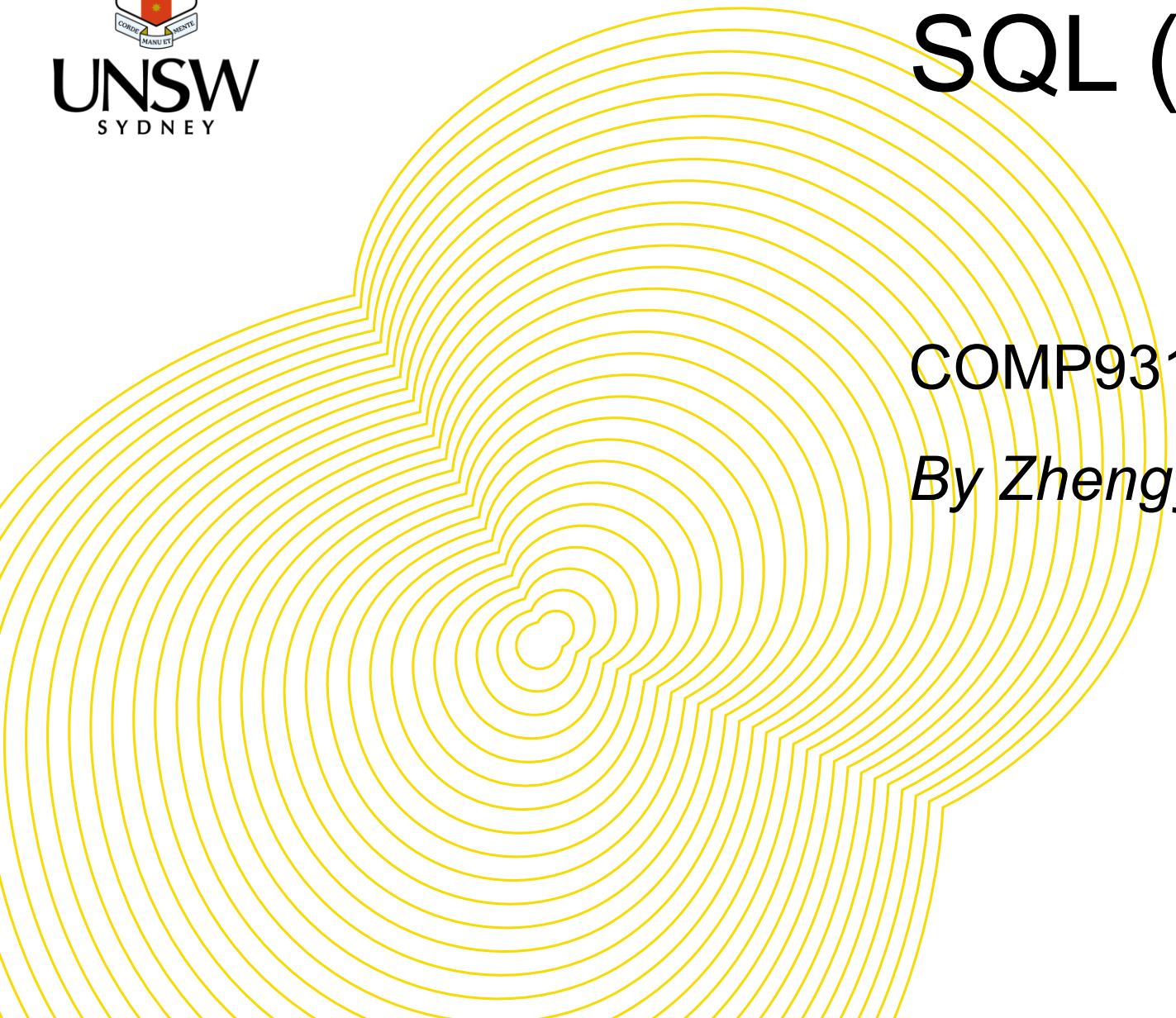

SQL (Part 2)

COMP9311 24T2; Week 3.2

By Zhengyi Yang, UNSW

Notice

- Assignment 1 due on **Sun 22:00 16th June**
- 5% reduction per day for late submission, up to 5 days
- Will be checked for plagiarism

Querying With Subqueries

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

Simplest Case: Subquery returns one tuple/value. 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

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 same price as the Coogee Bay Hotel charges for Victoria Bitter.

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’;
```

Query with Subquery

Regular Case: Subquery returns multiple tuples/a relation.

Typical usage of subqueries:

- compare the membership of one relation to that of another
- compare the 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 names 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 those beers that are the unique beer 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).

Quantifiers

ANY and **ALL** behave as existential and universal quantifiers respectively.

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ANY
        (SELECT column_name
        FROM table_name
        WHERE condition);
```

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ALL
        (SELECT column_name
        FROM table_name
        WHERE condition);
```

All

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

```
SELECT name
FROM instructor
WHERE salary > ALL
        (SELECT salary
        FROM instructor
        WHERE dept_name='Comp. Sci.');
```

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

Einstein's salary is 95,000, which is higher than all instructors from Comp. Sci.

ANY

Find the students who fail at least one course in T1.

```
SELECT zid, name
FROM students
WHERE 50 > ANY
        (SELECT score
        FROM enrollment
        WHERE students.zid= enrollment.zid AND term='T1');
```

Jack fails comp9024 in T1. He only gets 45, which is smaller than 50.

student

zid	name
z123	Deniel
z456	Jack

enrollment

zid	course_id	term	score
z123	comp9311	T1	80
z123	comp9315	T2	75
z456	comp9311	T1	55
z456	comp9024	T1	45

Set Operations

Find courses that ran in Fall 2009 or in Spring 2010

(**select** course_id **from** section **where** sem = 'Fall' **and** year = 2009)
union
(**select** course_id **from** section **where** sem = 'Spring' **and** year = 2010)

Find courses that ran in Fall 2009 and in Spring 2010

(**select** course_id **from** section **where** sem = 'Fall' **and** year = 2009)
intersect
(**select** course_id **from** section **where** sem = 'Spring' **and** year = 2010)

Find courses that ran in Fall 2009 but not in Spring 2010

(**select** course_id **from** section **where** sem = 'Fall' **and** year = 2009)
except
(**select** course_id **from** section **where** sem = 'Spring' **and** year = 2010)

Note: Each of the above operations will eliminate duplicates.
To keep duplicates, use **union all, intersect all, except all.**

DIVIDE in SQL

Example: Find bars each of which sells all the beers Justin likes.

Relational Algebra: $\pi_{bar,beer} Sells \div (\pi_{beer}(\sigma_{drinker='Justin'} Likes))$

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 in SQL

Example: Find bars each of which sells all the beers Justin likes.

How do we do this in SQL?

```
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 )
        );
```

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';
```

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

AVG(PRICE)

2.3875

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;
```

Common Aggregations

A 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: COUNT(*) gives the number of tuples in a relation.

Example: How many different beers are there?

```
SELECT COUNT(*) FROM Beers;

COUNT(*)
----------
18
```

Null Values and Aggregates

Aggregation ignores NULL values if present in the column specified.

Total all salaries: `select sum(salary) from instructor;`

- The above statement ignores null values
- Result is null if all salaries are null

All aggregate operations ignore tuples with null values on the aggregated attributes, except COUNT(*), which counts the number of rows.

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?

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? Answer: 0
- What would max(A1) if all values in column in A1 were NULL? Answer: NULL

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:

- Partition result relation into groups based on distinct values of attribute
- 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 query 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

Example: Find the cheapest beer price in each bar.

```
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
…	

Order By

The result relation of a SQL query has no particular order since it's a (multi-)set. We can enforce an order using order by.

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1, column2, ... ASC|DESC;
```

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

Create Table Statement

A relational database can store a lot of relations, each relation is created with the ***create table*** statement.

Example: create a relation that stores info about people.

```
CREATE TABLE person(
        drivers_license VARCHAR(20) PRIMARY KEY,
        name VARCHAR(20)
);
```

Drop Table Command

When a relational database no longer needs a relations, a relation can be deleted from the database using the ***drop table*** statement.

DROP TABLE person;

Note: Just like create, ***drop*** *is the main command, where* ***table*** *is one of the many arguments drop takes.*

Table Attributes - Domain

Within the create table command, we specify it's the columns/attributes, declare properties of the table and the properties of each attribute

```
CREATE TABLE table (
        attribute1  datatype [properties],
        attribute2  datatype [properties],
        …
        [table property1]
        [table property2]
        …
);
```

Table Attributes - Domain

The syntax requires that attribute in an SQL table must specify a domain.

```
CREATE TABLE table (
        attribute1  datatype [properties],
        attribute2  datatype [properties],
        …
        [table property1]
        [table property2]
        …
);
```

The domain puts constraints on the value that an attribute is allowed to take. Defining the ***data type*** *specifies the domain.*

Attribute Constraints

Attributes properties can be used to “enforce” more complex domain membership conditions.

```
CREATE TABLE table (
        attribute1  datatype [properties],
        attribute2  datatype [properties],
        …
        [table property1]
        [table property2]
        …
);
```

Attribute Constraints

By default, the NULL value is a member of all data types.

Example:

```
CREATE TABLE Likes (
        drinker VARCHAR(20) NOT NULL,
        beer VARCHAR(30)
);
```

More examples: UNIQUE, CHECK, DEFAULT, …

Declaring Primary Keys

Declare primary key constraint with the table property

- **primary key** (Ai, ...,)

Example: declare the name of a person to be primary key

```
CREATE TABLE Person (
        name VARCHAR(20),
        PRIMARY KEY (name)
);
```

Primary key declaration on an attribute ensures **not null** and **unique**

Declaring Foreign Keys

Declare foreign keys with the foreign key constraint

- **foreign key** (Ai, ...,) **references** R(Aj,…,)

Example: declare the name of a person to be primary key

```
CREATE TABLE employee (
        name VARCHAR(20),
        works_at VARCHAR(20),
        FOREIGN KEY (works_at)  REFERENCES company(id)
);
```

Declaring Foreign Keys

Declaring foreign keys assures referential integrity.

Foreign a key:

- specify Relation (Attribute) to which it refers.

For instance, if we want to delete a tuple from Beers, and there are tuples in Sells that refer to it, we could either:

- reject the deletion
- cascade the deletion and remove Sells records
- set-NULL the foreign key attribute

Can force cascade via **ON DELETE CASCADE** after REFERENCES.

Declaring Foreign Keys

The FOREIGN KEY constraint prevents actions that would destroy links between tables.

If you delete a value of a primary key that is referred to in other tables as a foreign key, you risk violating the referential integrity. There are several ways to handle this.

If you prefer to delete all records referring to the foreign key value in the database, you can specify ON DELETE CASCADE

```
CREATE TABLE Sells (
    bar VARCHAR(30),
    beer VARCHAR(30),
    price FLOAT,
    FOREIGN KEY(bar) REFERENCES Bars(name) ON DELETE CASCADE,
    FOREIGN KEY(beer) REFERENCES Beers(name) ON DELETE CASCADE
);
```

Insert Tuple(s) using values

The **INSERT** command inserts new tuple(s) into a table

INSERT INTO Relation **VALUES** (val1, val2, val3, ...)

(val1, val2, val3, ...) is a tuple of values

Example: Add the fact that Justin likes 'Old'.

```
INSERT INTO Likes VALUES ('Justin', 'Old');
```

The following are the same

```
INSERT INTO Sells (price,bar) VALUES (2.50, 'Coogee Bay Hotel');
INSERT INTO Sells (bar,price) VALUES ('Coogee Bay Hotel', 2.50);
```

Note: The order of the attributes in VALUES can be different from the SQL table definition as long as the order is specified in the INTO clause.

Insert Tuple(s) using values

Basic attribute constraint checking by SQL

E.g., suppose the table specified that drinkers' phone numbers cannot be NULL.

ALTER TABLE Drinkers **ALTER COLUMN** phone SET NOT NULL;

And then try to insert a new drinker whose phone number we don't know:

INSERT INTO Drinkers(name, addr) **VALUES** ('Zoe', 'Manly');

ERROR: null value in column "phone" violates not-null constraint
DETAIL: Failing row contains (Zoe, Manly, null).

Insert Tuple(s) using Select

We can use the result of a query to perform insertion of multiple tuples at once.

- **INSERT INTO** Relation (Subquery);

Condition: Tuples of subquery **must be projected into a suitable format** (by matching the tuple-type of Relation).

A ***subquery:***

- a query that is nested inside an outer query: SELECT , INSERT , UPDATE , or DELETE statement.

Subqueries:

- can also be nested inside another subqueries.
- are very helpful.

Insert Tuple(s) using Select

For Example: Populate a relation of John's potential drinking buddies (i.e. people who go to the same bars as John).

```
CREATE TABLE DrinkingBuddies (
        name varchar(20)
);

INSERT INTO DrinkingBuddies(
        SELECT DISTINCT f2.drinker
        FROM Frequents f1, Frequents f2
        WHERE f1.drinker = 'John'
                AND f2.drinker != 'John'
                AND f1.bar = f2.bar
);
```

Delete Tuple(s)

The DELETE operation removes all tuples from Table that satisfy a condition.

- **DELETE FROM** Table
  WHERE Condition

Example: Justin no longer likes Sparkling Ale.

- **DELETE FROM** Likes
  WHERE drinker = 'Justin'
  AND beer = 'Sparkling Ale';

Omitting the WHERE Clause deletes all tuples from relation R.

- **DELETE FROM** R;
- This doesn't drop the table, the table still remains

Note: Delete is not the same as Drop

Delete Tuple(s)

Semantics of the above Deletion:

Evaluation of DELETE FROM R WHERE Cond can be viewed as:

```
FOR EACH tuple T in R DO
        IF T satisfies Cond THEN
                make a note of this T
        END
END

FOR EACH noted tuple T DO
        remove T from relation R
END
```

Update Value(s) in Tuple(s)

If you have an tuple but want to change part of its values, use the UPDATE command.

Updates specified attributes in specified tuples of a relation:

```
UPDATE R
SET list of assignments
WHERE Condition
```

Example: John moves to Coogee.

```
UPDATE Drinkers
SET addr = 'Coogee' , phone = '9665-4321'
WHERE name = 'John';
```

Note: Careful because all tuples relation R that satisfies Condition has the assignments applied to it.

Update Value(s) in Tuple(s)

Can update many tuples at once (all tuples that satisfy condition)

Example: Make $3 the maximum price for beer.

```
UPDATE Sells
SET price = 3.00
WHERE price > 3.00;
```

Example: Increase all beer prices by 10%.

```
UPDATE Sells
SET price = price * 1.10;
```

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

- View are defined via:
  CREATE VIEW View_name AS Query
- 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 aren't really interested in 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 name FROM InnerCityHotels WHERE lic = '123456';
```

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

Views Update

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';
```

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

- Add column(s) of table
- Delete column(s) of table
- Modify column(s) of table

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

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.

Create Index

Index helps to speed up retrieval of tuples

```
CREATE INDEX index_name
ON table_name (column1, column2, ...);
```

```
CREATE UNIQUE INDEX index_name
ON table_name (column1, column2, ...);
```

Learning Outcomes

- Basic queries in SQL
- Querying multiple tables
- Aggregate queries
- Insert/delete/update
- Change schemas
- Views