SQL part2

(Chapters 4,5)

Ambiguous Name Columns

Two tables can have attributes with the same name.

*Beers (**name**, manf)*
*Bars (**name**, addr, license)*

Problem: this ambiguity can lead to confusion if you write a query involving two tables with common column names: if two same names appear in the WHERE clause

```
SELECT Bars.name
FROM Bars, Beers
WHERE name = name;
```

SQL: "**ERROR**: Ambiguous name column".

Qualified Column Names

Solution: **disambiguate** attributes by specifying the relation name (giving a ***qualified name*** of a column)

- e.g. Bars.name means the column name from table Bars

We typically qualify a column name to specify the table which the column comes from. (see previous example below)

```
SELECT Bars.name
FROM Bars, Beers
WHERE Bars.name = Beers.name;
```

Qualified Column Names

Question: can I use ***qualified names*** even if there is no ambiguity?

```
SELECT Sells.beer
FROM Sells
WHERE Sells.price > 3.00;
```

Giving an Alias (Temporary Name)

You can rename the attributes in a result of a query using **AS**

1. columns in SELECT part OR
2. on the table in the FROM part.

Example: Changing name of attributes in Beers

SELECT name **AS Brand**, manf **AS Brewer**
FROM Beers;

BRAND — *BREWER*

80/- — *Caledonian*
Bigfoot Barley Wine — *Sierra Nevada*
…

Beers:

Name	Manf
80/-	Caledonian
Bigfoot Barley Wine	Sierra Nevada
Burragorang Bock	George IV Inn
Crown Lager	Carlton
Fosters Lager	Carlton
Invalid Stout	Carlton
Melbourne Bitter	Carlton
New	Toohey's
Old	Toohey's
Old Admiral	Lord Nelson
Pale Ale	Sierra Nevada
Premium Lager	Cascade
Red	Toohey's
Sheaf Stout	Toohey's
Sparkling Ale	Cooper's
Stout	Cooper's
Three Sheets	Lord Nelson
Victoria Bitter	Carlton

Does not change the names of the underlying relation

Example

Query: What if we find **pairs of beers** by the same manufacturer.

We would have to employ a self-join

```
SELECT name, name
FROM Beers, Beers
WHERE manf = manf  AND name < name;
```

Issue: Similar problem with ambiguous name, the attributes have the same name…

1. can we fix it with a qualified column name?
2. what about a qualified table name?

Beers:

Name	Manf
80/-	Caledonian
Bigfoot Barley Wine	Sierra Nevada
Burragorang Bock	George IV Inn
Crown Lager	Carlton
Fosters Lager	Carlton
Invalid Stout	Carlton
Melbourne Bitter	Carlton
New	Toohey's
Old	Toohey's
Old Admiral	Lord Nelson
Pale Ale	Sierra Nevada
Premium Lag	
Red	
Sheaf Stout	
Sparkling Ale	
Stout	
Three Sheets	
Victoria Bitte	

Beers:

Name	Manf
80/-	Caledonian
Bigfoot Barley Wine	Sierra Nevada
Burragorang Bock	George IV Inn
Crown Lager	Carlton
Fosters Lager	Carlton
Invalid Stout	Carlton
Melbourne Bitter	Carlton
New	Toohey's
Old	Toohey's
Old Admiral	Lord Nelson
Pale Ale	Sierra Nevada
Premium Lager	Cascade
Red	Toohey's
Sheaf Stout	Toohey's
Sparkling Ale	Cooper's
Stout	Cooper's
Three Sheets	Lord Nelson
Victoria Bitter	Carlton

Example

To handle this, we need to define new names for each "instance" of the relation in the FROM clause.

Example: Find pairs of beers by the same manufacturer.

```
SELECT b1.name, b2.name
FROM Beers AS b1, Beers AS b2
WHERE b1.manf = b2.manf AND
b1.name < b2.name;
```

This does the same thing

```
SELECT b1.name, b2.name
FROM Beers b1, Beers b2
WHERE b1.manf = b2.manf AND
b1.name < b2.name;
```

Beers:

Name	Manf
80/-	Caledonian
Bigfoot Barley Wine	Sierra Nevada
Burragorang Bock	George IV Inn
Crown Lager	Carlton
Fosters Lager	Carlton
Invalid Stout	Carlton
Melbourne Bitter	Carlton
New	Toohey's
Old	Toohey's
Old Admiral	Lord Nelson
Pale Ale	Sierra Nevada
Premium Lag	
Red	
Sheaf Stout	
Sparkling Ale	
Stout	
Three Sheets	
Victoria Bitte	

Beers:

Name	Manf
80/-	Caledonian
Bigfoot Barley Wine	Sierra Nevada
Burragorang Bock	George IV Inn
Crown Lager	Carlton
Fosters Lager	Carlton
Invalid Stout	Carlton
Melbourne Bitter	Carlton
New	Toohey's
Old	Toohey's
Old Admiral	Lord Nelson
Pale Ale	Sierra Nevada
Premium Lager	Cascade
Red	Toohey's
Sheaf Stout	Toohey's
Sparkling Ale	Cooper's
Stout	Cooper's
Three Sheets	Lord Nelson
Victoria Bitter	Carlton

An alias is helpful

AS can **also be used to make a column name more readable**

SELECT bar, beer, price*120 FROM Sells;

BAR	*BEER*	*price*120*
Australia Hotel	*Burragorang Bock*	*420*
Coogee Bay Hotel	*New*	*270*

SELECT bar, beer, price*120 **AS PriceInYen** FROM Sells;

BAR	*BEER*	***PRICEINYEN***
Australia Hotel	*Burragorang Bock*	*420*
Coogee Bay Hotel	*New*	*270*

Aggregate Functions in SQL

Selection clauses can contain aggregation operations.

Example: What is the average price of New?

```
SELECT AVG(price)
FROM Sells
WHERE beer = 'New';
```

AVG(PRICE)

2.3875

Bar	Beer	Price
Coogee Bay Hotel	New	2.25
Marble Bar	New	2.8
Regent Hotel	New	2.2
Royal Hotel	New	2.3

Aggregation in SQL

Aggregation follows a multi-set semantic.

We can specify a set semantic by using DISTINCT in the aggregation function.

```
SELECT COUNT(DISTINCT bar)
FROM Sells;
```

By default

```
SELECT COUNT(ALL bar)
FROM Sells;
```

Null Values and Aggregates

Aggregation ignores NULL values if present in the column specified.

Total all salaries: select sum (salary) from instructor

- Above statement ignores null amounts
- Result is null if there is no non-null amount

All aggregate operations ignore tuples with null values on the aggregated attributes: except count(*)

Aggregation & Null Values

select count(A1) from R

- The answer is 3.

select count(distinct A1) from R

- The answer is 3.

As specified by SQL

- Count(*) counts null and non-null tuples
- Count(attribute) counts all non-null tuples

A1	A2	A3	A4
5	9	alpha	x
	4	beta	
2	4	gamma	
3		delta	x

Practice:

- What would count(A1) if all values in column in A1 were NULL?
- What would max(A1) if all values in column in A1 were NULL?

Common Aggregations

List of several built-in aggregate functions exist in SQL:

- SUM ()
- AVG ()
- MIN ()
- MAX ()
- COUNT ()

The above operators apply to numeric values in one column of a relation.

Exception: when using count(*).

Grouping and Aggregation

What is grouping? Grouping is the application of aggregate functions *to* ***subgroups of tuples*** *of a table*

Grouping is typically used in queries involving the phrase "for each"

Example: I now know the SUM() of money **all employees** has made, but I'm **more interested in** the money made by **each employee**…

Grouping and Aggregation

In many cases we want to apply the aggregate functions *to subgroups of tuples in a relation,* where the subgroups are based on some attribute values.

Syntax:
SELECT attributes and aggregations
FROM relations
WHERE condition
***GROUP BY** attribute*

Operational Semantics:

1. Partition result relation into groups based on distinct values of attribute
2. Apply the aggregation(s) on each group separately

Grouping and Aggregation

Example: For each drinker, find the average price of New at the bars they frequently go to.

```
SELECT drinker, AVG(price)
FROM Frequents, Sells
WHERE beer = 'New' AND Frequents.bar = Sells.bar
GROUP BY drinker;
```

DRINKER	*AVG(PRICE)*
Adam	*2.25*
John	*2.25*
Justin	*2.5*

Grouping and Aggregation

To use group by correctly, every attribute in the SELECT list must either

- be inside an aggregate function OR
- be in the GROUP-BY clause without an aggregate function

Is this correct? No

SELECT *dept_name*, *ID*, **AVG**(*salary*)
FROM *instructor*
GROUP BY *dept_name*;

ID	*name*	*dept_name*	*salary*
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

Grouping and Aggregation

```
SELECT bar, MIN(price)
FROM Sells
GROUP BY bar;
```

Bar	*MIN(PRICE)*
Australia Hotel	*3.5*
Coogee Bay Hotel	*2.25*
Lord Nelson	*3.75*
Marble Bar	*2.8*
…	

What if I only wanted results with a min(price) >2.3?

Group by and Having

SQL provides a HAVING clause, which can appear in conjunction with a GROUP BY clause.

SELECT attributes and aggregations
FROM relations
WHERE condition
***GROUP BY** attribute*
***HAVING** condition_on_group;*

Example:

SELECT bar, **MIN(price)**
FROM Sells
GROUP BY BAR
HAVING MIN(price) > 2.3;

Bar	*MIN(PRICE)*
Australia Hotel	*3.5*
Coogee Bay Hotel	*2.25*
Lord Nelson	*3.75*
Marble Bar	*2.8*
Royal Hotel	*2.3*
…	

Thought: can we put the condition on select instead?

Summary on SQL

There are a maximum of six clauses that could be used in a SELECT stmt.

SELECT *<attribute and function list>*

FROM *<table list>*

[***WHERE*** *<condition>]*

[***GROUP BY*** *<grouping attribute(s)>]*

[***HAVING*** *<group condition>]*

[***ORDER BY*** *<attribute list>];*

Note on notation: clauses between square brackets [...] being optional

Querying **With** Subqueries

Example: Find bars that sell New at the price same as the Coogee Bay Hotel charges for VB.

Simplest Case: Subquery returns one tuple. **IF** the subquery returns one tuple, you can treat the result as a constant value.

```
SELECT bar
FROM Sells
WHERE beer = 'New' AND price = (
SELECT price
FROM Sells
WHERE bar = 'Coogee Bay Hotel'
AND beer = 'Victoria Bitter'
);
```

Price
2.3

Sells:

Bar	Beer	Price
Australia Hotel	Burragorang Bock	3.5
Coogee Bay Hotel	New	2.25
Coogee Bay Hotel	Old	2.5
Coogee Bay Hotel	Sparkling Ale	2.8
Coogee Bay Hotel	Victoria Bitter	2.3
Lord Nelson	Three Sheets	3.75
Lord Nelson	Old Admiral	3.75
Marble Bar	New	2.8
Marble Bar	Old	2.8
Marble Bar	Victoria Bitter	2.8
Regent Hotel	New	2.2
Regent Hotel	Victoria Bitter	2.2
Royal Hotel	New	2.3
Royal Hotel	Old	2.3
Royal Hotel	Victoria Bitter	2.3

Querying **Without** Subqueries

Example: Find bars that sell New at the price same as the Coogee Bay Hotel charges for VB.

Can we do the one from before without subqueries? Of course…

```
SELECT b2.bar
FROM Sells b1, Sells b2
WHERE b1.beer = 'Victoria Bitter' and
        b1.bar = 'Coogee Bay Hotel' and
        b1.price = b2.price and b2.beer = 'New';
```

This will produce the same results, but what might it be doing here instead?

Query with Subquery

Regular Case: Subquery returns multiple tuples/a relation.

Common usage of subqueries:

- compare the membership of one relation to that of another,
- compare results of one set to that of another,

IN

The IN operator is typically used to filter a column. It can be used in subqueries to filter a column for a certain list of values.

```
SELECT * FROM Students WHERE Grade IN ('HD', 'D');
```

Example: find the name and brewers of beers that John likes.

```
SELECT *
FROM Beers
WHERE name IN
            (SELECT beer
             FROM Likes
             WHERE drinker = 'John');
```

Exists

Exists keyword returns **true** if the relation is non-empty.

Example: Find the beers uniquely made by their manufacturer.

```
SELECT name
FROM Beers b1
WHERE NOT EXISTS
        (SELECT *
         FROM Beers
         WHERE manf = b1.manf AND name != b1.name);
```

Note: add NOT before EXISTS to express the opposite condition (NOT EXISTS).

All

Find the names of all instructors whose salary is greater than the salary of all instructors in the Physics department.

ID	name	dept_name	salary
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

(a) The *instructor* table

```
SELECT name
FROM instructor
WHERE salary > ALL (SELECT salary
                    FROM instructor
                    WHERE dept name = 'Physics');
```

Divide from RA in SQL?

Example: Find bars each of which sell all beers Justin likes.
Relational Algebra: *πbar,beerSells ÷ (πbeer(σdrinker='Justin' Likes))*

Bar	Beer	Price
Australia Hotel	Burragorang Bock	3.5
Coogee Bay Hotel	New	2.25
Coogee Bay Hotel	Old	2.5
Coogee Bay Hotel	Sparkling Ale	2.8
Coogee Bay Hotel	Victoria Bitter	2.3
Lord Nelson	Three Sheets	3.75
Lord Nelson	Old Admiral	3.75
Marble Bar	New	2.8
Marble Bar	Old	2.8
Marble Bar	Victoria Bitter	2.8
Regent Hotel	New	2.2
Regent Hotel	Victoria Bitter	2.2
Royal Hotel	New	2.3
Royal Hotel	Old	2.3
Royal Hotel	Victoria Bitter	2.3

Drinker	Beer
Adam	Crown Lager
Adam	Fosters Lager
Adam	New
Gernot	Premium Lager
Gernot	Sparkling Ale
John	80/-
John	Bigfoot Barley Wine
John	Pale Ale
John	Three Sheets
Justin	Sparkling Ale
Justin	Victoria Bitter

Divide from RA in SQL

Question: How do we do this in SQL?

Answer: This gives you the same results as a divide operator

```
SELECT DISTINCT a.bar
FROM sells a
WHERE NOT EXISTS
        ( (SELECT b.beer FROM likes b WHERE b.drinker = 'Justin')
           EXCEPT
          (SELECT c.beer FROM sells c WHERE c.bar = a.bar )
        );
```

(Example: Find bars each of which sell all beers Justin likes)

SQL Views

Tables (created by CREATE TABLE) are ***base relations***
Views are ***virtual relations*** in SQL.

Views are derived from base relations and does not take up space in the DBMS. Base relations are physically stored in the DBMS

1. View are defined via:
   CREATE VIEW View_name AS Query
2. Views may be removed via:
   DROP VIEW View_name

Motivation

Example: an avid CUB drinker might not be interested in any other kinds of beer.

```
CREATE VIEW MyBeers AS
        SELECT name, manf
        FROM Beers
        WHERE manf = 'Carlton';
```

NAME	*MANF*
Crown Lager	*Carlton*
Fosters Lager	*Carlton*
Invalid Stout	*Carlton*
Melbourne Bitter	*Carlton*
Victoria Bitter	*Carlton*

Motivation

Example: we don't really all the other columns of inner-city hotels.

```
CREATE VIEW InnerCityHotels AS
        SELECT name, license
        FROM Bars
        WHERE addr = 'The Rocks' OR addr = 'Sydney';
```

NAME	*LICENSE*
Australia Hotel	*123456*
Lord Nelson	*123888*
Marble Bar	*122123*

Note: a view can help if you are not interested in all attributes of a base relation.

Querying Views

Example: Using the InnerCityHotels view.

```
CREATE VIEW InnerCityHotels AS
        SELECT name, license
        FROM Bars
        WHERE addr IN ('The Rocks', 'Sydney');
```

Views can be used in queries just as if they were base (stored) relations.

```
SELECT pub FROM InnerCityHotels WHERE lic = '123456';
```

This makes views especially useful, can you think of an usages?

Views Update

1. **SELECT * FROM *InnerCityHotels*;**

NAME	LICENSE
Australia Hotel	123456
Marble Bar	12212
Lord Nelson	13

Recall View Definition:

```
CREATE VIEW InnerCityHotels AS
        SELECT name, license
        FROM Bars
        WHERE     addr = 'The Rocks' OR
                  addr = 'Sydney';
```

2. UPDATE *Bars* SET license='111223' WHERE name='Lord Nelson';

3. **SELECT * FROM *InnerCityHotels*;**

NAME	LICENSE
Australia Hotel	123456
Marble Bar	12212
Lord Nelson	111223

Conclusion: Views update themselves automatically, if changes occur in the underlying relation(s).

Schema Change in SQL

Sometimes, we want to make changes to the table schema.

The definition of a base table or of other named schema elements can be changed by using the **ALTER** command.

1. Add column(s) of table
2. Delete column(s) of table
3. Modify column(s) of table

Can be accomplished via the **ALTER TABLE** operation:

(you don't have to alter tables in the upcoming project)

Example: Add New Column

Example: **Add column** phone numbers **to table** hotels.

ALTER TABLE Bars
ADD phone char(10) DEFAULT 'Unlisted';

This appends a new column to the table and sets value for this attribute to *'Unlisted*' in every tuple.

Note: If no default values is given, values new column is set to all NULL.

Example: Modify Column

Example: Changing the primary key

ALTER TABLE Persons DROP PRIMARY KEY;
OR
ALTER TABLE Persons ADD PRIMARY KEY (ID);

Note*: **Typically, any changes need to be allowable with respect to any existing data.***

In this case: any values in the new primary key column must obey NOT NULL, UNQIUE constraint.

So before you can specify the new primary key, you must update the table to delete all NULL, or non-unique values. e.g., *UPDATE table SET col = 0 WHERE col IS NULL;* (assuming col is numerical)