Creating Databases and Tables

SQL

- We’ve focused on querying and reading data from existing databases and tables.
- Let’s now shift our focus to creating our own databases and tables.

SQL

- Section Overview
    - Data Types
    - Primary and Foreign Keys
    - Constraints
    - CREATE
    - INSERT
    - UPDATE
    - DELETE, ALTER, DROP

SQL

- We first focus on learning a few theoretical concepts, such as choosing the correct data type for a stored value and setting possible constraints on it.
- We will also learn about primary and foreign keys.

Data Types

SQL

- We’ve already encountered a variety of data types, let’s quickly review the main data types in SQL.

SQL

- Boolean
  - True or False
- Character
  - char, varchar, and text
- Numeric
  - integer and floating-point number
- Temporal
  - date, time, timestamp, and interval

SQL

- UUID
    - Universally Unique Identifiers
- Array
    - Stores an array of strings, numbers, etc.
- JSON
- Hstore key-value pair
- Special types such as network address and geometric data.

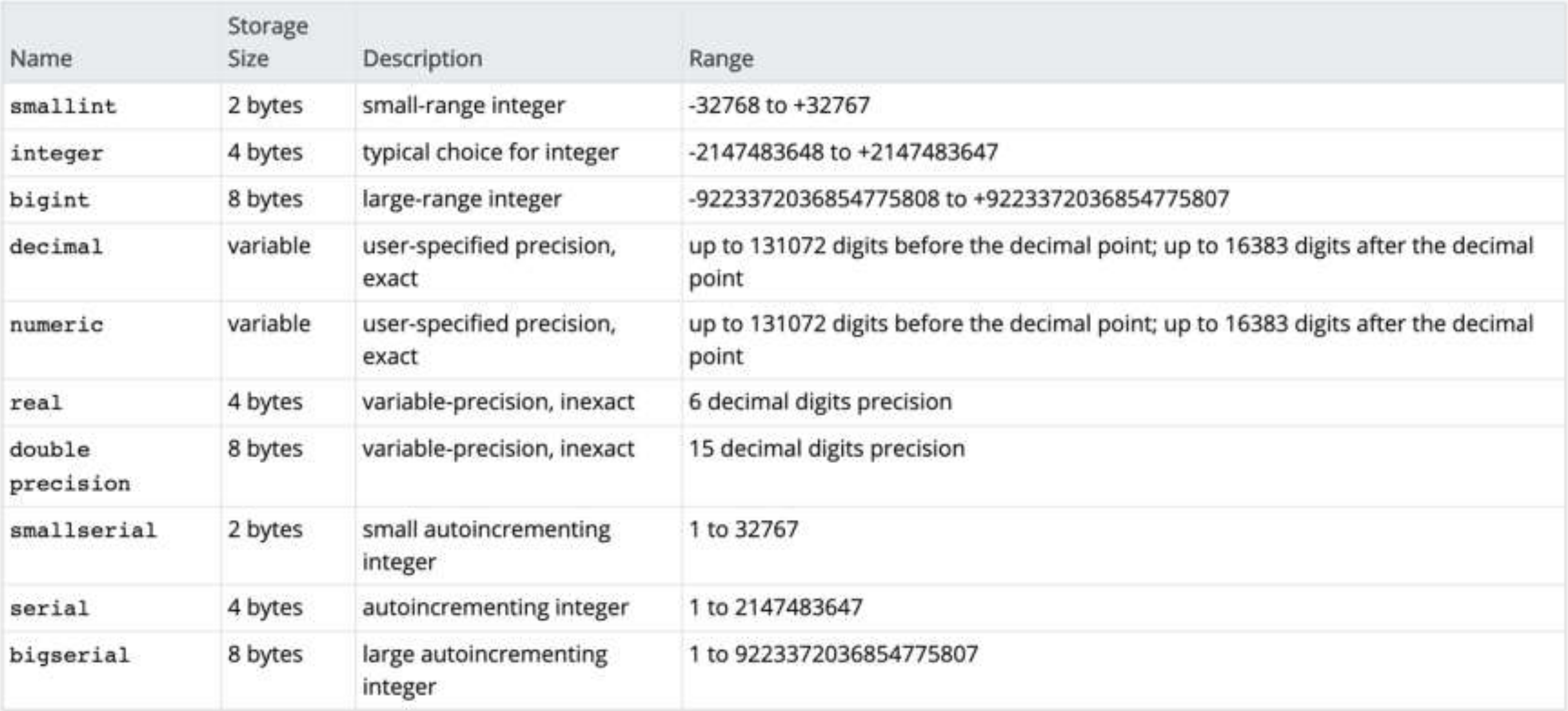
SQL

- When creating databases and tables, you should carefully consider which data types should be used for the data to be stored.
- Review the documentation to see limitations of data types:
- **postgresql.org/docs/current/datatype.html**

SQL

- For example
  - Imagine we want to store a phone number, should it be stored as numeric?
  - If so, which type of numeric?
- We could take a look at the documentation for options...

SQL



Name	Storage Size	Description	Range
`smallint`	2 bytes	small-range integer	-32768 to +32767
`integer`	4 bytes	typical choice for integer	-2147483648 to +2147483647
`bigint`	8 bytes	large-range integer	-9223372036854775808 to +9223372036854775807
`decimal`	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
`numeric`	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
`real`	4 bytes	variable-precision, inexact	6 decimal digits precision
`double precision`	8 bytes	variable-precision, inexact	15 decimal digits precision
`smallserial`	2 bytes	small autoincrementing integer	1 to 32767
`serial`	4 bytes	autoincrementing integer	1 to 2147483647
`bigserial`	8 bytes	large autoincrementing integer	1 to 9223372036854775807

SQL

- Based on the limitations, you may think it makes sense to store it as a **BIGINT** data type, but we should really be thinking what is best for the situation.
- Why bother with numerics at all?
- We don’t perform arithmetic with numbers, so it probably makes more sense as a **VARCHAR** data type instead.

SQL

- In fact, searching for best practice online, you will discover its usually recommended to store as a text based data type due to a variety of issues
    - No arithmetic performed
    - Leading zeros could cause issues, 7 and 07 treated same numerically, but are not the same phone number

SQL

- When creating a database and table, take your time to plan for long term storage
- Remember you can always remove historical information you’ve decided you aren’t using, but you can’t go back in time to add in information!

Primary and Foreign Keys

SQL

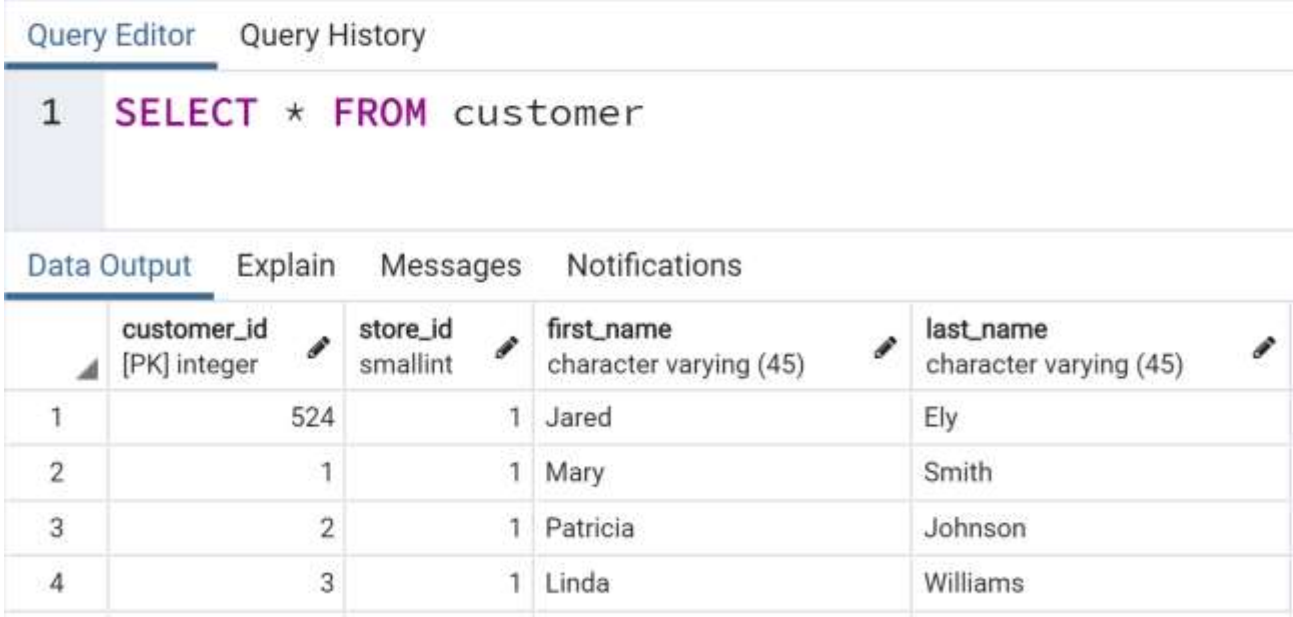
- A primary key is a column or a group of columns used to identify a row uniquely in a table.
- For example, in our dvdrental database we saw customers had a unique, non-null customer_id column as their primary key.

SQL

- Primary keys are also important since they allow us to easily discern what columns should be used for joining tables together.

SQL

- Example of Primary Key

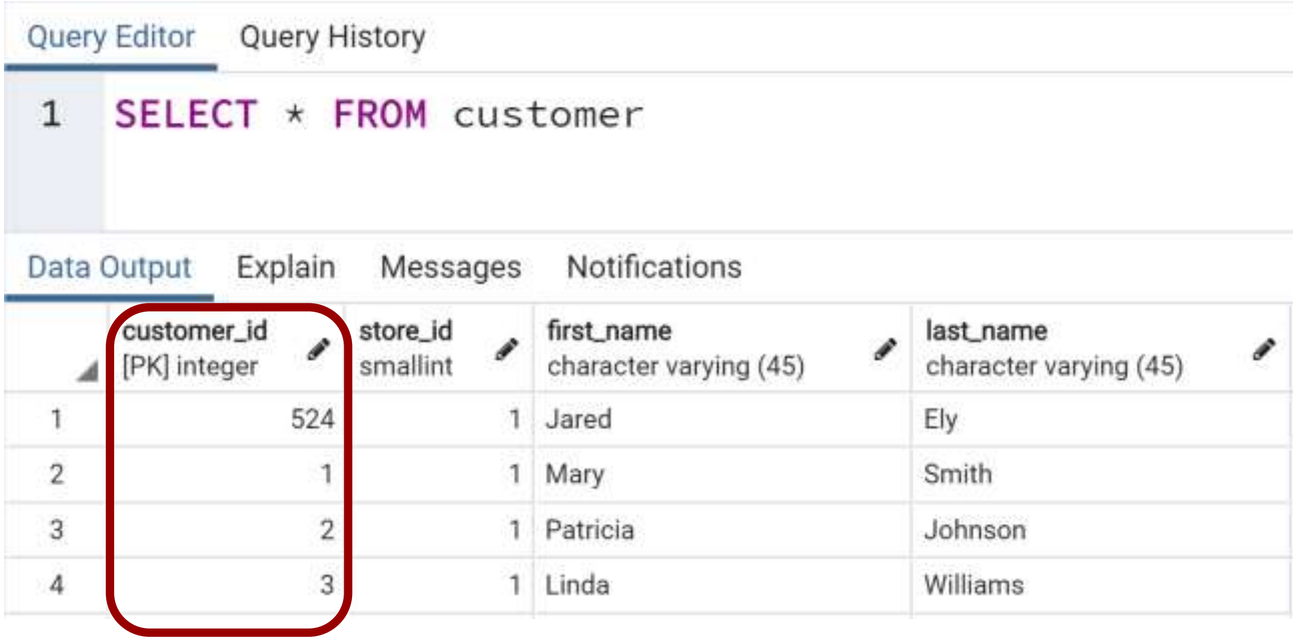


```sql
SELECT * FROM customer
```

	customer_id [PK] integer	store_id smallint	first_name character varying (45)	last_name character varying (45)
1	524	1	Jared	Ely
2	1	1	Mary	Smith
3	2	1	Patricia	Johnson
4	3	1	Linda	Williams

SQL

- Example of Primary Key

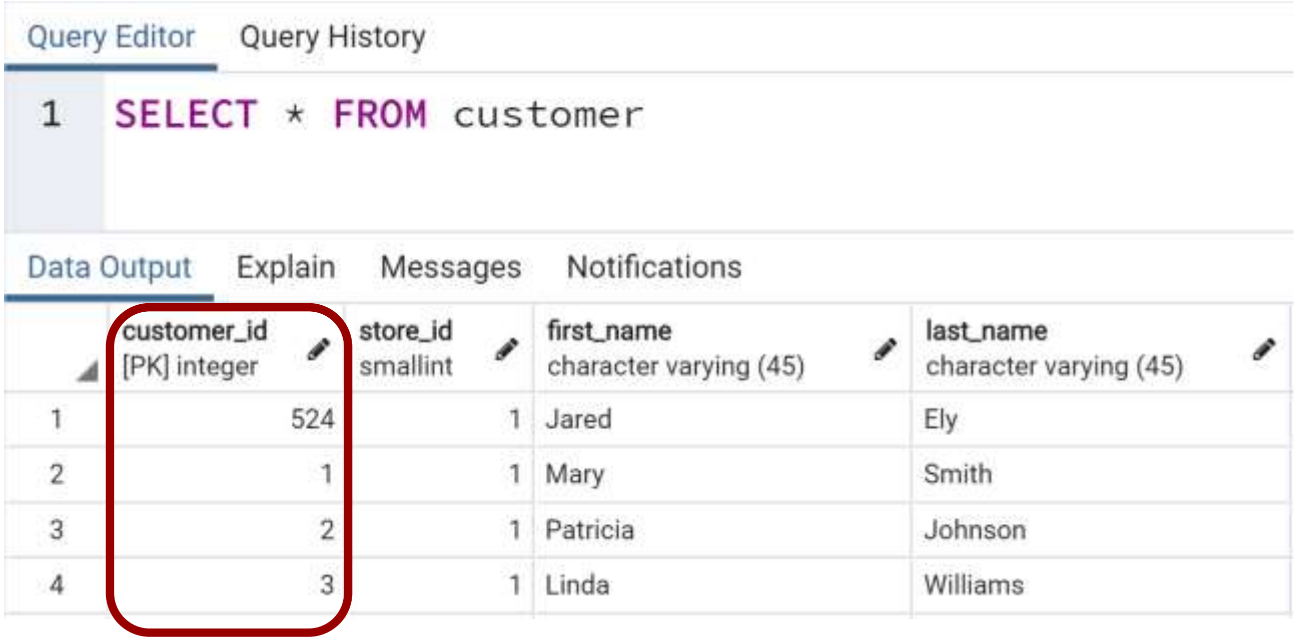


```
SELECT * FROM customer
```

	customer_id [PK] integer	store_id smallint	first_name character varying (45)	last_name character varying (45)
1	524	1	Jared	Ely
2	1	1	Mary	Smith
3	2	1	Patricia	Johnson
4	3	1	Linda	Williams

SQL

- Notice its integer based and unique

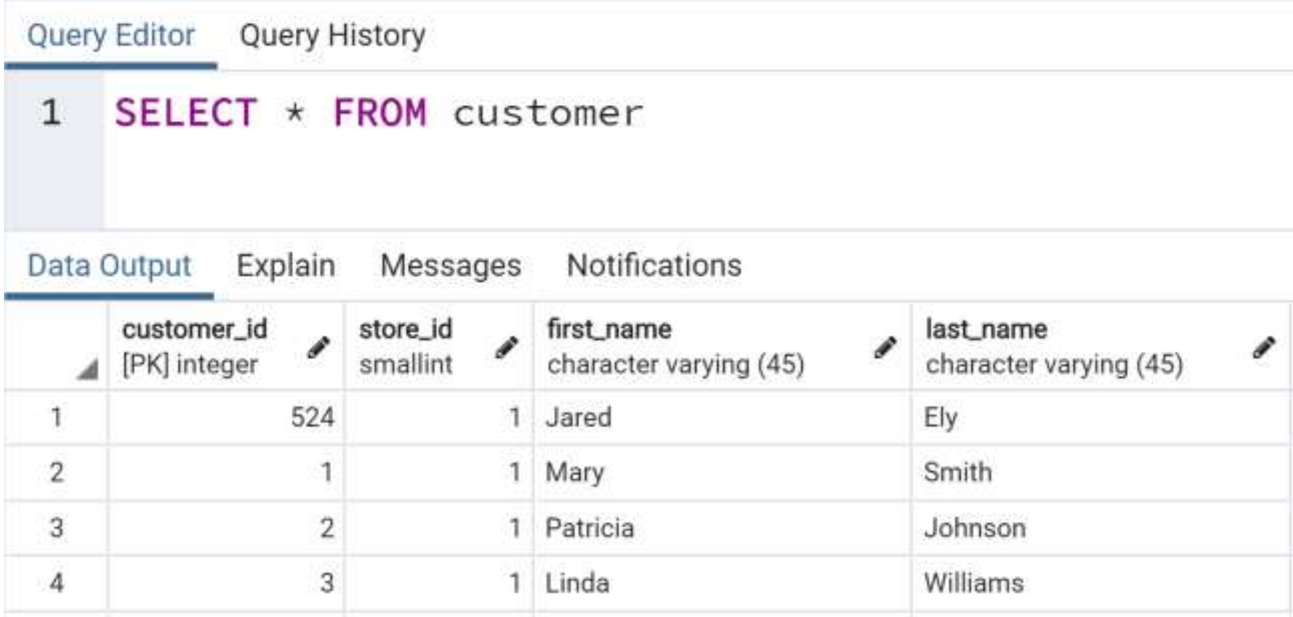


```
SELECT * FROM customer
```

	customer_id [PK] integer	store_id smallint	first_name character varying (45)	last_name character varying (45)
1	524	1	Jared	Ely
2	1	1	Mary	Smith
3	2	1	Patricia	Johnson
4	3	1	Linda	Williams

SQL

- Later we will learn about SERIAL data type



```
SELECT * FROM customer
```

	customer_id [PK] integer	store_id smallint	first_name character varying (45)	last_name character varying (45)
1	524	1	Jared	Ely
2	1	1	Mary	Smith
3	2	1	Patricia	Johnson
4	3	1	Linda	Williams

SQL

- A foreign key is a field or group of fields in a table that uniquely identifies a row in another table.
- A foreign key is defined in a table that references to the primary key of the other table.

SQL

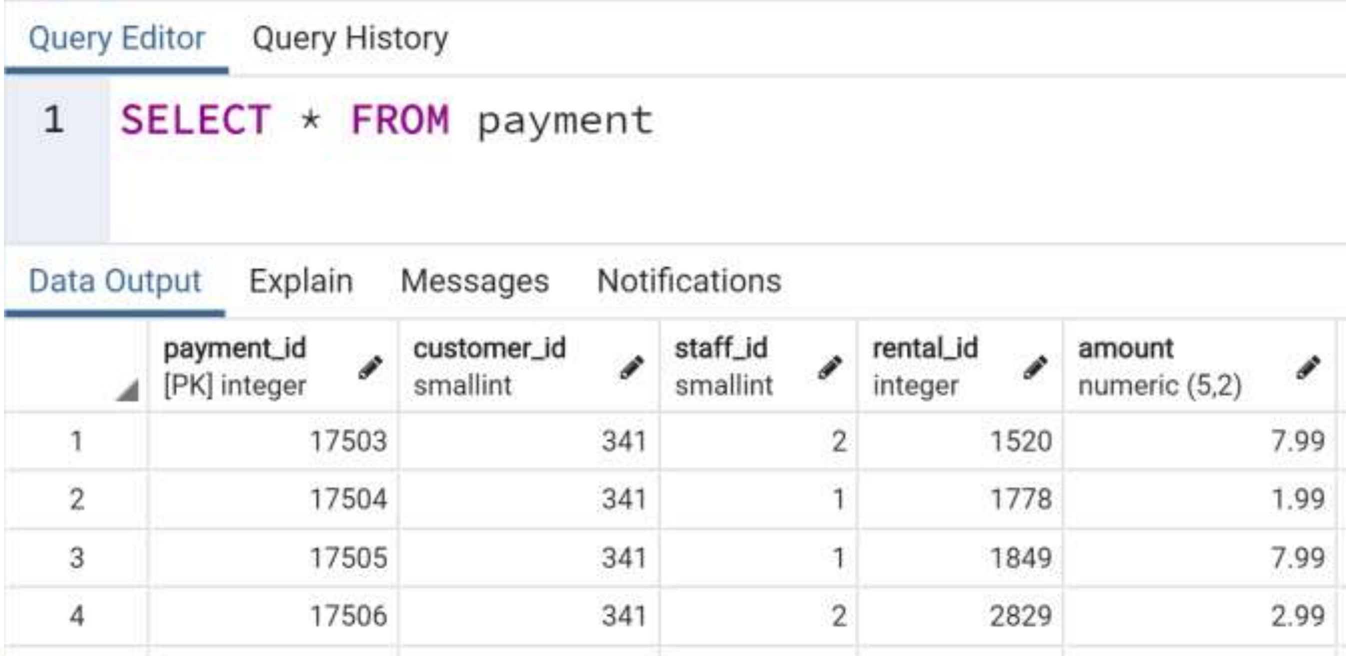
- The table that contains the foreign key is called referencing table or child table.
- The table to which the foreign key references is called referenced table or parent table.
- A table can have multiple foreign keys depending on its relationships with other tables.

SQL

- Recall in the dvdrental database payment table, each payment row had its unique payment_id (a primary key) and identified the customer that made the payment through the customer_id (a foreign key since it references the customer table's primary key)

SQL

- Example

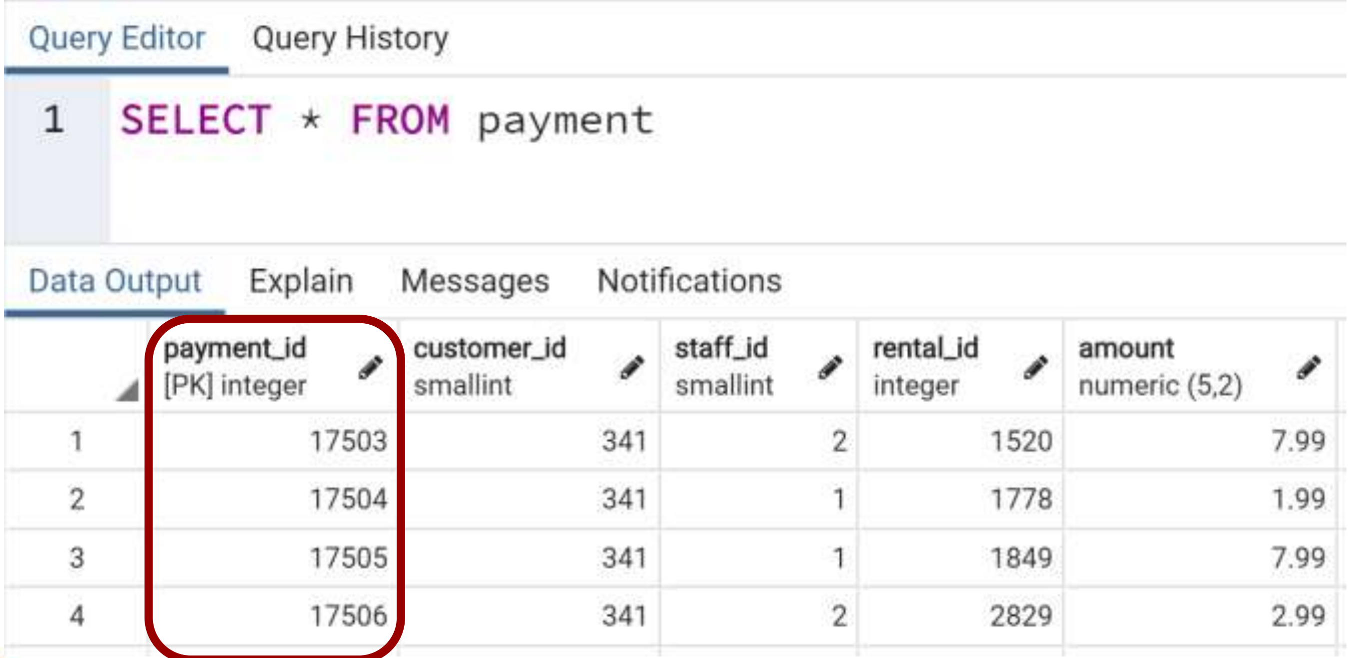


```
SELECT * FROM payment
```

	payment_id [PK] integer	customer_id smallint	staff_id smallint	rental_id integer	amount numeric (5,2)
1	17503	341	2	1520	7.99
2	17504	341	1	1778	1.99
3	17505	341	1	1849	7.99
4	17506	341	2	2829	2.99

SQL

- Primary Key for Payment Table

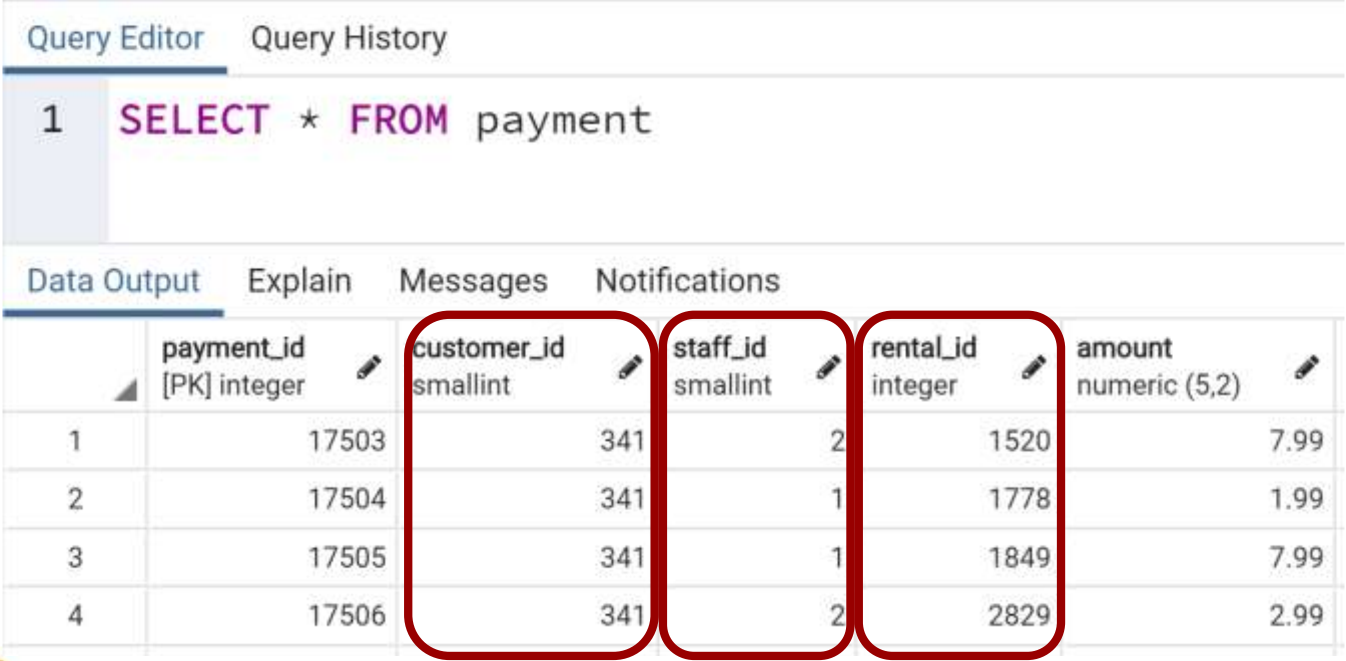


```
SELECT * FROM payment
```

	payment_id [PK] integer	customer_id smallint	staff_id smallint	rental_id integer	amount numeric (5,2)
1	17503	341	2	1520	7.99
2	17504	341	1	1778	1.99
3	17505	341	1	1849	7.99
4	17506	341	2	2829	2.99

SQL

- Multiple Foreign Key References

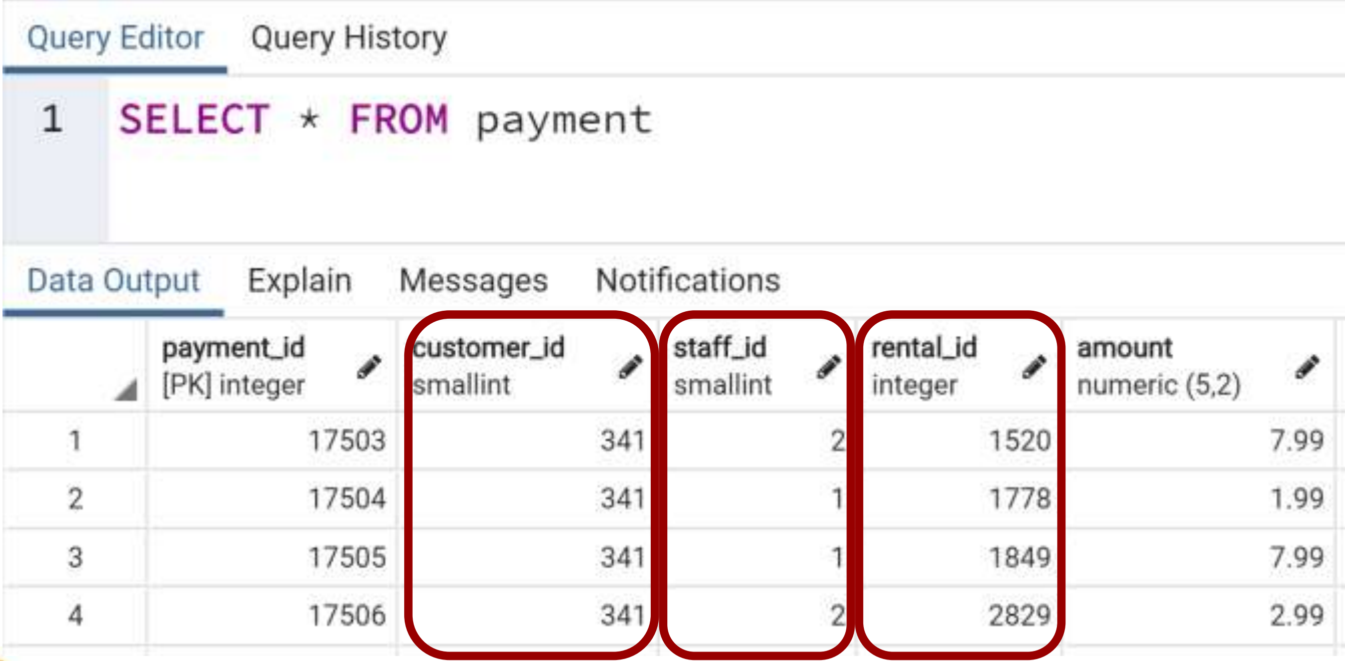


```
1  SELECT * FROM payment
```

	payment_id	customer_id	staff_id	rental_id	amount
	[PK] integer	smallint	smallint	integer	numeric (5,2)
1	17503	341	2	1520	7.99
2	17504	341	1	1778	1.99
3	17505	341	1	1849	7.99
4	17506	341	2	2829	2.99

SQL

- Note pgAdmin won’t alert you to FK



```
SELECT * FROM payment
```

	payment_id [PK] integer	customer_id smallint	staff_id smallint	rental_id integer	amount numeric (5,2)
1	17503	341	2	1520	7.99
2	17504	341	1	1778	1.99
3	17505	341	1	1849	7.99
4	17506	341	2	2829	2.99

SQL

- You may begin to realize primary key and foreign key typically make good column choices for joining together two or more tables.

SQL

- When creating tables and defining columns, we can use constraints to define columns as being a primary key, or attaching a foreign key relationship to another table.
- Let’s quickly explore table properties in pgAdmin to see how to get information on primary and foreign keys!

Constraints

SQL

- Constraints are the rules enforced on data columns on table.
- These are used to prevent invalid data from being entered into the database.
- This ensures the accuracy and reliability of the data in the database.

SQL

- Constraints can be divided into two main categories:
    - Column Constraints
        - Constrains the data in a column to adhere to certain conditions.
    - Table Constraints
        - applied to the entire table rather than to an individual column.

SQL

- The most common constraints used:
  - **NOT NULL** Constraint
    - Ensures that a column cannot have NULL value.
  - **UNIQUE** Constraint
    - Ensures that all values in a column are different.

SQL

- The most common constraints used:
    - PRIMARY Key
        - Uniquely identifies each row/record in a database table.
    - FOREIGN Key
        - Constrains data based on columns in other tables.

SQL

- The most common constraints used:
  - CHECK Constraint
    - Ensures that all values in a column satisfy certain conditions.

SQL

- The most common constraints used:
  - **EXCLUSION** Constraint
    - Ensures that if any two rows are compared on the specified column or expression using the specified operator, not all of these comparisons will return TRUE.

SQL

- Table Constraints
  - CHECK (condition)
    - to check a condition when inserting or updating data.
  - REFERENCES
    - to constrain the value stored in the column that must exist in a column in another table.

SQL

- Table Constraints
    - UNIQUE (column_list)
        - forces the values stored in the columns listed inside the parentheses to be unique.
    - PRIMARY KEY(column_list)
        - Allows you to define the primary key that consists of multiple columns.

SQL

- Now that we understand data types, primary keys, foreign keys, and constraints we are ready to begin using SQL syntax to create tables!

CREATE

SQL

- Let’s now learn the syntax to create a table in SQL using the CREATE keyword and column syntax.

SQL

- Full General Syntax

  -
    ```
    CREATE TABLE table_name (
      column_name TYPE column_constraint,
      column_name TYPE column_constraint,
       table_constraint table_constraint
    ) INHERITS existing_table_name;
    ```

SQL

- Full General Syntax

  -
    ```
    CREATE TABLE table_name (
        column_name TYPE column_constraint,
        column_name TYPE column_constraint,
        table_constraint table_constraint
    ) INHERITS existing_table_name;
    ```

SQL

- Full General Syntax

  -
    ```
    CREATE TABLE table_name (
      column_name TYPE column_constraint,
      column_name TYPE column_constraint,
       table_constraint table_constraint
    ) INHERITS existing_table_name;
    ```

SQL

- Full General Syntax
  - CREATE TABLE table_name (

    ```
    column_name TYPE column_constraint,
    column_name TYPE column_constraint,
    table_constraint table_constraint
    ) INHERITS existing_table_name;
    ```

SQL

- Full General Syntax
    - CREATE TABLE table_name (

      ```
      column_name TYPE column_constraint,
      column_name TYPE column_constraint,
      table_constraint table_constraint
      ) INHERITS existing_table_name;
      ```

SQL

- Full General Syntax

  -
    ```
    CREATE TABLE table_name (
      column_name TYPE column_constraint,
      column_name TYPE column_constraint,
       table_constraint table_constraint
    ) INHERITS existing_table_name;
    ```

SQL

- Full General Syntax

  -
    ```
    CREATE TABLE table_name (
     column_name TYPE column_constraint,
     column_name TYPE column_constraint,
      table_constraint table_constraint
    ) INHERITS existing_table_name;
    ```

SQL

- Common Simple Syntax
  - CREATE TABLE table_name (
    column_name TYPE column_constraint,
    column_name TYPE column_constraint,
    );

SQL

- Example Syntax
  - CREATE TABLE table_name (
    column_name TYPE column_constraint,
    column_name TYPE column_constraint,
    );

SQL

- Example Syntax

  -
    ```
    CREATE TABLE players(
        column_name TYPE column_constraint,
        column_name TYPE column_constraint,
     );
    ```

SQL

- Example Syntax
    - CREATE TABLE players(

```
column_name TYPE column_constraint,
column_name TYPE column_constraint,
);
```

SQL

- Example Syntax
  - CREATE TABLE players(

    column_name TYPE column_constraint,

    column_name TYPE column_constraint,

    );

SQL

- Example Syntax
    - CREATE TABLE players(
      player_id TYPE column_constraint,
      column_name TYPE column_constraint,
      );

SQL

- Example Syntax
  - CREATE TABLE players(
    player_id TYPE column_constraint,
    column_name TYPE column_constraint,
    );

SQL

- Example Syntax
  - CREATE TABLE players(

    player_id SERIAL column_constraint,

    column_name TYPE column_constraint,

    );

SQL

- SERIAL
  - In PostgreSQL, a sequence is a special kind of database object that generates a sequence of integers.
  - A sequence is often used as the primary key column in a table.

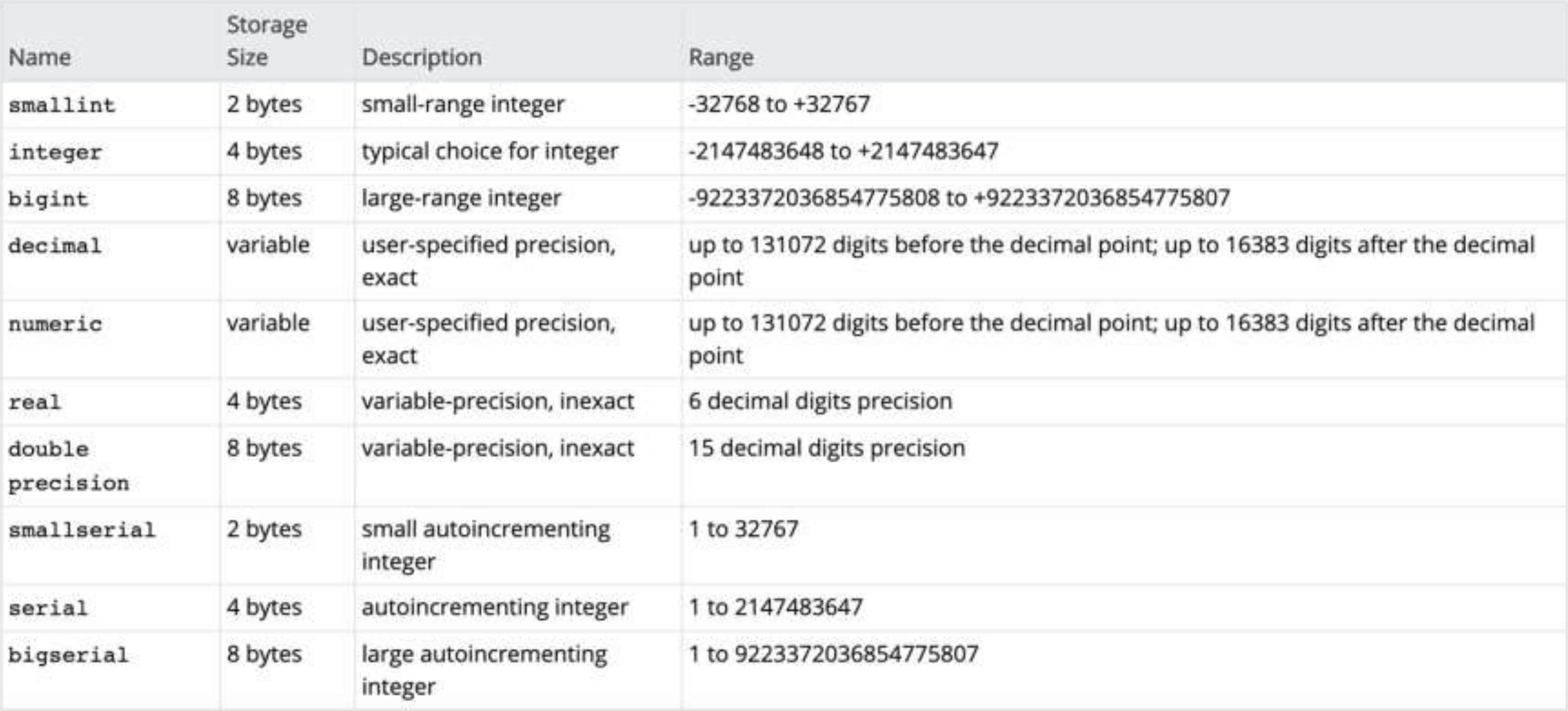
SQL

- SERIAL
  - It will create a sequence object and set the next value generated by the sequence as the default value for the column.
  - This is perfect for a primary key, because it logs unique integer entries for you automatically upon insertion.

SQL

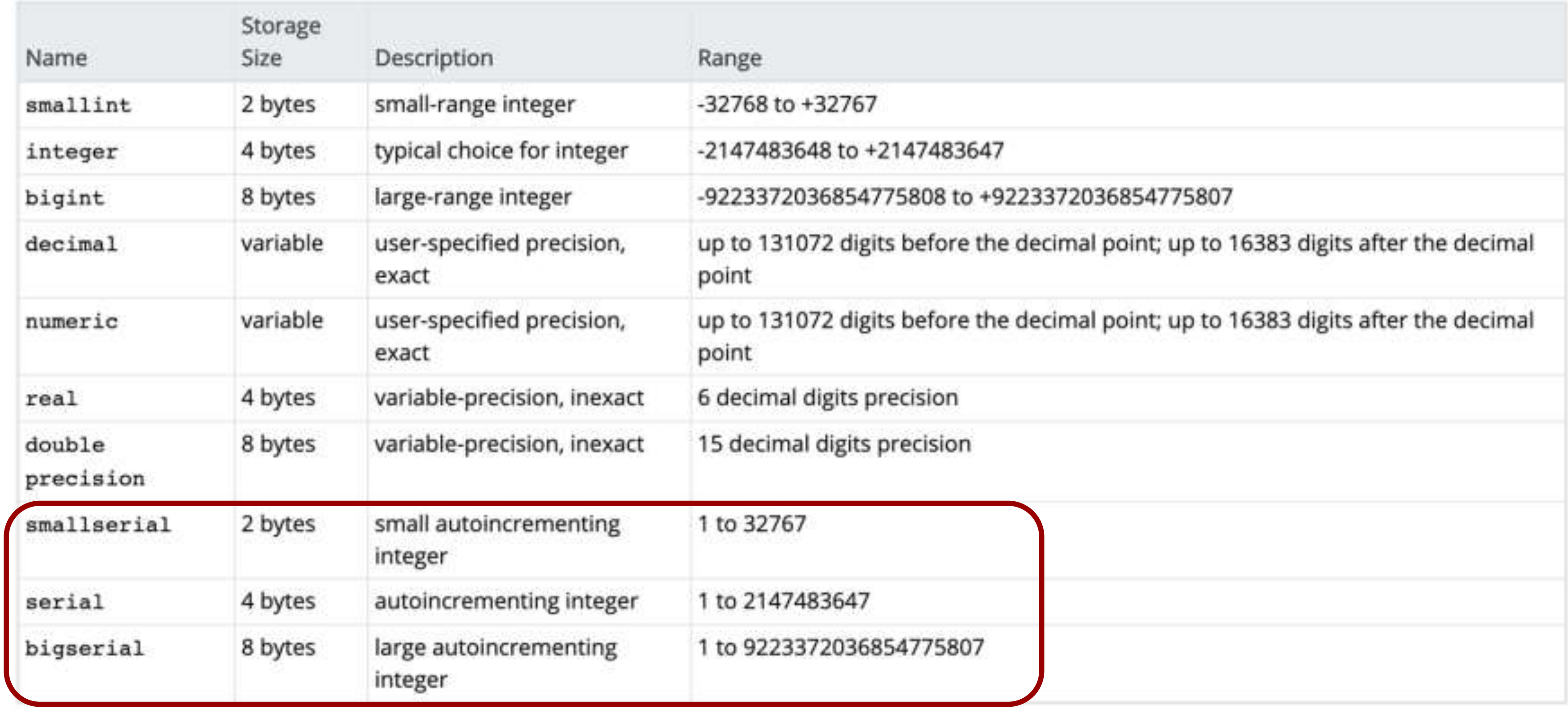
- SERIAL
  - If a row is later removed, the column with the SERIAL data type will <u>**not**</u> adjust, marking the fact that a row was removed from the sequence, for example
    - 1,2,3,5,6,7
      - You know row 4 was removed at some point

SQL



Name	Storage Size	Description	Range
`smallint`	2 bytes	small-range integer	-32768 to +32767
`integer`	4 bytes	typical choice for integer	-2147483648 to +2147483647
`bigint`	8 bytes	large-range integer	-9223372036854775808 to +9223372036854775807
`decimal`	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
`numeric`	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
`real`	4 bytes	variable-precision, inexact	6 decimal digits precision
`double precision`	8 bytes	variable-precision, inexact	15 decimal digits precision
`smallserial`	2 bytes	small autoincrementing integer	1 to 32767
`serial`	4 bytes	autoincrementing integer	1 to 2147483647
`bigserial`	8 bytes	large autoincrementing integer	1 to 9223372036854775807

SQL



Name	Storage Size	Description	Range
`smallint`	2 bytes	small-range integer	-32768 to +32767
`integer`	4 bytes	typical choice for integer	-2147483648 to +2147483647
`bigint`	8 bytes	large-range integer	-9223372036854775808 to +9223372036854775807
`decimal`	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
`numeric`	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
`real`	4 bytes	variable-precision, inexact	6 decimal digits precision
`double precision`	8 bytes	variable-precision, inexact	15 decimal digits precision
`smallserial`	2 bytes	small autoincrementing integer	1 to 32767
`serial`	4 bytes	autoincrementing integer	1 to 2147483647
`bigserial`	8 bytes	large autoincrementing integer	1 to 9223372036854775807

SQL

- Example Syntax

  -
    ```
    CREATE TABLE players(
      player_id SERIAL column_constraint,
      column_name TYPE column_constraint,
     );
    ```

- Example Syntax
  - CREATE TABLE players(

```
player_id SERIAL column_constraint,
column_name TYPE column_constraint,
);
```

SQL

- Example Syntax
  - CREATE TABLE players(

    player_id SERIAL PRIMARY KEY,

    column_name TYPE column_constraint,

    );

SQL

- Example Syntax

  -
    ```
    CREATE TABLE players(
      player_id SERIAL PRIMARY KEY,
      column_name TYPE column_constraint,
     );
    ```

SQL

- Example Syntax

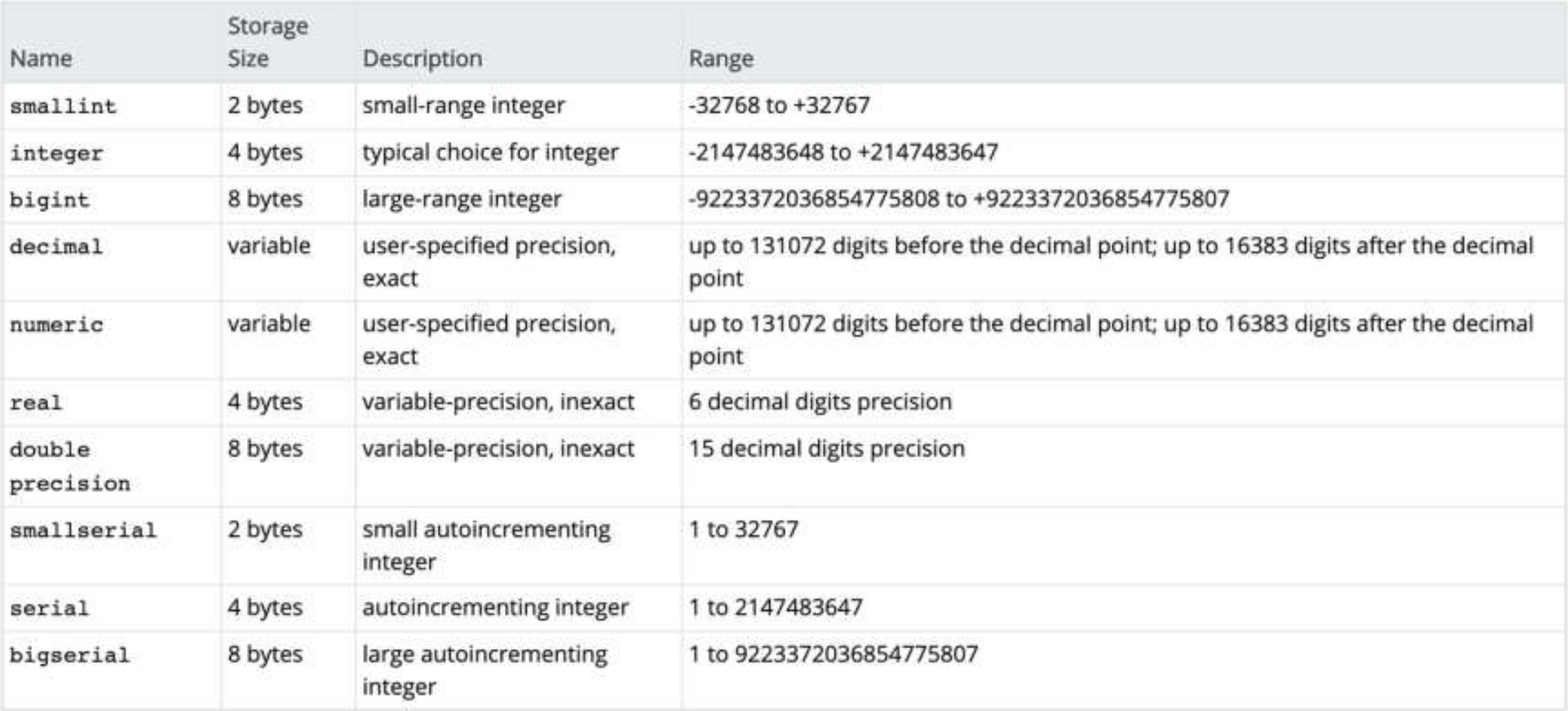
    -
      ```
      CREATE TABLE players(
        player_id SERIAL PRIMARY KEY,
        age TYPE column_constraint,
       );
      ```

SQL

- Example Syntax

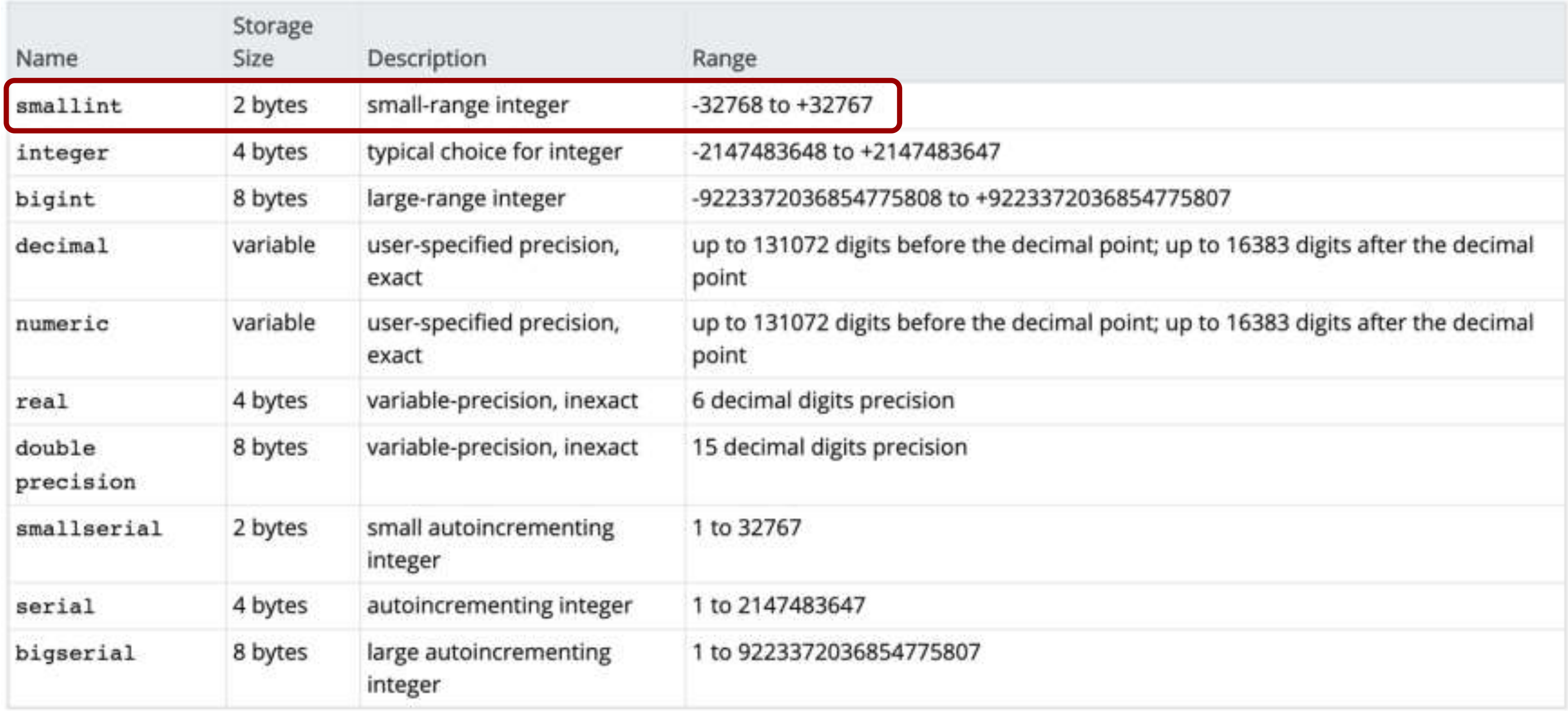
  -
    ```
    CREATE TABLE players(
      player_id SERIAL PRIMARY KEY,
      age TYPE column_constraint,
     );
    ```

SQL



Name	Storage Size	Description	Range
`smallint`	2 bytes	small-range integer	-32768 to +32767
`integer`	4 bytes	typical choice for integer	-2147483648 to +2147483647
`bigint`	8 bytes	large-range integer	-9223372036854775808 to +9223372036854775807
`decimal`	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
`numeric`	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
`real`	4 bytes	variable-precision, inexact	6 decimal digits precision
`double precision`	8 bytes	variable-precision, inexact	15 decimal digits precision
`smallserial`	2 bytes	small autoincrementing integer	1 to 32767
`serial`	4 bytes	autoincrementing integer	1 to 2147483647
`bigserial`	8 bytes	large autoincrementing integer	1 to 9223372036854775807

SQL



Name	Storage Size	Description	Range
`smallint`	2 bytes	small-range integer	-32768 to +32767
`integer`	4 bytes	typical choice for integer	-2147483648 to +2147483647
`bigint`	8 bytes	large-range integer	-9223372036854775808 to +9223372036854775807
`decimal`	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
`numeric`	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
`real`	4 bytes	variable-precision, inexact	6 decimal digits precision
`double precision`	8 bytes	variable-precision, inexact	15 decimal digits precision
`smallserial`	2 bytes	small autoincrementing integer	1 to 32767
`serial`	4 bytes	autoincrementing integer	1 to 2147483647
`bigserial`	8 bytes	large autoincrementing integer	1 to 9223372036854775807

SQL

- Example Syntax

  -
    ```
    CREATE TABLE players(
      player_id SERIAL PRIMARY KEY,
      age TYPE column_constraint
    );
    ```

SQL

- Example Syntax

  -
    ```
    CREATE TABLE players(
      player_id SERIAL PRIMARY KEY,
      age SMALLINT column_constraint
     );
    ```

SQL

- Example Syntax
  - CREATE TABLE players(

    player_id SERIAL PRIMARY KEY,

    age SMALLINT column_constraint

    );

SQL

- Example Syntax

  -
    ```
    CREATE TABLE players(
      player_id SERIAL PRIMARY KEY,
      age SMALLINT NOT NULL
     );
    ```

SQL

- Example Syntax
  - CREATE TABLE players(

    ```
    player_id SERIAL PRIMARY KEY,
    age SMALLINT NOT NULL
     );
    ```

SQL

- Let’s explore some examples in pgAdmin!

INSERT

SQL

- INSERT allows you to add in rows to a table.
- General Syntax
  - INSERT INTO table (column1, column2, …)
    VALUES
        (value1, value2, …),
        (value1, value2, …) ,…;

SQL

- INSERT allows you to add in rows to a table.
- Syntax for Inserting Values from another table:
  - INSERT INTO table(column1,column2,...)
    SELECT column1,column2,...
    FROM another_table
    WHERE condition;

SQL

- Keep in mind, the inserted row values must match up for the table, including constraints.
- SERIAL columns do not need to be provided a value.
- Let’s use INSERT in pgAdmin!

UPDATE

SQL

- The UPDATE keyword allows for the changing of values of the columns in a table.

SQL

- General Syntax

  -
    ```
    UPDATE table
    SET column1 = value1,
        column2 = value2 ,...
    WHERE
       condition;
    ```

SQL

- Example

  -
    ```
    UPDATE account
    SET last_login = CURRENT_TIMESTAMP
       WHERE last_login IS NULL;
    ```

SQL

- Reset everything without WHERE condition
  - UPDATE account
    SET last_login = CURRENT_TIMESTAMP

SQL

- Set based on another column
    - UPDATE account
      SET last_login = created_on

SQL

- Using another table’s values (UPDATE join)

  -
    ```
    UPDATE TableA
    SET original_col = TableB.new_col
    FROM tableB
    WHERE tableA.id = TableB.id
    ```

SQL

- Return affected rows
  - UPDATE account
    SET last_login = created_on
    RETURNING account_id,last_login

SQL

- Let’s explore this further in pgAdmin!

DELETE

SQL

- We can use the DELETE clause to remove rows from a table.
- For example:
    - DELETE FROM table
      WHERE row_id = 1

SQL

- We can delete rows based on their presence in other tables
- For example:
  - DELETE FROM tableA
    USING tableB
    WHERE tableA.id=TableB.id

SQL

- We can delete all rows from a table
- For example:
    - DELETE FROM table

SQL

- Similar to UPDATE command, you can also add in a RETURNING call to return rows that were removed.
- Let’s explore DELETE with pgAdmin!

ALTER

SQL

- The ALTER clause allows for changes to an existing table structure, such as:
    - Adding,dropping,or renaming columns
    - Changing a column’s data type
    - Set DEFAULT values for a column
    - Add CHECK constraints
    - Rename table

SQL

- General Syntax
  - ALTER TABLE table_name action

SQL

- Adding Columns
  - ALTER TABLE table_name

    ADD COLUMN new_col TYPE

SQL

- Removing Columns
  - ALTER TABLE table_name

    DROP COLUMN col_name

SQL

- Alter constraints
  - ALTER TABLE table_name

    ALTER COLUMN col_name

    SET DEFAULT value

SQL

- Alter constraints
  - ALTER TABLE table_name

    ALTER COLUMN col_name

    DROP DEFAULT

SQL

- Alter constraints
  - ALTER TABLE table_name

    ALTER COLUMN col_name

    SET NOT NULL

SQL

- Alter constraints
  - ALTER TABLE table_name

    ALTER COLUMN col_name

    DROP NOT NULL

SQL

- Alter constraints
    - ALTER TABLE table_name

      ALTER COLUMN col_name

      ADD CONSTRAINT constraint_name

SQL

- Let’s explore some examples in pgAdmin!

DROP

SQL

- DROP allows for the complete removal of a column in a table.
- In PostgreSQL this will also automatically remove all of its indexes and constraints involving the column.
- However, it will not remove columns used in views, triggers, or stored procedures without the additional CASCADE clause.

SQL

- General Syntax
  - ALTER TABLE table_name

    DROP COLUMN col_name

SQL

- Remove all dependencies
  - ALTER TABLE table_name

    DROP COLUMN col_name CASCADE

SQL

- Check for existence to avoid error
  - ALTER TABLE table_name

    DROP COLUMN IF EXISTS col_name

SQL

- Drop multiple columns
  - ALTER TABLE table_name

    DROP COLUMN col_one,

    DROP COLUMN col_two

SQL

- Let’s see a quick example in pgAdmin!

CHECK

SQL

- The CHECK constraint allows us to create more customized constraints that adhere to a certain condition.
- Such as making sure all inserted integer values fall below a certain threshold.

SQL

- General Syntax

  -
    ```
    CREATE TABLE example(
    ex_id SERIAL PRIMARY KEY,
    age SMALLINT CHECK (age > 21),
    parent_age SMALLINT CHECK (
    parent_age > age)
    );
    ```

SQL

- Let’s explore this concept in pgAdmin!