

# More SQL



Still Chapter 7!

# More Data Manipulation

- **ALTER TABLE** is going to be the workhorse
- No general syntax because it's too general, let's look at specifics

# Data Changes

- **ALTER TABLE** *table* **MODIFY** (*column datatype*) [*constraints*]
- Constraints are things like not null, unique, etc.
- Changing the type
  - Depends on DBMS whether this works
  - The column may have to be empty
  - Referential integrity must be kept!
- Changing the type's characteristics is always ok
  - Example, changing the amount of decimal precision

# Column Changes

- Can add or drop columns
- **ALTER TABLE *table* ADD (*column datatype*)**
  - Row entries get NULL
  - If you want NOT NULL or other constraints, add these after creating the column (see next slide)
- **ALTER TABLE *table* DROP COLUMN *column***
  - Referential integrity must be kept!
  - What about deleting the only column in a table?

# Renaming

- **ALTER TABLE *table* RENAME TO *newtable***
- What about columns?
  - Sorry about your luck!

# Adding Constraints

- **ALTER TABLE *table* ADD CONSTRAINT**  
constraintname **CONSTRAINTTYPE** *data*
  - The 'constrainttype' can be any of the constraints we learned about for CREATE TABLE: keys, unique, check, etc.
  - 'data' is whatever needs to go along with the constraint type
- ALTER TABLE product ADD CONSTRAINT  
chk\_nonzero CHECK p\_code > 0
- ALTER TABLE product ADD CONSTRAINT  
pk\_pcode PRIMARY KEY (p\_code)

# Making tables from other tables

- If you already have a table created...

```
INSERT INTO table[( columns1 )]  
SELECT columns2 FROM table2
```

- Data types must match and columns must be in same order
- If you haven't created the table yet...

```
SELECT col [AS newname][, more columns]  
INTO table  
FROM table2
```

  - This copies the data as well as the column definitions

# An example of that

- `SELECT p_code AS pcode, v_code AS vcode,  
p_min, p_qoh, p_descript AS descript  
INTO my_new_table  
FROM Product`

# Getting rid of tables

- **DROP TABLE** *table*
- Note that your DBMS may not let you drop a table that contains data

# Improving SELECT

- So far we've had just a taste of what SELECT can do
  - We know we can get a set of rows for any columns in a table, possibly based on some conditions we specify in the WHERE clause using boolean comparisons
- That's not all that great though is it, where's the POWER?



# Beyond WHERE

- Data will almost never be in a sensible order so we have to order it when we view it
- Syntax →  
**SELECT** *columnlist*  
**FROM** *table*  
**[WHERE** *conditionlist*  
**[ORDER BY** *columnlist* **[ASC|DESC]**
- If more than one column is specified you sort by the first column, and then within those sorted sets you sort by the next column, and so on

# Finding unique values

- What if I want to know all the vendors we carry?
- What if I want to know which vendors we have products for?
- What about vendors with products that are in stock?
- To answer the second two we need **DISTINCT**

# DISTINCT

- Syntax →  
**SELECT DISTINCT** *column[s]*  
**FROM** *table*
- When specifying more than one column, it finds all the distinct combinations
- Note that NULL counts as a distinct value
- Answers →
  - SELECT v\_name FROM Vendor
  - SELECT DISTINCT v\_code FROM Product
  - SELECT DISTINCT v\_code FROM Product WHERE p\_qoh > 0

# Aggregators

- What if we only need the *number* of vendors with current products rather than their codes?
- **COUNT** Syntax →  
**SELECT COUNT**(*option*)  
(rest of a select... from, where, etc.)
- The option can either be a column or \*
  - \* gives a count of all rows returned, even nulls
  - Column gives a count of all non-null rows returned;
- In the case of using a column can also specify an expression, such as using DISTINCT with it

# High and low

- To find the highest or lowest value for a column use **MAX** and **MIN**
- Syntax →  
**SELECT MAX**(*column*)  
**FROM** *table*
- Similar for **MIN**

# Gotcha

- It is easy to try something like the following to get the row that is the max value for a column:  
**SELECT \***  
**FROM** *table*  
**WHERE** *column* = **MAX**(*column*)
- Max/min can only be used in the column listing of the select, not in the where clause
- Solution: a nested query →  
**SELECT \***  
**FROM** *table*  
**WHERE** *column* = (**SELECT** **MAX**(*column*)  
**FROM** *table*)

# Other Numeric Operators

- **SUM** → calculate the sum of a column
  - Remember doing “inventory value” on a per-row basis? Now give the total inventory value.
  - Can use **WHERE** conditions to limit what gets summed
- **AVG** → Works similarly to, and with the same restrictions as, **MAX** and **MIN** but calculates the average of that column

# Groups!

- What if you want the minimum price... but not in the whole table, just for each vendor?
- We can tell SQL to group rows by the values in a certain column, and then perform a numeric operation on each group separately

# Groups

- **SELECT MIN( price ) FROM Product**
  - → give the minimum price in the whole table
- **SELECT v\_code, MIN( price ) FROM Product GROUP BY v\_code**
  - → give the vendor code and the minimum price of all rows with that vendor code
- Important: a **GROUP BY** must have count, min, max, avg, or sum in its column list
- You can have a WHERE clause in these queries, it goes BEFORE the GROUP BY

# More Examples

- Give me the number of products in the product table
- Give me the the number of products *per vendor*

# Answers

- `SELECT COUNT(*) FROM Product`
- `SELECT v_code, COUNT(DISTINCT(p_code))  
FROM Product GROUP BY v_code`
  - Is the distinct really needed?

# Let's get confusing

- We have a **WHERE** clause that tests all rows before they are returned in a query, but this is a row-by-row comparison
- A **HAVING** clause applies to the *groups* in a **GROUP BY** query and is listed after the **GROUP BY** part

# Examples

- **SELECT** v\_code, **COUNT(DISTINCT(p\_code)),**  
**AVG(p\_price) FROM** Product  
**WHERE** p\_price > 10  
**GROUP BY** v\_code
  - Grab all rows with price >10 then group them
- **SELECT** v\_code, **COUNT(DISTINCT(p\_code)),**  
**AVG(p\_price) FROM** Product  
**GROUP BY** v\_code  
**HAVING AVG( p\_price ) > 10**
  - Group all the rows and only return the groups with an average price over 10

# Playing nice with each other

- All this is fine but say our queries we don't want the `v_code` (how informative is that?), we need the vendor's name instead?
- A **join** will allow us to link two tables based on some attribute and grab data from both
- There are many types of joins, but we'll examine the most common, the inner join
- It's so common that you don't even need to use keywords!

# The Join

- Syntax →  
**SELECT** *columns*  
**FROM** *table1, table2*  
**WHERE** *table1.somecolumn = table2.somecolumn*
- The list of columns can be from either table
  - Put the table name before each column name, separated by a period (ex. Product.p\_code)
- Rows returned are dependent on the equality in the **WHERE**
  - Most common is joining a FK to its PK table

# Example

- Give the name, description, and vendor name for each product
- ```
SELECT Product.p_name, Product.p_descrip,  
Vendor.v_name  
FROM Product, Vendor  
WHERE Product.v_code = Vendor.v_code
```
- This can be shortened a bit using table aliases
- ```
SELECT P.p_name, P.p_descrip, V.v_name  
FROM Product P, Vendor V  
WHERE P.v_code = V.v_code
```

# Aside: Can do more than just join

- In the examples so far the only thing in the WHERE is how to do the join
- Keep in mind that the WHERE can contain more than one join related comparison as well as other non-join related comparison(s)
- Also, you can join on more than one table!

# Examining the results

- A similar query is run on pg 275 with results in Fig 7.29 on data from 7.2
- Do the results come out as expected?

# Outer Joins

- In the case of missing information (NULLS or non matches) we can still get the data from one of the tables using an outer join
- Outer joins show the results of an INNER JOIN the same, but in cases where there is no match they include the data from either the left or right table and fill in the data from the other table with nulls
- Figures 7.33 and 7.34

# Homework!

- Review: 18, 19, 20, 21, 22, 23, 24, 25
- Problems: 1, 2, 4, 5, 6, 7, 8