**PostgreSQL Tutorial**

00:30 Why Use Postgres?

Postgres is an object relational database that is just as fast as MySQL that adheres more closely to SQL standards and excels at concurrency. Postgres is also superior at avoiding data corruption. Postgres also provides more advanced data types and allows for the creation of custom types, operators and index types.

Postgres is normally the best option when extensibility, scalability and data integrity are most important to you.

**After Installation**

1. The difference between Windows / Linux and Mac is that on Mac you’ll have a username database along with the postgres database
2. PGAdmin4 : Can connect to and edit / view / change databases both locally and remote

**01:13 What is a Database**

A database is data that is structured into rows and columns like a spreadsheet. To receive or change data in a database you send it commands called queries. The database in turn returns a result based on that request.

Databases contain many tables of data organized into rows and columns. Each column represents one type of data the database stores. Each row contains multiple pieces of data specific to each entity you are describing. For example we store information on students here. Each individual value stored is called a cell.

Primary keys are used to define unique entities in your tables. Here id provides a unique value associated with each student.

**03:12 Change Database Theme**

**03:53 Create a Database**

Right click Databases -> Create -> Database

Name it and save sales\_db

Right click -> Query Tool -> Start writing SQL queries

**04:46 How to Design a Database**

**05:50 Turning an Invoice into a Database**

One way to define what your database needs to contain is to use a real world way of tracking an order. An invoice is a perfect example of that.

First I define all the information I want to track on the customer.

**07:04 Make a Table**

CREATE TABLE customer(

first\_name VARCHAR(30) NOT NULL,

last\_name VARCHAR(30) NOT NULL,

email VARCHAR(60) NOT NULL,

company VARCHAR(60) NULL,

street VARCHAR(50) NOT NULL,

city VARCHAR(40) NOT NULL,

state CHAR(2) NOT NULL DEFAULT 'PA',

zip SMALLINT NOT NULL,

phone VARCHAR(20) NOT NULL,

birth\_date DATE NULL,

sex CHAR(1) NOT NULL,

date\_entered TIMESTAMP NOT NULL,

id SERIAL PRIMARY KEY

);

You can find the table -> Schemas -> Public -> Tables

**What is Going On?**

Create table customer creates the table named customer

When defining what data goes in each cell you must define the type of data you plan to store

**12:13 Data Types**

**Character Types**

1. Char(5) : Stores up to a max number of 5 characters
2. Varchar : Store any length of characters
3. Varchar(20) : Store up to 20 characters
4. Text : Store any length of characters

**Numeric Types** : *Used when you need accuracy / precision*

**Serial** : Whole numbers that also auto increment. *Always used for column ids.*

1. Smallserial : 1 to 32,767
2. Serial : 1 to 2147483647
3. Bigserial : 1 to 9223372036854775807

**Integer** : Whole numbers only *Always used when you don’t need a decimal*

1. Smallint : -32,768 to 32, 767
2. Integer : -2,147,583,648 to 2,174,483,647
3. Bigint : -9223372036854775808 to 9223372036854775807

**Floats**

1. Decimal : 131072 whole digits and 16383 after decimal
2. Numeric : 131072 whole digits and 16383 after decimal
3. Real : 1E-37 to 1E37 (6 places of precision)
4. Double Precision : 1E-307 to 1E308 (15 places of precision) *Used when decimal doesn’t have to be very precise*
5. Float : Same as double

**Boolean**

1. True, 1, t, y, yes, on
2. False, 0, f, n, no, off
3. null

**Date / Time**

DATE

1. No matter what format you enter you get this : 1974-12-21

TIME

1. TIME WITHOUT TIME ZONE (Default)
2. ‘1:30:30 PM’:: TIME WITHOUT TIME ZONE -> 13:30:30
3. 01:30 AM EST -> 01:30-5:00 (UTC Format)
4. 01:30 PM PST -> 01:30-8:00
5. 01:30 PM UTC -> 01:30+00:00
6. ’01:30:30 PM EST’::TIME WITH TIME ZONE -> 13:30:30-5:00

TIMESTAMP

1. ‘DEC-21-1974 1:30 PM EST’::TIMESTAMP WITH TIME ZONE -> 1974-12-21 13:30-5:00

INTERVAL

1. Represents a duration of time
2. ‘1 day’::INTERVAL -> 01:00
3. ‘1 D 1 H 1 M 1 S’::INTERVAL -> 01:01:01:01
4. You can add and subtract intervals
5. You can add or subtract intervals from dates
6. (‘DEC-21-1974 1:30 PM EST’::TIMESTAMP WITH TIME ZONE) – (‘1 D’::INTERVAL)

Also Currency, Binary, JSON, Range, Geometric, Arrays, XML, UUID

**Data Constraints**

Some additional data constraints include Not Null. If you mark data as not null that means it must have a value when a new row of data is created. Default designates a default value when a row is created without data.

Primary keys are identified as unique values assigned to a row. They are auto incremented each time a row of data is created

**16:36 Adding Data to Table**

INSERT INTO customer(first\_name, last\_name, email, company, street, city, state, zip, phone, birth\_date, sex, date\_entered) VALUES ('Christopher', 'Jones', 'christopherjones@bp.com', 'BP', '347 Cedar St', 'Lawrenceville', 'GA', '30044', '348-848-8291', '1938-09-11', 'M', current\_timestamp);

**18:15 To See Data**

Right click customer -> View / Edit Data -> All Rows

The id is added by default and auto incremented

**18:25 SELECT**

**19:19 Create Enumerated Type Custom Data Type**

Right Click sales\_db -> Query Tool

CREATE TYPE sex\_type as enum

('M', 'F');

It is located in Types

Alter Enum to add ‘O’ for other

Right Click sex\_type -> Properties -> Definition -> + (Add Row) and type ‘O’

**20:48 Change Column Data Type**

alter table customer

alter column sex type sex\_type USING sex::sex\_type;

**SLIDE I then also track information on the sales person**

**21:37 Add sales\_person Table**

CREATE TABLE sales\_person(

first\_name VARCHAR(30) NOT NULL,

last\_name VARCHAR(30) NOT NULL,

email VARCHAR(60) NOT NULL,

street VARCHAR(50) NOT NULL,

city VARCHAR(40) NOT NULL,

state CHAR(2) NOT NULL DEFAULT 'PA',

zip SMALLINT NOT NULL,

phone VARCHAR(20) NOT NULL,

birth\_date DATE NULL,

sex sex\_type NOT NULL,

date\_hired TIMESTAMP NOT NULL,

id SERIAL PRIMARY KEY

);

**22:58 Thinking About Tables**

Now we look at a description of a product which will be a shoe in this situation. We define if it is business, casual or athletic. Brand, individual shoe name, size, color, price, discount, tax rate, and quantity.

**24:00 Create a Table product\_type**

This table will define if a product is considered business, casual or athletic

CREATE TABLE product\_type(

name VARCHAR(30) NOT NULL,

id SERIAL PRIMARY KEY);

**25:30 Create Product Table SLIDE**

We talked about how a primary key is used to uniquely identify a row in a table. A foreign key is used to identify 1 of a group of possible rows in another table.

If we create a product table and want to store a value from the product type table we can reference that information using a foreign key.

When creating a foreign key it has an integer type instead of a serial type. We can’t use serial because Postgres will try to assign a value to serial types.

-- type\_id references rows in the table product\_id and the row we are referencing matches the id

-- column

CREATE TABLE product(

type\_id INTEGER REFERENCES product\_type(id),

name VARCHAR(30) NOT NULL,

supplier VARCHAR(30) NOT NULL,

description TEXT NOT NULL,

id SERIAL PRIMARY KEY);

**25:37 Breaking Up Tables**

**Table with Information that Differentiates Items of the Same Type NO SLIDE**

This table describes just the quality of an item. If I were to list quantity here it would

make it hard to look at this as a single item. Quantity should be kept in a completely different

table if needed.

Anything that gets in the way of being able to model an individual object should be put in

another table.

When dealing with prices it is recommended to define Precision (Total number of digits) and Scale (How many digits in fraction).

The picture will be a url to the picture.

CREATE TABLE item(

product\_id INTEGER REFERENCES product(id),

size INTEGER NOT NULL,

color VARCHAR(30) NOT NULL,

picture VARCHAR(256) NOT NULL,

price NUMERIC(6,2) NOT NULL,

id SERIAL PRIMARY KEY);

**27:03 Primary & Foreign Keys**

**30:28 Sales Order Table**

Only information pertaining to the order is here aside from products and prices

It simulates 2 people agreeing to do business, the time of that event, a purchase order and

the means of payment

CREATE TABLE sales\_order(

cust\_id INTEGER REFERENCES customer(id),

sales\_person\_id INTEGER REFERENCES sales\_person(id),

time\_order\_taken TIMESTAMP NOT NULL,

purchase\_order\_number INTEGER NOT NULL,

credit\_card\_number VARCHAR(16) NOT NULL,

credit\_card\_exper\_month SMALLINT NOT NULL,

credit\_card\_exper\_day SMALLINT NOT NULL,

credit\_card\_secret\_code SMALLINT NOT NULL,

name\_on\_card VARCHAR(100) NOT NULL,

id SERIAL PRIMARY KEY

);

**31:54 Sales Item Table SLIDE**

Each item that is part of an order goes in its own table. It is linked to the order with

sales\_order\_id

This simulates picking up a quantity of an individual item, with a certain discount and tax rate

The item itself is hidden in the item table and 100% defined specifically there in a way that

makes it easy to refer to it just by its id

If I were to list color, size or anything else here we would break that ability to consider

items in a self contained way

CREATE TABLE sales\_item(

item\_id INTEGER REFERENCES item(id),

sales\_order\_id INTEGER REFERENCES sales\_order(id),

quantity INTEGER NOT NULL,

discount NUMERIC(3,2) NULL DEFAULT 0,

taxable BOOLEAN NOT NULL DEFAULT FALSE,

sales\_tax\_rate NUMERIC(5,2) NOT NULL DEFAULT 0,

id SERIAL PRIMARY KEY

);

**32:40 Foreign & Primary Keys**

You can see here how foreign keys allow us to merge our data. When we start issuing queries it will become more clear how to use these keys.

Product type is linked to the product. The product is linked to the item which is a more specific version of our product. Then both the item and sales order is linked to the sales item table. There are many other foreign keys linking tables, but I think this is enough for now.

**33:28 Altering Tables Many Examples**

// Add a new column

ALTER TABLE sales\_item ADD day\_of\_week VARCHAR(8)

// Modify a column (Change any Constraint this Way)

ALTER TABLE sales\_item ALTER COLUMN day\_of\_week SET NOT NULL;

// Change name of a column

ALTER TABLE sales\_item RENAME COLUMN day\_of\_week TO weekday;

// Drop a column

ALTER TABLE sales\_item DROP COLUMN weekday;

// Add a new table

CREATE TABLE transaction\_type(

name VARCHAR(30) NOT NULL,

payment\_type VARCHAR(30) NOT NULL,

id SERIAL PRIMARY KEY

);

// Rename table

ALTER TABLE transaction\_type RENAME TO transaction;

// Create index based on a single column (Use UNIQUE INDEX for a unique index)

// Indexes show under indexes tab

CREATE INDEX transaction\_id ON transaction(name)

// Create an index based on 2 columns

CREATE INDEX transaction\_id\_2 ON transaction(name, payment\_type)

// Delete data in a table

TRUNCATE TABLE transaction

// Drop a table

DROP TABLE transaction

**39:42 Inserting Data**

INSERT INTO product\_type (name) VALUES ('Business');

INSERT INTO product\_type (name) VALUES ('Casual');

INSERT INTO product\_type (name) VALUES ('Athletic');

select \* from product\_type;

**Insert into Products**

-- You can also insert multiple rows without defining column names if you put the values in the same order as the table data.

INSERT INTO product VALUES

(1, 'Grandview', 'Allen Edmonds', 'Classic broguing adds texture to a charming longwing derby crafted in America from lustrous leather'),

(1, 'Clarkston', 'Allen Edmonds', 'Sharp broguing touches up a charming, American-made derby fashioned from finely textured leather'),

(1, 'Derby', 'John Varvatos', 'Leather upper, manmade sole'),

(1, 'Ramsey', 'Johnston & Murphy', 'Leather upper, manmade sole'),

(1, 'Hollis', 'Johnston & Murphy', 'Leather upper, manmade sole'),

(2, 'Venetian Loafer', 'Mezlan', 'Suede upper, leather sole'),

(2, 'Malek', 'Johnston & Murphy', 'Contrast insets at the toe and sides bring updated attitude to a retro-inspired sneaker set on a sporty foam sole and triangle-lugged tread.'),

(3, 'Air Max 270 React', 'Nike', 'The reggae inspired Nike Air 270 React fuses forest green with shades of tan to reveal your righteous spirit'),

(3, 'Joyride', 'Nike', 'Tiny foam beads underfoot conform to your foot for cushioning that stands up to your mileage'),

(2, 'Air Force 1', 'Nike', 'A modern take on the icon that blends classic style and fresh, crisp details'),

(3, 'Ghost 12', 'Brooks', 'Just know that it still strikes a just-right balance of DNA LOFT softness and BioMoGo DNA responsiveness'),

(3, 'Revel 3', 'Brooks', 'Style to spare, now even softer.'),

(3, 'Glycerin 17', 'Brooks', 'A plush fit and super soft transitions make every stride luxurious');

select \* from product;

**Insert Customers**

**43:51 Changing Column Data Type**

// Have to first change data type for zip

ALTER TABLE customer ALTER COLUMN zip TYPE INTEGER;

INSERT INTO customer (first\_name, last\_name, email, company, street, city, state, zip, phone, birth\_date, sex, date\_entered) VALUES

('Matthew', 'Martinez', 'matthewmartinez@ge.com', 'GE', '602 Main Place', 'Fontana', 'CA', '92336', '117-997-7764', '1931-09-04', 'M', '2015-01-01 22:39:28'),

('Melissa', 'Moore', 'melissamoore@aramark.com', 'Aramark', '463 Park Rd', 'Lakewood', 'NJ', '08701', '269-720-7259', '1967-08-27', 'M', '2017-10-20 21:59:29'),

('Melissa', 'Brown', 'melissabrown@verizon.com', 'Verizon', '712 View Ave', 'Houston', 'TX', '77084', '280-570-5166', '1948-06-14', 'F', '2016-07-16 12:26:45'),

('Jennifer', 'Thomas', 'jenniferthomas@aramark.com', 'Aramark', '231 Elm St', 'Mission', 'TX', '78572', '976-147-9254', '1998-03-14', 'F', '2018-01-08 09:27:55'),

('Stephanie', 'Martinez', 'stephaniemartinez@albertsons.com', 'Albertsons', '386 Second St', 'Lakewood', 'NJ', '08701', '820-131-6053', '1998-01-24', 'M', '2016-06-18 13:27:34'),

('Daniel', 'Williams', 'danielwilliams@tjx.com', 'TJX', '107 Pine St', 'Katy', 'TX', '77449', '744-906-9837', '1985-07-20', 'F', '2015-07-03 10:40:18'),

('Lauren', 'Anderson', 'laurenanderson@pepsi.com', 'Pepsi', '13 Maple Ave', 'Riverside', 'CA', '92503', '747-993-2446', '1973-09-09', 'F', '2018-02-01 16:43:51'),

('Michael', 'Jackson', 'michaeljackson@disney.com', 'Disney', '818 Pine Ave', 'Mission', 'TX', '78572', '126-423-3144', '1951-03-03', 'F', '2017-04-02 21:57:36'),

('Ashley', 'Johnson', 'ashleyjohnson@boeing.com', 'Boeing', '874 Oak Ave', 'Pacoima', 'CA', '91331', '127-475-1658', '1937-05-10', 'F', '2015-01-04 08:58:56'),

('Brittany', 'Thomas', 'brittanythomas@walmart.com', 'Walmart', '187 Maple Ave', 'Brownsville', 'TX', '78521', '447-788-4913', '1986-10-22', 'F', '2018-05-23 08:04:32'),

('Matthew', 'Smith', 'matthewsmith@ups.com', 'UPS', '123 Lake St', 'Brownsville', 'TX', '78521', '961-108-3758', '1950-06-16', 'F', '2018-03-15 10:08:54'),

('Lauren', 'Wilson', 'laurenwilson@target.com', 'Target', '942 Fifth Ave', 'Mission', 'TX', '78572', '475-578-8519', '1965-12-26', 'M', '2017-07-16 11:01:01'),

('Justin', 'Smith', 'justinsmith@boeing.com', 'Boeing', '844 Lake Ave', 'Lawrenceville', 'GA', '30044', '671-957-1492', '1956-03-16', 'F', '2017-10-07 10:50:08'),

('Jessica', 'Garcia', 'jessicagarcia@toyota.com', 'Toyota', '123 Pine Place', 'Fontana', 'CA', '92336', '744-647-2359', '1996-08-05', 'F', '2016-09-14 12:33:05'),

('Matthew', 'Jackson', 'matthewjackson@bp.com', 'BP', '538 Cedar Ave', 'Katy', 'TX', '77449', '363-430-1813', '1966-02-26', 'F', '2016-05-01 19:25:17'),

('Stephanie', 'Thomas', 'stephaniethomas@apple.com', 'Apple', '804 Fourth Place', 'Brownsville', 'TX', '78521', '869-582-9955', '1988-08-26', 'F', '2018-10-21 22:01:57'),

('Jessica', 'Jackson', 'jessicajackson@aramark.com', 'Aramark', '235 Pine Place', 'Chicago', 'IL', '60629', '587-334-1054', '1991-07-22', 'F', '2015-08-28 03:11:35'),

('James', 'Martinez', 'jamesmartinez@kroger.com', 'Kroger', '831 Oak St', 'Brownsville', 'TX', '78521', '381-428-3119', '1927-12-22', 'F', '2018-01-27 07:41:48'),

('Christopher', 'Robinson', 'christopherrobinson@ibm.com', 'IBM', '754 Cedar St', 'Pharr', 'TX', '78577', '488-694-7677', '1932-06-25', 'F', '2016-08-19 16:11:31');

select \* from customer;

**Insert Salespersons**

// Have to first change data type for zip

ALTER TABLE sales\_person ALTER COLUMN zip TYPE INTEGER;

INSERT INTO sales\_person (first\_name, last\_name, email, street, city, state, zip, phone, birth\_date, sex, date\_hired) VALUES

('Jennifer', 'Smith', 'jennifersmith@volkswagen.com', '610 Maple Place', 'Hawthorne', 'CA', '90250', '215-901-2287', '1941-08-09', 'F', '2014-02-06 12:22:48'),

('Michael', 'Robinson', 'michaelrobinson@walmart.com', '164 Maple St', 'Pacoima', 'CA', '91331', '521-377-4462', '1956-04-23', 'M', '2014-09-12 17:27:23'),

('Brittany', 'Jackson', 'brittanyjackson@disney.com', '263 Park Rd', 'Riverside', 'CA', '92503', '672-708-7601', '1934-07-05', 'F', '2015-01-17 02:51:55'),

('Samantha', 'Moore', 'samanthamoore@ge.com', '107 Pine Place', 'Houston', 'TX', '77084', '893-423-2899', '1926-05-05', 'M', '2015-11-14 22:26:21'),

('Jessica', 'Thompson', 'jessicathompson@fedex.com', '691 Third Place', 'Sylmar', 'CA', '91342', '349-203-4736', '1938-12-18', 'M', '2014-12-13 06:54:39');

**Insert Items**

INSERT INTO item VALUES

(2, 10, 'Gray', 'Coming Soon', 199.60),

(11, 12, 'Red', 'Coming Soon', 155.65),

(2, 11, 'Red', 'Coming Soon', 128.87),

(11, 11, 'Green', 'Coming Soon', 117.52),

(5, 8, 'Black', 'Coming Soon', 165.39),

(7, 11, 'Brown', 'Coming Soon', 168.15),

(5, 8, 'Gray', 'Coming Soon', 139.48),

(5, 11, 'Blue', 'Coming Soon', 100.14),

(4, 10, 'Brown', 'Coming Soon', 117.66),

(8, 10, 'Brown', 'Coming Soon', 193.53),

(7, 8, 'Light Brown', 'Coming Soon', 154.62),

(12, 10, 'Green', 'Coming Soon', 188.32),

(3, 12, 'Green', 'Coming Soon', 101.49),

(7, 9, 'Black', 'Coming Soon', 106.39),

(8, 12, 'Red', 'Coming Soon', 124.77),

(5, 8, 'Black', 'Coming Soon', 86.19),

(8, 12, 'Blue', 'Coming Soon', 196.86),

(8, 8, 'Blue', 'Coming Soon', 123.27),

(7, 11, 'Red', 'Coming Soon', 130.76),

(9, 12, 'Black', 'Coming Soon', 152.98),

(11, 8, 'Blue', 'Coming Soon', 175.58),

(7, 11, 'Light Brown', 'Coming Soon', 146.83),

(4, 8, 'Green', 'Coming Soon', 159.82),

(12, 8, 'Light Brown', 'Coming Soon', 171.92),

(1, 12, 'Light Brown', 'Coming Soon', 128.77),

(2, 10, 'Gray', 'Coming Soon', 102.45),

(10, 8, 'Green', 'Coming Soon', 186.86),

(1, 8, 'Blue', 'Coming Soon', 139.73),

(9, 8, 'Light Brown', 'Coming Soon', 151.57),

(2, 10, 'Green', 'Coming Soon', 177.16),

(3, 9, 'Gray', 'Coming Soon', 124.87),

(8, 8, 'Black', 'Coming Soon', 129.40),

(5, 9, 'Black', 'Coming Soon', 107.55),

(5, 8, 'Light Brown', 'Coming Soon', 103.71),

(11, 10, 'Green', 'Coming Soon', 152.31),

(6, 12, 'Red', 'Coming Soon', 108.96),

(7, 12, 'Blue', 'Coming Soon', 173.14),

(3, 10, 'Green', 'Coming Soon', 198.44),

(1, 9, 'Light Brown', 'Coming Soon', 119.61),

(1, 10, 'Black', 'Coming Soon', 114.36),

(7, 9, 'Light Brown', 'Coming Soon', 181.93),

(5, 10, 'Black', 'Coming Soon', 108.32),

(1, 12, 'Black', 'Coming Soon', 153.97),

(2, 12, 'Gray', 'Coming Soon', 184.27),

(2, 9, 'Blue', 'Coming Soon', 151.63),

(6, 8, 'Brown', 'Coming Soon', 159.39),

(11, 9, 'Red', 'Coming Soon', 150.49),

(9, 10, 'Gray', 'Coming Soon', 139.26),

(4, 8, 'Gray', 'Coming Soon', 166.87),

(12, 9, 'Red', 'Coming Soon', 110.77);

select \* from item;

**Insert into Sales Order**

Change purchase\_order\_number to BIGINT

INSERT INTO sales\_order VALUES

(1, 2, '2018-03-23 10:26:23', 20183231026, 5440314057399014, 3, 5, 415, 'Ashley Martin'),

(8, 2, '2017-01-09 18:58:15', 2017191858, 6298551651340835, 10, 27, 962, 'Michael Smith'),

(9, 3, '2018-12-21 21:26:57', 201812212126, 3194084144609442, 7, 16, 220, 'Lauren Garcia'),

(8, 2, '2017-08-20 15:33:17', 20178201533, 2704487907300646, 7, 10, 430, 'Jessica Robinson'),

(3, 4, '2017-09-19 13:28:35', 20179191328, 8102877849444788, 4, 15, 529, 'Melissa Jones'),

(14, 1, '2016-10-02 18:30:13', 20161021830, 7294221943676784, 10, 22, 323, 'Lauren Moore'),

(4, 2, '2016-03-21 07:24:30', 2016321724, 1791316080799942, 1, 24, 693, 'Joshua Wilson'),

(1, 1, '2018-08-04 12:22:06', 2018841222, 4205390666512184, 5, 16, 758, 'Jennifer Garcia'),

(8, 4, '2016-08-25 10:36:09', 20168251036, 3925972513042074, 1, 10, 587, 'Michael Thomas'),

(8, 4, '2018-08-10 20:24:52', 20188102024, 2515001187633555, 10, 7, 354, 'David Martin'),

(5, 2, '2016-11-28 15:21:48', 201611281521, 6715538212478349, 5, 25, 565, 'Jennifer Johnson'),

(5, 3, '2016-12-07 10:20:05', 20161271020, 5125085038984547, 10, 27, 565, 'Brittany Garcia'),

(13, 3, '2018-10-11 16:27:04', 201810111627, 5559881213107031, 7, 14, 593, 'Sarah Jackson'),

(14, 1, '2018-04-26 20:35:34', 20184262035, 2170089500922701, 7, 26, 105, 'Daniel Harris'),

(3, 2, '2016-11-14 04:32:50', 20161114432, 6389550669359545, 7, 19, 431, 'Brittany Williams'),

(18, 3, '2016-07-10 17:55:01', 20167101755, 7693323933630220, 4, 22, 335, 'Christopher Thomas'),

(12, 2, '2018-05-13 06:20:56', 2018513620, 1634255384507587, 1, 4, 364, 'Megan Garcia'),

(3, 4, '2016-03-04 20:52:36', 2016342052, 7720584466409961, 2, 7, 546, 'Justin Taylor'),

(17, 1, '2017-02-16 15:44:27', 20172161544, 7573753924723630, 3, 15, 148, 'Michael White'),

(19, 3, '2017-08-04 07:24:30', 201784724, 9670036242643402, 10, 24, 803, 'Melissa Taylor'),

(8, 2, '2018-07-08 15:51:11', 2018781551, 5865443195522495, 2, 2, 793, 'James Thompson'),

(18, 1, '2017-03-02 03:08:03', 20173238, 9500873657482557, 6, 22, 793, 'Daniel Williams'),

(7, 1, '2018-03-19 10:54:30', 20183191054, 7685678049357511, 2, 9, 311, 'Joshua Martinez'),

(18, 1, '2017-07-04 18:48:02', 2017741848, 2254223828631172, 6, 18, 621, 'Justin Taylor'),

(16, 1, '2018-07-23 21:44:51', 20187232144, 8669971462260333, 10, 3, 404, 'Ashley Garcia'),

(8, 4, '2016-05-21 16:26:49', 20165211626, 9485792104395686, 2, 4, 270, 'Andrew Taylor'),

(19, 4, '2018-09-04 18:24:36', 2018941824, 5293753403622328, 8, 4, 362, 'Matthew Miller'),

(9, 2, '2018-07-01 18:19:10', 2018711819, 7480694928317516, 10, 5, 547, 'Justin Thompson'),

(8, 4, '2018-09-10 20:15:06', 20189102015, 7284020879927491, 4, 15, 418, 'Samantha Anderson'),

(17, 2, '2016-07-13 16:30:53', 20167131630, 7769197595493852, 1, 19, 404, 'Jessica Thomas'),

(17, 4, '2016-09-22 22:58:11', 20169222258, 1394443435119786, 7, 5, 955, 'James Wilson'),

(17, 4, '2017-10-28 11:35:05', 201710281135, 6788591532433513, 8, 13, 512, 'Michael Williams'),

(12, 4, '2018-11-11 04:55:50', 20181111455, 1854718494260005, 3, 26, 928, 'Melissa Jones'),

(15, 4, '2016-08-11 23:05:58', 2016811235, 7502173302686796, 3, 11, 836, 'Michael Thompson'),

(2, 3, '2018-07-13 07:50:24', 2018713750, 5243198834590551, 10, 12, 725, 'Joseph Thomas'),

(9, 3, '2017-09-28 11:42:16', 20179281142, 7221309687109696, 2, 5, 845, 'James Martinez'),

(7, 1, '2016-01-09 18:15:08', 2016191815, 9202139348760334, 4, 4, 339, 'Samantha Wilson'),

(18, 1, '2016-03-14 17:33:26', 20163141733, 3066530074499665, 6, 23, 835, 'David Garcia'),

(12, 3, '2017-08-21 18:14:01', 20178211814, 1160849457958425, 8, 19, 568, 'Samantha Miller'),

(8, 1, '2018-09-12 19:25:25', 20189121925, 6032844702934349, 8, 13, 662, 'Justin Brown'),

(19, 2, '2016-11-06 03:07:33', 201611637, 1369214097312715, 9, 23, 330, 'Joseph Jones'),

(3, 4, '2016-06-06 01:07:15', 20166617, 7103644598069058, 1, 5, 608, 'Brittany Thomas'),

(13, 4, '2017-05-15 01:02:57', 201751512, 2920333635602602, 11, 14, 139, 'Stephanie Smith'),

(15, 4, '2016-03-27 02:18:18', 2016327218, 7798214190926405, 5, 13, 809, 'Stephanie Taylor'),

(9, 2, '2018-01-25 14:43:01', 20181251443, 4196223548846892, 10, 17, 115, 'Melissa Martin'),

(6, 3, '2017-01-08 13:54:49', 2017181354, 8095784052038731, 8, 23, 416, 'Amanda White'),

(12, 2, '2017-09-24 15:24:44', 20179241524, 6319974420646022, 2, 4, 755, 'Megan Anderson'),

(11, 2, '2018-04-09 18:53:22', 2018491853, 3258192259182097, 11, 22, 730, 'Samantha Thompson'),

(10, 2, '2018-01-11 22:20:29', 20181112220, 8336712415869878, 3, 18, 872, 'Melissa Wilson'),

(14, 3, '2018-11-10 03:08:36', 2018111038, 6942550153605236, 9, 18, 250, 'Jessica Johnson'),

(6, 4, '2016-06-26 16:48:19', 20166261648, 5789348928562200, 2, 7, 458, 'Christopher Jones'),

(5, 1, '2018-06-23 02:25:16', 2018623225, 8550095429571317, 9, 25, 590, 'Samantha Wilson'),

(18, 2, '2017-07-01 01:16:04', 201771116, 2651011719468438, 11, 11, 107, 'Andrew Miller'),

(12, 4, '2017-01-17 21:42:51', 20171172142, 7354378345646144, 3, 14, 772, 'Andrew Moore'),

(7, 3, '2016-01-07 22:56:31', 2016172256, 3429850164043973, 2, 6, 295, 'Joseph Taylor'),

(10, 1, '2016-01-27 01:14:53', 2016127114, 2480926933843246, 7, 3, 704, 'Ashley Taylor'),

(13, 1, '2018-09-15 08:15:17', 2018915815, 6626319262681476, 4, 8, 837, 'Stephanie Thomas'),

(9, 1, '2018-04-06 15:40:28', 2018461540, 4226037621059886, 10, 26, 896, 'Stephanie Jones'),

(17, 3, '2016-10-17 21:31:09', 201610172131, 7862008338119027, 10, 25, 767, 'Amanda Robinson'),

(12, 2, '2016-06-04 22:27:57', 2016642227, 4472081783581101, 10, 9, 279, 'Justin Williams'),

(9, 3, '2018-01-27 06:57:23', 2018127657, 2384491606066483, 11, 23, 417, 'Joshua Garcia'),

(14, 2, '2018-07-19 22:11:23', 20187192211, 2680467440231722, 10, 8, 545, 'Ashley Wilson'),

(19, 4, '2018-11-06 03:12:35', 2018116312, 3973342791188144, 10, 9, 749, 'Megan Martinez'),

(11, 2, '2017-01-15 14:11:54', 20171151411, 3042008865691398, 8, 3, 695, 'Brittany White'),

(10, 4, '2018-10-07 01:26:57', 2018107126, 7226038495242154, 8, 7, 516, 'Stephanie White'),

(12, 3, '2018-10-02 16:13:23', 20181021613, 7474287104417454, 11, 1, 184, 'Daniel Davis'),

(8, 1, '2018-08-12 23:54:52', 20188122354, 6454271840792089, 1, 19, 914, 'Michael Robinson'),

(11, 2, '2016-07-06 04:57:33', 201676457, 6767948287515839, 8, 7, 127, 'Samantha Anderson'),

(12, 2, '2018-09-06 10:34:03', 2018961034, 2724397042248973, 11, 11, 686, 'Ashley Harris'),

(16, 1, '2017-11-12 07:05:38', 2017111275, 4832060124173185, 11, 27, 697, 'Brittany White'),

(16, 4, '2016-06-08 17:38:18', 2016681738, 2187337846675221, 5, 9, 895, 'Megan Wilson'),

(3, 3, '2016-02-08 21:46:46', 2016282146, 8361948319742012, 6, 26, 157, 'Jessica Taylor'),

(8, 1, '2016-10-22 03:01:13', 2016102231, 1748352966511490, 8, 7, 815, 'Justin Davis'),

(5, 4, '2018-12-06 12:51:24', 20181261251, 3987075017699453, 7, 18, 557, 'Andrew Martinez'),

(4, 1, '2017-09-23 07:14:32', 2017923714, 4497706297852239, 2, 12, 756, 'Justin Moore'),

(5, 3, '2016-02-28 23:16:42', 20162282316, 9406399694013062, 1, 26, 853, 'Joseph Moore'),

(11, 4, '2016-05-24 14:37:36', 20165241437, 4754563147105980, 8, 8, 742, 'Amanda Brown'),

(1, 2, '2018-04-08 09:35:58', 201848935, 5031182534686567, 2, 11, 760, 'Andrew Thompson'),

(11, 1, '2017-10-07 20:45:13', 20171072045, 9736660892936088, 5, 19, 240, 'Megan Robinson'),

(19, 2, '2017-03-19 23:03:38', 2017319233, 1154891936822433, 2, 14, 554, 'Christopher Davis'),

(1, 1, '2018-04-26 11:58:53', 20184261158, 5672494499371853, 11, 18, 692, 'James Thomas'),

(1, 3, '2018-07-20 10:05:17', 2018720105, 9695318985866569, 2, 12, 107, 'Jennifer Martin'),

(7, 3, '2018-06-21 18:41:12', 20186211841, 2824438494479373, 1, 12, 296, 'Joseph Miller'),

(6, 1, '2016-04-07 08:47:40', 201647847, 5608599820055114, 7, 2, 163, 'Brittany Brown'),

(15, 3, '2016-07-22 19:25:23', 20167221925, 3011298350076480, 1, 9, 352, 'Jessica Jackson'),

(16, 4, '2016-10-14 10:17:30', 201610141017, 5250543218399397, 9, 16, 975, 'David Wilson'),

(3, 4, '2018-05-15 03:51:28', 2018515351, 8835896606865589, 11, 4, 675, 'Andrew Garcia'),

(19, 3, '2017-05-25 07:44:57', 2017525744, 9159566098395188, 6, 23, 112, 'Ashley Brown'),

(11, 2, '2017-12-02 19:07:39', 2017122197, 9920715756046783, 2, 25, 490, 'Joshua Garcia'),

(7, 4, '2016-05-01 04:50:28', 201651450, 8393790616940265, 9, 22, 490, 'Matthew White'),

(15, 3, '2018-01-21 19:54:46', 20181211954, 8078408967493993, 6, 18, 316, 'Jessica Thomas'),

(6, 1, '2018-04-11 11:23:58', 20184111123, 3921559263693643, 11, 17, 221, 'Andrew Jackson'),

(13, 3, '2018-03-05 10:26:27', 2018351026, 4739593984654108, 10, 18, 925, 'Samantha White'),

(8, 4, '2018-11-15 14:53:55', 201811151453, 8752393645304583, 4, 14, 554, 'Daniel Jackson'),

(10, 1, '2017-09-03 12:57:29', 2017931257, 3434269111389638, 6, 18, 360, 'Megan Johnson'),

(7, 1, '2018-06-28 12:10:58', 20186281210, 6543388006451934, 5, 19, 491, 'Megan Thomas'),

(15, 3, '2018-07-13 12:21:29', 20187131221, 4717498129166869, 5, 21, 386, 'Megan Davis'),

(4, 1, '2016-08-01 16:26:39', 2016811626, 1822404586758111, 3, 2, 346, 'Joseph Davis'),

(3, 2, '2016-10-27 10:53:05', 201610271053, 8446943405552052, 11, 17, 266, 'Daniel Smith'),

(18, 3, '2018-10-20 15:28:54', 201810201528, 6433477195769821, 8, 26, 723, 'Lauren Smith');

**Insert Sales Item**

INSERT INTO sales\_item VALUES

(24, 70, 2, 0.11, false, 0.0),

(8, 37, 2, 0.16, false, 0.0),

(24, 90, 2, 0.06, false, 0.0),

(34, 83, 2, 0.13, false, 0.0),

(26, 55, 2, 0.13, false, 0.0),

(19, 26, 1, 0.19, false, 0.0),

(23, 2, 1, 0.13, false, 0.0),

(48, 24, 2, 0.15, false, 0.0),

(30, 11, 2, 0.06, false, 0.0),

(1, 60, 2, 0.18, false, 0.0),

(48, 2, 2, 0.12, false, 0.0),

(35, 34, 2, 0.07, false, 0.0),

(29, 13, 1, 0.15, false, 0.0),

(15, 98, 2, 0.13, false, 0.0),

(27, 35, 2, 0.07, false, 0.0),

(30, 5, 1, 0.05, false, 0.0),

(45, 33, 1, 0.09, false, 0.0),

(31, 20, 1, 0.18, false, 0.0),

(32, 88, 1, 0.13, false, 0.0),

(47, 43, 1, 0.09, false, 0.0),

(23, 20, 2, 0.16, false, 0.0),

(44, 86, 2, 0.18, false, 0.0),

(35, 75, 2, 0.12, false, 0.0),

(24, 49, 1, 0.08, false, 0.0),

(31, 37, 1, 0.14, false, 0.0),

(21, 11, 2, 0.14, false, 0.0),

(21, 71, 2, 0.06, false, 0.0),

(48, 1, 1, 0.06, false, 0.0),

(37, 87, 1, 0.11, false, 0.0),

(38, 66, 1, 0.13, false, 0.0),

(14, 7, 2, 0.13, false, 0.0),

(26, 85, 2, 0.2, false, 0.0),

(21, 83, 2, 0.16, false, 0.0),

(8, 15, 2, 0.18, false, 0.0),

(40, 32, 1, 0.19, false, 0.0),

(49, 38, 1, 0.15, false, 0.0),

(41, 13, 2, 0.06, false, 0.0),

(36, 59, 1, 0.1, false, 0.0),

(14, 46, 2, 0.14, false, 0.0),

(30, 77, 2, 0.19, false, 0.0),

(12, 78, 2, 0.18, false, 0.0),

(5, 21, 1, 0.18, false, 0.0),

(10, 13, 1, 0.09, false, 0.0),

(39, 9, 2, 0.2, false, 0.0),

(46, 51, 2, 0.13, false, 0.0),

(47, 98, 1, 0.15, false, 0.0),

(25, 83, 2, 0.09, false, 0.0),

(36, 56, 2, 0.12, false, 0.0),

(18, 8, 2, 0.12, false, 0.0),

(35, 17, 1, 0.14, false, 0.0),

(41, 70, 1, 0.14, false, 0.0),

(9, 21, 1, 0.07, false, 0.0),

(42, 46, 1, 0.09, false, 0.0),

(18, 74, 1, 0.1, false, 0.0),

(25, 14, 1, 0.16, false, 0.0),

(44, 57, 1, 0.13, false, 0.0),

(2, 84, 2, 0.06, false, 0.0),

(18, 68, 2, 0.08, false, 0.0),

(35, 64, 2, 0.16, false, 0.0),

(49, 79, 1, 0.07, false, 0.0),

(7, 3, 2, 0.14, false, 0.0),

(42, 40, 2, 0.15, false, 0.0),

(8, 48, 2, 0.18, false, 0.0),

(27, 82, 2, 0.08, false, 0.0),

(21, 63, 1, 0.1, false, 0.0),

(42, 21, 2, 0.08, false, 0.0),

(31, 23, 2, 0.18, false, 0.0),

(29, 7, 1, 0.11, false, 0.0),

(48, 29, 2, 0.14, false, 0.0),

(15, 49, 2, 0.15, false, 0.0),

(34, 37, 1, 0.16, false, 0.0),

(22, 35, 1, 0.19, false, 0.0),

(22, 29, 2, 0.11, false, 0.0),

(38, 92, 2, 0.08, false, 0.0),

(21, 11, 2, 0.17, false, 0.0),

(13, 72, 1, 0.09, false, 0.0),

(12, 7, 1, 0.17, false, 0.0),

(41, 11, 2, 0.13, false, 0.0),

(22, 26, 2, 0.09, false, 0.0),

(43, 91, 1, 0.13, false, 0.0),

(33, 60, 1, 0.1, false, 0.0),

(39, 82, 2, 0.2, false, 0.0),

(27, 72, 2, 0.17, false, 0.0),

(10, 79, 2, 0.12, false, 0.0),

(41, 78, 2, 0.15, false, 0.0),

(11, 43, 1, 0.05, false, 0.0),

(29, 76, 1, 0.08, false, 0.0),

(25, 60, 1, 0.15, false, 0.0),

(15, 83, 2, 0.09, false, 0.0),

(7, 46, 1, 0.07, false, 0.0),

(26, 24, 2, 0.1, false, 0.0),

(43, 22, 2, 0.08, false, 0.0),

(47, 99, 1, 0.06, false, 0.0),

(29, 26, 1, 0.12, false, 0.0),

(36, 36, 2, 0.06, false, 0.0),

(41, 15, 1, 0.08, false, 0.0),

(12, 47, 2, 0.15, false, 0.0),

(38, 17, 1, 0.05, false, 0.0),

(22, 32, 2, 0.13, false, 0.0),

(12, 99, 2, 0.11, false, 0.0),

(30, 27, 2, 0.15, false, 0.0),

(38, 40, 1, 0.15, false, 0.0),

(22, 36, 1, 0.09, false, 0.0),

(14, 55, 2, 0.07, false, 0.0),

(1, 69, 1, 0.07, false, 0.0),

(47, 88, 1, 0.1, false, 0.0),

(7, 72, 2, 0.07, false, 0.0),

(46, 13, 1, 0.18, false, 0.0),

(9, 10, 1, 0.15, false, 0.0),

(35, 40, 1, 0.13, false, 0.0),

(15, 82, 2, 0.07, false, 0.0),

(47, 34, 1, 0.14, false, 0.0),

(10, 53, 1, 0.08, false, 0.0),

(49, 34, 2, 0.06, false, 0.0),

(13, 43, 1, 0.19, false, 0.0),

(6, 67, 1, 0.08, false, 0.0),

(21, 11, 1, 0.12, false, 0.0),

(26, 94, 1, 0.13, false, 0.0),

(38, 66, 1, 0.19, false, 0.0),

(40, 68, 2, 0.16, false, 0.0),

(25, 84, 1, 0.18, false, 0.0),

(11, 28, 1, 0.18, false, 0.0),

(48, 20, 1, 0.12, false, 0.0),

(26, 3, 1, 0.12, false, 0.0),

(1, 75, 1, 0.19, false, 0.0),

(6, 58, 1, 0.12, false, 0.0),

(33, 43, 2, 0.11, false, 0.0),

(15, 70, 1, 0.15, false, 0.0),

(41, 72, 2, 0.14, false, 0.0),

(8, 77, 2, 0.18, false, 0.0),

(36, 85, 2, 0.18, false, 0.0),

(42, 57, 2, 0.18, false, 0.0),

(27, 71, 1, 0.19, false, 0.0),

(20, 40, 1, 0.18, false, 0.0),

(14, 23, 2, 0.16, false, 0.0),

(15, 73, 1, 0.12, false, 0.0),

(25, 60, 1, 0.12, false, 0.0),

(30, 10, 2, 0.11, false, 0.0),

(18, 90, 2, 0.09, false, 0.0),

(17, 6, 2, 0.13, false, 0.0),

(43, 17, 1, 0.08, false, 0.0),

(20, 33, 2, 0.11, false, 0.0),

(1, 94, 2, 0.1, false, 0.0),

(49, 22, 2, 0.09, false, 0.0),

(1, 55, 2, 0.1, false, 0.0),

(24, 59, 1, 0.14, false, 0.0),

(19, 45, 1, 0.17, false, 0.0),

(13, 80, 2, 0.1, false, 0.0),

(17, 50, 1, 0.08, false, 0.0),

(45, 3, 2, 0.13, false, 0.0),

(6, 92, 2, 0.19, false, 0.0),

(25, 4, 1, 0.08, false, 0.0),

(47, 81, 1, 0.16, false, 0.0),

(39, 39, 2, 0.17, false, 0.0),

(47, 79, 1, 0.12, false, 0.0),

(6, 8, 1, 0.17, false, 0.0),

(15, 60, 2, 0.11, false, 0.0),

(49, 66, 1, 0.15, false, 0.0),

(34, 44, 2, 0.09, false, 0.0),

(20, 10, 1, 0.1, false, 0.0),

(13, 35, 1, 0.12, false, 0.0),

(10, 43, 1, 0.13, false, 0.0),

(24, 51, 2, 0.09, false, 0.0),

(11, 42, 2, 0.14, false, 0.0),

(20, 54, 1, 0.17, false, 0.0),

(42, 35, 1, 0.1, false, 0.0),

(1, 47, 2, 0.17, false, 0.0),

(35, 98, 1, 0.11, false, 0.0),

(14, 25, 1, 0.18, false, 0.0),

(23, 41, 2, 0.13, false, 0.0),

(4, 74, 2, 0.15, false, 0.0),

(32, 47, 2, 0.11, false, 0.0),

(49, 72, 2, 0.17, false, 0.0),

(37, 59, 2, 0.11, false, 0.0),

(43, 98, 1, 0.16, false, 0.0),

(26, 28, 1, 0.15, false, 0.0),

(16, 87, 1, 0.16, false, 0.0),

(6, 49, 2, 0.07, false, 0.0),

(6, 14, 2, 0.2, false, 0.0),

(27, 88, 1, 0.19, false, 0.0),

(37, 38, 1, 0.13, false, 0.0),

(44, 8, 1, 0.18, false, 0.0),

(49, 13, 1, 0.11, false, 0.0),

(30, 61, 2, 0.09, false, 0.0),

(33, 45, 2, 0.09, false, 0.0),

(24, 70, 2, 0.05, false, 0.0),

(42, 49, 2, 0.14, false, 0.0),

(43, 83, 1, 0.16, false, 0.0),

(39, 77, 2, 0.12, false, 0.0),

(1, 65, 1, 0.19, false, 0.0),

(42, 77, 1, 0.1, false, 0.0),

(2, 37, 2, 0.11, false, 0.0),

(24, 59, 2, 0.07, false, 0.0),

(42, 88, 1, 0.17, false, 0.0),

(45, 21, 1, 0.18, false, 0.0),

(10, 75, 2, 0.05, false, 0.0),

(15, 9, 2, 0.15, false, 0.0),

(24, 82, 2, 0.09, false, 0.0),

(30, 87, 1, 0.15, false, 0.0),

(22, 57, 1, 0.19, false, 0.0);

**53:00 Getting Data from One Table**

**53:40 Where**

Here we will learn about SELECT, FROM, WHERE, ORDER BY and LIMIT. You've seen a few of these already. Here I'll retrieve all data from from the table sales\_item.

SELECT \* FROM sales\_item;

WHERE is used to define which rows are included in the results based on a condition. Show all sales with a discount greater than 15%

**54:30 Conditional Operators**

= : Equal

< : Less than

> : Greater than

<= : Less than or Equal

>= : Greater than or Equal

<> : Not Equal

!= : Not Equal

SELECT \* FROM sales\_item WHERE discount > .15;

**55:48 Logical Operators**

AND, OR and NOT are logical operators. Use them to combine conditions. Find the order dates for all orders in December, 2018.

SELECT time\_order\_taken

FROM sales\_order

WHERE time\_order\_taken > '2018-12-01' AND time\_order\_taken < '2018-12-31';

You can use BETWEEN to get the same results

SELECT time\_order\_taken

FROM sales\_order

WHERE time\_order\_taken BETWEEN '2018-12-01' AND '2018-12-31';

**58:12 Order By**

ORDER BY determines which column is used to define the order of results. The default order is from low to high.

SELECT \* FROM sales\_item WHERE discount > .15 ORDER BY discount;

The following gives results from high to low

SELECT \* FROM sales\_item WHERE discount > .15 ORDER BY discount DESC;

**59:32 Limit**

LIMIT limits the number of rows in the result. Get just the top 5. You could use LIMIT 5, 10 to get the next 5

SELECT \* FROM sales\_item WHERE discount > .15 ORDER BY discount DESC LIMIT 5;

You can limit the results. Get the name, phone number and state where state is Texas. We can use CONCAT to merge to columns. We can then use AS to define a new column name.

SELECT CONCAT(first\_name, ' ', last\_name) AS Name, phone, state FROM customer WHERE state = 'TX';

You can perform calculations. Get the total value of all business shoes in inventory.

**1:01:45 GROUP BY**

SELECT product\_id, SUM(price) AS Total FROM item WHERE product\_id=1 GROUP BY product\_id;

**1:03:11 Distinct**

You can use distinct to eliminate duplicates in results. Get a list of states we have customers in.

SELECT DISTINCT state

FROM customer

ORDER BY state;

Find all states where we have customers not including 'CA'

SELECT DISTINCT state

FROM customer

WHERE state != 'CA'

ORDER BY state;

The IN phrase can be used to test if a value is in a list. Find customer states that are in my list. You can also use NOT IN.

SELECT DISTINCT state

FROM customer

WHERE state IN ('CA', 'NJ')

ORDER BY state;

**1:05:00 Getting Data from Multiple Tables**

We can get results from multiple tables with either inner joins, outer joins, or unions. The most common join is the inner join. You join data from 2 tables in the FROM clause with the JOIN keyword. The ON keyword is used to define the join condition. Get all items ordered ever and sort them by id while listing their price :

**1:05:21 Inner Join**

SELECT item\_id, price

FROM item INNER JOIN sales\_item

ON item.id = sales\_item.item\_id

ORDER BY item\_id;

We use the join condition to find ids that are equal in the tables item and sales\_item. These joins are normally done using the primary and foreign keys in the tables as we did here. When we join tables while checking for equality between a common column this is called a equijoin.

You can define multiple join conditions with logical operators :

SELECT item\_id, price

FROM item INNER JOIN sales\_item

ON item.id = sales\_item.item\_id

AND price > 120.00

ORDER BY item\_id;

**1:08:50 Join 3 Tables**

Now let's join 3 tables. Get the orders, quantity and the total sale.

SELECT sales\_order.id, sales\_item.quantity, item.price,

(sales\_item.quantity \* item.price) AS Total

FROM sales\_order

JOIN sales\_item

ON sales\_item.sales\_order\_id = sales\_order.id

JOIN item

ON item.id = sales\_item.item\_id

ORDER BY sales\_order.id;

**1:13:15 Arithmetic Operators**

Other arithmetic operators include :

Addition : +

Subtraction : -

Division : /

Integer Division : DIV

Modulus : %

**1:13:45 Join with Where**

You can also define the join conditions using WHERE, but this is not considered to be a best practice.

SELECT item\_id, price

FROM item, sales\_item

WHERE item.id = sales\_item.item\_id

AND price > 120.00

ORDER BY item\_id;

**1:14:55 Outer Joins**

Outer joins return all of the rows from one of the tables being joined even if no matches are found.

A Left Outer Join returns all rows from the table being joined on the left. The Right Outer Join returns all rows from the table on the right. It's common practice to avoid Right Outer joins though.

Here I'll get product information from 2 tables

SELECT name, supplier, price

FROM product LEFT JOIN item

ON item.product\_id = product.id

ORDER BY name;

**1:17:03 Cross Joins**

Cross joins include data from each row in both tables. I'll grab information from the item and sales\_item table. This will produce many results. Since there are no join conditions in a Cross Join you will rarely use them.

SELECT sales\_order\_id, quantity, product\_id

FROM item CROSS JOIN sales\_item

ORDER BY sales\_order\_id;

**1:18:16 Unions**

**1:19:27 Extract**

Unions combine the results of 2 or more select statements in one result. Each result must return the same number of columns and data in each column must have the same data type.

Let's say we want to send birthday cards to all customers and sales persons for the month of December we could do this. Always put the Order By statement last. The column names are taken from those provided in the 1st select statement. (We use Extract to get just the month from the birth date)

SELECT first\_name, last\_name, street, city, zip, birth\_date

FROM customer

WHERE EXTRACT(MONTH FROM birth\_date) = 12

UNION

SELECT first\_name, last\_name, street, city, zip, birth\_date

FROM sales\_person

WHERE EXTRACT(MONTH FROM birth\_date) = 12

ORDER BY birth\_date;

**1:21:05 IS NULL**

Null is used when a value is not known. IS NULL can be used to search for potential problems.

Search for items with NULL prices

SELECT product\_id, price

FROM item

WHERE price = NULL;

You can also use IS NOT NULL

**1:22:03 SIMILAR LIKE ~ & REGEXP**

SIMILAR is used to search for simple string matches. Match any customers whose name begins with M

SELECT first\_name, last\_name

FROM customer

WHERE first\_name SIMILAR TO 'M%';

% matches for zero or more characters.

**\_ Matches any single character.**

We will check if there is an Ashley with 5 \_

SELECT first\_name, last\_name

FROM customer

WHERE first\_name LIKE 'A\_\_\_\_\_';

Return all customers whose 1st name begins with D, or whose last name ends with an n

SELECT first\_name, last\_name

FROM customer

WHERE first\_name SIMILAR TO 'D%' OR last\_name SIMILAR TO '%n';

**REGEXP SLIDE**

REGEXP is used to search for complex patterns using regular expressions. Match 1st name that starts with Ma using the match operator

SELECT first\_name, last\_name

FROM customer

WHERE first\_name ~ '^Ma';

Match names that end with ez

SELECT first\_name, last\_name

FROM customer

WHERE last\_name ~ 'ez$';

Match last names that end with ez or son

SELECT first\_name, last\_name

FROM customer

WHERE last\_name ~ 'ez|son';

Last names that contain w, x, y, or z

SELECT first\_name, last\_name

FROM customer

WHERE last\_name ~ '[w-z]';

**SUMMARIZING RESULTS**

**1:29:25 GROUP BY** defines how the results are grouped. COUNT returns the total number of records that match.

We'll use GROUP BY to return a single row for each unique value. How many customers have birthdays in certain months

SELECT EXTRACT(MONTH FROM birth\_date) AS Month, COUNT(\*) AS Amount

FROM customer

GROUP BY Month

ORDER BY Month;

**1:31:14 HAVING** narrows the results based on a condition. Let's only get months if more than 1 person has a birthday that month

SELECT EXTRACT(MONTH FROM birth\_date) AS Month, COUNT(\*)

FROM customer

GROUP BY Month

HAVING COUNT(\*) > 1

ORDER BY Month;

**1:32:18 AGGREGATE FUNCTIONS**

Aggregate functions return a single value from multiple parameters. For example sum all our inventory

SELECT SUM(price)

FROM item;

Get count, sum, min, max and average value of our items

SELECT COUNT(\*) AS Items,

SUM(price) AS Value,

ROUND(AVG(price), 2) AS Avg,

MIN(price) AS Min,

MAX(price) AS Max

FROM item;

**1:34:22 WORKING WITH VIEWS**

Views are select statements thats result is stored in your database. Let's create a view that contains our main purchase order info.

CREATE VIEW purchase\_order\_overview AS

SELECT sales\_order.purchase\_order\_number, customer.company,

sales\_item.quantity, product.supplier, product.name, item.price,

--Can’t use total if you want this to be updated Fix Below

(sales\_item.quantity \* item.price) AS Total,

--Remove concat if you want this to be updatable

CONCAT(sales\_person.first\_name, ' ', sales\_person.last\_name) AS Salesperson

FROM sales\_order -- Join some tables

JOIN sales\_item

ON sales\_item.sales\_order\_id = sales\_order.id -- Tables go together by joining on sales order id

-- Any time you join tables you need to find foreign and primary keys that match up

JOIN item

ON item.id = sales\_item.item\_id -- Join item as well using matching item id

JOIN customer

ON sales\_order.cust\_id = customer.id // Join customer using customer ids

JOIN product

ON product.id = item.product\_id

JOIN sales\_person

ON sales\_person.id = sales\_order.sales\_person\_id

ORDER BY purchase\_order\_number;

When data in the database is updated so is the view. You can use the view in all the same ways you can a regular table. If you want it to be updatable though it can’t include DISTINCT, UNION, Aggregate Functions, GROUP BY or HAVING.

SELECT \* FROM purchase\_order\_overview;

**Recalculate Total**

If we removed total above so it could be updated we can just calculate with total like this

SELECT \*, (quantity \* price) AS Total

FROM purchase\_order\_overview;

**Drop a View**

DROP VIEW purchase\_order\_overview;

**1:45:01 SQL Functions**

You can write programs that are similar to traditional programming languages. There are different types of stored programs. Stored Functions can be executed by SQL statements.

After creating the function they appear in the functions folder. You can see info on the function by using properties on the function.

CREATE OR REPLACE FUNCTION fn\_add\_ints(int, int)

RETURNS int as

'

--$1 refers to 1st parameter and $2 the 2nd

--The result is passed back as a string

SELECT $1 + $2;

'

LANGUAGE SQL

Execute like this

SELECT fn\_add\_ints(4,5);

After creating the function they appear in the functions folder. You can see info on the function by using properties on the function.

**1:49:00 Dollar Quotes**

You are going to want to escape the quotes that surround your SQL so you can use quotes in your queries. $$ allows you to do this.

CREATE OR REPLACE FUNCTION fn\_add\_ints(int, int)

RETURNS int as

$body$

--$1 refers to 1st parameter and $2 the 2nd

SELECT $1 + $2;

$body$

LANGUAGE SQL

**1:50:06 Functions that Return Void**

Check if sales\_person has a state assigned and if not change it to ‘PA’

CREATE OR REPLACE FUNCTION fn\_update\_employee\_state()

RETURNS void as

$body$

UPDATE sales\_person

SET state = 'PA'

WHERE state is null

$body$

LANGUAGE SQL

SELECT fn\_update\_employee\_state();

**1:52:38 Get Maximum Product Price**

CREATE OR REPLACE FUNCTION fn\_max\_product\_price()

RETURNS numeric as

$body$

SELECT MAX(price)

FROM item

$body$

LANGUAGE SQL

SELECT fn\_max\_product\_price();

**1:53:39 Get Total Value of Inventory**

CREATE OR REPLACE FUNCTION fn\_get\_value\_inventory()

RETURNS numeric as

$body$

SELECT SUM(price)

FROM item;

$body$

LANGUAGE SQL

SELECT fn\_get\_value\_inventory();

**1:54:26 Get Number of Customers**

CREATE OR REPLACE FUNCTION fn\_number\_customers()

RETURNS numeric as

$body$

SELECT count(\*)

FROM customer;

$body$

LANGUAGE SQL

SELECT fn\_number\_customers();

**Get Number of Customers with No Phone**

CREATE OR REPLACE FUNCTION fn\_number\_customers\_no\_phone()

RETURNS numeric as

$body$

SELECT count(\*)

FROM customer

WHERE phone is NULL;

$body$

LANGUAGE SQL

SELECT fn\_number\_customers\_no\_phone();

**1:56:15 Named Parameters**

**Get Number of Customers from Texas using a Named Parameter**

CREATE OR REPLACE FUNCTION fn\_get\_number\_customers\_from\_state(state\_name char(2))

RETURNS numeric as

$body$

SELECT count(\*)

FROM customer

WHERE state = state\_name;

$body$

LANGUAGE SQL

SELECT fn\_get\_number\_customers\_from\_state('TX');

**Get Number of Orders Using Customer Name**

SELECT COUNT(\*)

FROM sales\_order

NATURAL JOIN customer

WHERE customer.first\_name = 'Christopher' AND customer.last\_name = 'Jones';

CREATE OR REPLACE FUNCTION fn\_get\_number\_orders\_from\_customer(cus\_fname varchar, cus\_lname varchar)

RETURNS numeric as

$body$

SELECT COUNT(\*)

FROM sales\_order

NATURAL JOIN customer

WHERE customer.first\_name = cus\_fname AND customer.last\_name = cus\_lname;

$body$

LANGUAGE SQL

SELECT fn\_get\_number\_orders\_from\_customer('Christopher', 'Jones');

**2:01:30 Return a Row / Composite for the Latest Order**

CREATE OR REPLACE FUNCTION fn\_get\_last\_order()

RETURNS sales\_order as

$body$

SELECT \*

FROM sales\_order

ORDER BY time\_order\_taken DESC

LIMIT 1;

$body$

LANGUAGE SQL

SELECT fn\_get\_last\_order();

--Get in table format

SELECT (fn\_get\_last\_order()).\*;

--Get just the date

SELECT (fn\_get\_last\_order()).\*;

**2:03:38 Get Multiple Rows All Employees in CA**

SELECT \*

FROM sales\_person

WHERE state = 'CA';

CREATE OR REPLACE FUNCTION fn\_get\_employees\_location(loc varchar)

RETURNS SETOF sales\_person as

$body$

SELECT \*

FROM sales\_person

WHERE state = loc;

$body$

LANGUAGE SQL

SELECT (fn\_get\_employees\_location('CA')).\*;

--Get names and phone number using function results

SELECT first\_name, last\_name, phone

FROM fn\_get\_employees\_location('CA');

**2:07:08 PL/pgSQL**

PL/pgSQL is influenced by Oracle SQL. It allows for loops, conditionals, functions, data types and much more.

CREATE OR REPLACE FUNCTION func\_name(parameter par\_type) RETURNS ret\_type AS

$body$

BEGIN

--statements

END

$body$

LANGUAGE plpqsql

**Get Product Price by Name**

SELECT item.price

FROM item

NATURAL JOIN product

WHERE product.name = 'Grandview';

CREATE OR REPLACE FUNCTION fn\_get\_price\_product\_name(prod\_name varchar)

RETURNS numeric AS

$body$

BEGIN

RETURN item.price

FROM item

NATURAL JOIN product

WHERE product.name = prod\_name;

END

$body$

LANGUAGE plpgsql

SELECT fn\_get\_price\_product\_name('Grandview');

**2:11:35 Using Variables in Functions**

--Create variables in functions

CREATE OR REPLACE FUNCTION fn\_get\_sum(val1 int, val2 int)

RETURNS int AS

$body$

--Put variables here

DECLARE

ans int;

BEGIN

ans := val1 + val2;

RETURN ans;

END;

$body$

LANGUAGE plpgsql

SELECT fn\_get\_sum(4,5);

**Assign Variable Value with a Query**

--Get random number and assign it to a variable

CREATE OR REPLACE FUNCTION fn\_get\_random\_number(min\_val int, max\_val int)

RETURNS int AS

$body$

--Put variables here

DECLARE

rand int;

BEGIN

SELECT random()\*(max\_val - min\_val) + min\_val INTO rand;

RETURN rand;

END;

$body$

LANGUAGE plpgsql

SELECT fn\_get\_random\_number(1, 5);

**2:15:55 Store Rows in Variables & Concat**

--Get random sales person name

CREATE OR REPLACE FUNCTION fn\_get\_random\_salesperson()

RETURNS varchar AS

$body$

--Put variables here

DECLARE

rand int;

--Use record to store row data

emp record;

BEGIN

--Generate random number

SELECT random()\*(5 - 1) + 1 INTO rand;

--Get row data for a random sales person and store in emp

SELECT \*

FROM sales\_person

INTO emp

WHERE id = rand;

--Concat the first and last name and return it

RETURN CONCAT(emp.first\_name, ' ', emp.last\_name);

END;

$body$

LANGUAGE plpgsql

SELECT fn\_get\_random\_salesperson();

**2:19:17 IN INOUT and OUT**

--These can be used to except and return multiple values without return

CREATE OR REPLACE FUNCTION fn\_get\_sum\_2(IN v1 int, IN v2 int, OUT ans int) AS

$body$

BEGIN

ans := v1 + v2;

END;

$body$

LANGUAGE plpgsql

SELECT fn\_get\_sum\_2(4,5);

**2:21:01 Using Multiple Outs**

-- Get a customer born in given month

CREATE OR REPLACE FUNCTION fn\_get\_cust\_birthday(IN the\_month int, OUT bd\_month int, OUT bd\_day int, OUT f\_name varchar, OUT l\_name varchar) AS

$body$

BEGIN

SELECT EXTRACT(MONTH FROM birth\_date), EXTRACT(DAY FROM birth\_date),

first\_name, last\_name

INTO bd\_month, bd\_day, f\_name, l\_name

FROM customer

WHERE EXTRACT(MONTH FROM birth\_date) = the\_month

LIMIT 1;

END;

$body$

LANGUAGE plpgsql

SELECT fn\_get\_cust\_birthday(12);

**2:25:56 Return Query Results**

--Return sales person data using a Query

CREATE OR REPLACE FUNCTION fn\_get\_sales\_people()

RETURNS SETOF sales\_person AS

$body$

BEGIN

RETURN QUERY

SELECT \*

FROM sales\_person;

END;

$body$

LANGUAGE plpgsql

SELECT (fn\_get\_sales\_people()).\*;

**Return Specific Data from Query Using Multiple Tables**

--Get top 10 most expensive products

SELECT product.name, product.supplier, item.price

FROM item

NATURAL JOIN product

ORDER BY item.price DESC

LIMIT 10;

CREATE OR REPLACE FUNCTION fn\_get\_10\_expensive\_prods()

RETURNS TABLE (

name varchar,

supplier varchar,

price numeric

) AS

$body$

BEGIN

RETURN QUERY

SELECT product.name, product.supplier, item.price

FROM item

NATURAL JOIN product

ORDER BY item.price DESC

LIMIT 10;

END;

$body$

LANGUAGE plpgsql

SELECT (fn\_get\_10\_expensive\_prods()).\*;

**2:33:42 IF ELSEIF and ELSE**

**Check order status with IF ELSEIF and ELSE**

--Check order performance with IF ELSEIF and ELSE

CREATE OR REPLACE FUNCTION fn\_check\_month\_orders(the\_month int)

RETURNS varchar AS

$body$

--Put variables here

DECLARE

total\_orders int;

BEGIN

--Check total orders

SELECT COUNT(purchase\_order\_number)

INTO total\_orders

FROM sales\_order

WHERE EXTRACT(MONTH FROM time\_order\_taken) = the\_month;

--Use conditionals to provide different output

IF total\_orders > 5 THEN

RETURN CONCAT(total\_orders, ' Orders : Doing Good');

ELSEIF total\_orders < 5 THEN

RETURN CONCAT(total\_orders, ' Orders : Doing Bad');

ELSE

RETURN CONCAT(total\_orders, ' Orders : On Target');

END IF;

END;

$body$

LANGUAGE plpgsql

SELECT fn\_check\_month\_orders(12);

**2:38:48 CASE Statement**

--Do the same using the case statement

--Check order performance with IF ELSEIF and ELSE

CREATE OR REPLACE FUNCTION fn\_check\_month\_orders(the\_month int)

RETURNS varchar AS

$body$

--Put variables here

DECLARE

total\_orders int;

BEGIN

--Check total orders

SELECT COUNT(purchase\_order\_number)

INTO total\_orders

FROM sales\_order

WHERE EXTRACT(MONTH FROM time\_order\_taken) = the\_month;

-- Case executes different code depending on an exact value

-- for total\_orders or a range of values

CASE

WHEN total\_orders < 1 THEN

RETURN CONCAT(total\_orders, ' Orders : Terrible');

WHEN total\_orders > 1 AND total\_orders < 5 THEN

RETURN CONCAT(total\_orders, ' Orders : Get Better');

WHEN total\_orders = 5 THEN

RETURN CONCAT(total\_orders, ' Orders : On Target');

ELSE

RETURN CONCAT(total\_orders, ' Orders : Doing Good');

END CASE;

END;

$body$

LANGUAGE plpgsql

SELECT fn\_check\_month\_orders(11);

**2:42:01 Loop Statement**

LOOP

Statements

EXIT WHEN condition is true;

END LOOP;

You can also exit with EXIT; with no condition

--Sum values up to a max number using

CREATE OR REPLACE FUNCTION fn\_loop\_test(max\_num int)

RETURNS int AS

$body$

--Put variables here

DECLARE

j INT DEFAULT 1;

tot\_sum INT DEFAULT 0;

BEGIN

LOOP

tot\_sum := tot\_sum + j;

j := j + 1;

EXIT WHEN j > max\_num;

END LOOP;

RETURN tot\_sum;

END;

$body$

LANGUAGE plpgsql

SELECT fn\_loop\_test(5);

**2:45:20 FOR LOOP**

Iterates over range of values or data coming from a table.

FOR counter\_name IN start\_value .. end\_value BY stepping

LOOP

Statements

END LOOP;

--Sum odd values up to a max number

CREATE OR REPLACE FUNCTION fn\_for\_test(max\_num int)

RETURNS int AS

$body$

--Put variables here

DECLARE

tot\_sum INT DEFAULT 0;

BEGIN

FOR i IN 1 .. max\_num BY 2

LOOP

tot\_sum := tot\_sum + i;

END LOOP;

RETURN tot\_sum;

END;

$body$

LANGUAGE plpgsql

SELECT fn\_for\_test(5);

You can also count in reverse with FOR i IN REVERSE max\_num .. 1 BY 2

**2:48:34 For Loops with Result Sets Blocks and Raise Notice**

--Use a bloc to test this

DO

$$

DECLARE

rec record;

BEGIN

FOR rec IN

SELECT first\_name, last\_name

FROM sales\_person

LIMIT 5

LOOP

--Outputs info to Messages

RAISE NOTICE '%, %', rec.first\_name, rec.last\_name;

END LOOP;

END;

$$

LANGUAGE plpgsql

**2:51:11 For Each and Arrays**

FOREACH var IN ARRAY array\_name

-- Print all values in the array

DO

$body$

DECLARE

arr1 int[] := array[1,2,3];

i int;

BEGIN

FOREACH i IN ARRAY arr1

LOOP

RAISE NOTICE '%', i;

END LOOP;

END;

$body$

LANGUAGE plpgsql

**2:53:20 While Loop**

-- Sums values as long as a condition is true

DO

$body$

DECLARE

j INT DEFAULT 1;

tot\_sum INT DEFAULT 0;

BEGIN

WHILE j <= 10

LOOP

tot\_sum := tot\_sum + j;

j := j + 1;

END LOOP;

RAISE NOTICE '%', tot\_sum;

END;

$body$

LANGUAGE plpgsql

**2:54:54 Continue**

--Prints the odd numbers from 1 to 10

DO

$$

DECLARE

i int DEFAULT 1;

BEGIN

LOOP

i := i + 1;

EXIT WHEN i > 10;

CONTINUE WHEN MOD(i, 2) = 0;

RAISE NOTICE 'Num : %', i;

END LOOP;

END;

$$

LANGUAGE plpgsql

**Return Inventory Value by Supplier**

CREATE OR REPLACE FUNCTION fn\_get\_supplier\_value(the\_supplier varchar)

RETURNS varchar AS

$body$

DECLARE

supplier\_name varchar;

price\_sum numeric;

BEGIN

SELECT product.supplier, SUM(item.price)

INTO supplier\_name, price\_sum

FROM product, item

WHERE product.supplier = the\_supplier

GROUP BY product.supplier;

RETURN CONCAT(supplier\_name, ' Inventory Value : $', price\_sum);

END;

$body$

LANGUAGE plpgsql

SELECT fn\_get\_supplier\_value('Nike');

3:01:34

**----- Stored Procedures -----**

Stored Procedures can be executed by an application that has access to your database. For example PHP could call for this code to execute.

Stored procedures can also execute transactions, which you cannot do with functions. Procedures however traditionally can’t return values, but there is a work around with INOUT.

Procedures also can’t be called by Select. You can execute them with EXECUTE with parameters, or with CALL. If a SP doesn’t have parameters it is called static and those with parameters are called dynamic.

CREATE OR REPLACE PROCEDURE procedure\_name(parameters)

AS

$body$

DECLARE

BEGIN

END;

$body$

LANGUAGE PLPGSQL;

-- Create a sample table that stores customer ids with balances due

CREATE TABLE past\_due(

id SERIAL PRIMARY KEY,

cust\_id INTEGER NOT NULL,

balance NUMERIC(6,2) NOT NULL);

SELECT \* FROM customer;

INSERT INTO past\_due(cust\_id, balance)

VALUES

(1, 123.45),

(2, 324.50);

SELECT \* FROM past\_due;

CREATE OR REPLACE PROCEDURE pr\_debt\_paid(

past\_due\_id int,

payment numeric

)

AS

$body$

DECLARE

BEGIN

UPDATE past\_due

SET balance = balance - payment

WHERE id = past\_due\_id;

COMMIT;

END;

$body$

LANGUAGE PLPGSQL;

-- Execute procedure

CALL pr\_debt\_paid(1, 10.00);

SELECT \* FROM past\_due;

pr\_debt\_paid(

past\_due\_id int,

payment numeric,

INOUT msg VARCHAR

);

3:09:35

**----- TRIGGERS -----**

Triggers are used when you want an action to automatically occur when an event occurs. Common events include the commands insert, update, delete and truncate. Triggers can also be associated with tables, foreign tables or views.

Triggers can execute before or after an event executes. Triggers also can execute instead of another event.

You can put multiple triggers on a table and they execute in alphabetical order. They can’t be triggered manually by a user. Triggers also can’t receive parameters.

If a Trigger is Row Level the Trigger is called for each row that is modified. If a Trigger is Statement level it will execute once regardless of the number of rows.

**When can you perform certain actions with triggers SLIDE**

This table shows what triggers can execute based on when they are to execute.

For example if a trigger is to execute Before if an event is insert, update, or delete it can perform actions on tables if row level and on tables or views if at statement level.

**Pros of Triggers SLIDE**

* Can be used for auditing, so if something is deleted a trigger could save it in case it is needed later
* They can be used to validate data
* Make certain events always happen to maintain integrity of data
* Insure integrity between different databases
* They can call functions or procedures
* Triggers are recursive so a trigger on a table can call another table with a trigger

**Cons of Triggers SLIDE**

* Triggers add execution overhead
* Nested / recursive trigger errors can be hard to debug
* Invisible to the client which can cause confusion when actions aren’t allowed

-- Create trigger function

CREATE FUNCTION trigger\_function()

RETURNS TRIGGER

LANGUAGE PLPGSQL

AS

$body$

BEGIN

END;

$body$

-- Create trigger

CREATE TRIGGER trigger\_name

{BEFORE | AFTER} {event} -- Event : insert, update, insert

ON table\_name

[FOR [EACH] {ROW | STATEMENT}]

EXECUTE PROCEDURE trigger\_function

**Trigger Data Logging / Auditing**

-- Log changes to distributor table

CREATE TABLE distributor(

id SERIAL PRIMARY KEY,

name VARCHAR(100));

-- Insert distributors

INSERT INTO distributor (name) VALUES

('Parawholesale'),

('J & B Sales'),

('Steel City Clothing');

SELECT \* FROM distributor;

-- Table that stores changes to distributor

CREATE TABLE distributor\_audit(

id SERIAL PRIMARY KEY,

dist\_id INT NOT NULL,

name VARCHAR(100) NOT NULL,

edit\_date TIMESTAMP NOT NULL);

-- Create trigger function

CREATE OR REPLACE FUNCTION fn\_log\_dist\_name\_change()

RETURNS TRIGGER

LANGUAGE PLPGSQL

AS

$body$

BEGIN

-- If name changes log the change

IF NEW.name <> OLD.name THEN

INSERT INTO distributor\_audit

(dist\_id, name, edit\_date)

VALUES

(OLD.id, OLD.name, NOW());

END IF;

-- Trigger information Variables

RAISE NOTICE 'Trigger Name : %', TG\_NAME;

RAISE NOTICE 'Table Name : %', TG\_TABLE\_NAME;

RAISE NOTICE 'Operation : %', TG\_OP;

RAISE NOTICE 'When Executed : %', TG\_WHEN;

RAISE NOTICE 'Row or Statement : %', TG\_LEVEL;

RAISE NOTICE 'Table Schema : %', TG\_TABLE\_SCHEMA;

-- Return the updated new data

RETURN NEW;

END;

$body$

-- Bind function to trigger

CREATE TRIGGER tr\_dist\_name\_changed

-- Call function before name is updated

BEFORE UPDATE

ON distributor

-- We want to run this on every row where an update occurs

FOR EACH ROW

EXECUTE PROCEDURE fn\_log\_dist\_name\_change();

-- Update distributor name and log changes

UPDATE distributor

SET name = 'Western Clothing'

WHERE id = 2;

-- Check the log

SELECT \* FROM distributor\_audit;

**Conditional Triggers**

You can revoke delete on tables for some users, or you can use triggers.

-- Block insert, update and delete on the weekend

CREATE OR REPLACE FUNCTION fn\_block\_weekend\_changes()

RETURNS TRIGGER

LANGUAGE PLPGSQL

AS

$body$

BEGIN

RAISE NOTICE 'No database changes allowed on the weekend';

RETURN NULL;

END;

$body$

-- Bind function to trigger

CREATE TRIGGER tr\_block\_weekend\_changes

-- Call function before name is updated

BEFORE UPDATE OR INSERT OR DELETE OR TRUNCATE

ON distributor

-- We want to run this on statement level

FOR EACH STATEMENT

-- Block if on weekend

WHEN(

EXTRACT('DOW' FROM CURRENT\_TIMESTAMP) BETWEEN 6 AND 7

)

EXECUTE PROCEDURE fn\_block\_weekend\_changes();

UPDATE distributor

SET name = 'Western Clothing'

WHERE id = 2;

-- Drop triggers

DROP EVENT TRIGGER tr\_block\_weekend\_changes;

3:29:25

**----- CURSORS -----**

Cursors are used to step backwards or forwards through rows of data. They can be pointed at a row and then select, update or delete. Cursor gets data, pushes it to another language for processing operations that add, edit, or delete.

Cursors are first declared defining the selection options to be used. It is then opened so it retrieves the data. Then individual rows can be fetched. After use the cursor is closed freeing memory. When needed the cursor can be used as needed.

-- Declare cursor

DECLARE cursor\_name refcursor;

-- Cursor that references all the product data

DECLARE cur\_products refcursor;

-- Declare cursor tied to a query / SELECT

-- SCROLL / NO SCROLL : Whether it can scroll backward or not

-- The query is a SELECT statement

cursor\_name [scrollability] CURSOR (parameter datatype, ...) FOR the\_query

-- It is best to get as small a data set as possible

DECLARE cur\_products CURSOR FOR

SELECT name, supplier FROM product;

-- Create cursor that takes parameters

DECLARE cur\_products CURSOR (company)

FOR

SELECT name, supplier

FROM product

WHERE supplier = company;

**Opening Cursors**

-- Bound & Unbound Cursors

-- Create an unbound cursor that can be bound to any query

OPEN ub\_cursor\_var [NO SCROLL | SCROLL] FOR query;

select \* from customer;

DECLARE cur\_customers refcursor;

OPEN cur\_customers

FOR

SELECT first\_name, last\_name, phone, state

FROM customer

WHERE state = 'CA';

-- Create an unbound cursor and attach a query

OPEN ub\_cursor\_var [NO SCROLL | SCROLL]

FOR EXECUTE

query;

-- Bound Cursor

-- Since it is bound to a query we only pass arguments when we open it if required

OPEN bound\_cur\_name (para:=val,...);

OPEN cur\_customers;

**Example with Cursors**

DO

$body$

DECLARE

msg text DEFAULT '';

rec\_customer record;

-- Declare cursor with customer data

cur\_customers CURSOR

FOR

SELECT \* FROM customer;

BEGIN

-- Open cursor

OPEN cur\_customers;

LOOP

-- Fetch records from cursor

FETCH cur\_customers INTO rec\_customer;

-- Loop until nothing more is found

EXIT WHEN NOT FOUND;

-- Concatenates all customer names together

msg := msg || rec\_customer.first\_name || ' ' || rec\_customer.last\_name || ', ';

END LOOP;

RAISE NOTICE 'Customers : %', msg;

END;

$body$

**Using Cursors with Functions**

-- Cursurs & Functions

-- Function returns a list of all customers in provided state

CREATE OR REPLACE FUNCTION fn\_get\_cust\_by\_state(c\_state varchar)

RETURNS text

LANGUAGE PLPGSQL

AS

$body$

DECLARE

cust\_names text DEFAULT '';

rec\_customer record;

cur\_cust\_by\_state CURSOR (c\_state varchar)

FOR

SELECT

first\_name, last\_name, state

FROM customer

WHERE state = c\_state;

BEGIN

-- Open cursor and pass the parameter

OPEN cur\_cust\_by\_state(c\_state);

LOOP

-- Move row of data to rec\_customer

FETCH cur\_cust\_by\_state INTO rec\_customer;

-- Loop until nothing more is found

EXIT WHEN NOT FOUND;

-- Concat customer name for each row

cust\_names := cust\_names || rec\_customer.first\_name || ' ' || rec\_customer.last\_name || ', ';

END LOOP;

-- Close cursor

CLOSE cur\_cust\_by\_state;

RETURN cust\_names;

END;

$body$

SELECT fn\_get\_cust\_by\_state('CA');

INSTALLATION