `INSERT` statement specifies `OVERRIDING SYSTEM VALUE`. Note that this setting is ignored by the `COPY` statement.
  - **Managing table structure**: It's easier, from a syntactical point view, to manage the identity column. It's also easier to alter existing columns' default values if they're defined as an identity.

- Both identity columns and serial types use sequence objects behind the scenes.
- In most cases, it's straightforward to replace a serial type with an identity column, because both behave similarly.

- Create a table of the `SERIAL` type and perform an `INSERT` operation:

```
postgres=# CREATE TABLE test_serial (id SERIAL PRIMARY KEY, payload text);
CREATE TABLE
postgres=# INSERT INTO test_serial (payload) SELECT 'a' RETURNING *;
 id | payload
----+---------
  1 | a
(1 row)

INSERT 0 1
postgres=# INSERT INTO test_serial (id, payload) SELECT 2, 'a' RETURNING *;
 id | payload
----+---------
  2 | a
(1 row)

INSERT 0 1
postgres=# INSERT INTO test_serial (payload) SELECT 'a' RETURNING *;
ERROR:  duplicate key value violates unique constraint "test_serial_pkey"
DETAIL:  Key (id)=(2) already exists.
```

- Create a table with `IDENTITY` and perform an `INSERT` operation:

```
postgres=# CREATE TABLE test_identity ( id INTEGER generated by default as identity PRIMARY KEY, payload text);
CREATE TABLE
postgres=# INSERT INTO test_identity (payload) SELECT 'a' RETURNING *;
 id | payload
----+---------
  1 | a
(1 row)

INSERT 0 1
postgres=# INSERT INTO test_identity (id, payload) SELECT 1, 'a' RETURNING *;
ERROR:  duplicate key value violates unique constraint "test_identity_pkey"
DETAIL:  Key (id)=(1) already exists.
postgres=# INSERT INTO test_identity (id, payload) SELECT 100000, 'a' RETURNING *;
   id   | payload
--------+---------
 100000 | a
(1 row)

INSERT 0 1
```

- `IDENTITY` and `SERIAL` internal implementation depends on sequence:

```
postgres=# \ds
                    List of relations
 Schema |           Name           |   Type   |  Owner
--------+--------------------------+----------+----------
 public | customer_customer_id_seq | sequence | postgres
 public | test_identity_id_seq     | sequence | postgres
 public | test_serial_id_seq       | sequence | postgres
(3 rows)
```

- Create an `IDENTITY` column with the `ALWAYS` option:

```
postgres=# CREATE TABLE test_identity2 ( id INTEGER generated always as identity PRIMARY KEY, payload text);
CREATE TABLE
postgres=# INSERT INTO test_identity2 (id, payload) SELECT 1, 'a' RETURNING *;
ERROR:  cannot insert into column "id"
DETAIL:  Column "id" is an identity column defined as GENERATED ALWAYS.
HINT:  Use OVERRIDING SYSTEM VALUE to override.
```

# Character Types

| Name | Comments | Trailing spaces | Maximum length |
|---|---|---|---|
| `char` | Equivalent to `char(1)`, it must be quoted as shown in the name. | Semantically insignificant | 1 |
| `name` | Equivalent to `varchar(64)`. Used by Postgres for object names. | Semantically significant | 64 |
| `char(n)` | Alias: `character(n)`. Fixed-length character where the length is `n`. Internally called **blank padded character (bpchar)**. | Semantically insignificant | 1 to 10485760 |
| `varchar(n)` | Alias: character `varying(n)`. Variable-length character where the maximum length is `n`. | Semantically significant | 1 to 10485760 |
| `text` | Variable-length character. | Semantically significant | Unlimited |

- PostgreSQL provides two general text types—the `char(n)` and `varchar(n)` data types—where `n` is the number of characters allowed.
- In the `char` data type, if a value is less than the specified length, trailing spaces are padded at the end of the value.
- Operations on the `char` data types ignore the trailing spaces.

```
postgres=# SELECT 'a'::CHAR(2) = 'a '::CHAR(3) AS "Trailing space is ignored" ,length('a'::CHAR(10));
 Trailing space is ignored | length
---------------------------+--------
 t                         |      1
(1 row)
```

- For both the char and varchar data types, if the string is longer than the maximum allowed length, an error will be raised in the case of INSERT or UPDATE unless the extra characters are all spaces.
  - In the latter case, the string will be truncated.
- In the case of casting, extra characters will be truncated automatically without raising an error.

- The following example shows how mixing different data types might cause problems:

```
postgres=# SELECT
postgres-# 'a '::VARCHAR(2) = 'a '::TEXT AS "Text and varchar",
postgres-# 'a '::CHAR(2) = 'a '::TEXT AS "Char and text",
postgres-# 'a '::CHAR(2) = 'a '::VARCHAR(2) AS "Char and varchar";
 Text and varchar | Char and text | Char and varchar
------------------+---------------+------------------
 t                | f             | t
(1 row)

postgres=# SELECT length ('a '::CHAR(2)), length ('a '::VARCHAR(2));
 length | length
--------+--------
      1 |      2
(1 row)
```

- The preceding example shows that `'a '::CHAR(2)` equals `'a '::VARCHAR(2)`, but both have different lengths, which isn't logical.
- Also, it shows that `'a'::CHAR(2)` isn't equal to `'a '::text`.
- Finally, `'a '::VARCHAR(2)` equals `'a'::text`.
- The preceding example causes confusion because if variable `a` is equal to `b`, and `b` is equal to `c`, `a` should be equal to `c` according to mathematics.

- The `text` data type can be considered an unlimited `varchar()` type.
- The maximum text size that can be stored is 1 GB, which is the maximum column size.

- The following code shows the storage consumption for the character and character varying data types.

```
postgres=# CREATE TABLE char_size_test (
postgres(#  size CHAR(10)
postgres(# );
CREATE TABLE
postgres=# CREATE TABLE varchar_size_test(
postgres(#  size varchar(10)
postgres(# );
CREATE TABLE
postgres=# WITH
postgres-# test_data AS (
postgres(#  SELECT substring(md5(random()::text), 1, 5)
postgres(#  FROM generate_series (1, 1000000)),
postgres-# char_data_insert AS (
postgres(#  INSERT INTO char_size_test SELECT * FROM test_data)
postgres-# INSERT INTO varchar_size_test SELECT * FROM test_data;
INSERT 0 1000000
```

- Use this code to get the table size:

```
postgres=# \dt+ varchar_size_test
                         List of relations
 Schema |       Name        | Type  |  Owner   | Size  | Description
--------+-------------------+-------+----------+-------+-------------
 public | varchar_size_test | table | postgres | 35 MB |
(1 row)

postgres=# \dt+ char_size_test
                       List of relations
 Schema |      Name      | Type  |  Owner   | Size  | Description
--------+----------------+-------+----------+-------+-------------
 public | char_size_test | table | postgres | 42 MB |
(1 row)
```

- The `varchar` data type can be emulated by the `text` data type and a check constraint to check the text length.
- For example, the following code snippets are semantically equivalent:
  - ```
    CREATE TABLE emulate_varchar(
     test VARCHAR(4)
    );
    ```
  - ```
    --semantically equivalent to
    CREATE TABLE emulate_varchar (
     test TEXT,
     CONSTRAINT test_length CHECK (length(test) <= 4)
    );
    ```

- In PostgreSQL, there's no difference in performance between the different character types, so it's recommended you use the `text` data type.
- This allows the developer to react quickly to the changes in business requirements.
- For example, one common business case is changing the text length, such as changing the length of a customer ticket number from six to eight characters due to length limitation, or changing how certain information is stored in the database.
  - In such a scenario, if the data type is `text`, this could be done by amending the check constraint without altering the table structure.

# Date and Time Types

| Name | Size in bytes | Description | Low value | High value |
|---|---|---|---|---|
| Timestamp without time zone | 8 | Date and time without time zone, equivalent to timestamp | 4713 BC | 294276 AD |
| Timestamp with time zone | 8 | Date and time with time zone, equivalent to timestamptz | 4713 BC | 294276 AD |
| Date | 4 | Date only | 4713 BC | 294276 AD |
| Time without time zone | 8 | Time of day | 00:00:00 | 24:00:00 |
| Time with time zone | 12 | Time of day with time zone | 00:00:00+1459 | 24:00:00-1459 |
| Interval | 16 | Time interval | -178,000,000 years | +178,000,000 years |

- PostgreSQL stores the timestamp with and without the time zone in the **Coordinated Universal Time (UTC)** format, and only the time is stored without the time zone.
- This explains the identical storage size for both the timestamp with the time zone and the timestamp without the time zone.

- There are two approaches for handling the timestamp correctly.
- The first approach is to use the timestamp without the time zone, and let the client side handle the time zone differences.
- This is useful for in-house development, applications with only one-time zone, and when the clients know the time zone differences.

- The other approach is to use the timestamp with the time zone.
- In PostgreSQL, this is given the `timestamptz` extension.
- The following are some of the best practices to avoid the common pitfalls when using `timestamptz`:
  - Make sure to set the default time zone for all connections. This is done by setting the time zone configuration in the `postgresql.conf` file. Since PostgreSQL stores the timestamp with the time zone in UTC format internally, it's a good practice to set the default connection to UTC as well. Also, UTC helps us to overcome the potential problems due to **Daylight Savings Time (DST)**.
  - The time zone should be specified in each `CRUD` operation.
  - Don't perform operations on the timestamp without time zone and the timestamp with time zone, this will normally lead to the wrong results due to implicit `conversion`.
  - Don't invent your own conversion; instead, use the database server to convert between the different time zones.
  - Investigate the data types of high-level languages to determine which type could be used with PostgreSQL without extra handling.

- PostgreSQL has two important settings: `timezone` and `DATESTYLE`.
- `DATESTYLE` has two purposes:
  - **Setting the display format**: `DATESTYLE` specifies the timestamp and `timestamptz` rendering style
  - **Interpreting ambiguous data**: DATESTYLE specifies how to interpret timestamp and `timestamptz`

- The `pg_timezone_names` and `pg_timezone_abbrevs` views provide a list of the time zone names and abbreviations, respectively.
- They also provide information regarding the time offset from UTC, and whether the time zone observes DST.

```
postgres=# select *
postgres-# from   pg_timezone_names
postgres-# where  name like 'Asia/Jerusalem';
      name      | abbrev | utc_offset | is_dst
----------------+--------+------------+--------
 Asia/Jerusalem | IST    | 02:00:00   | f
(1 row)
```

- For example, the following code snippet sets the `timezone` setting to Jerusalem, and then retrieves the local date and time in Jerusalem:

```
postgres=# SET timezone TO 'Asia/Jerusalem';
SET
postgres=# SELECT now();
              now
-------------------------------
 2019-02-22 07:18:25.002316+02
(1 row)
```

- The PostgreSQL `AT TIME ZONE` statement converts the timestamp with or without the `timezone` to a specified time zone; its behavior depends on the converted type.

```
postgres=# SHOW timezone;
    TimeZone
----------------
 Asia/Jerusalem
(1 row)

postgres=# \x
Expanded display is on.
postgres=# SELECT
now() AS "Return current timestap in Jerusalem",
now()::timestamp AS "Return current timestap in Jerusalem without time zone",
now() AT TIME ZONE 'CST' AS "Return current time in Central Standard Time without time zone",
'2019-02-22:00:00:00'::timestamp AT TIME ZONE 'CST' AS "Convert the time in CST to Jerusalem time zone";
-[ RECORD 1 ]--------------------------------------------------+------------------------------
Return current timestap in Jerusalem                            | 2019-02-22 07:25:30.573308+02
Return current timestap in Jerusalem without time zone          | 2019-02-22 07:25:30.573308
Return current time in Central Standard Time without time zone | 2019-02-21 23:25:30.573308
Convert the time in CST to Jerusalem time zone                  | 2019-02-22 08:00:00+02
```

- The date is recommended when there is no need to specify the time, such as birthdays, holidays, and absence days.
- Time with time-zone storage is 12 bytes: 8 bytes are used to store the time, and 4 bytes are used to store the time zone.
- The time without a time zone consumes only 8 bytes.
- Conversions between time zones can be made using the `AT TIME ZONE` construct.

- Finally, the interval data type is very important in handling timestamp operations, as well as describing some business cases.
- From the point of view of functional requirements, the interval data type can represent a period of time, such as estimation time for the completion of a certain task.

- The following example shows `timestamptz` and date subtraction.

```
postgres=# SELECT
postgres-# '2014-10-11'::date - '2014-10-10'::date = 1 AS "date Subtraction",
postgres-# '2014-09-01 23:30:00'::timestamptz - '2014-09-01 22:00:00'::timestamptz = Interval '1 hour, 30 minutes' AS "Time stamp subtraction";
-[ RECORD 1 ]----------+--
date Subtraction       | t
Time stamp subtraction | t
```