

Agenda _prof only

- Case Study (5m)
- Key Concepts Review & Advanced Functions (20m)
- Database Setup (5m)
- Schemas (5m)
- Functions (30m)
- Scripts (30m)
- Inheritance and Partitioning (30m)
 - Utilizing indexes to improve performance (15m)
- Extended Use Cases (5m)
- Review with Q & A (10m)

Digging Deeper with PostgreSQL

Automating processes and partitioning data with inheritance

Case Study Overview

- Shopping experience
- Primary keys will always be 'id' column in tables
- All foreign keys will refer to: <table_name>_id

Case Study: Shopping site

Shopping Components

Products
Product Types
Users
Orders
Carts

Case Study: Shopping site

Shopping Components

- Products
- Product Types
- Users
- Orders
- Carts

products

- id
- product_type_id
- price
- name
- grams
- address

Case Study: Shopping site

Shopping Components

Products

Product Types

Users

Orders

Carts

products

id
product_type_id
price
name
grams
address

product_types

id
name
description

Case Study: Shopping site

Shopping Components

- Products
- Product Types
- Users
- Orders
- Carts

products

- id
- product_type_id
- price
- name
- grams
- address

product_types

- id
- name
- description



Case Study: Shopping site

Shopping Components

- Products
- Product Types
- Users
- Orders
- Carts

products

- id
- product_type_id
- price
- name
- grams
- address

product_types

- id
- name
- description

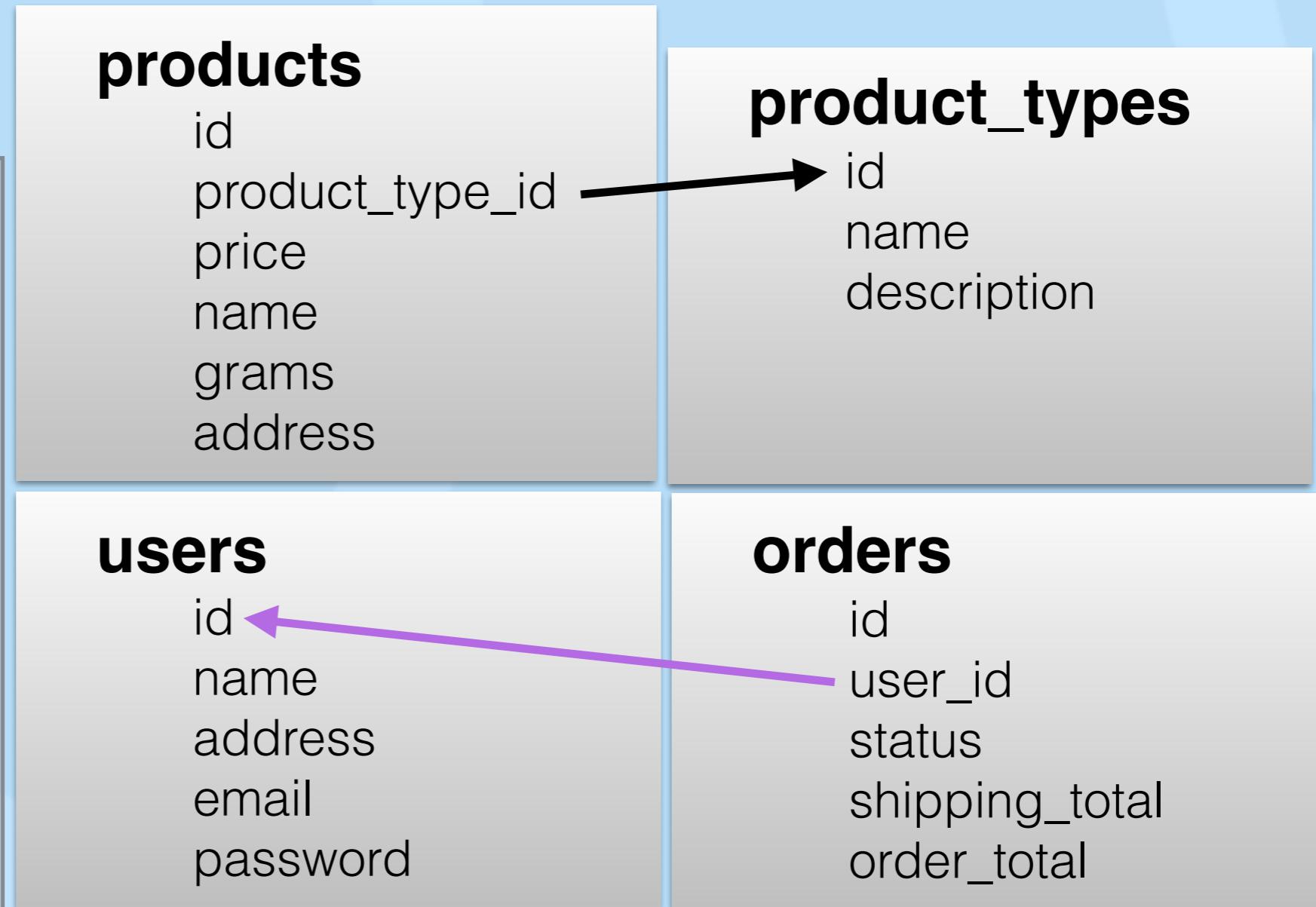
users

- id
- name
- address
- email
- password

Case Study: Shopping site

Shopping Components

- Products
- Product Types
- Users
- Orders
- Carts



Case Study: Shopping site

Shopping Components

- Products
- Product Types
- Users
- Orders
- Carts



Case Study: Shopping site

Shopping Components

- Products
- Product Types
- Users
- Orders
- Carts



Case Study: Shopping site

Shopping Components

- Products
- Product Types
- Users
- Orders
- Carts



Case Study: Shopping site

products

id
product_type_id
price
name
grams
address

product_types

id
name
description

users

id
name
address
email
password

orders

id
user_id
status
shipping_total
order_total

carts

id
user_id
last_updated

Case Study: Shopping site

products

id
product_type_id
price
name
grams
address

product_types

id
name
description

users

id
name
address
email
password

orders

id
user_id
status
shipping_total
order_total

order_products

id
product_id
order_id

carts

id
user_id
last_updated

Case Study: Shopping site

products

id ←
product_type_id →
price
name
grams
address

product_types

id
name
description

users

id ←
name
address
email
password

orders

id ←
user_id
status
shipping_total
order_total

order_products

id
product_id
order_id

carts

id
user_id
last_updated

Case Study: Shopping site

products

id ←
product_type_id →
price
name
grams
address

product_types

id
name
description

users

id ←
name
address
email
password

orders

id ←
user_id
status
shipping_total
order_total

order_products

id
product_id
order_id

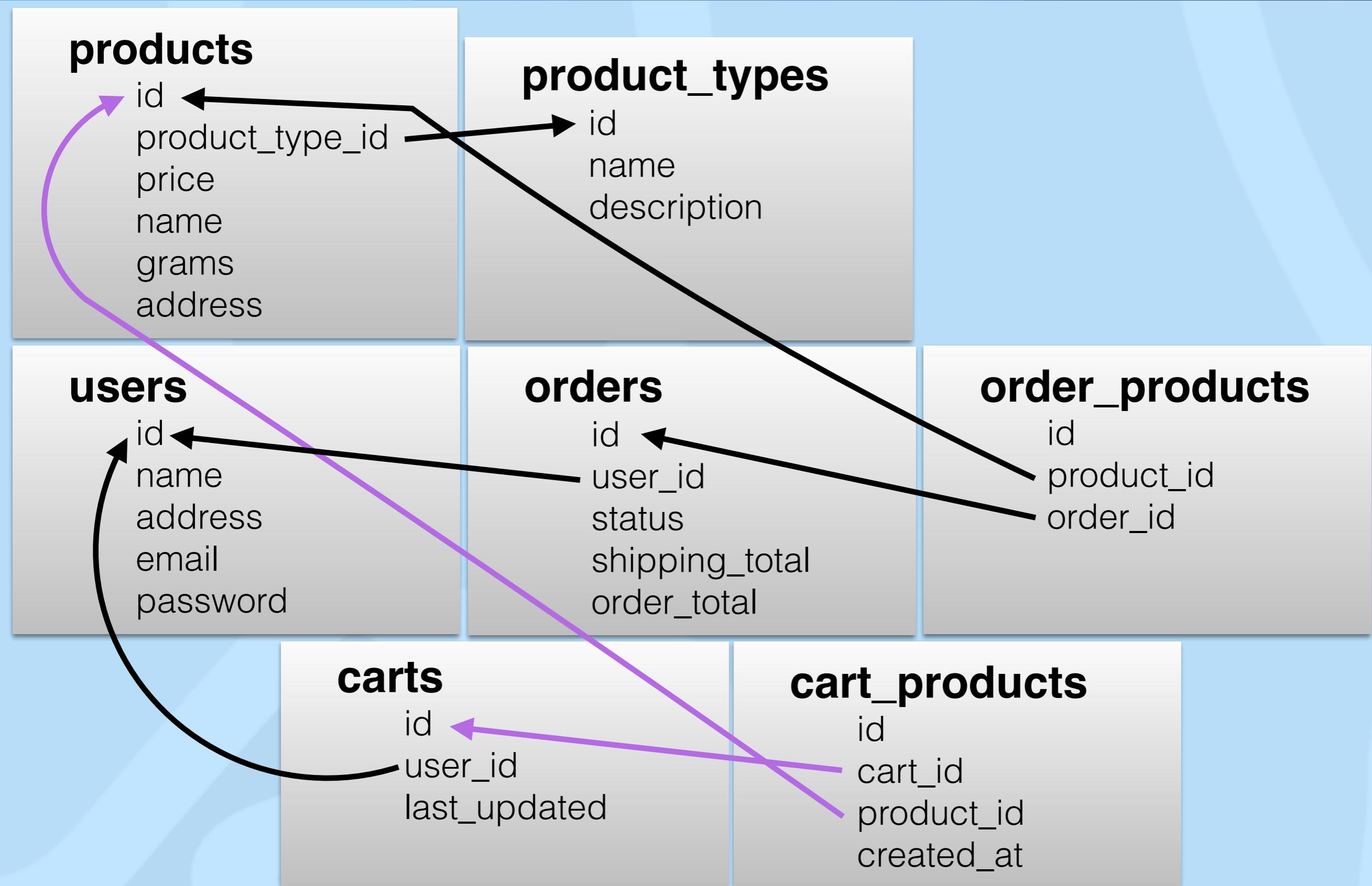
carts

id
user_id
last_updated

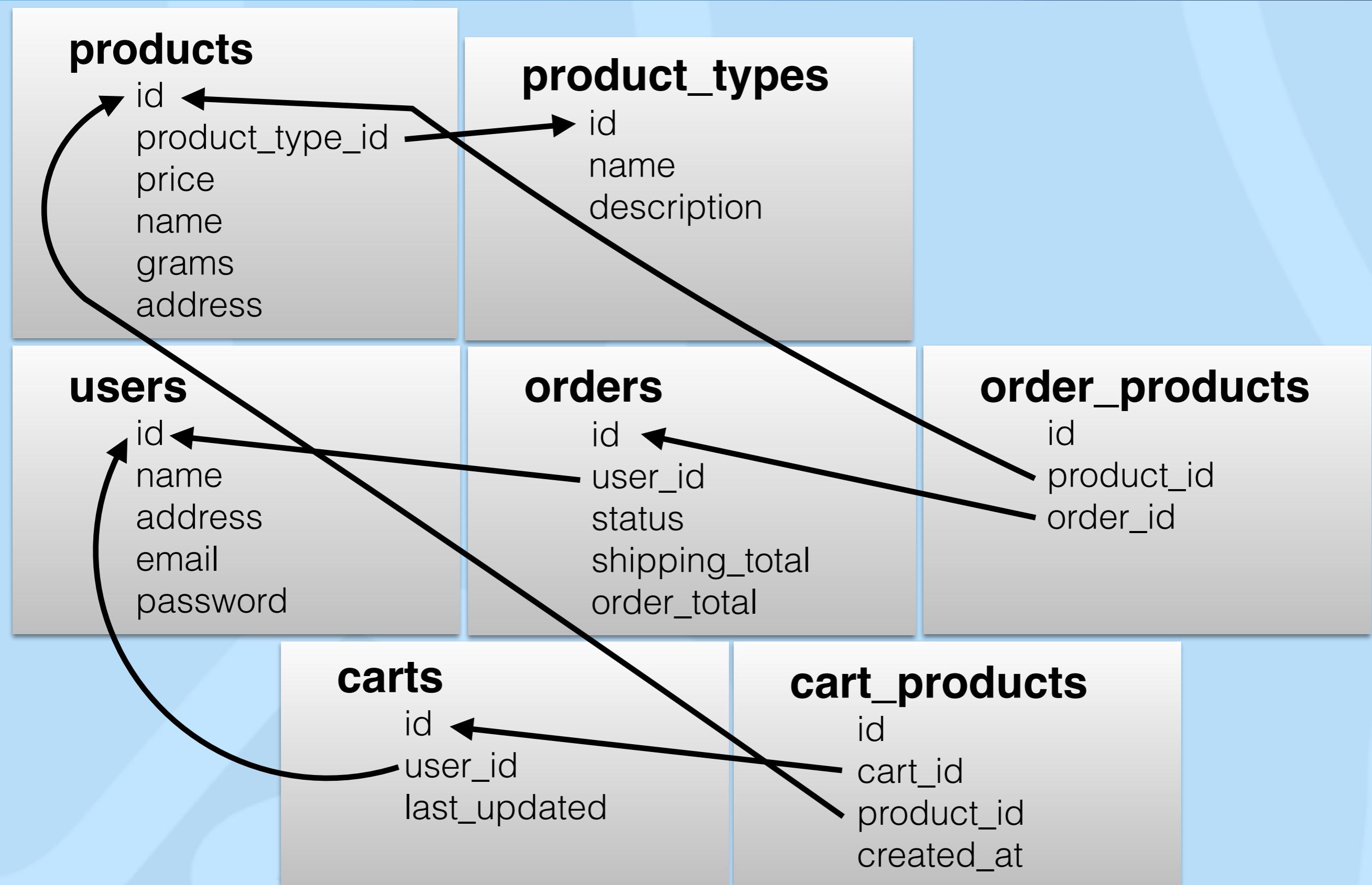
cart_products

id
cart_id
product_id
created_at

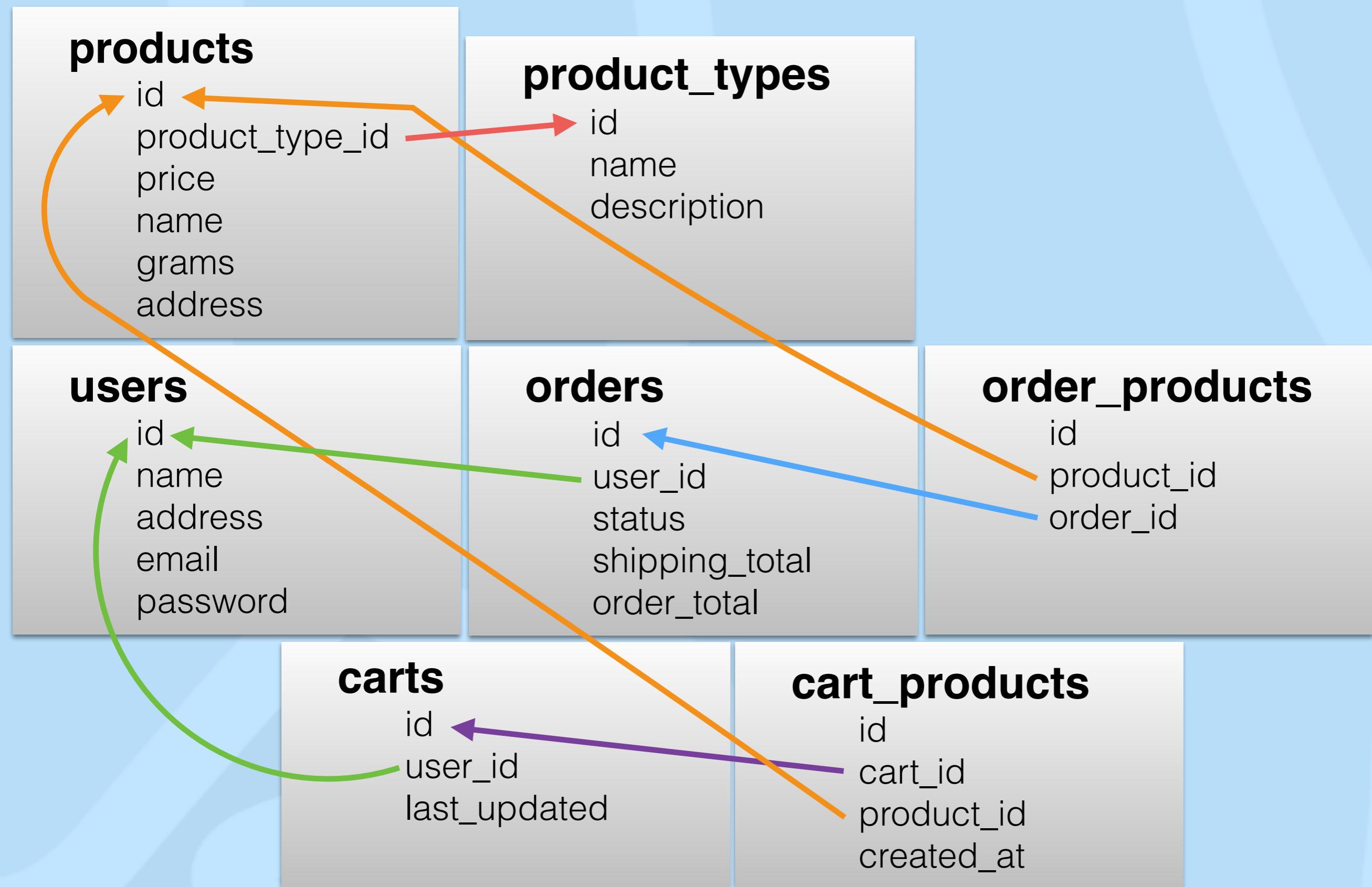
Case Study: Shopping site



Case Study: Shopping site



Case Study: Shopping site



Key Concept Review

- Select Functions
- Casting & data types
- Sets
- Creating tables

Key Concept Review

- select functions
 - distinct and group by
 - math and date functions
 - case statements
 - aliasing

Key Concept Review

- distinct and group by

```
SELECT DISTINCT product_type_id FROM products;
```

```
SELECT product_type_id FROM products  
ORDER BY product_type_id  
GROUP BY product_type_id;
```

Key Concept Review

- math functions: avg, count, sum; date functions
 - `SELECT SUM(shipping_total) as shipping_total,
COUNT(*) as total_orders, AVG(order_total) as
avg_order,
EXTRACT(MONTH FROM created_at) as month,
EXTRACT(YEAR FROM created_at) as year
FROM orders
GROUP BY year, month
ORDER BY year, month;`

Key Concept Review

- select functions
 - aliasing - setting name for columns, tables and values to be referred to more
 - ex: `SELECT o.shipping_total AS total FROM orders o;`

Key Concept Review

- select functions
 - aliasing - setting name for columns, tables and values to be referred to more
 - ex:

```
SELECT o.shipping_total AS total FROM
FROM orders o
JOIN order_products op on op.order_id = o.id
JOIN cart_products cp on cp.product_id =
op.product_id
```

Casting: Data Types

- Casting values - common 'types'
 - Text - a 'string' of characters, words/characters (i.e. from a keyboard)
 - Integer - whole number
 - Decimal - Float (i.e. floating point)
 - Date - a calendar date
 - Timestamp - a date with specific time, can also have a specific timezone

Casting

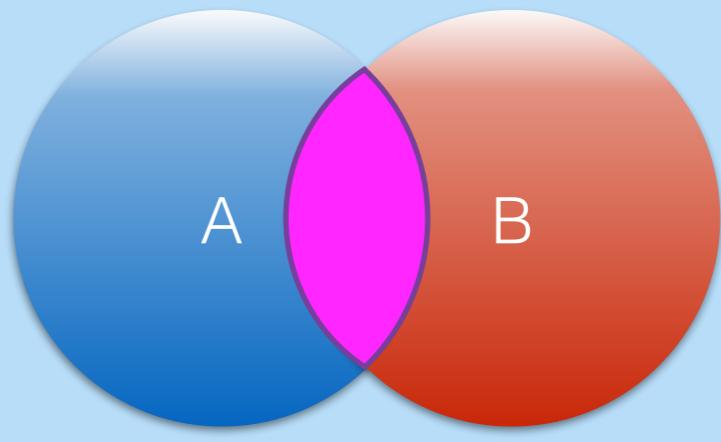
- Casting values
 - 2 methods:
 - **CAST(<column/result> AS <type>)**
 - **<column/result>::<type>**
 - SELECT
CAST(o.shipping_total AS int) as int_total,
o.shipping_total::INTEGER as int_total_v2
FROM orders o;

Key Concept Review

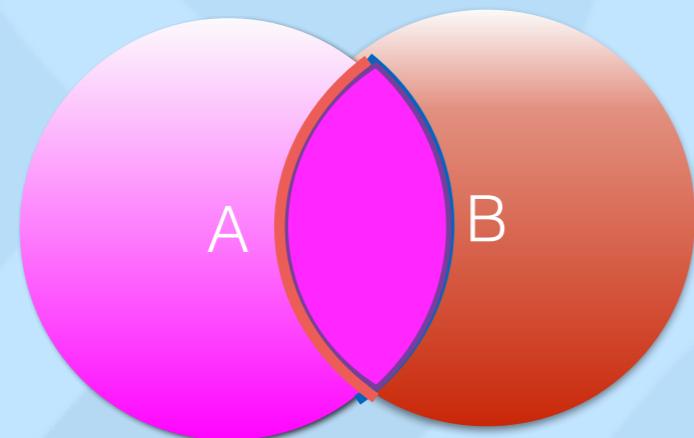
- Sets
 - Outer vs Inner
 - Left vs Right
 - Union, Intersect Except

Key Concept Review

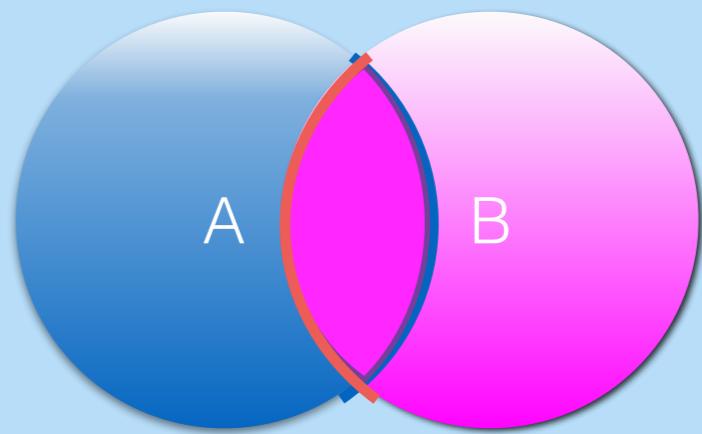
- Sets overview
 - Outer vs Inner
 - Left vs Right



A INNER JOIN B



A LEFT OUTER B



A RIGHT OUTER B

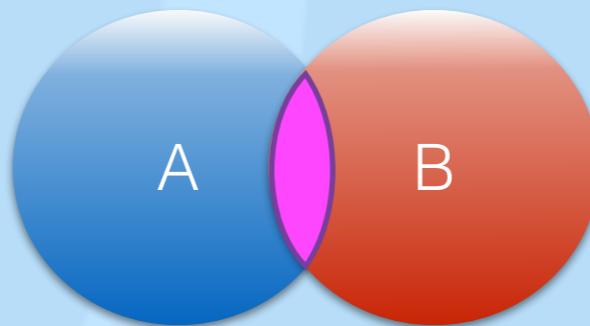
Inner Join

A = Users

id	integer
name	text
address	text
created_at	date

B = Orders

id	integer
user_id	integer
order_total	float
address	text



```
SELECT * FROM
users INNER JOIN orders ON users.id = orders.user_id
```

AA

id	name	address	created_at	id	user_id	order_total	address
1	John Nob...	123 nowhe...	2016-03-19	8	1	147.33	678 nowhe...
2	John Som...	234 a pla...	2016-05-23	23	2	43.27	372 other...
3	John Doe	345 Doesv...	2016-05-22	52	3	57.9	345 Doesv...

BB1

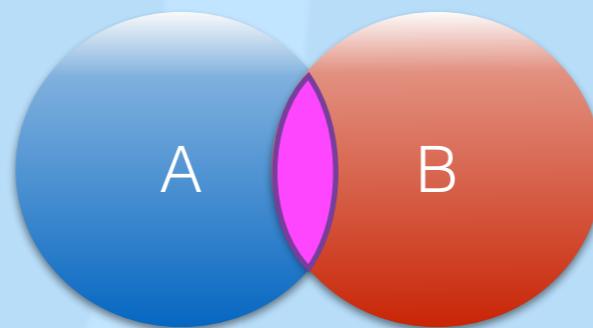
Inner Join

A = Users

id	integer
name	text
address	text
created_at	date

B = Orders

id	integer
user_id	integer
order_total	float
address	text



```
SELECT * FROM
    users, orders
WHERE users.id = orders.user_id
```

A

id	name	address	created_at
1	John Nob...	123 nowhe...	2016-03-19
2	John Som...	234 a pla...	2016-05-23
3	John Doe	345 Doesv...	2016-05-22

B-1

id	user_id	order_total	address
8	1	147.33	678 nowhe...
23	2	43.27	372 other...
52	3	57.9	345 Doesv...

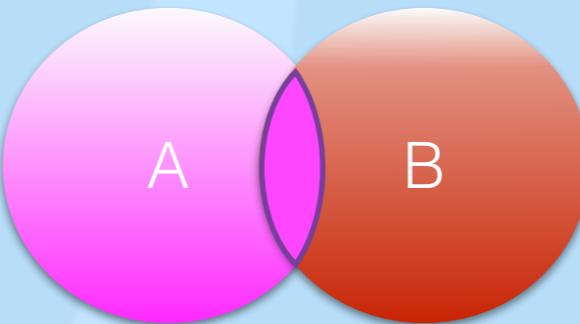
Left Outer Join

A = Users

id	integer
name	text
address	text
created_at	date

B = Orders

id	integer
user_id	integer
order_total	float
address	text



```
SELECT * FROM users
LEFT OUTER JOIN orders ON users.id = orders.user_id
```

A

id	name	address	created_at		id	user_id	order_total	address
1	John Nob...	123 nowhe...	2016-03-19		8	1	147.33	678 nowhe...
2	John Som...	234 a pla...	2016-05-23		23	2	43.27	372 other...
3	John Doe	345 Doesv...	2016-05-22		52	3	57.9	345 Doesv...
14	Jane Smith	502 downt...	2017-10-21	null	null		null	null
17	Janet Farm	402 lastt...	2018-01-24	null	null		null	null

B

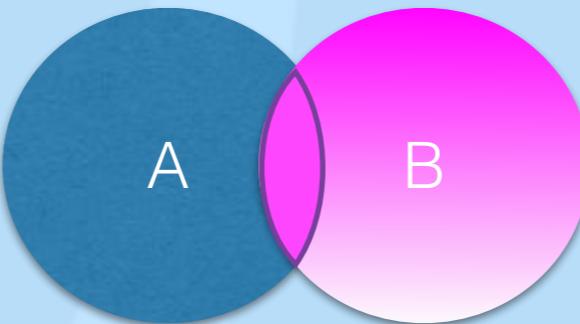
Right Join

A = Users

id	integer
name	text
address	text
created_at	date

B = Orders

id	integer
user_id	integer
order_total	float
address	text



```
SELECT * FROM users
RIGHT OUTER JOIN orders ON users.id = orders.user_id
```

A

id	name	address	created_at		id	user_id	order_total	address
1	John Nob...	123 nowhe...	2016-03-19		8	1	147.33	678 nowhe...
2	John Som...	234 a pla...	2016-05-23		23	2	43.27	372 other...
3	John Doe	345 Doesv...	2016-05-22		52	3	57.9	345 Doesv...

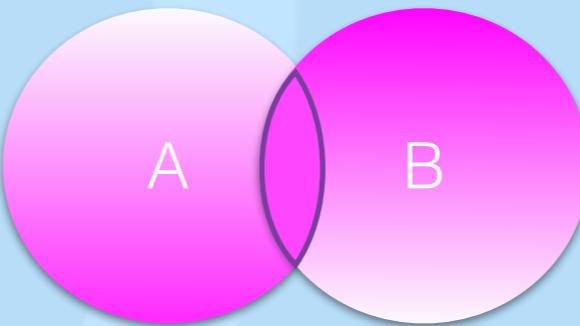
Full Join

A = Users

<code>id</code>	<code>integer</code>
<code>name</code>	<code>text</code>
<code>address</code>	<code>text</code>
<code>created_at</code>	<code>date</code>

B = Orders

<code>id</code>	<code>integer</code>
<code>user_id</code>	<code>integer</code>
<code>order_total</code>	<code>float</code>
<code>address</code>	<code>text</code>



Users FULL OUTER JOIN orders ON users.address =
orders.address

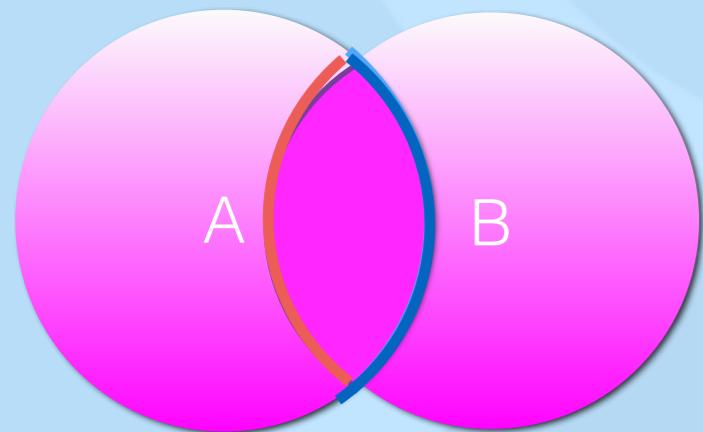
A

B

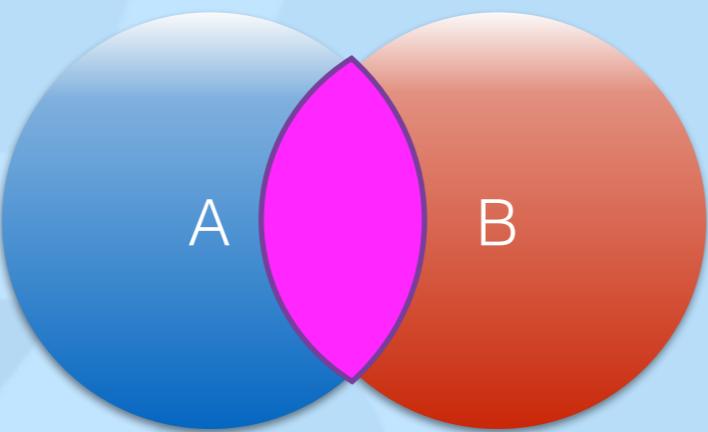
<code>id</code>	<code>name</code>	<code>address</code>	<code>created_at</code>	<code>id</code>	<code>user_id</code>	<code>order_total</code>	<code>address</code>
1	John Nob...	123	2016-03-19	null	null	null	null
14	Jane Smith	502	2017-10-21	null	null	null	null
2	John Som...	234 a	2016-05-23	35	2	113.99	234 a pla...
3	John Doe	345	2016-05-22	52	3	57.9	345 Doesv..
4	John Moe	456	2016-05-22	61	4	74.99	456 somew..
null	null	null	null	109	9	19.24	36127 av..
null	null	null	null ³⁴	20	1	64.23	678 nowh..

Key Concept Review

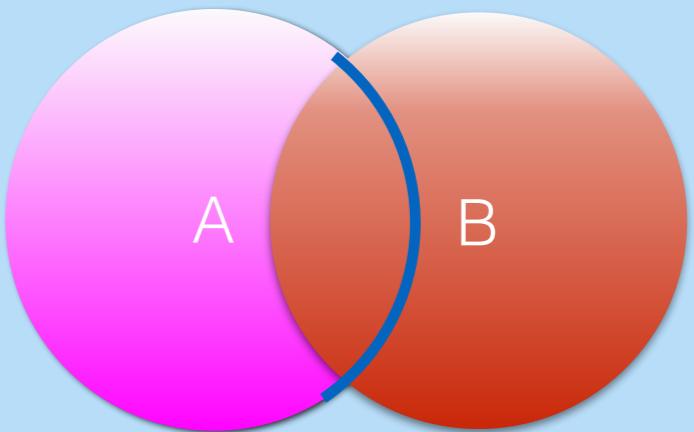
- Sets overview
 - Union, Intersect Except



A union B



A intersect B



A except B



Q & A

Advanced Functions

- Having
- String Functions
- Random

Having

- Having allows you to filter the results on the result of the rest of the query
 - Useful for filtering results of aggregate and math results only
 - Won't work with case statement results for example

Having

- Having allows you to filter the results on the RESULT of the rest of the query
- WHERE
Select product_id,
count(*) from
order_products
WHERE count(*) > 2
GROUP BY product_id
ORDER BY count desc
- HAVING
Select product_id,
count(*) from
order_products
GROUP BY product_id
HAVING count(*) > 2
ORDER BY count(*) desc

Advanced functions con't.

- String Functions
 - `||`, concatenate

Advanced functions con't.

- String Functions
 - `||`, concatenate - push two text values together

Advanced functions con't.

- String Functions
 - `||`, concatenate - push two text values together

Example: `select 'add text' || name from products;`

Concatenate: Quick Exercise

- String Functions
 - `||`, `concatenate` - push two text values together

Example: `select 'add text' || name from products;`

- EXERCISE: Return all users' full address with their name and address concatenated together.

Concatenate: Quick Exercise

- String Functions
 - `||`, `concatenate` - push two text values together

Example: `select 'add text' || name from products;`

- EXERCISE: Return all users' full address with their name and address concatenated together.

`SELECT name || ' ' || address from users;`

Advanced functions con't.

- Random

Advanced functions con't.

- Random - returns a random value between 0 and 1
EX: Select random()

Advanced functions con't.

- Random - returns a random value between 0 and 1
EX: Select random() * 10

Advanced functions con't.

- Random - returns a random value between 0 and 1
EX: Select random() * 10
- Getting whole numbers
 - round - round to the nearest whole number
 - floor - round down to the nearest whole number less than the value
 - ceiling - round up to the nearest whole number above the value

Advanced functions con't.

- Random - returns a random value between 0 and 1
EX: Select random() * 10
- Getting whole numbers
 - round - round to the nearest whole number
 - floor - round down to the nearest whole number less than the value
 - ceiling - round up to the nearest whole number above the value
- SELECT random_value,
round(random_value),
floor(random_value),
ceiling(random_value)
FROM (select random() * 10 AS random_value) random_query

Random: Quick Exercise

- Random - returns a random value between 0 and 1
EX: Select random()
- Write a query to return a random whole number value between 5 and 10

Random: Quick Exercise

- Random - returns a random value between 0 and 1
EX: Select random()
- Write a query to return a random whole number between 5 and 10
(HINT: 5 + <random whole # from 0-5>)

Random: Quick Exercise

- Random - returns a random value between 0 and 1
EX: Select random()
- Write a query to return a random value between 5 and 10

```
SELECT 5 + round(random() * 5)
```

PgAdmin4 Overview

- PG Admin4 intro



Q & A

Functions and Scripts

- Purpose/use cases
 - Automation
 - Share/Create re-usable common translations/ process

Functions

Functions

- CREATE FUNCTION <function_name>()
RETURNS <data type of result>
AS '<query>'
LANGUAGE SQL
IMMUTABLE/STABLE/VOLATILE
;

Functions

- CREATE FUNCTION <function_name>()
RETURNS <data type of result returned>
AS '<query>'
-- will only use SQL or plpgsql functions
LANGUAGE SQL
<IMMUTABLE/STABLE/VOLATILE>
;

Functions

- CREATE FUNCTION <function_name>()
RETURNS <data type of result returned>
AS \$\$ BEGIN
(<plpgsql_statement>)
END \$\$
-- will only use SQL or plpgsql functions
LANGUAGE plpgsql
<IMMUTABLE/STABLE/VOLATILE>;

Functions

- CREATE FUNCTION <function_name>()
RETURNS <data type of result returned>
AS <'query'/plpgsql statement>
-- will only use SQL or plpgsql functions
LANGUAGE SQL
-- **define if the query will interact with tables/random**
<IMMUTABLE/STABLE/VOLATILE>
;

Functions

- CREATE FUNCTION <function_name>()
RETURNS <data type of result returned>
AS <'query'/plpgsql statement>
-- will only use SQL or plpgsql functions
LANGUAGE SQL
-- **define if the query will interact with tables/random**
<IMMUTABLE/STABLE/VOLATILE>
;
- IMMUTABLE - function cannot alter or read from database
STABLE - function may read from the database, but not alter it
VOLATILE - function read from database may change within the function:
Ex: Use of these w/in a query:
random(), now(), timeofday()

Functions

- CREATE FUNCTION <function_name>()
RETURNS <data type of result returned>
AS <'query'/plpgsql statement>
-- will only use SQL or plpgsql functions
LANGUAGE SQL
-- **define if the query will interact with tables/random**
<IMMUTABLE/STABLE/VOLATILE>
;
- IMMUTABLE - function cannot alter or read from database
STABLE - function may read from the database, but not alter it
VOLATILE - function read from database may change within the function:
Ex: Use of these w/in a query:
random(), curval(), timeofday()

Functions

- CREATE FUNCTION <function_name>()
RETURNS <data type of result returned>
AS <'query'/plpgsql statement>
-- will only use SQL or plpgsql functions
LANGUAGE SQL/<pgplsql>
-- define if the query will interact with tables/random
<IMMUTABLE/STABLE/VOLATILE>
- ;

Functions

- CREATE FUNCTION <function_name>()
RETURNS <data type of result returned>
AS <'query'/plpgsql statement>
-- will only use SQL or plpgsql functions
LANGUAGE SQL
-- define if the query will interact with tables/random
<**IMMUTABLE**/STABLE/VOLATILE>
- ;

Immutable Functions

- CREATE FUNCTION **test()**
RETURNS **TEXT**
AS '**SELECT "some text"**' ;
LANGUAGE SQL
-- query won't alter or read from the database
IMMUTABLE
;

Immutable Functions

- CREATE FUNCTION **hundred()**
RETURNS **INTEGER**
AS '**SELECT 100;**'
LANGUAGE SQL
-- query won't alter or read from the database
IMMUTABLE
;

Immutable Functions: Quick Exercise

- Write a function to return Christmas Day of 2018 as a date (HINT use Cast/::DATE)

```
CREATE FUNCTION christmas()
RETURNS DATE
AS '<query>'
LANGUAGE SQL
-- query won't alter or read from the database
IMMUTABLE;
```

Immutable Functions: Quick Exercise

- Write a function to return Christmas Day of 2018 as a date (HINT use Cast/::DATE, don't forget to escape ')

```
CREATE FUNCTION christmas()
RETURNS DATE
AS 'SELECT "2018-12-25"::DATE'
LANGUAGE SQL
-- query won't alter or read from the database
IMMUTABLE;
```

Immutable pgsql Functions

- use the PostgreSQL functions and calculations without selecting from the database
- CREATE FUNCTION **hundred()**
RETURNS **INTEGER**
AS 'SELECT 100;'
LANGUAGE SQL
-- query won't alter or read from the database
IMMUTABLE;

Immutable pgsql Functions

- Use the PostgreSQL functions and calculations without selecting from the database
- ```
CREATE FUNCTION hundred()
RETURNS INTEGER
AS $$ BEGIN
 RETURN (100);
END $$

LANGUAGE 'plpgsql'
IMMUTABLE; -- query won't alter or read from the db
```

# Immutable pgplsql Functions: Quick Exercise

- Write a function to return Christmas Day of 2018 as a date (HINT use Cast/::DATE)

```
CREATE FUNCTION pg_christmas()
RETURNS DATE
AS $$ BEGIN
 RETURN (<result>)
END $$

LANGUAGE 'plpgsql'
IMMUTABLE; -- query won't alter or read from the db
```

# Immutable pgplsql Functions: Quick Exercise

- Write a function to return Christmas Day of 2018 as a date (HINT use Cast/::DATE)

```
CREATE FUNCTION pg_christmas()
RETURNS DATE
AS $$ BEGIN
 RETURN ('2018-12-25'::DATE);
END $$

LANGUAGE 'plpgsql'
IMMUTABLE; -- query won't alter or read from the db
```



**Q & A**



# 10 Min Break

# Stable Functions

- CREATE FUNCTION <function\_name>()  
RETURNS <data type of result returned>  
AS <'query'/plpgsql statement>  
LANGUAGE <SQL/plpgsql'>  
-- define if the query will interact with tables/random  
<IMMUTABLE/**STABLE**/VOLATILE>;

# Stable Functions

- CREATE FUNCTION <function\_name>()  
RETURNS <data type of result returned>  
AS <'query'/plpgsql statement>  
LANGUAGE <SQL/'plpgsql'>  
-- query will interact with the DB  
**STABLE**  
;

# Stable Functions

- Function to return the current time in your timezone

```
CREATE FUNCTION now_with_timezone()
RETURNS timestamp
```

```
AS <'query'/plpgsql statement>
LANGUAGE <SQL/'plpgsql'>
-- query will interact with the DB
STABLE
```

```
;
```

# Stable Functions:

- Function to return the current time in your timezone

```
CREATE FUNCTION now_with_timezone()
RETURNS timestamp
AS 'SELECT now::timestamp with time zone'
LANGUAGE SQL
-- query will interact with the DB
STABLE;
```

# Stable Functions:

- Function to return the current time in your timezone

```
CREATE FUNCTION now_with_timezone()
RETURNS timestamp
AS $$ BEGIN
 RETURN <result>
END $$
LANGUAGE 'plpgsql'
-- query will interact with the DB
STABLE;
```

# Stable Functions:

- Function to return the current time in your timezone

```
CREATE FUNCTION now_with_timezone()
RETURNS timestamp
AS $$ BEGIN
 RETURN now::timestamp with timezone
END $$;
LANGUAGE 'plpgsql'
-- query will interact with the DB
STABLE;
```

# Stable Functions: Quick Exercise

- Create a function to return the current hour of the day  
(HINT: Use Extract and now())

# Stable Functions: Quick Exercise

- Create a function to return the current hour of the day  
(HINT: Use Extract and now())

```
CREATE FUNCTION current_hour()
RETURNS integer
```

AS <'query'/plpgsql statement>

LANGUAGE SQL/'plpgsql'

STABLE; -- query will interact with the DB, but not alter

# Stable Functions: Quick Exercise

- Create a function to return the current hour of the day

pgplsql:

```
CREATE FUNCTION current_hour()
RETURNS integer
AS $$ BEGIN
 RETURN EXTRACT(hour from now());
END $$;
LANGUAGE 'pgplsql'
STABLE; -- query will interact with the DB, but not alter
```

SQL:

```
CREATE FUNCTION current_hour()
RETURNS integer
AS 'SELECT EXTRACT(hour from now())'
LANGUAGE SQL
STABLE; -- query will interact with the DB, but not alter
```



**Q & A**

# Volatile Functions

- VOLATILE - function read from database may change within the function:  
Ex: Use of these w/in a query:  
random(), currval(), timeofday()
- CREATE FUNCTION <function\_name>()  
RETURNS <data type of result returned>  
AS <'query'/plpg statement>  
LANGUAGE <SQL/pgplsql'>  
-- define if the query will interact with tables/random  
<IMMUTABLE/STABLE/**VOLATILE**>  
;

# Volatile Function

- use the PostgreSQL functions and calculations without selecting from the database
- CREATE FUNCTION **random\_to\_5()**  
RETURNS **float**  
AS '**SELECT random() \* 5**'  
LANGUAGE **SQL**  
**VOLATILE**; -- random could change w/in query

# Volatile plpgsql Function

- use the PostgreSQL functions and calculations without selecting from the database
- CREATE FUNCTION **random\_to\_5()**  
RETURNS **float**  
AS \$\$ BEGIN  
    (**random()** \* 5)  
END \$\$  
LANGUAGE '**plpgsql**'  
VOLATILE; -- random could change w/in query

# Volatile Function - Advanced Example

- Function to pick a random date in the next 30 days.

```
CREATE FUNCTION random_date_next_month()
RETURNS date
AS <'query'/plpgsql statement>
LANGUAGE <SQL/'plpgsql'>
-- query value could change
VOLATILE;
```

# Volatile Function - Advanced Example

- Function to pick a random date in the next 30 days.

```
CREATE FUNCTION random_date_next_month()
RETURNS date
AS '<query>';
LANGUAGE SQL
VOLATILE;
```

# Volatile Function - advanced

## Example

- Function to pick a random date in the next 30 days.

```
CREATE FUNCTION random_date_next_month()
RETURNS date
```

```
AS 'SELECT CAST(now() +
(round(random()*30) * interval "1 day") AS
DATE);'
```

```
LANGUAGE SQL
```

```
VOLATILE; -- random could change w/in query
```

# Volatile Function - Advanced Example

- Function to pick a random date in the next 30 days.

```
CREATE FUNCTION random_date_next_month()
RETURNS date
AS <plpgsql statement>
LANGUAGE 'plpgsql'
VOLATILE; -- query value could change
```

# Volatile Function - advanced

## Example

- Function to pick a random date in the next 30 days.

```
CREATE FUNCTION random_date_next_month()
RETURNS date
AS $$ BEGIN
 RETURN (now() + (floor(random())*30) *
interval '1 day')::DATE;
END $$;
LANGUAGE SQL
VOLATILE; -- random could change w/in query
```

# Volatile Function: Quick Exercise

- Write a function that will return a whole number from 1 to 10
- CREATE FUNCTION **random\_1\_10()**  
RETURNS **integer**  
AS **<'query'/plpg statement>**  
**LANGUAGE SQL/'plpgsql'**  
**VOLATILE;** -- random could change w/in query

# Volatile Function: Quick Exercise

- Write a function that will return a whole number from 1 to 10
- CREATE FUNCTION **random\_1\_10()**  
RETURNS **integer**  
AS \$\$ BEGIN  
 RETURN (1+ floor(random() \* 10))  
END \$\$  
LANGUAGE '**pgplsql**'  
VOLATILE; -- random could change w/in query
- CREATE FUNCTION **random\_1\_10()**  
RETURNS **integer**  
AS '**SELECT 1+ floor(random() \* 10)**'  
LANGUAGE **SQL**  
VOLATILE; -- random could change w/in query



**Q & A**

# Functions - Variables

# Functions - Variables

- CREATE FUNCTION **<function\_name>(<data\_type>, ...)**  
RETURNS <data type of result>  
AS <'query'/plpg statement>  
LANGUAGE <SQL/'pgplsql'>  
<IMMUTABLE/STABLE>;

# Functions - Variables

- CREATE FUNCTION **<function\_name>(<data\_type>, ...)**  
RETURNS <data type of result>  
AS <'query'/plpg statement>  
LANGUAGE <SQL/'pgplsql'>  
<IMMUTABLE/STABLE>;

# Functions - Variables

- CREATE FUNCTION **sample(text, text, integer, integer)**  
RETURNS text  
    AS 'SELECT \$1 || \$2 || \$3 || \$4;'  
    LANGUAGE SQL  
    IMMUTABLE;

```
SELECT sample('first', 'second', 3, 4);
```

# Functions - Variables

- CREATE FUNCTION **<function\_name>(<data\_type>, ...)**  
RETURNS <data type of result>  
AS '<query>;'  
LANGUAGE <SQL/pgplsql>  
<IMMUTABLE/STABLE>  
-- optionally can deal with null values specifically  
**RETURNS NULL ON NULL INPUT;**

# Functions - Variables

- Example: Create a function for addition:

```
CREATE FUNCTION addition(<data_type>, ...)
RETURNS <data type of result>
AS <'query'/plpgsql statement>
LANGUAGE <SQL/'pgplsql'>
<IMMUTABLE/STABLE>;
```

# Functions - Variables

- Example: Create a function for addition:

```
CREATE FUNCTION addition(integer, integer)
RETURNS <data type of result>
AS <'query'/plpgsql statement>
LANGUAGE <SQL/'pgplsql'>
<IMMUTABLE/STABLE>;
```

# Functions - Variables

- Example: Create a function for addition:

```
CREATE FUNCTION addition(integer, integer)
RETURNS integer
AS <'query'/plpg statement>
LANGUAGE <SQL/'pgplsql'>
<IMMUTABLE/STABLE/VOLATILE>;
```

# Functions - Variables

- Example: Create a function for addition:

```
CREATE FUNCTION addition(integer, integer)
RETURNS integer
AS 'SELECT $1 + $2;'
LANGUAGE SQL
<IMMUTABLE/STABLE/VOLATILE>;
```

# Functions - Variables

- Example: Create a function to add two integers

```
CREATE FUNCTION addition(integer, integer)
RETURNS integer
AS 'SELECT $1 + $2;'
LANGUAGE SQL
IMMUTABLE; -- query won't use DB
```

# Functions - Variables

- Example: Create a function to add two integers

```
CREATE FUNCTION addition(integer, integer)
RETURNS integer
AS $$ BEGIN
 RETURN ($1 + $2);
END; $$

LANGUAGE 'plpgsql'
IMMUTABLE; -- function won't use database
```

# Functions: Quick Exercise

- Create a function that multiplies two integers using the SQL language

# Functions: Quick Exercise

- Create a function that multiplies two integers using the SQL language

```
CREATE FUNCTION multiply(integer, integer)
RETURNS integer
AS '<query>'
LANGUAGE SQL
IMMUTABLE;
```

# Functions: Quick Exercise

- Create a function that multiplies two integers using the SQL language

```
CREATE FUNCTION multiply(integer, integer)
RETURNS integer
AS 'select $1 * $2;'
LANGUAGE SQL
IMMUTABLE;
```

# Functions: Quick Exercise

- Create a function that multiplies two integers using the plpgsql language

# Functions: Quick Exercise

- Create a function that multiplies two integers using the plpgsql language

```
CREATE FUNCTION multiply2(integer, integer)
RETURNS integer
AS $$ BEGIN
 RETURN (<result>);
END; $$

LANGUAGE 'plpgsql'
IMMUTABLE;
```

# Functions: Quick Exercise

- Create a function that multiplies two integers using the plpgsql language

```
CREATE FUNCTION multiply2(integer, integer)
RETURNS integer
AS $$ BEGIN
 RETURN ($1 * $2);
END; $$

LANGUAGE 'plpgsql'
IMMUTABLE;
```

# Functions - Variables

- Example: Create a function to add two integers - store the value into a variable

```
CREATE FUNCTION addition_w_tot(integer, integer)
RETURNS integer
AS $$
 DECLARE total integer;
BEGIN
 SELECT ($1 + $2) INTO total;
 RETURN total;
END; $$
LANGUAGE 'pgsql'
IMMUTABLE; -- query won't use DB
```



**Q & A**

# Functions Exercises

1. Create a function that returns the current week with year value as text: "year\_week" using SQL  
HINT(Use EXTRACT and now() )
2. ... Also using pgplsql

# Functions Exercises

1. Create a function that returns the current week with year value as text: "year\_week"  
HINT(Use EXTRACT and now() )

```
CREATE OR REPLACE FUNCTION year_week()
 RETURNS text
 AS '<...>'
 LANGUAGE SQL
STABLE;
```

2. using plpgsql:

```
CREATE OR REPLACE FUNCTION year_week()
 RETURNS text
 AS $$ BEGIN
 RETURN <...>;
 END $$;
 LANGUAGE SQL
STABLE;
```

# Functions Exercises

1. Create a function that returns the current week with year value as text: "year\_week"

HINT(Use EXTRACT and now() ):

**SELECT EXTRACT(YEAR FROM now()) || '\_' || EXTRACT (WEEK from now())**

CREATE OR REPLACE FUNCTION year\_week()

RETURNS text

AS '<...>'

LANGUAGE SQL

STABLE;

2. using plpgsql:

CREATE OR REPLACE FUNCTION year\_week()

RETURNS text

AS \$\$ BEGIN

    RETURN (<...>);

END \$\$

LANGUAGE SQL

STABLE;

# Functions Exercises

1.Create a function that returns the current week with year value as text: "year\_week"  
HINT(Use EXTRACT and now() )

```
CREATE OR REPLACE FUNCTION year_week()
RETURNS text
AS 'SELECT EXTRACT(YEAR from now()) ||
 "_" || EXTRACT(WEEK from
now());'
LANGUAGE SQL
STABLE;
```

# Functions Exercises

2.Create a function that returns the current week with year value as text: "year\_week"  
HINT(Use EXTRACT and now() )

```
CREATE OR REPLACE FUNCTION year_week_pl()
RETURNS text AS $$ BEGIN
RETURN EXTRACT(YEAR from now()) || '_' ||
EXTRACT(WEEK from now());
END $$;
LANGUAGE 'plpgsql'
STABLE;
```



**Q & A**



# 10 Min Break

# Scripts

- Scripts are a series of commands stored in a file.

# Scripts

- Scripts are a series of commands stored in a file.
- Useful with tools such as psql a command line version of pgAdmin (no graphical interface).

# Scripts

- Scripts are a series of commands stored in a file.
- Useful with tools such as psql a command line version of pgAdmin (no graphical interface).
- Separate each command with ;

# Scripts

- Scripts are a series of commands stored in a file.
- Useful with tools such as psql a command line version of pgAdmin (no graphical interface).
- Separate each command with ;
- Each command will run fully before going to the next one

# Scripts

- Scripts are a series of commands stored in a file.
- Useful with tools such as psql a command line tool (no graphical interface) or can be ran from the Query Tool version of pgAdmin.
- Separate each command with ;
- Each command will run fully before going to the next one
- The script won't run if any command has an error

# Scripts

- Script commands can be 'wrapped' to execute a generated query:
- DO \$\$ BEGIN  
  <execute/command>  
END \$\$

# Scripts

- Script commands can be 'wrapped' to execute a command:
- DO \$\$ BEGIN  
<execute/command>  
END \$\$

# Scripts

- Example
- Create a function, then run a command to utilize it

# Scripts

- Example
- Create a function, then run a command to utilize it
- Using the multiply function from earlier:
  - CREATE FUNCTION multiply(integer, integer)  
RETURNS integer  
AS 'select \$1 \* \$2;'  
LANGUAGE SQL  
IMMUTABLE;

```
SELECT multiply(4,3)
UNION
SELECT multiply(10,2);
```

# Scripts

- Example
- Create a function, then run a command to utilize it
- Using the multiply function from earlier:
  - **CREATE OR REPLACE** FUNCTION multiply(integer, integer)  
RETURNS integer  
AS 'select \$1 \* \$2;'  
LANGUAGE SQL  
IMMUTABLE;
  - SELECT multiply(4,3)  
UNION  
SELECT multiply(10,2);

# Scripts

- Script can have any number of commands that will be ran.

# Functions and Scripts

- Advanced Use case:

Since carts can be updated at any time, want to keep history. Let's create a script that will create a snapshot of the products that are in a cart each week, and record the cart they are in with the date of the snapshot.

# Functions and Scripts

- Create a snapshot of the products that are in a cart each week, and record the cart they are in with the date of the snapshot.
- Step 1: Query to get the current products in a cart.  
SELECT products.id as product\_id,  
products.price,  
products.grams,  
products.name as product,  
cart\_products.cart\_id,  
cart\_products.created\_at as cart\_added\_at  
FROM cart\_products  
JOIN products on products.id = cart\_products.product\_id

# Functions and Scripts

- Create a snapshot of the products that are in a cart each week, and record the cart they are in with the date of the snapshot.
- Step 2: Include now() as the snapshot date  
SELECT products.id as product\_id,  
products.price,  
products.grams,  
products.name as product,  
cart\_products.cart\_id,  
cart\_products.created\_at as cart\_added\_at,  
**now() as snapshot\_date**  
FROM cart\_products  
JOIN products on products.id = cart\_products.product\_id

# Functions and Scripts

- Create a snapshot of the products that are in a cart each week, and record the cart they are in with the date of the snapshot.
- Step 3: Create a table for with the results of the snapshot

**CREATE TABLE product\_details AS**

```
SELECT products.id as product_id,
products.price,
products.grams,
products.name as product,
cart_products.cart_id,
cart_products.created_at as cart_added_at,
now() as snapshot_date
FROM cart_products
JOIN products on products.id = cart_products.product_id
```

# Functions and Scripts

- Create a snapshot of the products that are in a cart each week, and record the cart they are in with the date of the snapshot.
- Step 4: Ensure the table names are unique

```
CREATE TABLE product_details_YEAR_WEEK AS
SELECT products.id as product_id,
products.price,
products.grams,
products.name as product,
cart_products.cart_id,
cart_products.created_at as cart_added_at,
now() as snapshot_date
FROM cart_products
JOIN products on products.id = cart_products.product_id
```

# Functions and Scripts

- Create a snapshot of the products that are in a cart each week, and record the cart they are in with the date of the snapshot.
- Step 5: Utilize a function to create the name - Start with query from earlier!

```
CREATE OR REPLACE FUNCTION year_week()
 RETURNS text
 AS 'SELECT "EXTRACT(YEAR from now()) ||
 " - " || EXTRACT(WEEK from now());'
 LANGUAGE SQL
STABLE;
```

```
SELECT product_detail_table();
```

# Functions and Scripts

- Create a snapshot of the products that are in a cart each week, and record the cart they are in with the date of the snapshot.
- Step 5: Utilize a function to create the name  
CREATE OR REPLACE FUNCTION **product\_detail\_tablename()**  
RETURNS text  
AS 'SELECT "**product\_detail\_**" ||  
EXTRACT(YEAR from now()) ||  
"\_" || EXTRACT(WEEK from now());'  
LANGUAGE SQL  
STABLE;

**SELECT product\_detail\_tablename();**

# Functions and Scripts

- Create a snapshot of the products that are in a cart each week, and record the cart they are in with the date of the snapshot.

- Step 5: Utilize a function to create the name

```
CREATE OR REPLACE FUNCTION product_detail_tablename()
```

```
RETURNS text
```

```
AS 'SELECT "product_detail_" ||
```

```
 EXTRACT(YEAR from now()) ||
```

```
 " _ " || EXTRACT(WEEK from now());'
```

```
LANGUAGE SQL
```

```
STABLE;
```

```
SELECT product_detail_tablename();
```

- DO \$\$ BEGIN

```
EXECUTE 'CREATE TABLE ' || product_detail_tablename() ||
```

```
 ' AS SELECT 100 as hundred;';
```

```
END $$
```

# Scripts

- Create a snapshot of the products that are in a cart each week, and record the cart they are in with the date of the snapshot.
- Step 6: Bring it all together

```
CREATE OR REPLACE FUNCTION product_detail_tablename()
RETURNS text
AS 'SELECT "product_detail_" ||
 EXTRACT(YEAR from now()) ||
 "_" || EXTRACT(WEEK from now());'
LANGUAGE SQL
STABLE;
DO $$ BEGIN
EXECUTE 'CREATE TABLE ' || product_details_tablename() || ' AS
SELECT products.id as product_id,
 products.price,
 products.grams,
 products.name as product,
 cart_products.cart_id,
 cart_products.created_at as cart_added_at,
 now() as snapshot_date
 FROM cart_products
 JOIN products on products.id = cart_products.product_id;';
END $$;
```

# Scripts

- Create a snapshot of the products that are in a cart each week, and record the cart they are in with the date of the snapshot.
- Step 6: Bring it all together

```
CREATE OR REPLACE FUNCTION product_detail_tablename()
RETURNS text
AS 'SELECT "product_detail_" ||
 EXTRACT(YEAR from now()) || "_" || EXTRACT(WEEK from now());'
LANGUAGE SQL
STABLE;
DO $$ BEGIN
EXECUTE 'DROP TABLE IF EXISTS ' || product_detail_tablename() || '
'; CREATE TABLE ' || product_details_tablename() || ' AS
SELECT products.id as product_id,
 products.price,
 products.grams,
 products.name as product,
 cart_products.cart_id,
 cart_products.created_at as cart_added_at,
 now() as snapshot_date
 FROM cart_products
 JOIN products on products.id = cart_products.product_id;';
END $$;
```



**Q & A**

# Schema

- Schemas are a named 'space' for data to be stored
  - If a database is like a large excel spreadsheet
  - Schema is like a single tab

# Schemas

- Common use cases:
  - separation of 'type' of data (from different sources or by concept)
  - need to 'segment'/separate data to improve performance
  - Simplify master queries, etc.

# Schema

- CREATE SCHEMA historical\_cart\_details;

# Schemas

- Referring to tables in the schema:
  - <schema>.<table>

# Schema

- CREATE OR REPLACE FUNCTION product\_detail\_tablename()  
RETURNS text  
AS 'SELECT "**historical\_cart\_details**.product\_detail\_" ||  
EXTRACT(YEAR from now()) || "\_" || EXTRACT(WEEK from now());'  
LANGUAGE SQL  
STABLE;  
DO \$\$ BEGIN  
EXECUTE EXECUTE 'DROP TABLE IF EXISTS ' || product\_detail\_tablename() ||  
'; CREATE TABLE ' || product\_details\_tablename() || ' AS  
SELECT products.id as product\_id,  
products.price,  
products.grams,  
products.name as product,  
cart\_products.cart\_id,  
cart\_products.created\_at as cart\_added\_at,  
now() as snapshot\_date  
FROM cart\_products  
JOIN products on products.id = cart\_products.product\_id';  
END \$\$;

# Inheritance

- Creating a table will 'inherit' the schema from the source table

# Inheritance

- First let's create a master table
- CREATE TABLE cart\_product\_details

```
CREATE TABLE cart_details
(
 product_id integer,
 price double precision,
 grams double precision,
 product character varying(100),
 cart_id integer,
 cart_added_at timestamp without time zone,
 snapshot_date timestamp with time zone
)
```



# Inheritance

- `CREATE TABLE historical_cart_details.test_table()  
INHERITS(cart_details)`

# Inheritance

- `CREATE TABLE historical_cart_details.test_table  
INHERITS(cart_details)`
- `SELECT * FROM historical_cart_details.test_table`

# Inheritance

- **INSERT INTO historical\_cart\_details.test\_table**  
(SELECT 2000 as product\_id,  
0.00 as price,  
0 as grams,  
'test' as product,  
2000 as cart\_id,  
now() as cart\_added\_at,  
now()::DATE as snapshot\_date  
);

# Inheritance

- `SELECT * FROM historical_cart_details.test_table;`

# Inheritance

- `SELECT * FROM historical_cart_details.test_table;`

# Inheritance

- DO \$\$ BEGIN  
EXECUTE 'DROP TABLE IF EXISTS ' || product\_detail\_tablename() ||  
'; **CREATE TABLE** ' || **product\_details\_tablename()** || ' AS  
SELECT products.id as product\_id,  
products.price,  
products.grams,  
products.name as product,  
cart\_products.cart\_id,  
cart\_products.created\_at as cart\_added\_at,  
now() as snapshot\_date  
FROM cart\_products  
JOIN products on products.id = cart\_products.product\_id;';  
END \$\$;

# Inheritance

- **INSERT INTO <table>**  
(SELECT products.id as product\_id,  
products.price,  
products.grams,  
products.name as product,  
cart\_products.cart\_id,  
cart\_products.created\_at as cart\_added\_at,  
now()::DATE as snapshot\_date  
FROM cart\_products  
JOIN products on products.id =  
cart\_products.product\_id);

# Inheritance

- DO \$\$ BEGIN  
EXECUTE EXECUTE 'DROP TABLE IF EXISTS ' || product\_detail\_tablename() ||  
'; **CREATE TABLE** ' || product\_details\_tablename() || ' **INHERITS**  
**cart\_details;**  
**INSERT INTO cart\_details**  
(SELECT products.id as product\_id,  
products.price,  
products.grams,  
products.name as product,  
cart\_products.cart\_id,  
cart\_products.created\_at as cart\_added\_at,  
now() as snapshot\_date  
FROM cart\_products  
JOIN products on products.id = cart\_products.product\_id);' ;  
END \$\$;

# Inheritance

- Now I can select from the parent table and see rows from both child tables.

```
SELECT * FROM cart_details;
```



**Q & A**

# Day 2 Wrap-up