London Business School

AM05 Data Management

03. Data Analysis with SQL

Dr. David Tilson

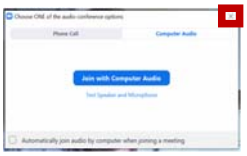
London Business School

AM05: Data Management

David Tilson

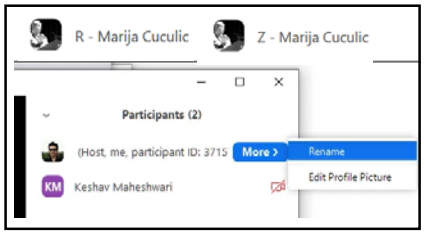
Welcome!
Class starts at 12:45 (London). Thank you for being early!

Zoom Classroom Etiquette



Roomies
When prompted, click "X"

1. Roomies: Please **turn on** your cameras and **join without audio**



2. Rename yourself
   a. Add “R” (for Roomie) in front of your name if you’re in the LT
   b. Add “Z” (for Zoomie) in front of your name if you’re remote

3. Questions
   a. Raise you (digital) hand if you want to speak or ask a question that needs an answer ASAP
   b. Use chat to ask questions that can wait for a few minutes
   c. I will ask you to answer questions using options on “participants” panel
   d. Technical issues - message the facilitator privately in Zoom chat

4. If you are in a breakout room, engage with your colleagues to extract the most out the class

5. Session will be recorded

¶SQL in context

¶Working with multiple relations (JOINS and UNION)

¶Aggregation and GROUP BY

¶Sub-queries and Views

¶For next time

SQL first standardized in 1986 … ANSI SQL*

Major updates: 1989, 1992, 1999, 2003, 2006, 2008, 2011, 2016,2019

¶ Specifies syntax/semantics for data definition (DDL) and manipulation (DML)

¶ Defines data structures and basic operations

¶ Designed to allow for enhancement (e.g. referential integrity, transaction management, user-defined functions, extended join operations, national character sets)

Possible benefits of standardized relational language

¶ Reduced training costs

¶ Productivity

¶ Application portability

¶ Application longevity

¶ Reduced dependence on a single vendor

¶ Cross-system communication

See https://blog.ansi.org/2018/10/sql-standard-iso-iec-9075-2016-ansi-x3-135/#gref

SQL used in wide range of applications…

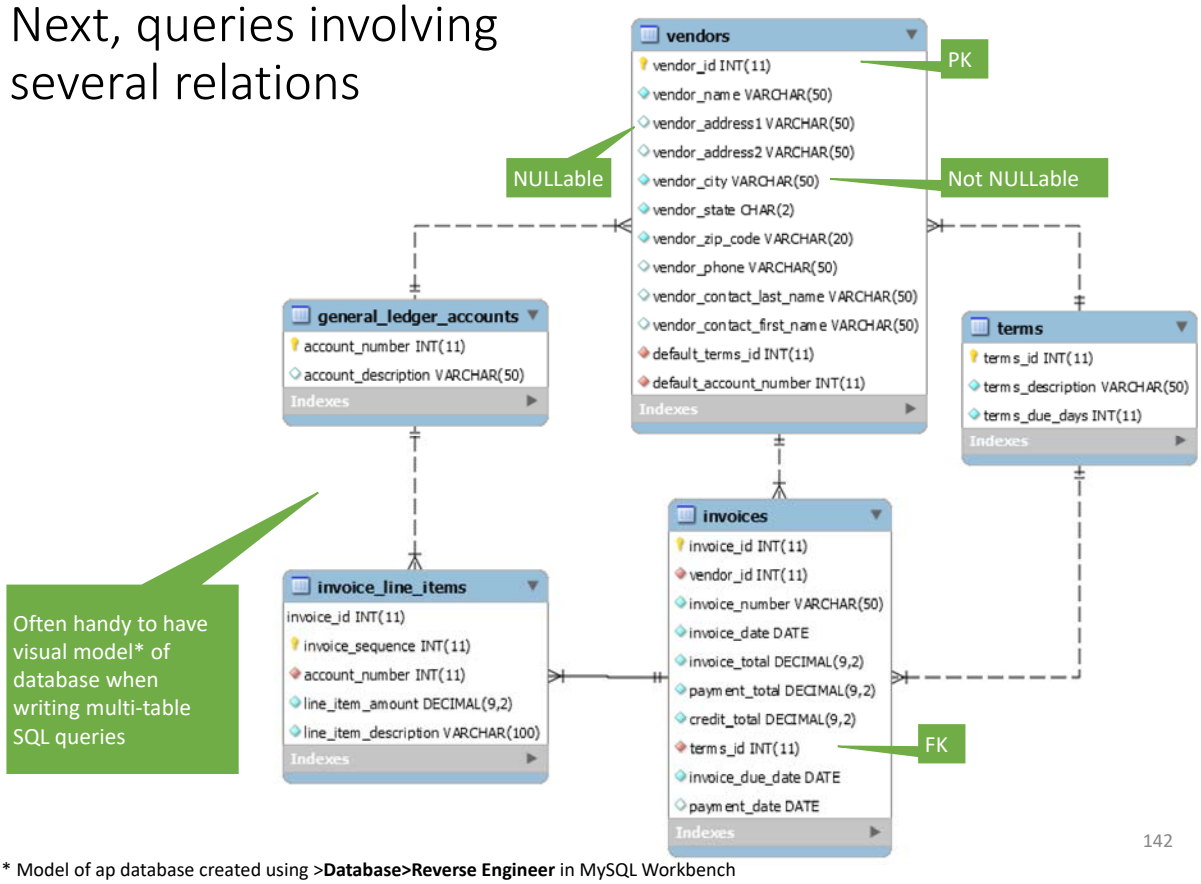
- Small database embedded in devices e.g. SQLite
- Person databases on laptop e.g. Microsoft Access
- Operational databases e.g. MySQL, ORACLE, MS SQL Server
  - Transaction processing – large volumes of small amounts of data
  - Some reporting – Resolve order delays, Schedule employees
- Business Intelligence / Data warehouses
  - ETL workloads to load data
  - Tactical issues – choose supplier, forecast sales
  - Strategic – Find new markets, choose new locations
- Big data e.g. Hadoop, HIVE /HQL , Apache Spark Web scale analytics
  - Big science
  - Machine learning

snowflake

… so worth spending time to get to know the key topics discussed today

- SQL in context
- Working with multiple relations (JOINS and UNION)
- Aggregation and GROUP BY
- Sub-queries and Views
- For next time

Next, queries involving several relations



* Model of ap database created using >**Database>Reverse Engineer** in MySQL Workbench

INNER JOIN combines columns from two or more tables

Example query

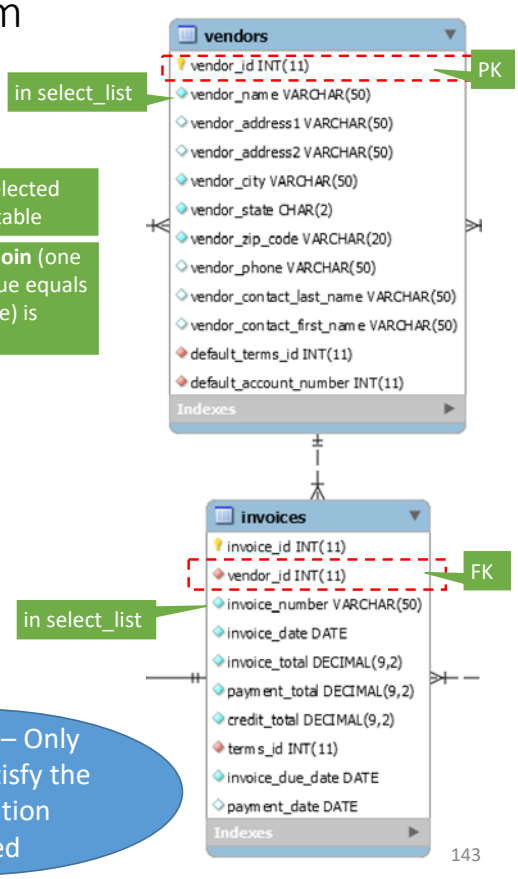
```
35    -- An inner join of the Vendors and Invoices tables
36 •  SELECT invoice_number, vendor_name
37    FROM vendors INNER JOIN invoices
38        ON vendors.vendor_id = invoices.vendor_id
39    ORDER BY invoice_number;
```

Columns selected from each table

PK-FK **equijoin** (one column value equals another one) is common

Result Grid | Filter Rows: | Export: | Wrap Cell Cor

invoice_number	vendor_name
0-2058	Malloy Lithographing Inc
0-2060	Malloy Lithographing Inc
0-2436	Malloy Lithographing Inc
1-200-5164	Federal Express Corporation
1-202-2978	Federal Express Corporation
10843	Yesmed, Inc
109596	Coffee Break Service
111-92R-10092	Pacific Bell
111-92R-10093	Pacific Bell
111-92R-10094	Pacific Bell
111-92R-10095	Pacific Bell



Syntax for EXPLICIT INNER JOIN

```
SELECT select_list
FROM table_1
    [INNER] JOIN table_2
        ON join_condition_1
   [[INNER] JOIN table_3
        ON join_condition_2]...
```

INNER JOIN – Only rows that satisfy the join condition returned

Column names must be **qualified** to remove ambiguity

```
40      -- An inner join of the Vendors and Invoices tables
41  •   SELECT vendors.vendor_id, invoice_number, vendor_name
42      FROM vendors INNER JOIN invoices
43          ON vendors.vendor_id = invoices.vendor_id
44      ORDER BY invoice_number;
45
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

vendor_id	invoice_number	vendor_name
110	0-2058	Malloy Lithographing Inc
110	0-2060	Malloy Lithographing Inc
110	0-2436	Malloy Lithographing Inc
123	1-200-5164	Federal Express Corporation
123	1-202-2978	Federal Express Corporation
119	10843	Yesmed, Inc
102	109596	Coffee Break Service
95	111-92R-10092	Pacific Bell
95	111-92R-10093	Pacific Bell
95	111-92R-10094	Pacific Bell
95	111-92R-10095	Pacific Bell
95	111-92R-10096	Pacific Bell
95	111-92R-10097	Pacific Bell

- **EXPLICIT JOIN** since JOIN keyword used
- **INNER** keyword is optional

Qualified column
Indicates that this version of the vendor_id column is from the invoices table

Query A

Table aliases make queries easier to write / read

Example query

```
50      -- An inner join with aliases for all tables
51  •   SELECT invoice_number, vendor_name, invoice_due_date,
52          invoice_total - payment_total - credit_total
53          AS balance_due
54      FROM vendors v JOIN invoices i
55          ON v.vendor_id = i.vendor_id
56      WHERE invoice_total - payment_total - credit_total > 0
57      ORDER BY invoice_due_date DESC;
```

Table alias is alternative table name
- Assigned in FROM clause
- Usually just one or two letters
- Less typing

Table alias must be used throughout the query once assigned (cannot also use original table name)

Can join to table in another database by **qualifying it with its database name**

```
69      -- The syntax of a table name that's qualified with a database name
70      -- A join to a table in another database
71      -- Also shows that queries based on ad-hoc relationships are possbile
72  •   SELECT vendor_name, customer_last_name,
73          customer_first_name, vendor_state AS state,
74          vendor_city AS city
75      FROM vendors v
76          JOIN om.customers c
77          ON v.vendor_zip_code = c.customer_zip
78      ORDER BY state, city;
79
```

The customers table is in the om database
It is qualified here using **database_name.table_name**

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

vendor_name	customer_last_name	custom
Wells Fargo Bank	Marissa	
Aztek Label	Irvin	
Zylka Design	Holbrooke	
Lou Gentile's Flower Basket	Damien	

Also note

- This join built around an **ad-hoc relationship** (i.e. not using PKs & FKs)
- Very handy in analytics. Joining tables **across sources** can bring more insight

JOINS can include more that one condition

Examine Customer Table

```
83  •   SELECT customer_id, customer_last_name, customer_first_name
84      FROM ex.customers
85      ORDER BY customer_id;
86
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

customer_id	customer_last_name	customer_first_name
1	Anders	Maria
2	Trujillo	Ana
3	Moreno	Antonio
4	Hardy	Thomas
5	Berglund	Christina
6	Moos	Hanna
7	Citeaux	Fred
8	Summer	Martin
9	Lebihan	Laurence
10	Lincoln	Elizabeth
11	Snyder	Howard

Examine Employee Table

```
87  •   SELECT * from ex.employees;
88
```

Result Grid | Filter Rows: | Export: | Wrap Cell Cor

employee_id	last_name	first_name	department_number	manager_id
1	Smith	Cindy	2	NULL
2	Jones	Elmer	4	1
3	Simonian	Ralph	2	2
4	Hernandez	Olivia	1	9
5	Aaronsen	Robert	2	4
6	Watson	Denise	6	8
7	Hardy	Thomas	5	2
8	O'Leary	Rhea	4	9
9	Locario	Paulo	6	1

Example join with compound condition

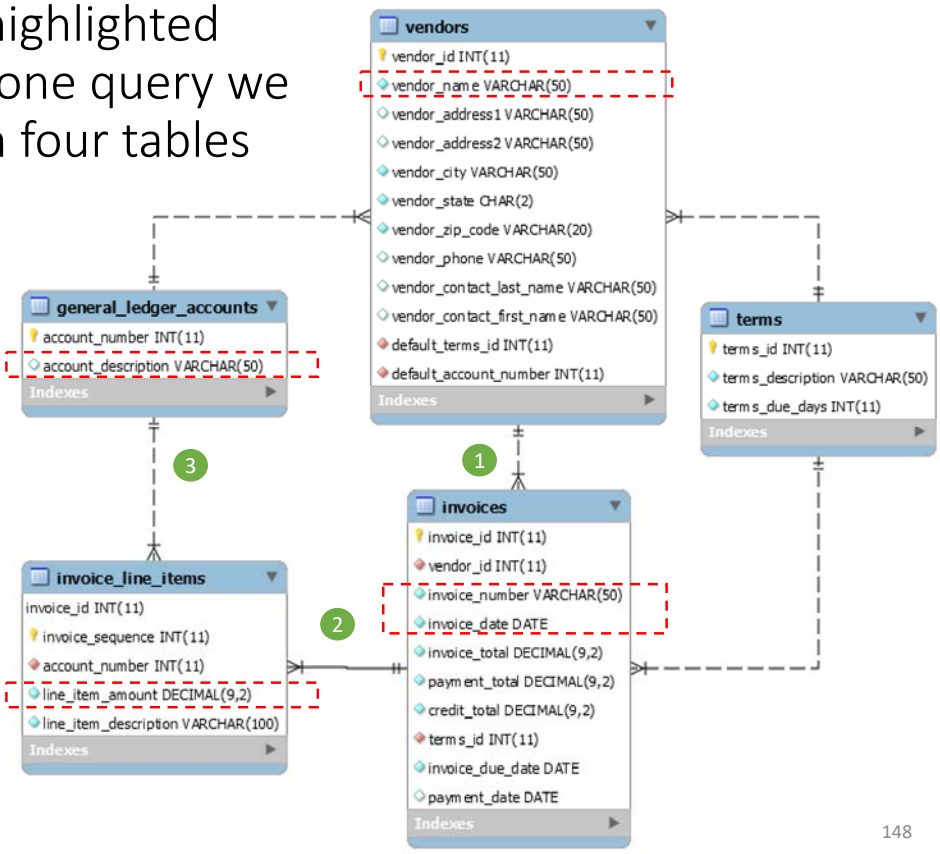
```
90      -- An inner join with two conditions
91  •   SELECT customer_first_name, customer_last_name
92      FROM ex.customers c JOIN ex.employees e
93          ON c.customer_first_name = e.first_name
94          AND c.customer_last_name = e.last_name;
```

COMPOUND JOIN condition

Result Grid | Filter Rows: | Export: | Wrap C

customer_first_name	customer_last_name
Thomas	Hardy

To include highlighted columns in one query we need to join four tables



Multi-table joins can be thought of as series of two-table joins

```
121     -- A statement that joins four tables
122 •   SELECT vendor_name, invoice_number, invoice_date,
123         line_item_amount, account_description
124     FROM vendors v
125     1 JOIN invoices i
126           ON v.vendor_id = i.vendor_id
127     2 JOIN invoice_line_items li
128           ON i.invoice_id = li.invoice_id
129     3 JOIN general_ledger_accounts gl
130           ON li.account_number = gl.account_number
131     WHERE invoice_total - payment_total - credit_total > 0
132     ORDER BY vendor_name, line_item_amount DESC;
```

Each join based on relationship between PK of one table and FK of another

Generally need one less JOIN than the number of tables being joined

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

vendor_name	invoice_number	invoice_date	line_item_amount	account_description
Blue Cross	547480102	2018-08-01	224.00	Group Insurance
Cardinal Business Media, Inc.	134116	2018-07-28	90.36	Direct Mail Advertising
Data Reproductions Corp	39104	2018-07-10	85.31	Book Printing Costs
Federal Express Corporation	263253270	2018-07-22	67.92	Freight
Federal Express Corporation	263253268	2018-07-21	59.97	Freight
Federal Express Corporation	963253264	2018-07-18	52.25	Freight
Federal Express Corporation	263253273	2018-07-22	30.75	Freight
Ford Motor Credit Company	9982771	2018-07-24	503.20	Travel and Accomodations

Query B

Implicit join includes the join conditions in the WHERE clause (mixed with other conditions)

IMPLICIT JOIN since JOIN keyword not used

```
142     -- Join the Vendors and Invoices tables
143 •   SELECT invoice_number, vendor_name
144     FROM vendors v, invoices i
145     WHERE v.vendor_id = i.vendor_id
146     ORDER BY invoice_number;
147
```

Result Grid | Filter Rows: | Export: | Wr

invoice_number	vendor_name
0-2058	Malloy Lithographing Inc
0-2060	Malloy Lithographing Inc
0-2436	Malloy Lithographing Inc
1-200-5164	Federal Express Corporation
1-202-2978	Federal Express Corporation
10843	Yesmed, Inc
109596	Coffee Break Service
111-92R-10092	Pacific Bell
111-92R-10093	Pacific Bell

Equivalent to explicit join in Query A

JOIN conditions mixed with others

```
148     -- Join four tables
149 •   SELECT vendor_name, invoice_number, invoice_date,
150         line_item_amount, account_description
151     FROM  vendors v, invoices i, invoice_line_items li,
152         general_ledger_accounts gl
153     WHERE v.vendor_id = i.vendor_id
154       AND i.invoice_id = li.invoice_id
155       AND li.account_number = gl.account_number
156       AND invoice_total - payment_total - credit_total > 0
157     ORDER BY vendor_name, line_item_amount DESC;
158
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

vendor_name	invoice_number	invoice_date	line_item_amount	account_description
Blue Cross	547480102	2018-08-01	224.00	Group Insurance
Cardinal Business Media, Inc.	134116	2018-07-28	90.36	Direct Mail Advertising
Data Reproductions Corp	39104	2018-07-10	85.31	Book Printing Costs
Federal Express Corporation	263253270	2018-07-22	67.92	Freight
Federal Express Corporation	263253268	2018-07-21	59.97	Freight
Federal Express Corporation	963253264	2018-07-18	52.25	Freight
Federal Express Corporation	263253273	2018-07-22	30.75	Freight
Ford Motor Credit Company	9982771	2018-07-24	503.20	Travel and Accomodat
Ingram	31361833	2018-07-21	579.42	Books, Dues, and Sub
Malloy Lithographing Inc	P-0608	2018-07-23	20551.18	Book Printing Costs
Malloy Lithographing Inc	0-2436	2018-07-31	10976.06	Book Printing Costs

Equivalent to explicit join in Query B

This inner join makes it easy to see that only rows that satisfy the join condition are returned

customers table

cust_id	name	country
a	Alice	us
b	Bob	ca
c	Carlos	mx
d	Dieter	de

orders table

order_id	cust_id	total
1	a	1539
2	c	1871
3	a	6352
4	b	1456
5	z	2137

```
SELECT c.cust_id, name, total
FROM customers c
JOIN orders o
ON (c.cust_id = o.cust_id);
```

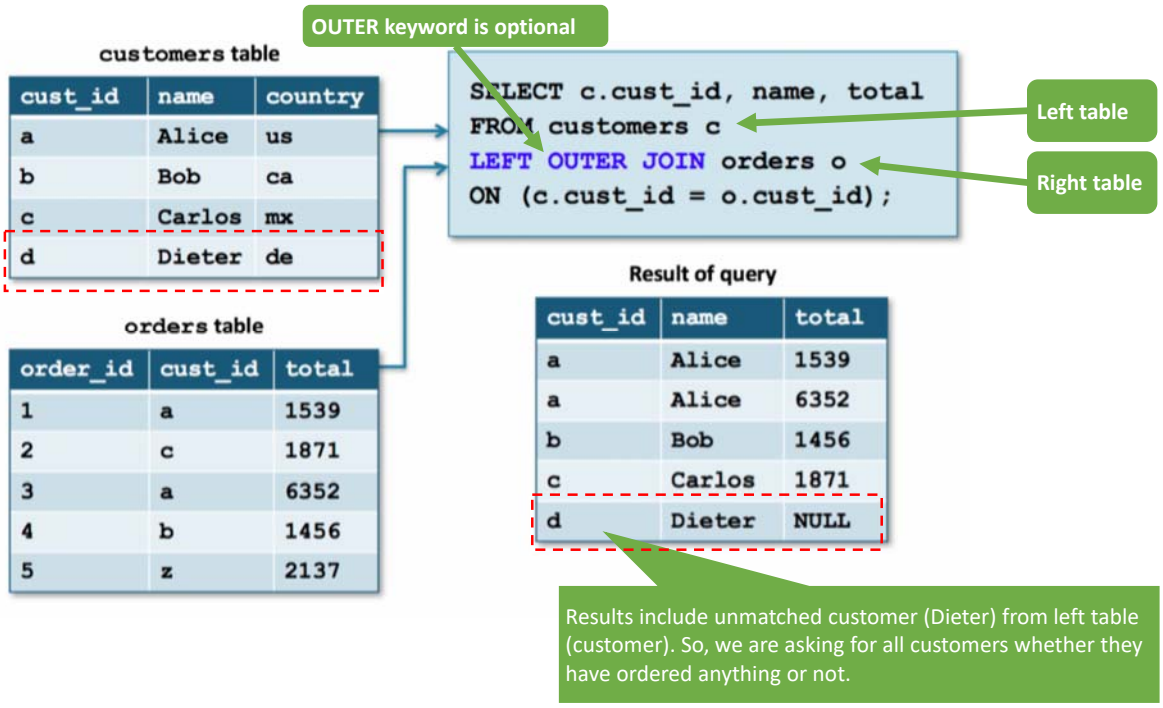
Result of query

cust_id	name	total
a	Alice	1539
a	Alice	6352
b	Bob	1456
c	Carlos	1871

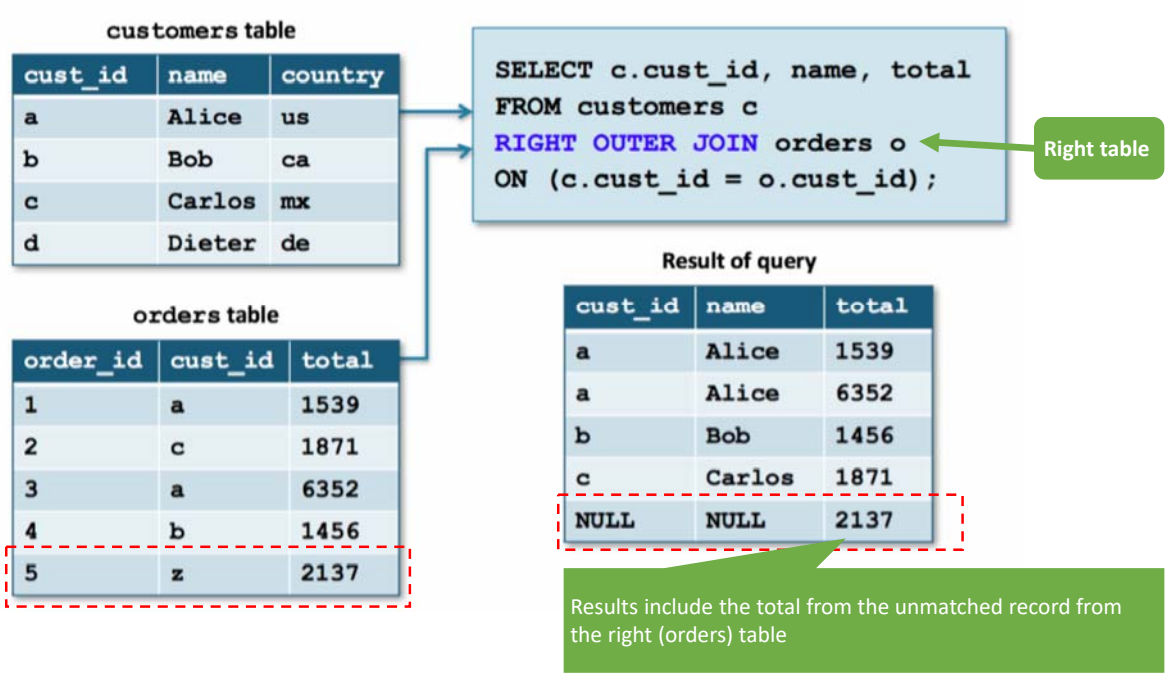
Dieter has not placed an order. So, he does not appear in the inner join result.

Order_id = 5 with customer_id = 'z' does not appear in the inner join result, since customer_id = 'z' is not in the customers table

In a **LEFT OUTER JOIN** only rows that satisfy the join condition **plus unmatched rows in the left table** are returned



In a **RIGHT OUTER JOIN** only rows that satisfy the join condition **plus unmatched rows in the right table** are returned



A FULL OUTER Join returns unmatched rows from left and right tables. It is not supported in MySQL

FULL OUTER join simulated using UNION of LEFT and RIGHT OUTER joins (duplicates eliminated by default)

```
233     -- Note there is no FULL OUTER JOIN in MySQL
234     -- However, we can simulate it using UNION
235 •       SELECT c.cust_id, name, total
236         FROM customers c
237             LEFT OUTER JOIN orders o
238             ON (c.cust_id = o.cust_id)
239     UNION
240         SELECT c.cust_id, name, total
241         FROM customers c
242             RIGHT JOIN orders o
243             ON (c.cust_id = o.cust_id);
```

customers table

cust_id	name	country
a	Alice	us
b	Bob	ca
c	Carlos	mx
d	Dieter	de

orders table

order_id	cust_id	total
1	a	1539
2	c	1871
3	a	6352
4	b	1456
5	z	2137

Result of query

cust_id	name	total
a	Alice	1539
a	Alice	6352
b	Bob	1456
c	Carlos	1871
d	Dieter	NULL
NULL	NULL	2137

Cross joins create results that join each row from one table with each row of the other table (be careful!)

Example of EXPLICIT CROSS JOIN

```
239     -- Cross joins of these two small tables show the principle
240     -- Produces result set that includes each row from the first
241     -- table joined with each row of the second table
242 •   SELECT c.cust_id, name, total, order_id, total
243     FROM customers c CROSS JOIN orders o
244     ORDER BY c.cust_id;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

cust_id	name	total	order_id	total
a	Alice	1539	1	1539
a	Alice	1871	2	1871
a	Alice	6352	3	6352
a	Alice	1456	4	1456
a	Alice	2137	5	2137
b	Bob	1539	1	1539
b	Bob	1871	2	1871
b	Bob	6352	3	6352
b	Bob	1456	4	1456
b	Bob	2137	5	2137
c	Carlos	1539	1	1539
c	Carlos	1871	2	1871

Not many practical uses

Example of IMPLICIT CROSS JOIN

```
247     -- Implicit cross join created when multi-table
248     -- SELECT query does not have a WHERE clause
249 •   SELECT c.cust_id, name, total, order_id, total
250     FROM customers c,  orders o
251     ORDER BY c.cust_id;
252
```

Result Grid | Filter Rows: | Export: | Wrap Ce

cust_id	name	total	order_id	total
a	Alice	1539	1	1539
a	Alice	1871	2	1871
a	Alice	6352	3	6352
a	Alice	1456	4	1456
a	Alice	2137	5	2137
b	Bob	1539	1	1539
b	Bob	1871	2	1871
b	Bob	6352	3	6352
b	Bob	1456	4	1456
b	Bob	2137	5	2137
c	Carlos	1539	1	1539

Cross joins created accidentally when JOIN conditions omitted… So, better to use explicit JOINS

UNION combines results from two or more SELECT statements into one result set (think of stacking results)

```
388 •  USE ex;
389    -- A union that combines result sets from two different tables
390 •  SELECT 'Active' AS source, invoice_number, invoice_date, invoice_total
391    FROM active_invoices
392    WHERE invoice_date >= '2018-06-01'
393  UNION
394    SELECT 'Paid' AS source, invoice_number, invoice_date, invoice_total
395    FROM paid_invoices
396    WHERE invoice_date >= '2018-06-01'
397  ORDER BY invoice_total DESC;
```

Column names for results set taken from the first SELECT statement

Each SELECT statement must return same number of columns and with corresponding data types (same DOMAIN)

Result Grid | Filter Rows: | Export:

source	invoice_number	invoice_date	invoice_total
Active	40318	2018-07-18	21842.00
Paid	P02-3772	2018-06-03	7125.34
Paid	10843	2018-06-04	4901.26
Paid	77290	2018-06-04	1750.00
Paid	RTR-72-3662-X	2018-06-04	1600.00
Paid	75C-90227	2018-06-06	1367.50
Paid	P02-88D77S7	2018-06-06	856.92
Active	I77271-O01	2018-06-05	662.00
Active	9982771	2018-06-03	503.20
Paid	121897	2018-06-01	450.00
Paid	CBM9920-M-T...	2018-06-07	290.00
Paid	133560	2018-06-01	175.00

Note:
- Duplicate rows eliminated by default
- Can use ORDER BY to sort combine set

In-class exercise: JOINS

10-minute breakout

¶ The following tables are in a university registrar's relational database

FACULTY (FacultyID, FacultyName)

FacultyID	FacultyName
2143	Birkin
3467	Berndt
4756	Collins
...	

QUALIFIED (FacultyID, CourseID, DateQualified)

FacultyID	CourseID	DateQualified
2143	ISM 3112	9/2005
2143	ISM 3113	9/2005
3467	ISM 4212	9/2012
3467	ISM 4930	9/2013
4756	ISM 3113	9/2008
4756	ISM 3112	9/2008
...		

COURSE (CourseID, CourseName)

CourseID	CourseName
ISM 3113	Syst Analysis
ISM 3112	Syst Design
ISM 4212	Database
ISM 4930	Networking
...	

¶ Write implicit and explicit SQL queries to return CourseID and CourseName for all courses that Professor Birkin in qualified to teach

¶ Submit your solutions to https://forms.gle/XLwkA2cPqE5U7cU36 One submission per breakout team

¶SQL in context

¶Working with multiple relations (JOINS and UNION)

¶Aggregation and GROUP BY

¶Sub-queries and Views

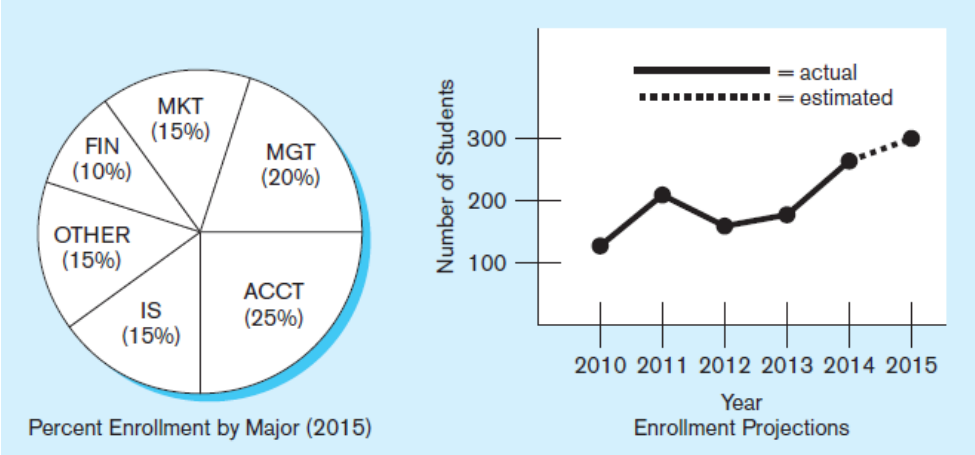
¶For next time

We already learned how to use some functions and operators while querying single tables

Math	* / DIV % + - ROUND
Strings	CONCAT, LEFT
Dates	DATE FORMAT
Comparisons	= < > <= >= <> != IN BETWEEN
Logical	AND OR NOT
Nulls	IS NULL IS NOT NULL
Other	DISTINCT LIMIT

… there are many more

Next, we examine aggregate functions which operate on series of values and return a summary value



Here we see summaries of the data (rather than individual data units)

- Data has been processed into **aggregates (e.g. sums and averages)** and categories
- It is one way of turning raw data into useful and actionable information
- Graphical representations are easier to absorb than tabular versions of the same information - aids managers' interpretation and decision making

Aggregation

Invoices table has 114 rows and one field that contains NULLs . . . Not very insightful!

```
22 •  USE ap;
23    -- Inspect the invoices table.
24    -- How many rows does it have?
25    -- Do any fields have NULLs?
26 •  SELECT * from invoices;
27
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

invoice_id	vendor_id	invoice_number	invoice_date	invoice_total	payment_total	credit_total	terms_id	invoice_due_date	payment_date
108	123	963253240	2018-07-24	67.00	67.00	0.00	3	2018-08-23	2018-08-23
109	121	97/222	2018-07-25	1000.46	1000.46	0.00	3	2018-08-24	2018-08-22
110	80	134116	2018-07-28	90.36	0.00	0.00	2	2018-08-17	NULL
111	123	263253257	2018-07-30	22.57	22.57	0.00	3	2018-08-29	2018-09-03
112	110	0-2436	2018-07-31	10976.06	0.00	0.00	3	2018-08-30	NULL
113	37	547480102	2018-08-01	224.00	0.00	0.00	3	2018-08-31	NULL
114	123	963253249	2018-08-02	127.75	127.75	0.00	3	2018-09-01	2018-09-04
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

payment_date column contains some NULLs

Aggregation

SQL has several aggregations functions that operate on series of values and return a single value → more insight

Syntax of most common SQL aggregate functions

AVG ([ALL|DISTINCT] expression)	Average of the non-null values in the expression
SUM ([ALL|DISTINCT] expression)	Total of the non-null values in the expression
MIN ([ALL|DISTINCT] expression)	Lowest non-null values in the expression
MAX ([ALL|DISTINCT] expression)	Highest non-null values in the expression
COUNT ([ALL|DISTINCT] expression)	Number of the non-null values in the expression
COUNT (*)	Number of rows selected by the query

* See https://dev.mysql.com/doc/refman/8.0/en/group-by-functions.html for full set of aggregation functions in MySQL

Counting is a basic part of analysis . . . but make sure you know what you are counting

```
28    -- The aggregatio functions (aka column functions) operate on values in selected rows
29    -- Here we are counting the number of invoices
30    --      COUNT(*) counts the number of rows selected (all of them in this case)
31    --      COUNT(payment_date) only counts the number of non-NULL values in that column
32    SELECT COUNT(*) AS number_of_invoices, COUNT(payment_date) AS number_of_rows_with_payment_date
33    FROM invoices;
34
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

number_of_invoices	number_of_rows_with_payment_date
114	103

COUNT(*) gives the number of **rows** returned

COUNT(column_name) gives the number of **non-null values** returned in a **specific column**

Non-aggregate query lists invoice with outstanding invoices

```
35      -- This query lists invoices with an outstanding balance
36      -- Note the number of rows returned (11)
37 •    SELECT * FROM invoices
38      WHERE invoice_total - payment_total - credit_total > 0;
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

invoice_id	vendor_id	invoice_number	invoice_date	invoice_total	payment_total	credit_total	terms_id	invoice_due_date	payment_date
89	72	39104	2018-07-10	85.31	0.00	0.00	3	2018-08-09	NULL
94	123	963253264	2018-07-18	52.25	0.00	0.00	3	2018-08-17	NULL
98	83	31361833	2018-07-21	579.42	0.00	0.00	2	2018-08-10	NULL
99	123	263253268	2018-07-21	59.97	0.00	0.00	3	2018-08-20	NULL
100	123	263253270	2018-07-22	67.92	0.00	0.00	3	2018-08-21	NULL
101	123	263253273	2018-07-22	30.75	0.00	0.00	3	2018-08-21	NULL
102	110	P-0608	2018-07-23	20551.18	0.00	1200.00	3	2018-08-22	NULL
105	106	9982771	2018-07-24	503.20	0.00	0.00	3	2018-08-23	NULL
110	80	134116	2018-07-28	90.36	0.00	0.00	2	2018-08-17	NULL
112	110	0-2436	2018-07-31	10976.06	0.00	0.00	3	2018-08-30	NULL
113	37	547480102	2018-08-01	224.00	0.00	0.00	3	2018-08-31	NULL

Aggregation query to count and calculate total of outstanding invoices

```
41      -- Aggregation functions operate on these rows that satisfy the condition
42      -- in the WHERE clause
43 •    SELECT          COUNT(*) AS number_of_invoices,
44                      SUM(invoice_total - payment_total - credit_total) AS total_due
45      FROM invoices
46      WHERE invoice_total - payment_total - credit_total > 0;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

number_of_invoices	total_due
11	32020.42

more insightful / actionable

Aggregation

WHERE clause can be used to restrict range of summaries

Note: Literal values can be included in a summary query

Summary query with AVG and SUM

```
49      -- A summary query with COUNT(*), AVG, and SUM
50 •    SELECT  'After 1 May 2018' AS selection_date,
51              COUNT(*) AS number_of_invoices,
52              ROUND(AVG(invoice_total), 2) AS avg_invoice_amt,
53              SUM(invoice_total) AS total_invoice_amt
54      FROM invoices
55      WHERE invoice_date > '2018-05-01';
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

selection_date	number_of_invoices	avg_invoice_amt	total_invoice_amt
After 1 May 2018	100	2084.56	208456.33

Note: Comparison operators like ">" can be used with dates and strings

Summary query with MIN and MAX

```
58      -- A summary query with MIN and MAX
59 •    SELECT 'After 1 May 2018' AS selection_date,
60          COUNT(*) AS number_of_invoices,
61          MAX(invoice_total) AS highest_invoice_total,
62          MIN(invoice_total) AS lowest_invoice_total
63      FROM invoices
64      WHERE invoice_date > '2018-05-01';
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

selection_date	number_of_invoices	highest_invoice_total	lowest_invoice_total
After 1 May 2018	100	37966.19	6.00

Aggregation

Some summary queries also work on **non-numeric** columns

```
66      -- A summary query for non-numeric columns
67  •   SELECT MIN(vendor_name) AS first_vendor,
68          MAX(vendor_name) AS last_vendor,
69          COUNT(vendor_name) AS number_of_vendors
70      FROM vendors;
```

MIN and MAX also work with dates

Result Grid | Filter Rows: | Export: | Wra

first_vendor	last_vendor	number_of_vendors
Abbey Office Furnishings	Zylka Design	122

In this case COUNT(*) would give the same count as there are no NULLs in the vendor_name column

Use **COUNT(DISTINCT column_name)** to only count rows with unique values in a specified column

```
73      -- A summary query with the DISTINCT keyword
74  •   SELECT COUNT(DISTINCT vendor_id) AS number_of_vendors,
75          COUNT(vendor_id) AS number_of_invoices,
76          ROUND(AVG(invoice_total), 2) AS avg_invoice_amt,
77          SUM(invoice_total) AS total_invoice_amt
78      FROM invoices
79      WHERE invoice_date > '2018-01-01';
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

number_of_vendors	number_of_invoices	avg_invoice_amt	total_invoice_amt
34	114	1879.74	214290.51

GROUP BY calculates aggregate summary values by group

Query that calculates the average invoice amount by vendor

```
90      -- A summary query that calculates the average invoice amount by vendor
91      SELECT vendor_id, ROUND(AVG(invoice_total), 2)
92          AS average_invoice_amount
93      FROM invoices
94      GROUP BY vendor_id
95      ORDER BY average_invoice_amount DESC;
96
```

GROUP BY clause groups rows of a result set based on more or more columns (or expressions)

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

vendor_id	average_invoice_amount
110	23978.48
72	10963.66
104	7125.34
99	6940.25
119	4901.26
122	2575.33
86	2433.00
100	2184.50
113	1750.00
107	1600.00
103	1367.50

Aggregate calculations for each group specified by GROUP BY clause

GROUP BY can include several columns

```
132     -- A query that finds summaries for vendors by state and city
133     SELECT vendor_state, vendor_city, COUNT(*) AS invoice_qty,
134         ROUND(AVG(invoice_total), 2) AS invoice_avg
135     FROM invoices JOIN vendors
136         ON invoices.vendor_id = vendors.vendor_id
137     GROUP BY vendor_state, vendor_city
138     ORDER BY vendor_state, vendor_city;
```

Of course we can work with data from multiple tables

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

vendor_state	vendor_city	invoice_qty	invoice_avg
AZ	Phoenix	1	662.00
CA	Fresno	19	1208.75
CA	Los Angeles	1	503.20
CA	Oxnard	3	188.00
CA	Pasadena	5	196.12
CA	Sacramento	7	253.00
CA	San Francisco	3	1211.04
CA	Turlock	1	95.00
CA	Valencia	1	6940.25
DC	Washington	1	600.00

HAVING clause specifies a search condition based on group aggregates calculated by the GROUP BY clause

Query that calculates the average invoice amount by vendor

```
97      -- Same summary but only returning results for average
98      -- invoice amounts larger than 2000
99  •   SELECT vendor_id, ROUND(AVG(invoice_total), 2)
100         AS average_invoice_amount
101     FROM invoices
102     GROUP BY vendor_id
103     HAVING AVG(invoice_total) > 2000
104     ORDER BY average_invoice_amount DESC;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

vendor_id	average_invoice_amount
110	23978.48
72	10963.66
104	7125.34
99	6940.25
119	4901.26
122	2575.33
86	2433.00
100	2184.50

With aggregate functions you can't have single-valued columns included in the SELECT clause, unless they are included in the GROUP BY clause*

HAVING clause determines which groups are included in the final results.

Only those vendors with invoices that average over $2000 are included in final results . . . as specified in the search condition in the HAVING clause

Grouping

* See sql script for example of one exception

Make sure you understand that WHERE and HAVING do different things

Search condition in WHERE clause

```
161     -- A summary query with a search condition in
162     -- the WHERE clause
163 •   SELECT vendor_name,
164         COUNT(*) AS invoice_qty,
165         ROUND(AVG(invoice_total), 2) AS invoice_avg
166     FROM vendors JOIN invoices
167         ON vendors.vendor_id = invoices.vendor_id
168     WHERE invoice_total > 500
169     GROUP BY vendor_name
170     ORDER BY invoice_qty DESC;
```

Result Grid | Filter Rows: | Export: | Wrap Ce

vendor_name	invoice_qty	invoice_avg
United Parcel Service	9	2575.33
Zylka Design	7	946.67
Malloy Lithographing Inc	5	23978.48
Ingram	2	1077.21
Data Reproductions Corp	1	21842.00
Federal Express Corporation	1	739.20

- Filtering done before grouping
- Could refer to any column in base tables
- Cannot contain aggregate functions

Search condition in HAVING clause

```
161     -- A summary query with a search condition in
162     -- the HAVING clause
163 •   SELECT vendor_name,
164         COUNT(*) AS invoice_qty,
165         ROUND(AVG(invoice_total),2) AS invoice_avg
166     FROM vendors JOIN invoices
167         ON vendors.vendor_id = invoices.vendor_id
168     GROUP BY vendor_name
169     HAVING AVG(invoice_total) > 500
170     ORDER BY invoice_qty DESC;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Cont

vendor_name	invoice_qty	invoice_avg
United Parcel Service	9	2575.33
Zylka Design	8	867.53
Malloy Lithographing Inc	5	23978.48
IBM	2	600.06
Data Reproductions Corp	2	10963.66
Ingram	2	1077.21

Note differences

- Filtering done after grouping
- Can only refer to columns in SELECT clause
- Can contain aggregate functions
- Can include non-aggregation conditions but documentation recommends against

Grouping

The SELECT statement includes many features that allow you to perform sophisticated queries

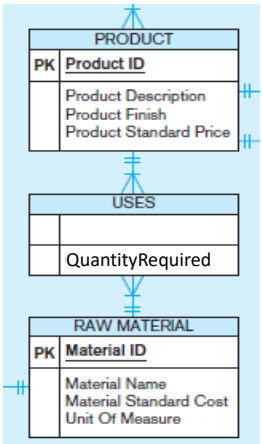
General syntax of SELECT statement

SELECT [ALL/DISTINCT] column_list	Columns (and expressions) to be returned
FROM table_list	Table(s) and view(s) from which data will be obtained
[**WHERE** conditional expression]	Conditions under which a **row** will be included (before any grouping occurs)
[**GROUP BY** group_by_column_list]	categorization of results
[**HAVING** conditional expression]	Conditions under which a **category (group)** will be included
[**ORDER BY** order_by_column_list]	Sorts results according to specified criteria

In-class exercise: Aggregation

10-minute breakout

- These tables in a relational database model the materials needed to build various products



- Write an SQL query to calculate the total raw material cost (label TotCost) for each product. Include product ID, product description, Product Standard Price, and the TotCost in the output
- Submit your solution to https://forms.gle/CP4Wo57rM74satiPA One submission per breakout team

¶SQL in context

¶Working with multiple relations (JOINS and UNION)

¶Aggregation and GROUP BY

¶Sub-queries and Views

¶For next time

A subquery is a SELECT statement embedded in another SQL statement

There are several ways to *introduce* a subquery in a SELECT statement
- In a WHERE or HAVING clause as a search condition
- In the FROM clause as a table specification
- In the SELECT clause as a column specification

Search condition in WHERE clause

```
3     -- A subquery in the WHERE clause
4     SELECT invoice_number, invoice_date, invoice_total
5     FROM invoices
6     WHERE invoice_total >
7         (SELECT AVG(invoice_total)
8          FROM invoices)
9     ORDER BY invoice_total;
```

- Subquery in parenthesis
- This subquery produces single value

Result Grid | Filter Rows: | Export: | Wrap Cell Con

invoice_number	invoice_date	invoice_total
989319-487	2018-06-20	1927.54
97/522	2018-06-28	1962.13
989319-417	2018-07-23	2051.59
989319-427	2018-06-16	2115.81
989319-477	2018-06-08	2184.11
587056	2018-06-30	2184.50
989319-497	2018-06-12	2312.20

Value returned by subquery

```
12 •  SELECT AVG(invoice_total)
```

Result Grid | Filter Rows:

AVG(invoice_total)
1879.741316

Some queries can be written as JOINS or subqueries

Using an INNER join

```
15    -- A query that uses an inner join
16    SELECT invoice_number, invoice_date, invoice_total
17    FROM invoices JOIN vendors
18        ON invoices.vendor_id = vendors.vendor_id
19    WHERE vendor_state = 'CA'
20    ORDER BY invoice_date;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Cont

invoice_number	invoice_date	invoice_total
125520-1	2018-04-24	95.00
97/488	2018-04-24	601.95
111-92R-10096	2018-04-30	16.33
25022117	2018-05-01	6.00
P02-88D77S7	2018-05-03	856.92
QP58872	2018-05-07	116.54
24863706	2018-05-10	6.00
10843	2018-05-11	4901.26

Using subquery

```
23    -- The same query restated with a subquery
24    SELECT invoice_number, invoice_date, invoice_total
25    FROM invoices
26    WHERE vendor_id IN
27        (SELECT vendor_id
28        FROM vendors
29        WHERE vendor_state = 'CA')
30    ORDER BY invoice_date;
```

This subquery produces a list of values (result set with one column)

Result Grid | Filter Rows: | Export: | Wrap Cell Con

invoice_number	invoice_date	invoice_total
125520-1	2018-04-24	95.00
97/488	2018-04-24	601.95
111-92R-10096	2018-04-30	16.33
25022117	2018-05-01	6.00
P02-88D77S7	2018-05-03	856.92
QP58872	2018-05-07	116.54
24863706	2018-05-10	6.00
10843	2018-05-11	4901.26

Advantages of joins

- Can include columns from both tables
- More intuitive when using an existing relationship

Advantages of subqueries

- Can pass an aggregate value to main query (see example on previous page)
- More intuitive if using ad hoc relationship
- Long, complex queries can be easier to code using subqueries

Subquery in WHERE clause

Subquery can introduce list of values that can be tested against an expression using [NOT] IN

Syntax of a WHERE clause that uses an IN phrase

```
WHERE test_expression [NOT] IN (subquery)
```

```
35    -- A query that gets vendors without invoices
36    SELECT vendor_id, vendor_name, vendor_state
37    FROM vendors
38    WHERE vendor_id NOT IN
39        (SELECT DISTINCT vendor_id
40        FROM invoices)
41    ORDER BY vendor_id;
```

[NOT] IN will test whether an expression is [NOT] contained in a list of values

A subquery must produce a list of values (result set with one column) to work in this context

Result Grid | Filter Rows: | Edit:

vendor_id	vendor_name	vendor_state
32	RR Bowker	NJ
33	Nielson	OH
35	Cal State Termite	CA
36	Graylift	CA
38	Venture Communications Int'l	NY
39	Custom Printing Company	MO
40	Nat Assoc of College Stores	OH
41	Shields Design	CA

Subquery in WHERE clause

A comparison operator in a WHERE clause can be used to compare an expression with the results of a subquery

Query to get invoices with a balance due less than the average

```
56 •  SELECT invoice_number, invoice_date,
57        invoice_total - payment_total - credit_total AS balance_due
58    FROM invoices
59    WHERE invoice_total - payment_total - credit_total > 0
60      AND invoice_total - payment_total - credit_total <
61        (SELECT AVG(invoice_total - payment_total - credit_total)
62         FROM invoices
63         WHERE invoice_total - payment_total - credit_total > 0)
64    ORDER BY invoice_total DESC;
```

Subquery returns the average balance due across all invoices

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

invoice_number	invoice_date	balance_due
31361833	2018-07-21	579.42
9982771	2018-07-24	503.20
547480102	2018-08-01	224.00
134116	2018-07-28	90.36
39104	2018-07-10	85.31
263253270	2018-07-22	67.92
263253268	2018-07-21	59.97

Subquery in WHERE clause

The ALL keyword allow comparisons with lists of values returned from a subquery

```
69    -- A query that uses ALL
70    -- Get invoices larger than the largest invoice for vendor 34
71 •  SELECT vendor_name, invoice_number, invoice_total
72    FROM invoices i JOIN vendors v ON i.vendor_id = v.vendor_id
73    WHERE invoice_total > ALL
74        (SELECT invoice_total
75         FROM invoices
76         WHERE vendor_id = 34)
77    ORDER BY vendor_name;
```

- The subquery can return a list of values if the ALL keyword are used
- This subquery returns two values for use in the comparison (116.54, 1083.58)
- So, the overall query returns all invoices with invoice_total > 1083.58

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

vendor_name	invoice_number	invoice_total
Bertelsmann Industry Svcs. Inc	509786	6940.25
Cahners Publishing Company	587056	2184.50
Computerworld	367447	2433.00
Data Reproductions Corp	40318	21842.00
Dean Witter Reynolds	75C-90227	1367.50
Digital Dreamworks	P02-3772	7125.34
Franchise Tax Board	RTR-72-3662-X	1600.00
Ingram	31359783	1575.00

Subquery in WHERE clause

The ANY and SOME keywords allow comparisons with lists of values returned from a subquery

```
79    -- A query that uses ANY
80    -- Get invoices smaller than largest invoice for vendor 115
81    SELECT vendor_name, invoice_number, invoice_total
82    FROM vendors JOIN invoices
83      ON vendors.vendor_id = invoices.vendor_id
84    WHERE invoice_total < ANY
85        (SELECT invoice_total
86         FROM invoices
87         WHERE vendor_id = 115);
```

Result Grid | Filter Rows: | Export: | Wrap Cell Con

vendor_name	invoice_number	invoice_total
Federal Express Corporation	963253251	15.50
Pacific Bell	111-92R-10096	16.33
Roadway Package System, Inc	25022117	6.00
Compuserve	21-4748363	9.95
Federal Express Corporation	4-321-2596	10.00
Roadway Package System, Inc	24863706	6.00
Compuserve	21-4923721	9.95

- The subquery can return a **list of values** if the ANY or SOME keywords are used
- This subquery returns four values for use in the comparison (6.00, 6.00, 25.67, 6.00)
- So, the overall query returns all invoices with invoice_total < 25.67
- ANY and SOME are synonyms and can be used interchangeably

Subquery in WHERE clause

Correlated subquery executed once for each row in main query (unlike uncorrelated subquery which is executed only once)

Get invoice amount that's higher than the vendor's average invoice amount

```
93    SELECT vendor_id, invoice_number, invoice_total
94    FROM invoices i
95    WHERE invoice_total >
96        (SELECT AVG(invoice_total)
97         FROM invoices
98         WHERE vendor_id = i.vendor_id)
99    ORDER BY vendor_id, invoice_total;
```

Result Grid | Filter Rows: | Export: | Wrap Cel

vendor_id	invoice_number	invoice_total
72	40318	21842.00
80	133560	175.00
83	31359783	1575.00
95	111-92R-10095	32.70
95	111-92R-10093	39.77
95	111-92R-10092	46.21
110	P-0259	26881.40

- Correlated subquery refers to a value provided by the main query
- For each different value returned by the main query for that column, the subquery returns a different result

In this case a column from the main query is qualified with a table name alias (since subquery also refers to a table with the same name)

Value returned from subquery for vendor 95

```
101   -- The value returned by the subquery
102   -- for vendor 95 (28.501667)
103   SELECT AVG(invoice_total)
104   FROM invoices
105   WHERE vendor_id = 95;
```

Result Grid | Filter Rows: | Export:

AVG(invoice_total)
28.501667

Should execute subquery on its own to ensure you get the sorts of results you expect

Correlated subquery in WHERE clause

Can use **EXISTS** operator to test that one or more rows are returned by the subquery

Get all vendors that do not have invoices

```
109     -- A query that gets vendors without invoices
110 •   SELECT vendor_id, vendor_name, vendor_state
111     FROM vendors
112     WHERE NOT EXISTS
113         (SELECT *
114          FROM invoices
115          WHERE vendor_id = vendors.vendor_id);
```

In this case NOT EXISTS tests that no rows are returned by the subquery

correlation

Result Grid | Filter Rows: | Edit: | Export/Import:

vendor_id	vendor_name	vendor_state
1	US Postal Service	WI
2	National Information Data Ctr	DC
3	Register of Copyrights	DC
4	Jobtrak	CA
5	Newbrige Book Clubs	NJ
6	California Chamber Of Commerce	CA
7	Towne Advertiser's Mailing Svcs	CA
8	BFI Industries	CA
9	Pacific Gas & Electric	CA

Note: Since subquery does not actually return rows when EXISTS is used
- Query runs quickly (efficient)
- Can just use * in the SELECT clause of subqery

Correlated subquery in WHERE clause

When using a **subquery in the SELECT clause** it must return a single value

Get the most recent invoice date for each vendor

```
119 •   SELECT vendor_name,
120         (SELECT MAX(invoice_date) FROM invoices
121          WHERE vendor_id = vendors.vendor_id) AS latest_inv
122     FROM vendors
123     ORDER BY latest_inv DESC;
```

This subquery
- Calculates the maximum invoice date for each vendor in the vendors table
- Returns one value per row in the main query

correlation

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

vendor_name	latest_inv
Federal Express Corporation	2018-08-02
Blue Cross	2018-08-01
Malloy Lithographing Inc	2018-07-31
Cardinal Business Media, Inc.	2018-07-28
Zylka Design	2018-07-25
Ford Motor Credit Company	2018-07-24

- Subqueries in SELECT clause considered difficult to read and seldom used
- Same result can be obtained using a JOIN

Correlated subquery in SELECT clause

Subquery in the FROM clause returns a result set that acts like a table (or view)…. referred to as an *inline view*

Get the largest invoice total for the vendor in each state (using inline view)

```
135 •  SELECT vendor_state,
136        MAX(sum_of_invoices) AS max_sum_of_invoices
137    FROM
138    (
139        SELECT vendor_state, vendor_name,
140            SUM(invoice_total) AS sum_of_invoices
141        FROM vendors v JOIN invoices i
142            ON v.vendor_id = i.vendor_id
143        GROUP BY vendor_state, vendor_name
144    ) t
145    GROUP BY vendor_state
146    ORDER BY vendor_state;
```

This subquery provides summarized data to the main query

Must assign an alias for the subquery in the FROM clause

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

vendor_state	max_sum_of_invoices
AZ	662.00
CA	7125.34
DC	600.00
MA	1367.50
MI	119892.41
NV	23177.96

Query C

Subquery in FROM clause

A VIEW is a virtual table that can be used in queries

Create a VIEW equivalent to the subquery in previous query (Query C)

```
160 •  CREATE VIEW invoice_totals_by_vendor_V AS
161        SELECT vendor_state, vendor_name,
162            SUM(invoice_total) AS sum_of_invoices
163        FROM vendors v JOIN invoices i
164            ON v.vendor_id = i.vendor_id
165        GROUP BY vendor_state, vendor_name;
166
167    -- Show what the view returns
168 •  SELECT * FROM invoice_totals_by_vendor_V;
```

A view is a SELECT statement stored in the RDBMS as a database object, it

- Can be thought of as a virtual table
- Refers to base tables in its FROM clause
- Does not store data itself. So, when invoked it reflects the latest data from those tables

The view can be used pretty much as a table in other queries.

Result Grid | Filter Rows: | Export: | Wrap

vendor_state	vendor_name	sum_of_invoices
NV	United Parcel Service	23177.96
TN	Federal Express Corporation	4378.02
CA	Evans Executone Inc	95.00
CA	Zylka Design	6940.25
AZ	Wells Fargo Bank	662.00
CA	Pacific Bell	171.01
CA	Roadway Package System, Inc	43.67

Can allow changes in the underlying base tables when using INSERT, UPDATE, and DELETE queries with a VIEW

VIEWS

VIEWs can make queries easier to read

Get the largest invoice total for the vendor in each state (using a VIEW)

```
172 •  SELECT vendor_state,
173        MAX(sum_of_invoices) AS max_sum_of_invoices
174    FROM invoice_totals_by_vendor_V
175    GROUP BY vendor_state
176    ORDER BY vendor_state;
177
```

VIEW

Result Grid | Filter Rows: | Export: | Wrap Cel

vendor_state	max_sum_of_invoices
AZ	662.00
CA	7125.34
DC	600.00
MA	1367.50
MI	119892.41
NV	23177.96
OH	207.78
PA	265.36
TN	4378.02

Compare to Query C

Advantages of Views

- Simplify queries
- Assist with data security (only access subset of base table content)
- Enhance programming productivity
- Contain most current base table data
- Use little storage
- Provide customized view for user
- Establish physical data independence (some changes in base tables could be hidden from users of views)

Disadvantages of Views

- Use processing time each time view is referenced
- May or may not be directly updateable

A Common Table Expression (CTE) can also be used to make queries easier to read

Get the largest invoice total for the vendor in each state (using a Common Table Expression)

```
180 •  WITH invoice_totals_by_vendor AS
181    (
182        SELECT vendor_state, vendor_name,
183            SUM(invoice_total) AS sum_of_invoices
184        FROM vendors v JOIN invoices i
185            ON v.vendor_id = i.vendor_id
186        GROUP BY vendor_state, vendor_name
187    )
188    SELECT vendor_state,
189        MAX(sum_of_invoices) AS max_sum_of_invoices
190    FROM invoice_totals_by_vendor
191    GROUP BY vendor_state
192    ORDER BY vendor_state;
```

Definition of CTE only usable in this query

Use of CTE just as if it were a table

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

vendor_state	max_sum_of_invoices
AZ	662.00
CA	7125.34
DC	600.00
MA	1367.50
MI	119892.41
NV	23177.96
OH	207.78
PA	265.36

Compare to Query C

¶SQL in context

¶Working with multiple relations (JOINS and UNION)

¶Aggregation and GROUP BY

¶Sub-queries and Views

¶For next time

For next time

¶Attend Thursday workshop (optional) if you think you might need more support before completing the homework

¶Check out readings for this session and next

¶Submit Homework Assignment #3 (team)

- Covers the concepts discussed today (details on Canvas)
- Submit via Canvas by 10pm on Sunday

- Last year's exam posted on Canvas
- Heads up . . . This year's exam will ask you to write some SQL queries. You will need to be able to run SQL script to create the required database

Appendix

Additional Topics

1. Implement unary relationships and self joins
2. Execution order of SQL statements (why you cannot refer to a column alias in a WHERE clause in MySQL)
3. More aggregation options with ROLLUP
4. Alternative ways of expressing INNER JOIN (USING and NATURAL) that you might see in code

Might need to look at for homework

A **self-join** is one that joins a table with itself

```
98    -- A self-join that returns vendors from cities in common with other vendors
99    SELECT DISTINCT v1.vendor_name AS vendor, v1.vendor_city, v1.vendor_state
100   FROM vendors v1 JOIN vendors v2
101       ON v1.vendor_city = v2.vendor_city AND
102          v1.vendor_state = v2.vendor_state AND
103          v1.vendor_name <> v2.vendor_name
104   ORDER BY v1.vendor_state, v1.vendor_city, v1.vendor_name;
105
```

Must
- Assign table aliases for the two versions of the table used in the query
- Qualify each column name with an alias

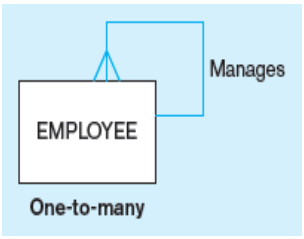
Result Grid | Filter Rows: | Export: | Wrap Cell Content:

vendor	vendor_city	vendor_state
AT&T	Phoenix	AZ
Computer Library	Phoenix	AZ
Wells Fargo Bank	Phoenix	AZ
Aztek Label	Anaheim	CA
Blue Shield of California	Anaheim	CA
Abbey Office Furnishings	Fresno	CA
ASC Signs	Fresno	CA
BFI Industries	Fresno	CA
Bill Marvin Electric Inc	Fresno	CA

1. Removes duplicates
2. Additional condition to prevent vendor matching with itself. So, will only get cities with at least two vendors in the results

A **self-join** is often associated with a unary relationship

Unary 1:N



Question: What are the employee IDs and names of each employee and the name of her/her supervisor?

```
SELECT E.EmployeeID, E.EmployeeName, M.EmployeeName AS Manager
  FROM Employee_T E, Employee_T M
  WHERE E.EmployeeSupervisor = M.EmployeeID;
```

Same table on both sides of join; distinguished using table aliases

Result:

EMPLOYEEID	EMPLOYEENAME	MANAGER
123-44-347	Jim Jason	Robert Lewis

Employees (E)

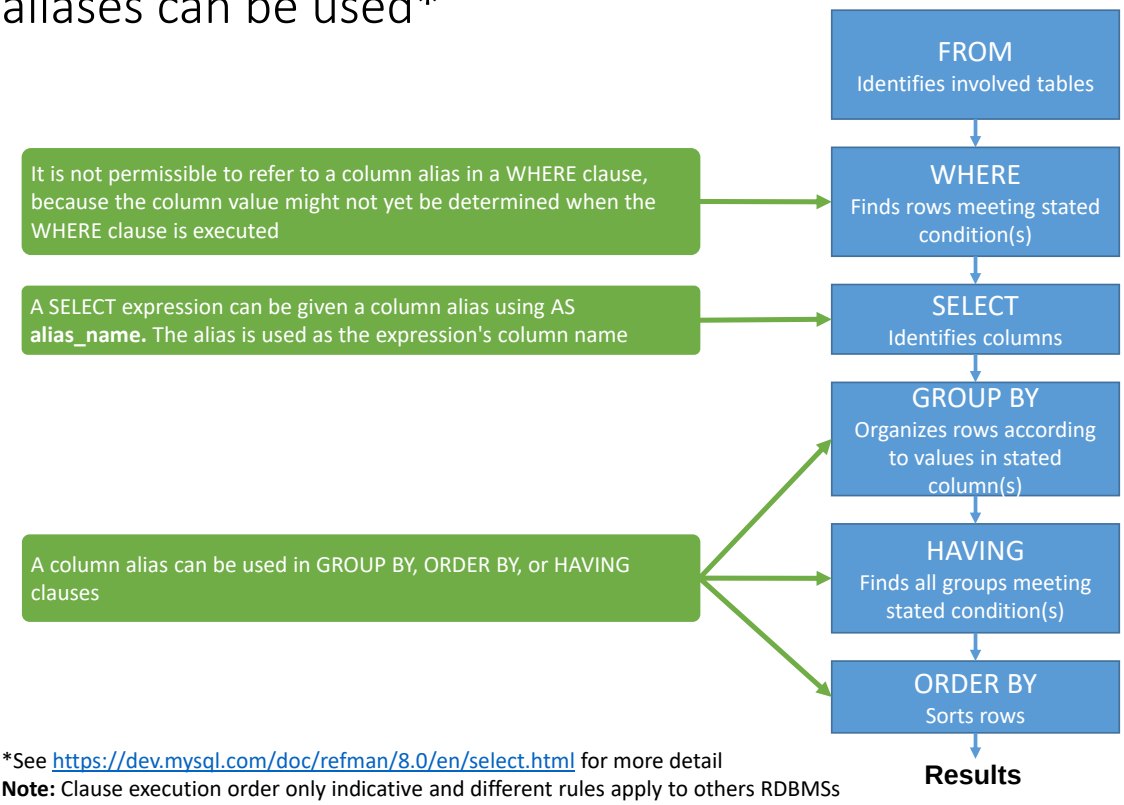
EmployeeID	EmployeeName	EmployeeSupervisor
098-23-456	Sue Miller	
107-55-789	Stan Getz	
123-44-347	Jim Jason	678-44-546
547-33-243	Bill Blass	
678-44-546	Robert Lewis	

Managers (M)

EmployeeID	EmployeeName	EmployeeSupervisor
098-23-456	Sue Miller	
107-55-789	Stan Getz	
123-44-347	Jim Jason	678-44-546
547-33-243	Bill Blass	
678-44-546	Robert Lewis	

Execution order influences where aliases can be used*

MySQL SELECT Clause Execution Order



*See https://dev.mysql.com/doc/refman/8.0/en/select.html for more detail
Note: Clause execution order only indicative and different rules apply to others RDBMSs

Column aliases can be used in the HAVING clause

```
150     -- A summary query with a search condition in the HAVING clause
151     -- Note the use of a column alias in the HAVING clause
152 •   SELECT vendor_name,
153         COUNT(*) AS invoice_qty,
154         AVG(invoice_total) AS invoice_avg
155     FROM vendors JOIN invoices
156         ON vendors.vendor_id = invoices.vendor_id
157     GROUP BY vendor_name
158     HAVING invoice_avg > 500
159     ORDER BY invoice_qty DESC;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

vendor_name	invoice_qty	invoice_avg
United Parcel Service	9	2575.328889
Zylka Design	8	867.531250
Malloy Lithographing Inc	5	23978.482000
IBM	2	600.060000
Data Reproductions Corp	2	10963.655000
Ingram	2	1077.210000
Ford Motor Credit Company	1	503.200000
Yesmed, Inc	1	4901.260000
Pollstar	1	1750.000000
Wells Fargo Bank	1	662.000000
Cahners Publishing Company	1	2184.500000

WITH ROLLUP can add summary rows*

```
213     -- A summary query with a final summary row
214 •   SELECT vendor_id, COUNT(*) AS invoice_count,
215         SUM(invoice_total) AS invoice_total
216     FROM invoices
217     GROUP BY vendor_id WITH ROLLUP;
218
```

Result Grid | Filter Rows: | Export: | Wrap Cell C

vendor_id	invoice_count	invoice_total
113	1	1750.00
114	1	290.00
115	4	43.67
117	1	16.62
119	1	4901.26
121	8	6940.25
122	9	23177.96
123	47	4378.02
NULL	114	214290.51

vendor_id cannot be summarized and is therefore NULL

This row in the result give summary aggregations across all vendors

* WITH ROLLUP is not standard SQL

WITH ROLLUP adds summary row for each group specified in GROUP BY. Also adds summary row at end for entire result set

```
220     -- A summary query with a summary row  for each grouping level
221 •   SELECT vendor_state, vendor_city, COUNT(*) AS qty_vendors
222     FROM vendors
223     WHERE vendor_state IN ('IA', 'NJ')
224     GROUP BY vendor_state, vendor_city WITH ROLLUP;
225
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

vendor_state	vendor_city	qty_vendors
IA	Fairfield	1
IA	Washington	1
IA	NULL	2
NJ	East Brunswick	2
NJ	Fairfield	1
NJ	Washington	1
NJ	NULL	4
NULL	NULL	6

Summary for IA

Summary for NJ

Summary for all rows in the results

Can be hard to identify which rows are WITH ROLLUP summaries versus those referring to NULLs in the data

```
226     -- The basic syntax of the GROUPING function GROUPING(expression)
227     -- A summary query that uses WITH ROLLUP on a table with null values
228 •   SELECT invoice_date, payment_date,
229            SUM(invoice_total) AS invoice_total,
230            SUM(invoice_total - credit_total - payment_total) AS balance_due
231     FROM invoices
232     WHERE invoice_date BETWEEN '2018-07-24' AND '2018-07-31'
233     GROUP BY invoice_date, payment_date WITH ROLLUP;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

invoice_date	payment_date	invoice_total	balance_due
2018-07-24	NULL	503.20	503.20
2018-07-24	2018-08-19	3689.99	0.00
2018-07-24	2018-08-23	67.00	0.00
2018-07-24	2018-08-27	23517.58	0.00
2018-07-24	NULL	27777.77	503.20
2018-07-25	2018-08-22	1000.46	0.00
2018-07-25	NULL	1000.46	0.00
2018-07-28	NULL	90.36	90.36
2018-07-28	NULL	90.36	90.36
2018-07-30	2018-09-03	22.57	0.00
2018-07-30	NULL	22.57	0.00
2018-07-31	NULL	10976.06	10976.06
2018-07-31	NULL	10976.06	10976.06
NULL	NULL	39867.22	11569.62

Summary for one or more invoices with NULL payment_date

Summary for 24 July 2018

GROUPING function allows you to tell which rows are created by WITH ROLLUP

```
235    -- A query that substitutes literals for nulls in summary rows
236    SELECT IF(GROUPING(invoice_date) = 1, 'Grand totals',
237              invoice_date) AS invoice_date,
238           IF(GROUPING(payment_date) = 1, 'Invoice date totals',
239              payment_date) AS payment_date,
240           SUM(invoice_total) AS invoice_total,
241           SUM(invoice_total - credit_total - payment_total)
242              AS balance_due
243    FROM invoices
244    WHERE invoice_date BETWEEN '2018-07-24' AND '2018-07-31'
245    GROUP BY invoice_date, payment_date WITH ROLLUP;
246
```

IF function will replace default value (NULL) with stated string literal if the row was created by WITH ROLLUP

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

invoice_date	payment_date	invoice_total	balance_due
2018-07-24	NULL	503.20	503.20
2018-07-24	2018-08-19	3689.99	0.00
2018-07-24	2018-08-23	67.00	0.00
2018-07-24	2018-08-27	23517.58	0.00
2018-07-24	Invoice date totals	27777.77	503.20
2018-07-25	2018-08-22	1000.46	0.00
2018-07-25	Invoice date totals	1000.46	0.00
2018-07-28	NULL	90.36	90.36
2018-07-28	Invoice date totals	90.36	90.36
2018-07-30	2018-09-03	22.57	0.00
2018-07-30	Invoice date totals	22.57	0.00
2018-07-31	NULL	10976.06	10976.06
2018-07-31	Invoice date totals	10976.06	10976.06
Grand totals	Invoice date totals	39867.22	11569.62

Remaining NULLS refer to one or more invoices with NULLs in the payment_date column in the underlying data for each invoice_date

Substituted string literals make it easier to read summaries created by WITH ROLLUP

USING can be used for equijoins with common field names

Can join with **USING** when joining columns sharing same name in both tables

```
296    -- Use the USING keyword to join two tables
297    SELECT invoice_number, vendor_name
298    FROM vendors
299        JOIN invoices USING (vendor_id)
300    ORDER BY invoice_number;
```

Result Grid | Filter Rows: | Export: | Wra

invoice_number	vendor_name
0-2058	Malloy Lithographing Inc
0-2060	Malloy Lithographing Inc
0-2436	Malloy Lithographing Inc
1-200-5164	Federal Express Corporation
1-202-2978	Federal Express Corporation
10843	Yesmed, Inc
109596	Coffee Break Service
111-92R-10092	Pacific Bell
111-92R-10093	Pacific Bell
111-92R-10094	Pacific Bell
111-92R-10095	Pacific Bell
111-92R-10096	Pacific Bell

Compare to Query A

Example with **three tables** and both **inner and outer** joins

```
302    -- Use the USING keyword to join three tables
303    SELECT department_name, last_name, project_number
304    FROM ex.departments
305        JOIN ex.employees USING (department_number)
306        LEFT JOIN ex.projects USING (employee_id)
307    ORDER BY department_name;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Co

department_name	last_name	project_number
Accounting	Hernandez	P1011
Maintenance	Hardy	NULL
Payroll	Simonian	P1012
Payroll	Smith	P1012
Payroll	Aaronsen	P1012
Personnel	O'Leary	P1011
Personnel	Jones	NULL

Reminder: equijoin means that the equals operator is used to compare the two columns in a join condition

A NATURAL JOIN relies on tables having the right field names in common to perform inner joins

```
316     -- Use the NATURAL keyword to join tables
317 •   SELECT invoice_number, vendor_name
318     FROM vendors
319         NATURAL JOIN invoices
320     ORDER BY invoice_number;
```

Result Grid | Filter Rows: | Export:

invoice_number	vendor_name
0-2058	Malloy Lithographing Inc
0-2060	Malloy Lithographing Inc
0-2436	Malloy Lithographing Inc
1-200-5164	Federal Express Corporation
1-202-2978	Federal Express Corporation
10843	Yesmed, Inc
109596	Coffee Break Service
111-92R-10092	Pacific Bell
111-92R-10093	Pacific Bell
111-92R-10094	Pacific Bell

JOINs tables based on all the columns in the two tables that have the same name

In this case just the vendor_id fields in both tables

Relies on careful naming of columns. Not as commonly used as ON or USING

Compare to Query A