ISYS2120

# Week 1

**Data**

* Fact that can be recorded
  + Important for users
  + Requires persistent management

**Database**

* A collection of data
  + Usually large in quantity
  + Normally contains all the information necessary to operate an organization

**Database Management Syustem (DBMS)**

* Software package designed to store and manage one or more databases
  + Allows shared access from many programs

**System Catalogue, Data Dictionary**

* Storing descriptions of the format of the data
* Often also called “metadata”

**Relational DBMS**

* Most DBMS stores table specific information that are linked via special key
* Nominalized database

**Instance**

* An instance is the content of the database at a single time
  + Specific values which describe a specific situation in the world
  + Every update changes the instance

**Schema**

* The schema describes the **Structure** of the data in a database
  + What tables exists
  + What attributes are present and in what form and datatype
* It is also called “integrity constraints” which restricts the possible instances that go against logical structure

“**instance at any given point must fit the pattern of the schema”**

**Data Design**

* Decision of schema is a very important task which focuses on what information is necessary and in what format.
* Decisions of such charasteristics are called “Data design”
* Stages of Data Design
  + Fist produce a conceptual or semantic model
  + Translate them into a relational schema
  + Evaluate the schema for quality and improve it as needed

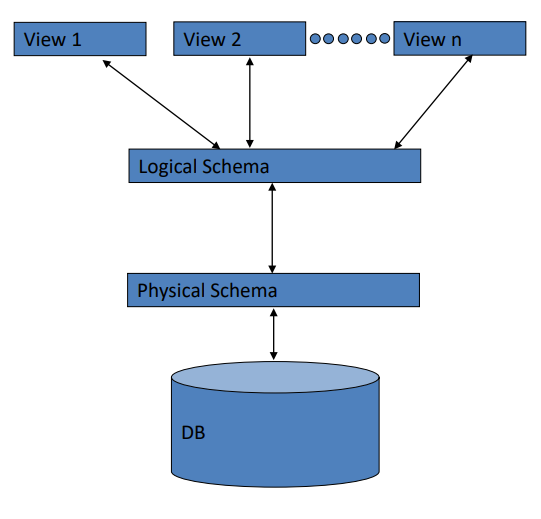
**Data Definition language**

* Used to define the schema
* Allows one to tell DBMS what tables exist and what structure they have
* Create Drop or Alter the Relation Schema

**Data Manipulation Language** (DML)

* Used to access and change data in DBMS
  + Update/Modify
  + Retrieve
  + Insert
  + Delete

**Data Control Language (DCL)**

* Commands that control database, including **administering** **privileges** **and users**

**VIEW:**

* Description of how a user sees the data

**Logical schema**

* Definition of the structure of data as it is shared among all users

**Physical schema**

* Description of the files and indezes used for storage on disk

**Data Independence**

**Logical data independence**

* Protection from changes in logical schema
  + Introducing an extra column in a table

**Physical data independence**

* Protection from changes in physical structure and location of data

Data independence is one of the most important benefit of using a dbms

**Roles within DBMS**

**End Users**

* End users are people who do something that advances the organization’s purpose
  + They run applications that **present** the data to them and allow them to **make controlled changes**
* **Categories**
  + **Offline naïve users**
    - People who gets reports based from the DBMS status
      * Store manager getting weekly profit report
  + **Parametric naïve users**
    - People who execute pre-written applications
      * Deli managers who run application to reorder some items
  + **Ad hoc users**
    - People who explore the data
      * Division manager looking for trendsA

**Application Programmers**

* IT professionals who produce the application that end users can run
  + This usually fit into a broader software development process, with system analysts, project managers, testers, etc.
* Programmers need to understand how to create an application that accesses data through a DBMS
  + As well, they need a range of software engineering skills
    - Quality assurance
    - SDLC processes
    - User interface

**Database Administrator**

* Responsible for management of Organization’s Databse for effective and efficient use of resources in proving access to data
* Example tasks
  + Design local/physical schemas
    - Make trasde-offs between different choices, to get good performance for all users, at reasonable cost in hardware and software
  + Handle security and authorization
    - Set up accounts and permissions
  + Data availability, crash recovery
    - Make sure backups are taken, and used when needed
  + Database tuning
    - Monitor performance and adjust parameters or redesign schemas as needed

**Files vs DBMS**

|  |  |
| --- | --- |
| **Files** | **DBMS** |
| Data definitions repeats in each program | Data is defined once in the central data dictionary of a BDMS |
| You have to program file readings and line managements manually | Declarative query is easy to read, efficient evaluation due to automatic optimizations |
| Data Integrity can be made from anywhere including incorrect input or missing synchronization | Violation of integrity is only possible in a relational sense:   * Record is deleted from the file with the product without checking the orders first |
| You need to keep track of which programs use which files, and modify all those programs when the file structure changes | You only need to change the description in the database, not in the program that use the data |
| Only a file accessibility is controllable | Declarative access control regarding individual database users and user groups  **Views** |

# Disadvantages of using file system

* Data redundancy and inconsistency
  + Multiple file format, duplication of information in different files
* Difficult in accessing data
  + Need to write algorithm each time you want to access a file
* No central authority, or security
  + Different programmers might want different choices of files and formats, and there is no easy way to enforce organizational control over the valuable data
* Integrity problems
  + Integrity constraints become a part of program code
* Concurrent access by multiple users may lead to undesired outcomes or unsynchronized file managements
* Security problems
* Atomicity of updates
  + Failure may leave database in an inconsistent state with partial updates carried out

SQL statement is in the “What information is needed” than “How to retrieve information”

**SQL Select Statement**

**SELECT** List the columns that should be returned from the query

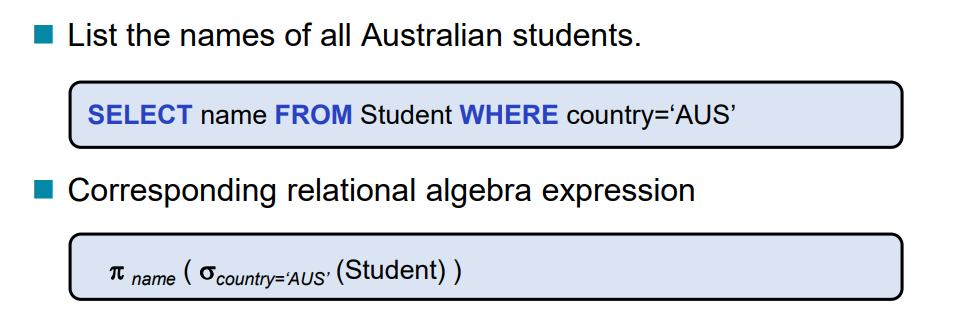
**FROM** Indicate the tables from which data will be obtained from

**WHERE** Indicate the conditions to include a tuple in the result

**GROUP BY** Indicate the categorization of tuples

**HAVING** Indicate the conditions to include a category

**ORDER BY** Sorts the result according to specified criteria



**SELECT**

* **Distinct** To force the elimination of duplicates
* **All** To allow duplicates to be included
* **As** Renaming relations and attributes using the **as** claese
  + - Only in the front end, does not change the name of actual back end attribute name
* **Aggregate Functions**
  + **Avg** Average Value
  + **Min** Minimum value
  + **Max** Maximum value
  + **Sum** Sum of values
  + **Count** Number of values

**FROM**

* **JOIN operations**

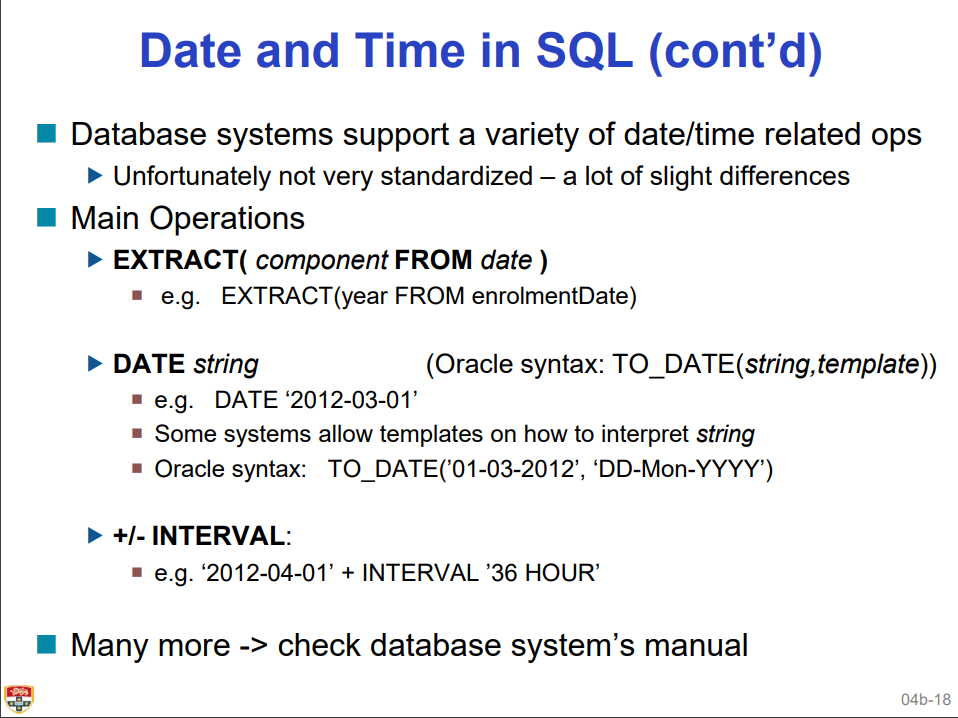
**WHERE**

* Comparison operators in SQL
  + =, >, <, <=, !=, <>
* **AND, OR, NOT**
* **BETWEEN** \* **AND** \* Allows specific range definitions
* \* **LIKE “”** Allows String Wild Card operations
  + % Any number of Substrings
  + \_ One Character
* **||** String Concatenations

**ORDER BY**

* **ASC** Ascending order (default)
* **DESC** Descending order

**Date and Time in SQL**



**Null**

* Aggrivate functions except Count() ifnores null values on the aggregated attributes
* **PRO**
  + Useful because using an ordinary value with special meaning does not always work
    - Put -1 for instance, if you want to put -1 for unknown cell value, then if you want to call aggregate function sum() then it goes to hell.
* **Cons**
  + Null causes complications in the definition of many operations

Database Design Sequence

**Requirement Analysis**

* Understand
  + What data is to be stored
  + What applications must be built
  + What operations are most frequent

**Conceptual Design**

* Develop
  + High level description of the data closely matching how users think of the data
  + Works as communication vehicle

**Logical Design**

* Convert
  + Conceptual design into a logical database schema

**Physical Design**

* Convert
  + Logical schema into a physical schema for a specific DBMS and tune for app

**Conceptual Data Model**

**Aim:** specification of database schema

**Conceptual Data Model**

* Technique for understanding and capturing business information requirements graphically
* It does not imply how data is implemented, created, modified, used or deleted
* Focuses on the internal relationships between databases

**Entity Relationship Model**

* High level graphical representation of what Data needs to be contained in the system
* Associations among different categories of data within a business or information system
  + What are the **entities** and **relationships** in the enterprise

**Entity**

* Object about wghich you want o gather and store data
* Distinguishable from other entities

**Entity Type**

* Collection of entities that share common properties or characteristics
  + Eg: Students, courses , accounts
  + Rectangles represent entity type

**Attribute**

* Describes one aspect of an entity type
  + People have names and addresses
  + Ellipses represent Attributes

**Entity Type**

* Described by a set of attributes
  + Descriptive properties possessed by all members of an entity type

**Domain**

* Possible values of an attribute

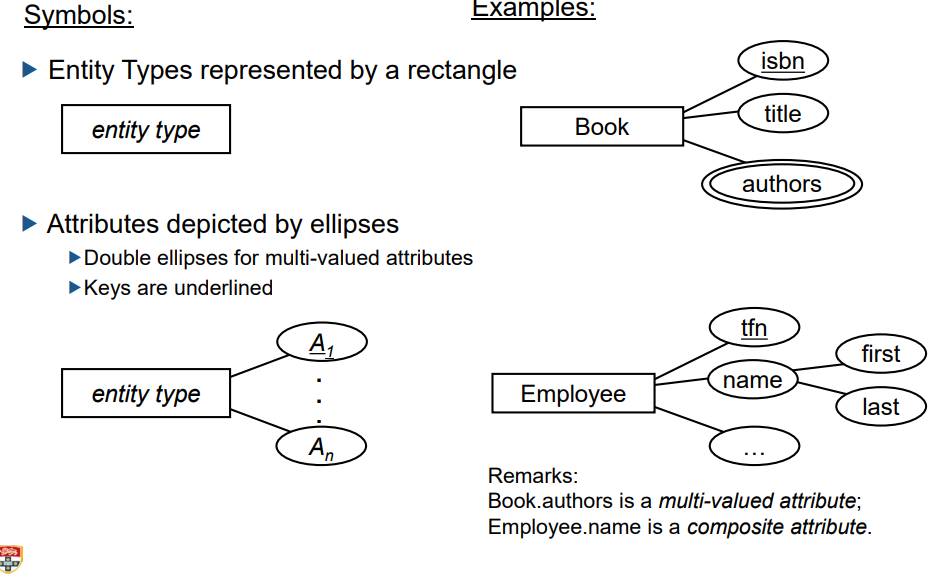
**Key**

* Minimal set of attributes that uniquely identifies an entity in the set]
* Underline in the attribute represents a primary key **PK**

**Entity Schema**

* Entity type name, attributes and PK

**ER Diagram representation**



**Relationships**

* Relates two or more entities
* Number of enttiies is also known as the **degree** of the relationship

**Relationship Type**

* Set of similar relationships
* Shown with a Diamond

**Distinction**

* Relation set of tuples
* Relationship describes relationship between entities

**Relationship Attributes and Roles**

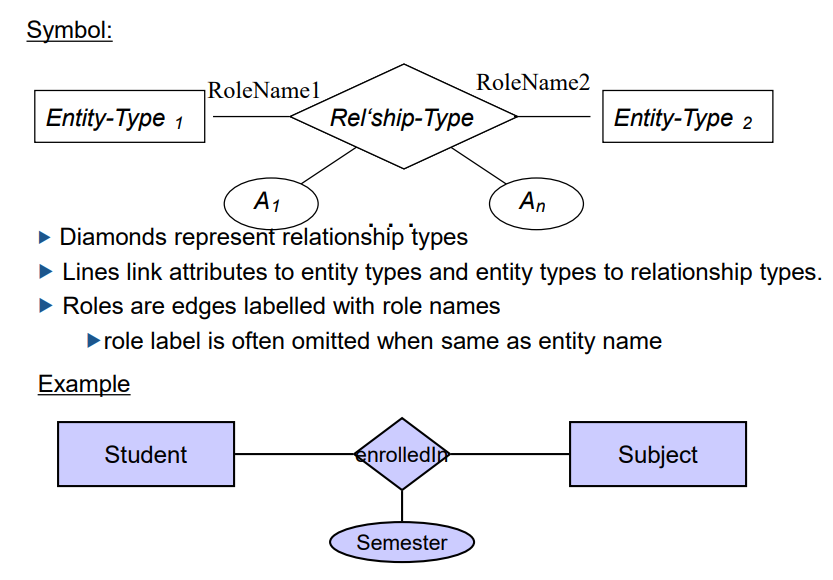
**Relationship-Attribute**

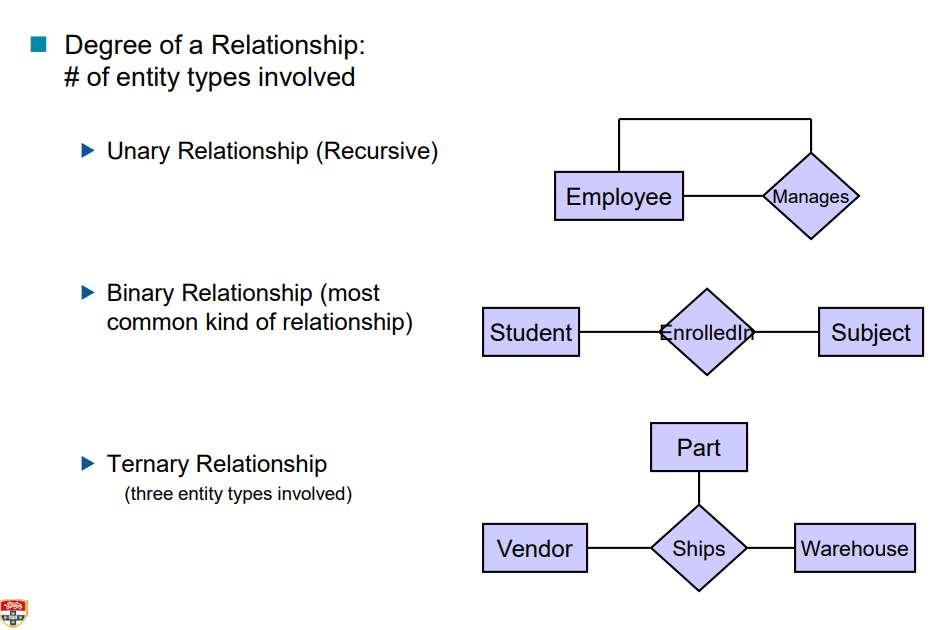
* Relationships can also have additional properties

**Relationship-Role**

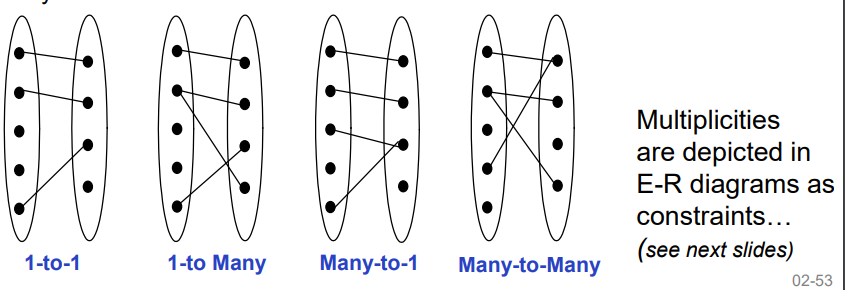
* Each participating entity can be named with an explicit role

**Diamonds** represent **Relationship Type**

****

****

**Relationship Degree is the number of entity types involved**

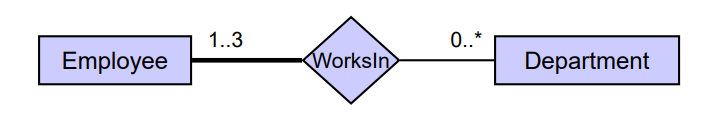
**Multiplicity**

**Model comes from domain knowledge**

* Conceptual data model should indicate whatever we know about the organizations business rules

**Different types of relationships**

* **Key Constraint**
  + **AT MOST ONE**
  + Shown as a thin arrow from key side to the relationship diamond
  + While mapping, you can put this as NULL which is as default
* **Participation Constraint**
  + **AT LEAST ONE**
    - **Thick line from entity type that must participate to relationship Diamond**
  + Make the two id’s be primary key, this would allow at least one relationship
* **Combination of Participation and Key Constraint**
  + **EXACTLY ONE**
    - **Thick arrow from key entity type that must participate to relationship diamond**
    - **NOT NULL**
* **Cardinality Constraints**
  + **Specification** of **Maximum and minimum** number of entity that is able to participate.

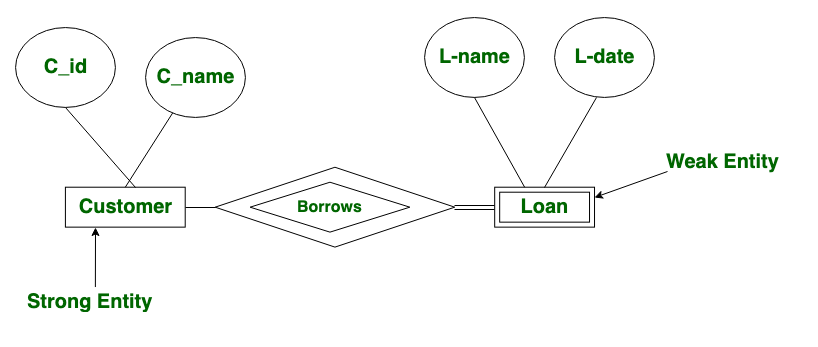
****

**Employee** works at **1-3 department**

**Department** can have **0 to many** employees

**Weak entity**

* These are an entity that cannot exist without existance of a dependent entity
* These entity have **TOTAL PARTICIPATION CONSTRAINT** in it’s idenfigying relationship ith owner identity





**Descriminator** of a weak entity type is the set of attributes that distinguishes among all the entities of a weak entity type related to the same owning entity

# Week 2

**Data Model**

* A collection of concepts for describing data
  + Structure of the data
  + Operations on the data
  + Constraints on the data

**Relation**

* Two dimentional tables of data
  + Row/ record
  + Column / attribute / field

**Requirement to becoming a relation (1NF)**

* Every relation must have a **unique name**
* Attributes in a table must have **unique name**
* All tuples in a relation have the **same structure**
* Every attribute value is **atomic**
* Every row is **unique**
* The order of the rows is **immaterial**

**Relation Schema**

* Specifies name of the relation as well as it’s attributes and data types

**Relation instance**

* Set of tuples for a schema, Table

If you think about relation as a table **object**, then relation schema and relation instance is completely Different

#rows = **cardinality**

# Columns = **degree (arity)\_of relation**

**Relational Database**

**Data Structure**

* Set of Relation instances

**Creating Relations in SQL**

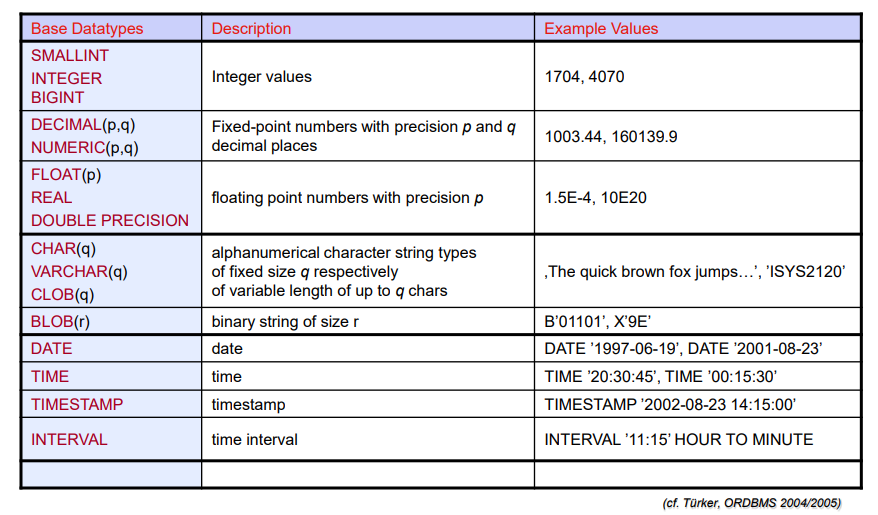
**CREATE TABLE** table\_name (

Attribute\_name data\_type **Integrity\_constraints**

)

**Deleting Tables**

**DROP TABLE** table\_name



Modifying Instances using SQL

**INSERT INTO** table [“(List of Columns) “] VALUES (“List of expressions”)

INSERT INTO Student (sid, name) VALUES (480222279, ‘Josh’)

**UPDATE** table\_name SET column = ‘expression’ ,{ column = ‘expression’ } [WHERE condition]

UPDATE Student SET address = “55 Duke Street’ WHERE sid = ‘490223789’

**DELETE FROM** table [**WHERE** condition]

DELETE FROM student WHERE name = ‘smith’

**Integrity Constraints (domain constraints)**

* Conditions that must be true for any instances of the database of that attribute
* Legal instance of a relation is one that satisfies all specified IC

**INTEGRITY CONSTRAINTS**

**NON-NULL**

* No value in a given column can be null

**PRIMARY KEY**

* Unique, minimal identifiers in a relation
* There may be several **candidate key** to choose from
* DOES NOT ALLOW NULL
* **PRIMARY KEY (sid,cid)**

**UNIQUE**

* A attribute(s) that can only have a unique values/pairs/combinations
* ALLOWS NULL
* **UNIQUE (sid,grade)**
  + A single student can have a certain grade **only once**

**Foreign keys**

* **Reference Integrity**
  + Must refer to a **existing candidate key of the parent relation**
* Identifiers that enable a **dependent relation** to refer to its **parent relation**’
  + **FOREIGN KEY** (sid) **REFERENCES** Student
* ALLOWS NULL WITH CERTAIN CONDITIONS

**CASCADE**

Update values according to any changes made in the parent tuple

**REFERENCES** Professor

**ON UPDATE CASCADE**

**ON DELETE SET DEFAULT**

**ER DIAGRAM**

**Relations (table)** correspond to entity types

**Rows** correspond to Entity’s **cardinality**

**Column** corresponds to **attributes**

**Entity Types**

* **Simple Attributes**
  + Directly mapped onto the relation
* **Composite attributes**
  + Flattened out by creating a separate field for each component attribute
* **Multi-valued attribute**
  + Becomes a separate relation with a foreign key taken from the superior entity
  + Becomes a weak entity

**Mapping of Relationship Types**

* **Many to Many**
  + Create a new relation with primary key of the two entity types as it’s primary key
* **One to many**
  + Primary key on the one side becomes a foreign key on the **many side**
* **One to one**
  + Primary key on the mandatory side becomes a foreign key on the optional side
* **Relationship attributes**
  + Becomes fields of either the dependent, respectively new relation

**Things to remember**

* **Every table** should have a **primary key** so that each **row** is **unique** and **no replications are made**
* Key constraint = NULLABLE
* Participation constraint = combined primary key
* Combination of both = NOT NULL

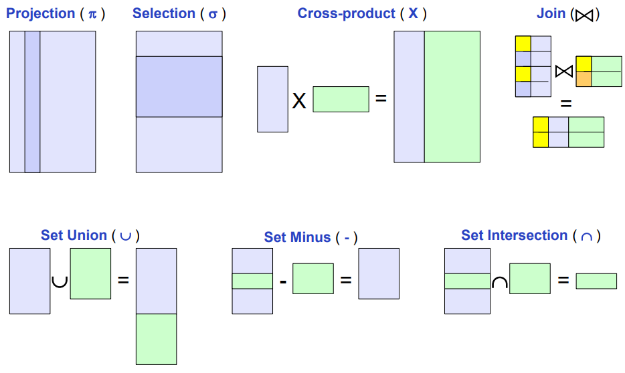
# WEEK 3

Relational Algebra

* Defines some basic oeprators that can be used to show general expression of a query

The aim of Relational Algebra is to allow simpler specification of Query without worry about implementation

Further extraction of “what” than “how”

Relational Algebra

**Union**  tuples in relation 1 or in relation 2

**Intersection** tuples in relation 1 as well as relation 2

- **Difference** tuples in relation 1 but not in relation 2

**Projection** Deletes unwanted columns from relations

* + - * SELECT

**Selection** Selects a subset of rows from relation

* + - * FROM or WHERE

X **Cross** P**roduct** allows us to fully combine two relations

**Join** To combine matching tuples from two relations

**rename** rename fields or even whole relation

* + - * AS



Get the names from student relation where country = Australia

Conditional join

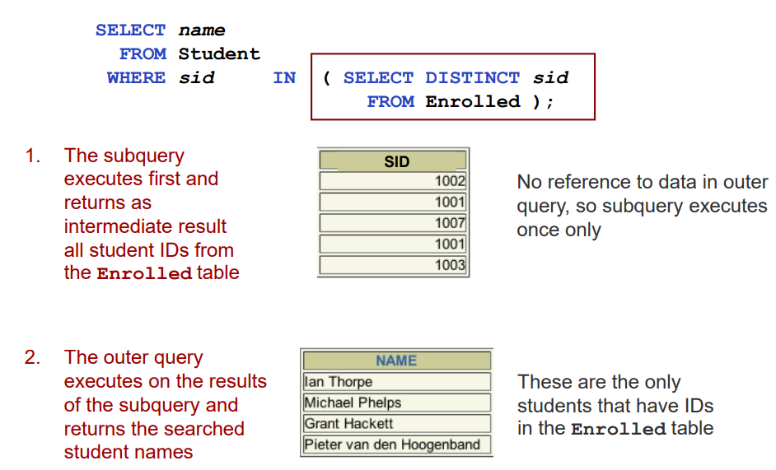
Conditions can be any comparative operations (=,>,<,>=,!=)

Conditions are called **EQUI-JOIN** if conditions are all equalities

Conditions by definitions are called **THETA-JOIN**

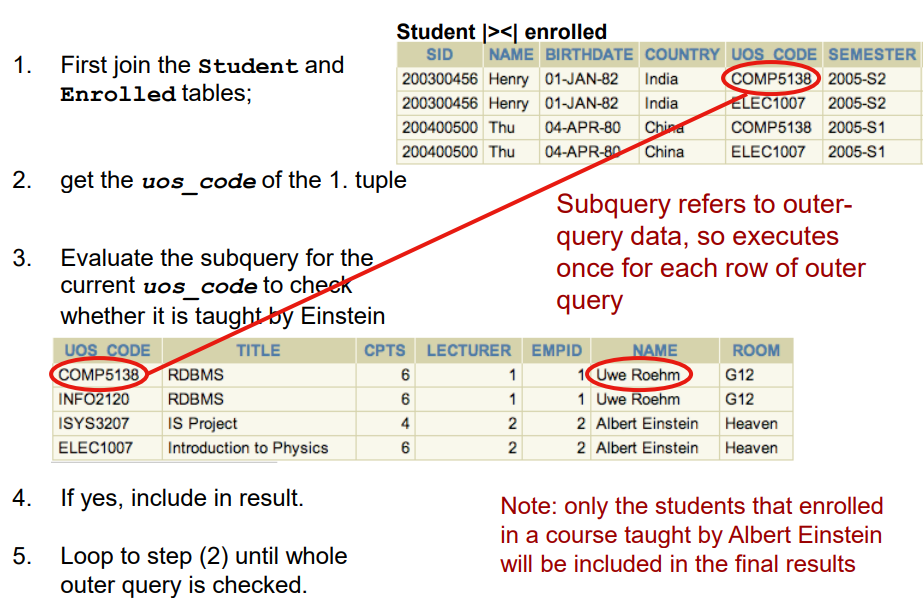


Get all the Entity from Enrolled who were enrolled in ISYS2120, show sid as Student in the display

**Subqueries**

**Correlated Subquery**

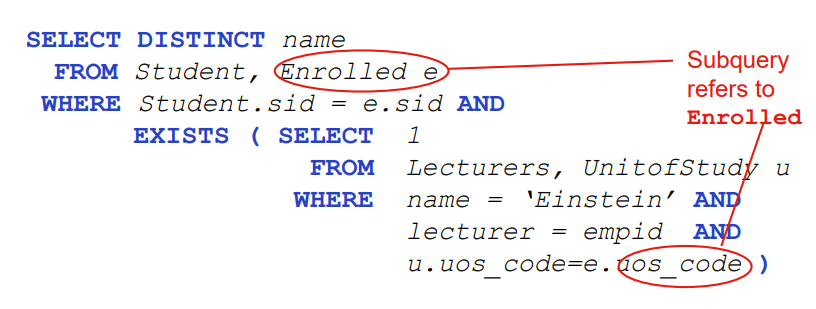
* Do not depend on data from the outer query
* Execute once for the entire outer query

**Correlated Subquery**

* Make use of data from the outer query
* Execute once for each row of the outer query
* Can use the EXISTS operator

**Find all the students who have enrolled in the lecture**

**Given by ‘Einstein’**

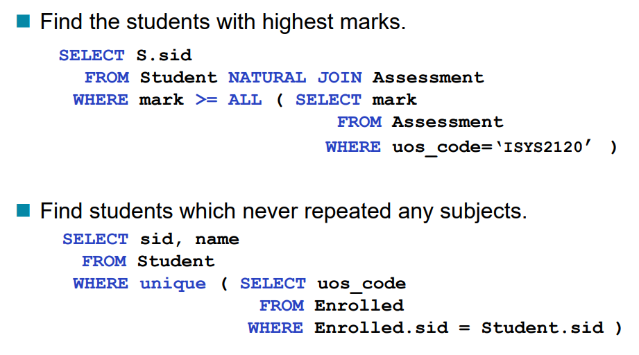
****

**IN**

Compares a value **v** with a set returned from a subquery, if **v** is one of the element in subquery, it returns true

**EXISTS**

Checks whether there is a returning value from a **correlated subquery**

**SET comparison operators**

**Not exists**

* Tests whether the returning set is empty

**Unique**

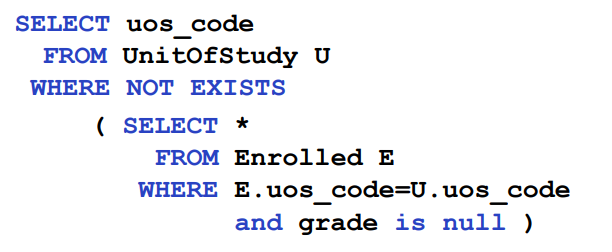
* Tests whether a subquery has any duplicate tuples in it’s result

**All**

* Tests whether a predicate is true for the whole set

**Some**

* Tests whether some comparison holds for at least one set element

****

**Week 4: Enhanced ER, Groupings and Divisions**

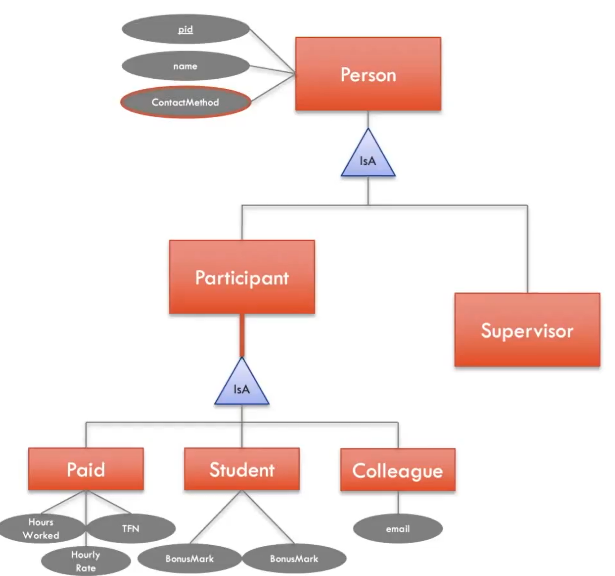
Enhanced ER model

Enhanced ER Diagram incorporates Object oriented Characteristics such as Abstraction and Super entities (Specialization / Generalization / Abstractions)

**Attribute Inheritance**

A lower level entity type inherits all the attributes and relationship participations of it’s supertype

If **F** ISA **E**

Set of attributes of F is a **superset** of the set of attributes of E

**F** is a subset of the entity set of **E**

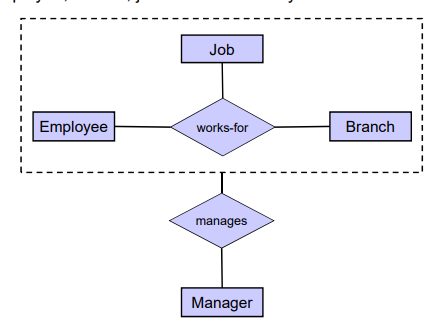
**F** is a **specialization** of **E** and **E** is a generalization of **F**

Constraints on ISA Hierarchy

**Overlap Constraints**

* Disjoint
  + An entity can belong to **only one lower level entity**
* Overlapping **DEFAULT**
  + An entity can belong to **more than one lower level entity**

**Covering Constraints**

* **Total**
  + An entity **must belong** to one of the **lower level entity sets**
* **Partial**
  + An entity **does not need to belong** to one of the **lower level entity sets**

**Aggregation**

* Relationship set to be treated as an **abstract entity set** for the purpose of participating in another relationship
* **Abstraction of relationship** into a new entity

**Mapping of ISA Hierarchy**

* Distinct Relations for the superclass and for each subclasses
* Superclass attributes go into superclass relations
* Subclass attributes go into each sub-relation
* Primary key of superclass relation also becomes primary key of subclass relations

**DIVISION**

Find the items in a set that are related to ALL tuples in another set

**Relational Algebra**

**Division operator (R/S)**

* + R, Dividend, numerator
    - Set that includes everything
  + S, Divisor, denominator
    - Subset that should ALL be included

**SELECT** name

**FROM** Student **JOIN** Enrolled **USING** (stdid)

**WHERE** uosCode **LIKE** ‘ISYS2%’

**GROUP BY** name

**HAVING** Count(unique(uosCode)) = (**SELECT** COUNT(\*)

**FROM** UnitOfStudy

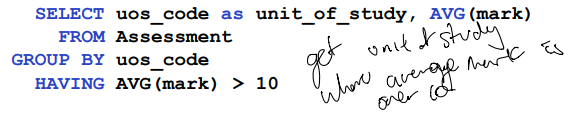
**WHERE** uosCode **LIKE** ‘ISYS2%’)’

The best method against this kind of idea is to find the number of element that are in the DIVISOR and then find element that has that amount of element in that context

**GROUP BY, HAVING**

Partition a relation into groups according to the values of one or more attributes

* Attributes in **SELECT** clause outside of aggregate functions must appear in the grouping list



**The order of Query Clause Evaluation**

FROM > WHERE > GROUP BY > HAVING > SELECT > ORDER BY

1. **Tables** are **collected and joined**
2. **Tuples** that fail the **WHERE** condition are **discarded**
3. Remaining tuples are **partitioned** into **groups** by the **value of attributes in the grouping-list**
4. **Groups** which fail the **HAVING** condition are **discarded**
5. **Answer table** is **generated**

**Week 5** Schema Normalization

**Schema Design process**

* Initially stemmed from a **conceptual design**
* Most important requirement is **ADEQUACY**
  + Ability to portray all the facts **completely**
* If Adequacy is satisfied then we seek to **avoid redundancy**

Aftereffect of **REDUNDANCY**

* **Insertion anomaly**
  + You are forced to create duplicate data or null value when inserting
* **Deletion Anomaly**
  + Deleting rows may delete data that could be needed in the future
* **Update Anomaly**
  + Update on one cell needed update on many other cells due to duplication

**FUNCTIONAL DEPENDENCY**

* Value of **one attribute** determining the value of **another attribute**
* **Tool to detect redundancies in schemas**
* X -> Y
  + X determines Y
  + Y is functionally dependent on X

**FD** is an **assertion** about the schema of a **relation** not about an **instance**

**Type of dependencies**

****

**Partial dependency**

* It is when the an attribute is determined only by a part of a candidate key and not the entire candidate key
* Mine -> state-code
  + It is a partial dependency because, in here (mine, commodity, company) is the candidate key but state is **only** determined by **mine**

**Transitive dependency**

* When a dependency has MIDDLE MAN
* MINE -> State-code -> state-name
  + MINE can go to State-Name

**Candidate key**

* **Minimal set of attributes** with ability to **uniquely identity a row/entity**

**Super key**

* **Any number of attributes** with ability to **uniquely identity a row/entity**

**Normalization**

* A process of improving upon schema design in order to reduce redundancy and efficiency
* **Putting up constraints to remove any redundancies in data and reduce anomalies**

1. Table with multivalued attributes

**REMOVAL OF MULTIVALUED ATTRIBUTES**

1. First normal form

**REMOVAL OF PARTIAL DEPENDENCIES**

1. Second normal form

**REMOVAL OF TRANSITIVE DEPENDENCIES**

1. Third Normal form

**Removing remaining anomalies resulting from functional dependencies**

1. Boyce-Codd normal Form

**REMOVAL OF MULTIVALUED ATTRIBUTES**

* Domain is atomic if its elements are considered to be **indivisible units**

**Table decomposition**

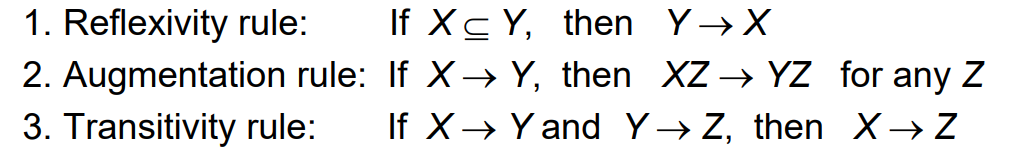
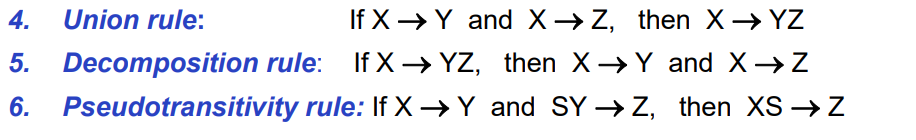
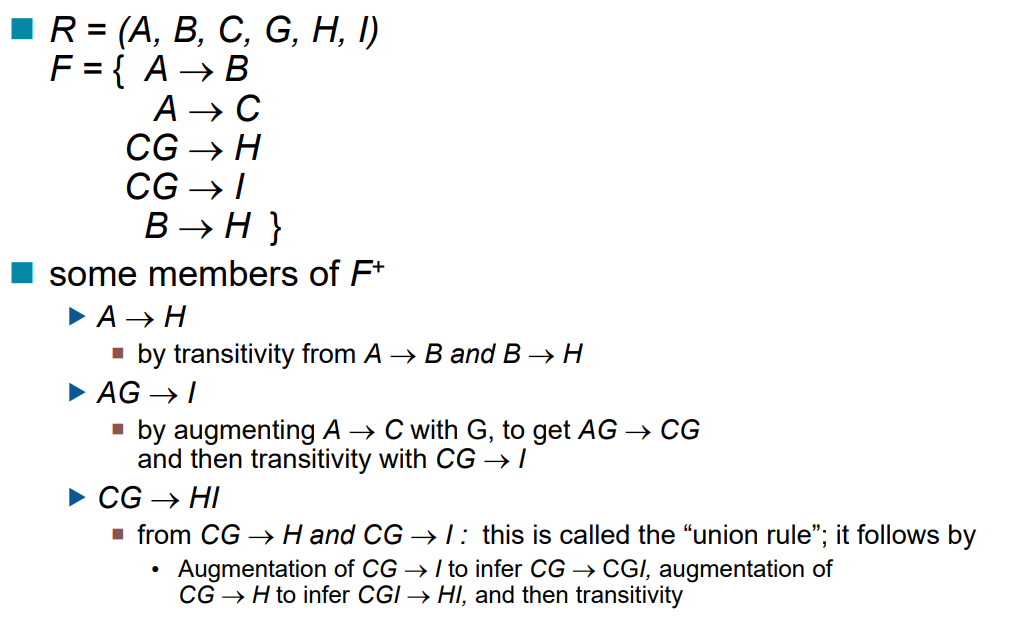
* An act of breaking down tables into many components such that
  + Each new relation scheme contains a subset of the attributes of the original table
  + Every attribute of original table appears as an attribute in one of the new tables
  + All the Dependencies are still present

**Each table decompositions should be**

* **Lossless join**
  + Any form of fact that could be found in the original table should also be reachable by the new decomposed table with joins
* **Dependency preserving**
  + All the functional dependency should hold true even after decompositions

**Method to find keys from Functional Dependencies**

1. Attribute closure/ The chase
   * Determining the **close of attributes (X+)**
     + This represents all the attributes accessible by X
     1. Find the first that **cannot be accessed** by any **Attribute**
     2. Then find **another attribute** that **cannot** be accessed by that **initial attribute**
     3. Keep **listing attributes you have access to** and keep finding elements that **you cannot access yet** until you have **all the attributes**
     4. That is the result
2. Closure of Functional Dependencies / Armstrong Axioms
   * **Reflexivity rule**
   * **Augmentation rule**
   * **Transitivity rule**

********

**Week 6: Views, Database Security and Integrity**

Database should ensure

* **Secrecy**
  + Users should not be able to see things they are not suppose to
* **Integrity**
  + Users should not be able to modify things they are not suppose to
* **Availability**
  + Users should be able to see and modify things they are allowed to

**Access Control**

* **Mandatory Access Control / Authentication**
  + Every connection must log in with log in and password
  + CREATE USERor CREATE LOG IN
* **Discretionary access control / Authorization**
  + Access rights or privileges for objects and mechanisms for giving user privileges
  + Creator of a table or a view automatically gets all privileges on it
    1. DMBS keeps track of who subsequently gains and loses privileges, and ensures that only requests from users who have the necessary privileges (at the time the request is issued) are allowed.

**GRANT** privilege list {SELECT, INSERT, DELETE, UPDATE, REFERENCES}

**ON** table

**TO** user list

**[WITH GRANT OPTION]**

You can also specify attributes on the privileges as well

**GRANT** SELECT(grade)

**REVOKE** privilege list

**ON** table

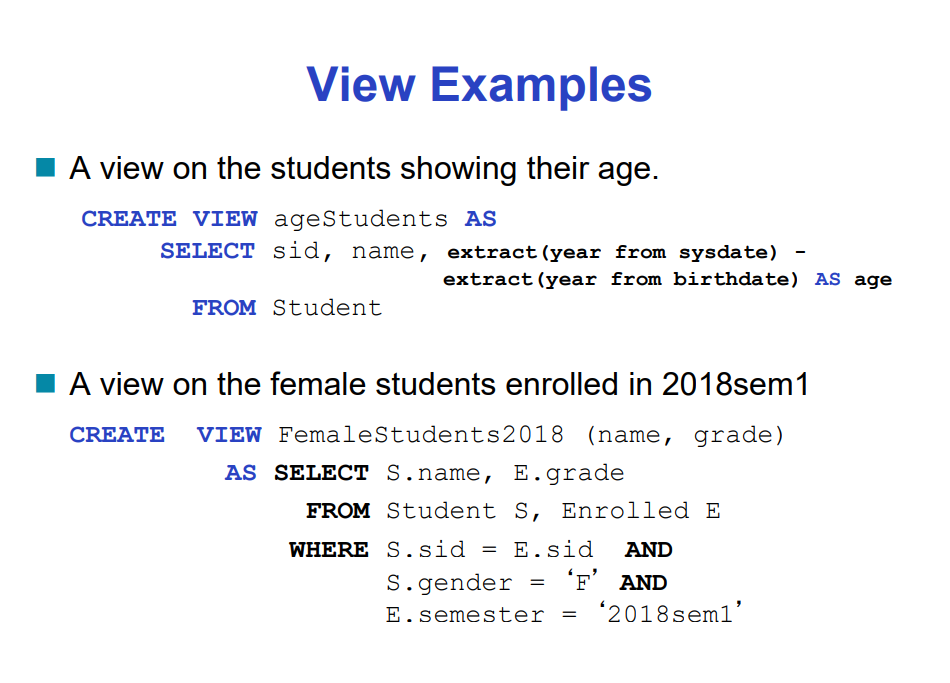
**FROM** user list

When a privilege is revoked from X, it is also revoked from all users who got it solely from X.

If User has a privilege with the **GRANT OPTION**, they can pass privilege onto other users (with or without passing on the GRANT OPTION)

**GRANT** INSERT **ON** STUDENT **TO** UWE **WITH GRANT OPTION**

This means that UWE can Insert into Student as well as to grant other users Insert privileges

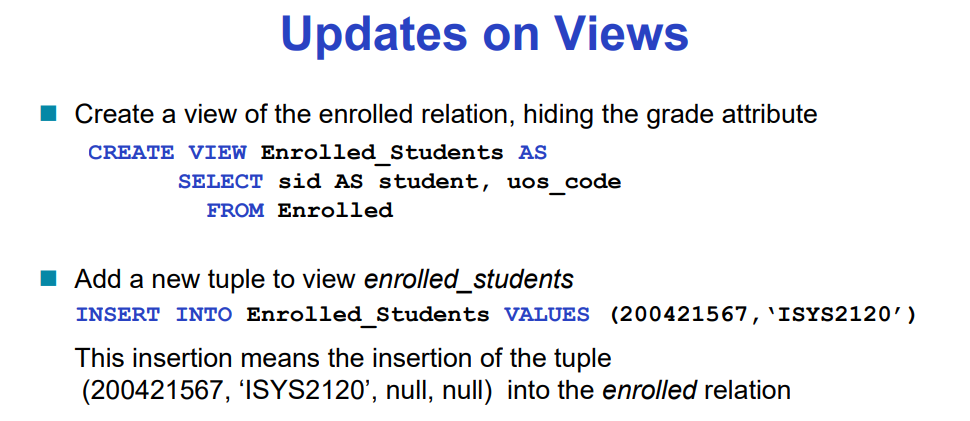
**VIEWS**

Virtual relation, but it stores **definition** of the SQL code rather than a **set of tuples**

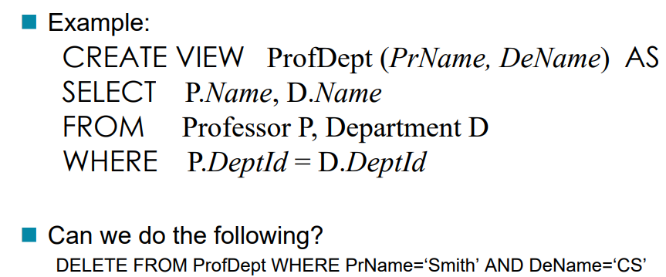
* This is useful because it means that only users with **authorization** can access and see **VIEW CONTENTS**

**CREATE VIWEW** name **AS** <query expression>

**Updating or inserting** into View will either add **NULL** or **DEFAUILT** values to attributes that were not specified in the **VIEW DEFINITION**



**PROBLEM WITH VIEWS**

1. Updates will cause null or default value insertion
2. New tuples will not be visible to view **unless they have insertion access**
3. Because View may be ambiguous in relation definition
   * Update to a view might not uniquely specify the change to the base table that results in the desired modifications of view

**VIEWS AND REFERENECS**

Because of the **Reference integrity constraint** that **foreign key** must always refer to an entity in it’s **super table,** this can be exploited by:

* If **insert is successful**, this means that an **entity** with that **specific id exist**
* If **insert** **fails**, this means that an entity with that **specific id does not exist**

Now in a view with foreign key, if you specify **GRANT** REFERENCESto the grant option, then they can insert, update element that holds foreign key

**Role based authorization**

Roles can be created and granted to a user

**CREATE ROLE** manager

* Role of manager is created

**GRANT** SELECT, INSERT **ON** Student **TO** manager

* Manager’s access authorization definition

**GRANT** manager **TO** shari

* Granting manager role to Shari

Limitation of SQL Authorization

* Does not support authorization at a tuple level

This is managed by the **application program layer** in the **front end**

* This is **advantageous** because of the **fine-grained** **authorization** potentials
* However, **authorizations** must be done in **application** code and may be **dispersed all over an application**

**Data minimalism**

* Best **protection** against **unauthorized** **access** to **data** in your **database** is to consider very carefully **what you store in the first place**

**Semantic Integrity Constraints**

Aim

* Ensure that authorized changes to the database do not result in **data inconsistency**
* Guard against **accidental damage to the database**

Advantages of centralized, automatic mechanism to ensure semantic integrity constraints

* More effective integrity control
* Easier application development
* Better maintainability
* Store data is more faithful to real world meaning

**Integrity Constraints**

* Conditions that must be true for every instance of a database
* Specified in the **database schema**
* Checked when **database is modified**
* If it is violated
  + Undoing of database operation
  + Abort of transaction
  + Execution of **maintenance** operations to make **database stable again**

**Static integrity constrant**

* Describe conditions that every legal instance of database must satisfy
  + Domain constraint
  + Key constraint
  + Referential integrity
  + Semantic integrity constraint
  + Assertion

**Dynamic Integrity constraint**

* Predicated on database state changes
  + Triggers

**Domain Constraints**

* Field must be of right data domain / data type
  + This is also the moment to reflect upon
    1. DEFAULT default-value
    2. NOT NULL - may not be null
    3. NULL (default) – may be null

**User Defined Domains**

* **CREATE DOMAIN** domain name, SQL data type
  + **CREATE DOMAIN** Dollar **NUMERIC**(12,2)
  + **CREATE DOMAIN** Grade **CHAR CHECK**(value in (‘F’,’P’,’C’,’D’,’HD’)

**Primary Key Constraint**

* UNIQUE AND NOT NULL by default

**Foreign Keys & referential integrity**

* Must refer to a candidate key of the parent relation
  + This is where constraint control comes into play
    1. **NO ACTION**
    2. **CASCADE**
    3. **SET NULL**
    4. **SET DEFAULT**
  + These can be set along side with modifications at the parent table to specify reactive actions to take if parent table changes
    1. FOREGIN KEY (sid) REFERENCES Student

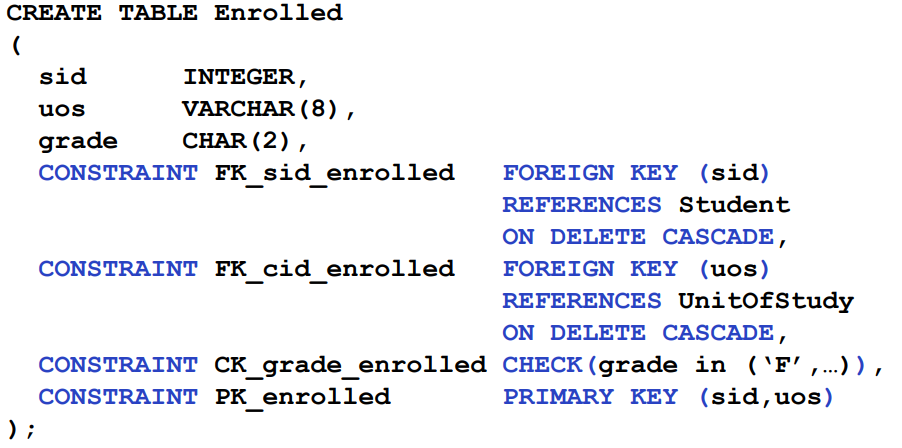
ON DELETE **CASCADE**

ON UPDATE NO **ACTION**

**MULTI-ATTRIBUTE Constraint**

**CONSTRAINT** name **CHECK** (semantic condition)

**CONSTRAINT** maxMark **CHECK** (mark between 0 and 100);

**Postgresql** does not support check’s **semantic** **condition** to be a **subquery**

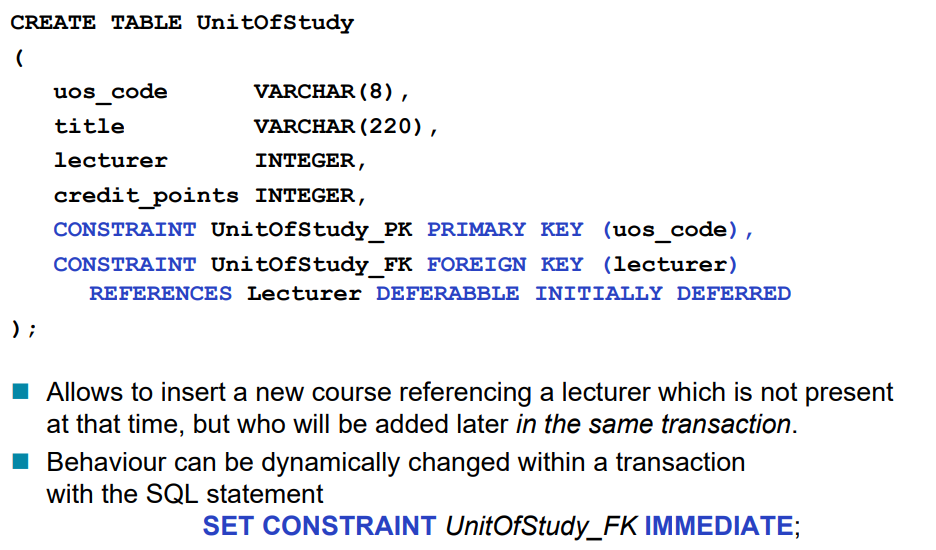
**Naming Integrity Constraints**

**CONSTRAINT**  name **CONSTRAINT**

**DEFERRING CONSTRAINT CHECKING**

Deferring refers to the ability to delay constraint checking until after transaction is complete

* **NOT DEFERRABLE**
  + Every modification will check the constraints **immediately**
* **DEFERRABLE**
  + **INITIALLY DEFERRED**
    1. Wait untill transaction end, but allow dynamic change later
  + **INITIALLY IMMEDIATE**
    1. Check immediate, but allow dynamic change later

****

**ALTER TABLE STATEMENT**

Integrity constraints can be added, modified and removed from an existing schema using **ALTER TABLE** statement

**ALTER TABLE** table-name **constraint-modification**

**Constraint-modification**

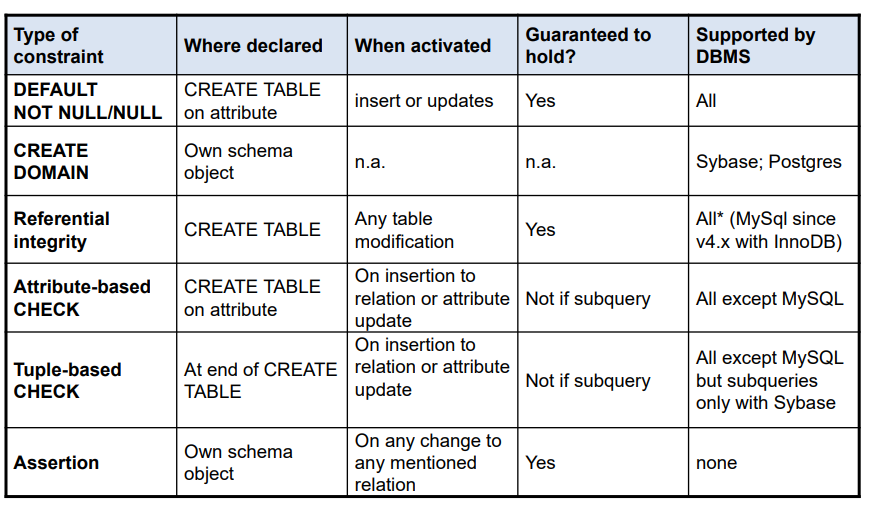
* **ADD CONSTRAINT** constraint-name new-constraint
* **DROP CONSTRAINT** constraint-name
* **RENAME CONSTRAINT** old name **TO** new-name
* **ALTER COLUMN** attribute-name **SET** domain-constraint

**ALTER TABLE** Enrolled **ALTER COLUMN** grade **SET NOT NULL**

**ASSERT**

* Predicate expressing a condition that we wish the database to always satisfy

**CREATE ASSERTION** assert-name **CHECK** conditions

****

**CREATE** **TABLE** Sailors (

sid INTEGER ,

sname CHAR (10) ,

rating INTEGER PRIMARY KEY (sid) ,

**CHECK** (rating >=1 AND rating <=10) ,

**CHECK** (

(**SELECT** count(s.sid) **FROM** Sailors s)

+ (**SELECT** count(b.bid) **FROM** Boats b) **< 100**

);

**CREATE** **ASSERTION** smallclub **CHECK** (

(**SELECT** **COUNT**(s.sid) **FROM** Sailors s)

+ (**SELECT** **COUNT**(b.bid) **FROM** Boats b) < 100)

)

Dynamic Integrity Constraint

**TRIGGERS**

* Statement that is executed automatically if specified modifications occur to the DBMS

**ON** event **IF** precondition **THEN** action

Event - event that activates the trigger

Precondition - guard/test whether the trigger shall be executed

Action - what happens if the trigger is run

Aim

* Constraint maintenance
  + Being able to maintain foreign-key and semantic constraint
  + Commonly seen with **ON DELETE** and **ON UPDATE**
* Business rules
  + Assertion can be implemented using **two triggers**
* Monitoring
* Maintenance of auxiliary cached data
* Simplified application design

Trigger creation

**CREATE TRIGGER** trigger name

**{BEFORE,AFTER} {INSERT, DELETE, UPDATE} ON** relation-name

**FOR EACH ROW/STATEMENT** -- optional, only for row/statement trigger

**WHEN** (condition) -- optional

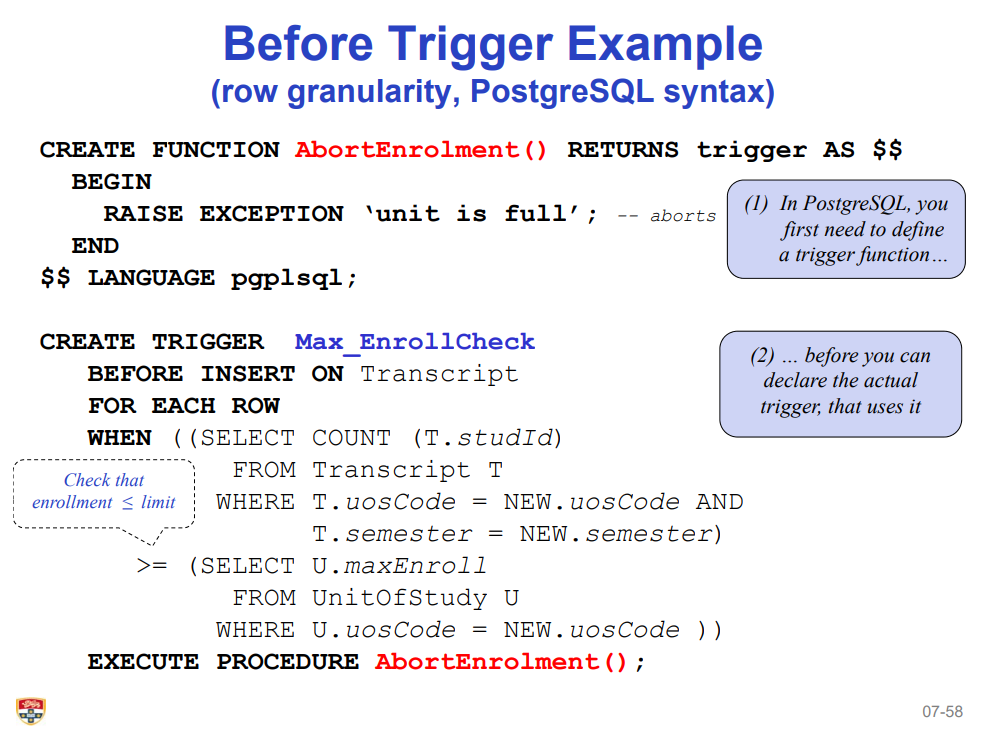
**EXECUTE PRODECURE** store-prodecure-name(); -- needs to be defined first

**CREATE TRIGGER** overdraft-trigger **AFTER UPDATE OF** balance **ON** account

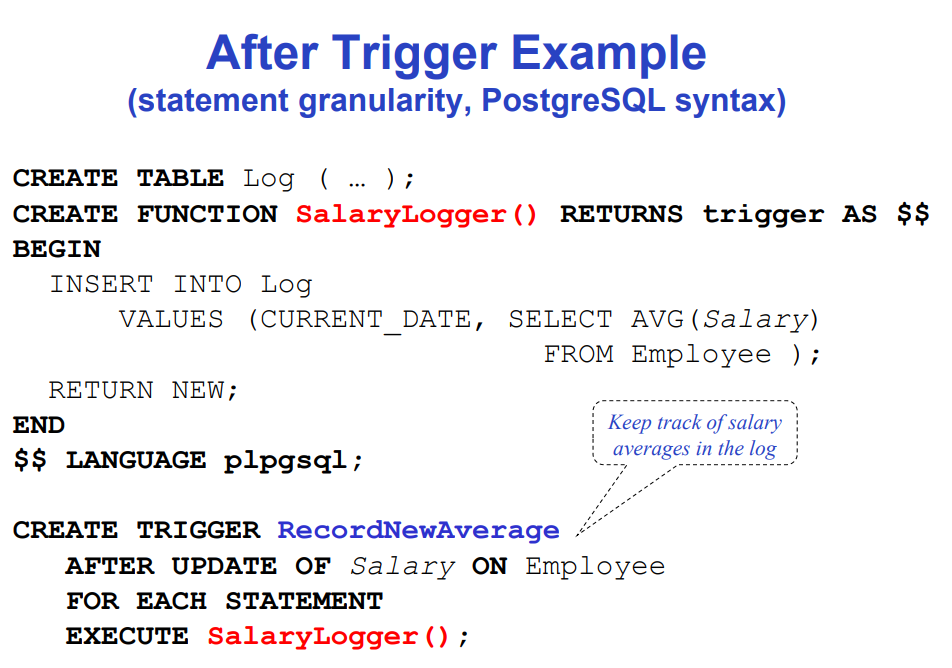
**GRANULARITY**

* Row level granularity
  + Change of a single row is an event
    1. Single **UPDATE** statement might result in multiple events
* Statement level granularity
  + Events are statements
    1. Single **UPDATE** statement that changes multiple row is considered **single event**

Before Trigger Example



After Trigger Example



Use **BEFORE** trigger for **checking integrity constraints**

User **AFTER** trigger for **Integrity maintenance and update propagation**

Reasons against **triggers**

* Initially triggers were used to
  + Maintain summary data (total salary of each department)
  + Replicating databases by recording changes to special relations (change relations or delta relations).
* There are better ways of doing these now
  + Built in materialized view facilities to maintain summary data
  + Databases provide built-in support for replications
* The need for triggers are replaced by build in supports

Week 7: Data security in Web SQL Injection Attacks

Usually, SQL Codes are done in a String Based statements

Sql =

‘’’

SELECT \*

FROM users

WHERE username = ‘$\_request[‘user.name]’

’’’

SQL injection is a input of SQL statements as a user entry field in order to manipulate the backend database of the application.

Example = “ OR 1 OR username = ‘’ ”

This will result in the Database to always follow up as true as the first OR statement means always true.

**Databases** requires security access restriction because it can reveal

* Internal logics
* Internal relations
* Database vulnerable
* Potential SQL Injection attacks

**PROTECTION AGAINST SQL INJECTION**

**Anonymous Parameters**

“

SELECT name

FROM Student

WHERE sid = %s

”,(stdid,)

**Named parameters**

“

SELECT name

FROM student

WHERE sid = %{sid}s

“,{‘sid’:stdid}

**ERROR MESSAGES SHOULD NEVER BE VISIBLE TO USERS**

* Error messages may show
  + The logics behind the back end databases.
  + Handling methodologies of databases
  + Relations names

**Access Controls**

* Application logins should be restricted to the bare minimum of access privileges needed
  + **CREATE LOG IN**
  + **GRANT/REVOKE**
  + **VIEWS**

**Stored procedures/ DB-API**

Another method to restrict access to SQL is to implement **stored procedures.**

Stored procedure is the SQL functions that store all the SQL procedures in the Database Schema and users have access to **calling** that function but nothing else

* This means the front end does not know the logic behind the features and functionality.

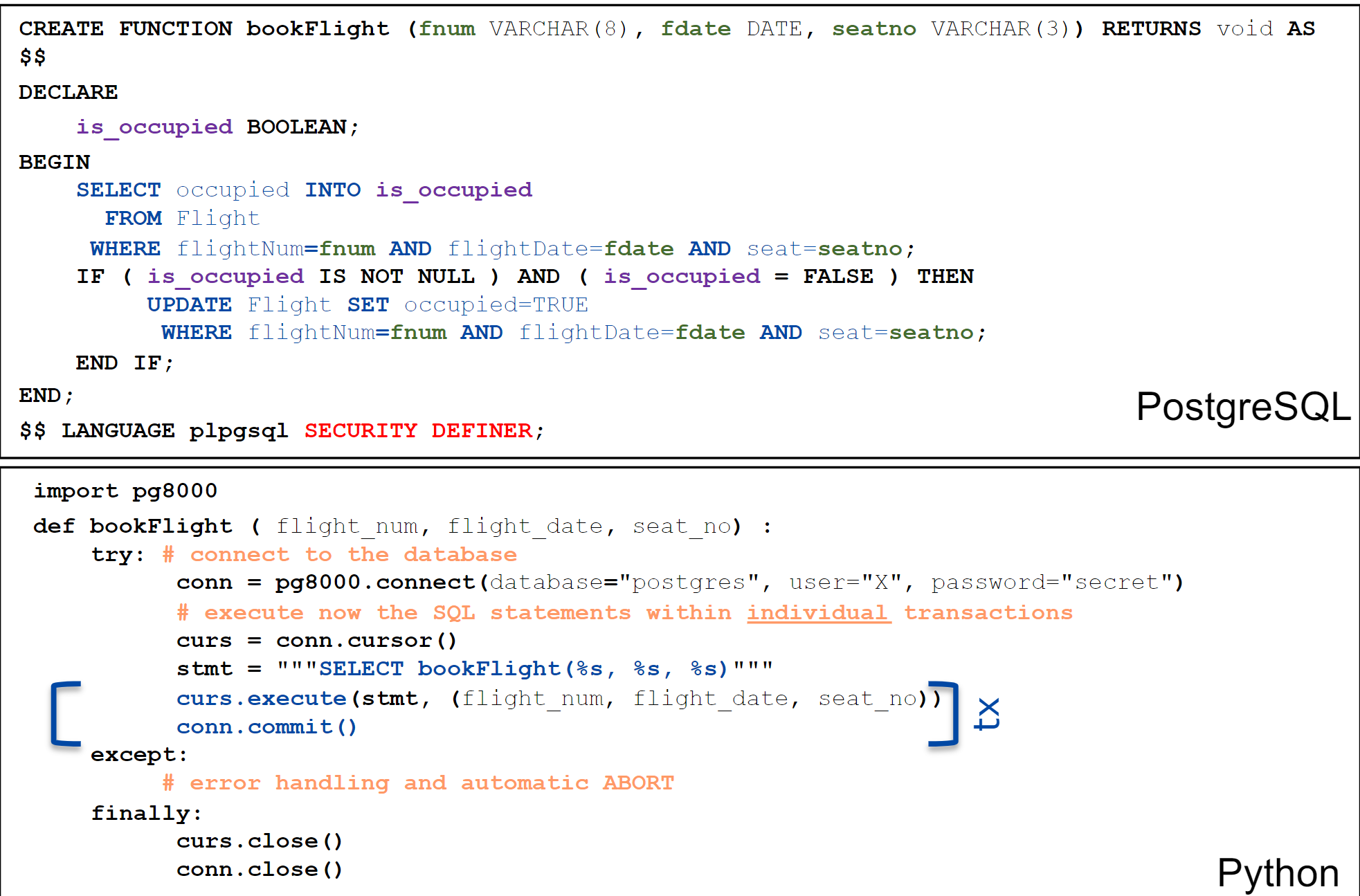
**PRO**

* **Security**
  + Access privileges on stored procedures and not any of the databases that the logic requires
* **Efficiency**
  + Instead of sending multiple SQL statements, you can call the procedure once and it’s parameters.
    1. Reduces the number of sensitive informations passed
* **Correctness**
  + Avoid pitfall of wrong transaction boundaries

**CONS**

* **Good design needed**
* **Difficult to debug**
* **Separate language to learn**

**CREATE FUNCTION**



Database Servers and Internets should be kept separately

* Keep database servers private protected by firewall, VPN, VPC
* Restriction of database access

**Password Security**

* **Hash functions**
  + Hashing is a method in which you convert given textfield value of the password into a function to convert it into non-human readable format that either gets **verified/validated** to check for user authorization or to **store passwords**
  + Should be done in **application** layer and not in **DBMS** layer
    1. Faster access
    2. Less sensitive data transmitted through internet
* **Salting**
  + Salting is a method of inserting a random text into the password to the inputted textfield value into another value that would result in a more comples process for the hash function.
* Both processes already have **libraries** that is available to the **public**. This is to **avoid** any **incompetent** **data** **encryptions** done by **users** who are **not experts in the field.**

**Data minimalism**

* The theory that states

The best way to protect against unauthorized access to data is to limit the data stored in the database itself.

**Information modelling, ontologies and linked data**

**Data modelling** split into three different types

* **Conceptual modelling**
  + Which focuses on abstract representation of the key entity types in the business domain of interest, their attributes and their interrelationships
  + Mainly based on Business requirements
  + ER Diagrams
* **Logical modelling**
  + More detailed descriptions of each attributes, relations, key values and relationships
  + This is where Functional Dependencies will be visible
* **Physical data model**
  + Detail of physical storage of the data including indexings etc.

**Data structures**

* **Structured data**
  + Data that is structured in **an interative format** with **rows and columns** and can be mapped to predefined **schema**
    1. Relational tables
    2. Excel spreadsheet
* **Semi-structured data**
  + Mix of both types of data that can have both structured and unstructured section in the data
    1. Email
       - Sender, receiver, cc, are formatted but the body of the email is not
* **Unstructured data**
  + Data that cannot be easily organized into rows and columns
  + Difficult to search, manage and analyse
    1. Text document
    2. Videos
    3. Audio files

Method to analysis

* **Structured data analysis**
  + Schema First > reading data > interpret the data > implement data facts
* **Unstructured data analysis**
  + First get the data > try to understand the format of the content/ read data definition> interpret the data > implement data fact

**ONTOLOGY**

* Method to establish a common vocabulary for unstructured data
* Explicit and formal specification of information in a domain
  + Concept
  + Properties and attributes of concepts
  + Constraints on properties and attributes
  + Individual instances
* **Formal, explicit specification of a shared conceptualization**
  + **Explicit**
    1. Something in concrete form
  + **Formal**
    1. Machine readable and processable
  + **Conceptualization**
    1. Defines an **abstract mod**el describing a **field** of **knowledge** or **domain**
* Main purpose of ontology is to **capture** the knowledge of a **particular domain**

**Responsibility** of an **ontology**

* Defining the **concept**/**classes** in the **domain** **of** **interest**
* Arranging the **classes** into **class-subclass format**
* Specifying the **relationships between concepts**
* Creation of **Knowledge** **graph**

**Knowledge Graph**

* **Formal description** of a **certain knowledg**e that can be **accessed and explored by machine**
* GOOGLE SEARCH ENGINE IS A KNOWLWEDGE GRAPH

**Semantic** **Web**

* Allow **web-based** knowledge data available for analysis and linking

Week 8: Database Application Development

Three types of functionality

* **Presentation logic (GUI)**
* **Processing Logic (Procedures, functions and programs)**
* **Data Management (DBMS activities)**

**Interactive SQL**

* SQL statements input from terminal
  + DBMS outputs to screen

**Non-Interactive SQL**

* SQL statements are included in an application program written in a host language

**Client side DB Application Development**

* Statement level interface
  + Embed SQL in the host language
  + Application program is a mixture of host language statements and SQL statements and directives
* Call level interface
  + Create special API to call SQL commands
  + SQL statements are passed as arguments to host language procedures

**Python is a call level interface**

* SQL statements are passed as parameters which are then inserted into a call function in the library to handle SQL statement proceduers.

**Difficulties** with python SQL Interface

* **Establishing Database Connection**
* **Static vs Dynamic SQL**
* **Parameterized SQL** and mapping of domain types to data types of host
* **Impedance Mismatch, Buffer Mismatch**
  + Cursor concept (pointer)
* **Error handling**

**Connecting to database**

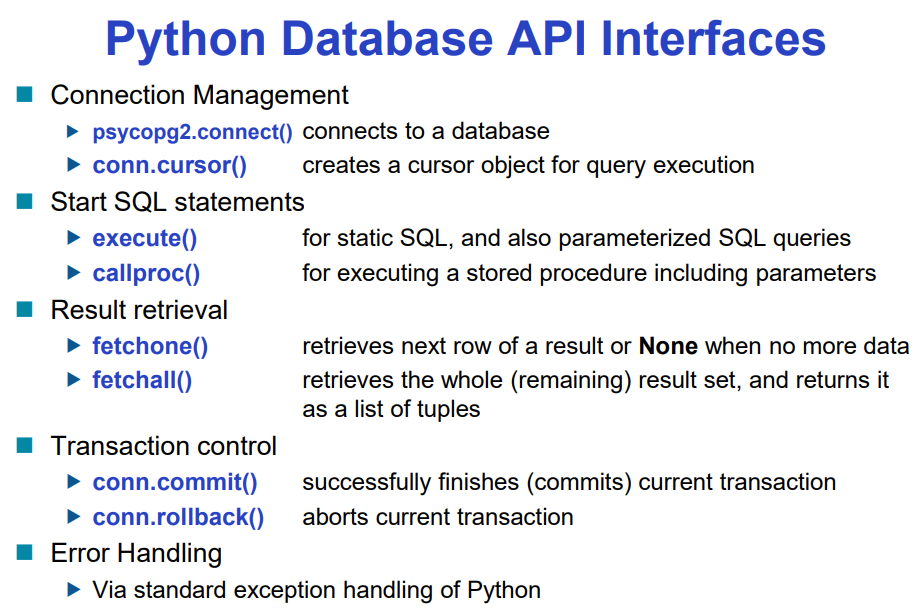
* **Sessions** start when **connection is created**
* Conn = psycopg2.connect(host = ‘’, database = ‘’, user = ‘’, password = ‘’)
* conn = psycopg2.connect(

"host=postgres.usyd.edu.au dbname=unidb user=U password=secret" )

Conenction parameter is given as one string.

**Python Database API Interface**

|  |  |
| --- | --- |
| **Connection Management** | |
| **Psycopg2.connect()** | Connects to a database |
| **conn.cursor()** | Creates a cursor object for query execution |
| **Start SQL Statements** | |
| **Execute()** | For static SQL, and also for parameterized SQL query |
| **Callproc()** | For executing a stored procedure including parameters |
| **Result Retrieval** | |
| **Fetchone()** | Retrieves next row of a result or **none** when no more data |
| **Fetchall()** | Retrieves the whole (remaining) result set, and returns it as a list of tuples |
| **Transaction control** | |
| **Conn.commit()** | Successfully finish current transaction |
| **Conn.rollback()** | Abort current transaction |

****

**Static and Dynamic SQL**

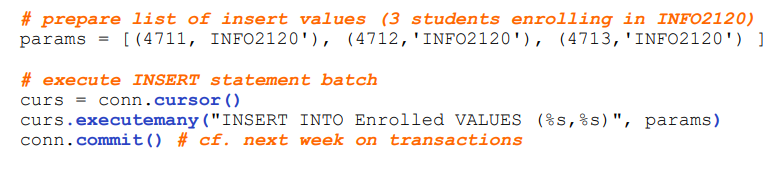
**Static SQL**

* Useful when SQL statement is known at all time

**Dynamic SQL**

* Application constructs SQL statements at run time as values of host language variables that are manipulated by directives
* As **python** is a interpreted language it constructs **SQL statements at run time** which makes it prone to SQL injection

**Batch insert operations**



**NEVER**  use **String Concatenation** as a SQL statement as this will lead to SQL Injection

* Possibilities **with connection** 
  + **Injecting SQL** via **unchecked user input**
  + Exploiting **buffer overflow**
  + Navigate **output** on **hacker’s screen**
* Possibility **without connection** 
  + **SQL injection** in **built-in** or **user-defined procedures.**
  + **Buffer overflow** in **built in** or **user-defined procedures**

**Parameterized SQL statements**

* When you pass SQL statement and parameters and SQL values as a parameter to conn.execute()
* **Each time** a SQL is required, you need to create new **cursor** and re**-**issue **SQL statement each time** when **parameter is changed**

**Buffer Mismatch problem (impedance mismatch)**

* This is a problem that happens when SQL deals with **tables with arbitrary** size while **host** **language** program deals with **fixed sized buffers**
* This is fixed by concept of **cursor** or **pointers**
* **Cursor points** to the **result set**
  + **Result set** is the set of rows produced by the **SELECT** statement

**Cursor** is then passed **SQL statement** to run and returns a **dictionary tuples** as a result

Null values are supported by **NONE** value.

**Error handling**

* Errors such as:
  + Failure to connect
  + Wrong log in
  + Missing privileges
  + SQL syntax error
  + Empty result
  + Null value
* ALWAYS check the RETURN VALUE
* Always use TRY CATCH statement and cover every error types.
* **NEVER SHOW ERRORS TO END USERS**

**ERROR HANDLING WITH PSYCOPG2**

Psycopg2 has built in error handling methods

Try:

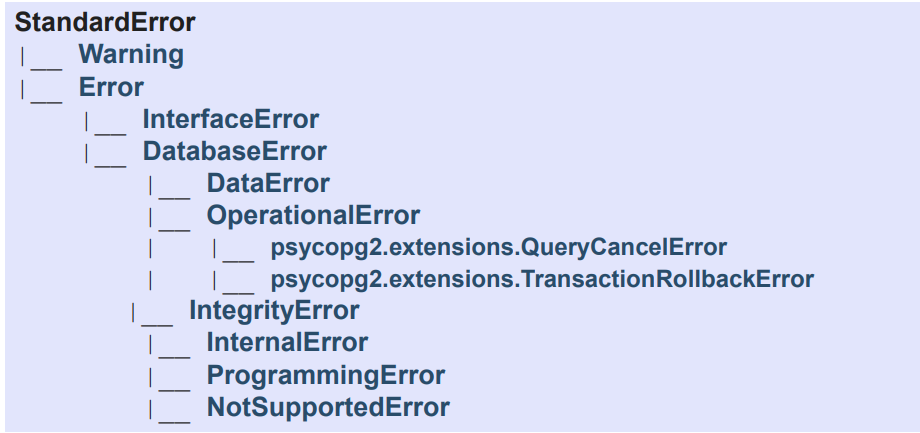
Psycopg2.connect()

Except psycopg2.Error as e:

Print(“error was caught”)

Print(e.pgerror)

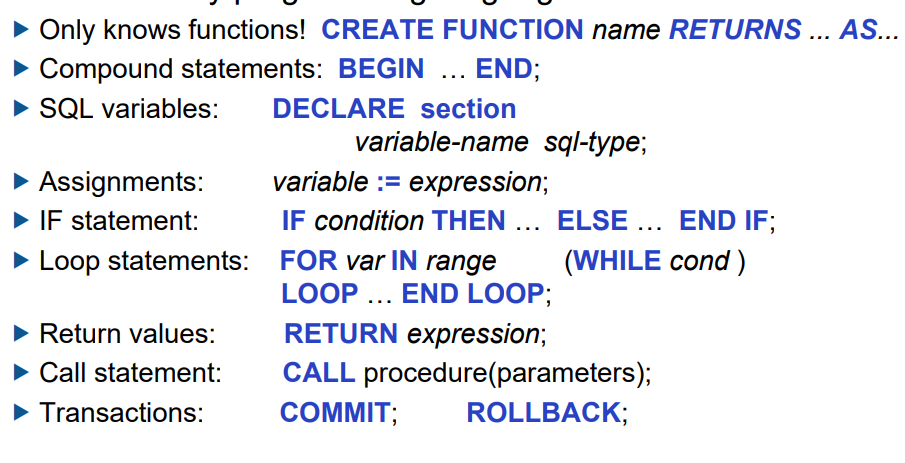
Print(e.pgcode)



Server side DB application development

* Stored procedures
  + Creating a function in the database server schema element
* Advantages
  + Central code-base for all applications
  + Improved maintainability
  + Additional abstraction layer
    1. Programmers does not know the logic behind processes
  + Reduced data transfer
  + Lessen Long-held locks
  + DBMS centric security
  + Has access to all the tables
* **CALL** ShowNumberOfEnrolments()

FUNCTION OPERATORS



**CREATE OR REPLACE FUNCTION** tryFunctionCalling(parm) **RETURNS** type AS $$

DECLARE

BEGINE

SELECT …

FROM….

END

$$ **LANGUAGE** plpgsql

Cur.callproc(“tryFunctionCalling**”, [parm])**

**Week 9: Transactions**

ACID Properties of transaction

* ATOMIC
* CONSISTENT
* ISOLATED
* DURABLE

**Transaction** is a **collection** **of one or more operations** on **one** **or** **more** **databases** which reflects a **single real-world transaction**

Due to the lack of operations that a SINGLE SQL can do, we need to program TRANSACTIONS

**COMMIT**

* Is when you want to store changes made by the transaction
* Commit is a **REQUEST** therefore, the system might not do it.

**ROLLBACK**

* Is when you want to **cancel/abort** **transaction** **process** and **revert** to the **most recent stable state** **before** **transaction** beginning
* Motivations
  + User’s decisions
  + Explicit program calls when error handling
  + Integrity constraints
  + System crash

**AUTOCOMMIT**

* Application calling commit after EACH SQL CODES

By default PYTHON DB-API is **explicit mode** meaning it **requires commit()** to complete transaction

The state of table is restricted by **integrity constraints** that are **explicitly declared in the schema**

**Dynamic Integrity Constraints**

* Transactions might **transform** thedatabase from one consistent state to another, but **transition** might not be permissible
  + Transformations from one state to another
  + Transition is going back and forth between states

**Transaction is consistent if**

* All **static integrity** is **satisfied**
* **New state** satisfies specification of **transaction**
* No **dynamic constraints** have been violated

Transaction can be checked

* Automatically
  + Using CHECK, ASSERTION, TRIGGER
  + Not always desirable since unnecessary checking might result in slower processing time
* Manual
  + Perform checks in application code only when it is needed
  + This becomes difficult to maintain as transactions are **modified/added**

Integrity Constraints in Transactions

* DIFERRABLE
  + Integrity constraints are checked after transaction is complete

System obligations

* To make sure that **each application** is mostly **independent** and **no mallicious interferences** are present **between processes**

**Data integrity** may be **threatened** if **transaction is not complete**

When **System Crash** happens, the transaction processes may be undergoing. Therefore,restart process must clean this up and call **rollback** to revert changes Transaction made.

Each item holds value for the last committed transaction

**CONCURRENCY**

* To allow transactions to be **MULTI-THREAD SAFE**, **serializability** is necessary
  + **SERIALIZABILITY** is the process of finding elements that do not require **changing the internal state of the database** and only **locking** on those that **do update**

**Transaction** is considered **perfect if**

* Each transaction is atomic, isolated, curable and follow all the integrity constraints

**LOGGING**

* **Log** is an append only collection of entries showing all the changes to data that happened in order as they happened.
* These are used to **abort and rollback** Transactions as well as to be looked **after crash,** to figure out the **last stable state to revert to.**

**DISTRIBUTED COMMIT**

* When transaction changes are done onto more than one tables, they require two batches of information to be relayed
  + Checking whether all databases have log entries safe on disk
  + Inform each database of outcome and tell them to commit

**LOGGING**

* Requires Extra storage
* Possible delay in transaction commit as it needs to store information

**LOCKS**

* Blocking transactions if another transaction is already in place for that specific database
* Risk of DEADLOCK

**DISTRIBUTED COMMIT**

* Distributed transactions have very long delays

**ISOLATION**

What information should Transactions read?

* **Set Transaction to read COMMITTED**
  + The data being read is always clean and stable
  + It can be inconsistent because commit could be taken place **as reading is done**
* **Set Transaction to read UNCOMMITTED**
  + Data being read can be DIRTY and inconsistent with data integrity
* **Set Transaction to have REPEATABLE READ**
  + Unless that transaction updates the record, it is always the same.

**WEEK 10: OLAP AND DATA WAREHOUSE**

**Data warehouse**

* Consolidate data from many sources in one large repository

**OLAP**

* **Complex SQL Queries** and **views** in order to **create and produce an analytical report** based on the data and facts

**Data Mining**

* Exploratory search for interesting trends and anomalies

**OLTP** Online Transaction Processing

* Maintains a database that is an accurate model of some real-world enterprise
  + Constructed to support day-to-day operations and constant updating of facts
    - Short simple transactions
    - Relatively frequent updates
    - Only accesses a fraction of the databases

**OLAP online analytic processing**

* Uses information in database to guide strategic decisions
  + Complex queries
  + Infrequent updates
  + Large transactions that would take a lot of computer power
  + More historic data

**DATA WAREHOUSE**

* A special server where **OLAP** and **Data** **mining** **databases** are **stored** in
  + Subject oriented, integrated, time variant, non-updatable collections of data used to support management decision making processes
  + Can accommodate **large sets of data**
  + **Allow OLAP and Data mining** to be operated separately from the main **OLTP server**

**OLAP processes** takes a lot of **computer power**, so if done in **parallel** with **everyday Transaction processes**, it will slow down the server **significantly**

Therefore, **Data warehouse** is there to separate two servers and to allow different optimization for OLTP and OLAP

**Data Warehouse** is updated **periodically** and not **everytime that OLTP** is updated on. This is because, updating a **Data Warehouse** is a pain in the ass.

**ISSUES WITH DATA WAREHOUSING**

**Semantic integration**

* When getting data from **multiple sources**, must eliminate **mismatching data** or **different data integrities.**

**Heterogenous Sources**

* Must **access data** from variety of **source format** and **repository**

**Load, Refresh, Purge**

* Must **load data**, periodically **refresh** it and **purge too old data**
  + Due to the volume of data, updating is a significant task, especially in checking for Data integrity and consistency

**Metadata Management**

* Must keep track all the **metadata information** about **all the data in the warehouse**

**Data Lake**

* Enterprises getting a lot of cheap CPU and Storage unit to store **old historic data**.
  + This is because getting **ONE MASSIVE COMPUTER** will cost them more than buying **several small cheap ones**

**Relational OLAP**

* Data is stored in relational database

**Multidimensional OLAP**

* One **BIG ASS TABLE** that holds all the informations
* NON-RELATIONAL

**Fact Table**

* Table that covers all the tables relevant to the Business Processes
* Collection of numeric measures, which depend on a set of dimensions

**Star Schema**

* The fact and dimensions relations can be displayed in an ER diagram which look like a star
  + 1 Central **fact table**
  + N Dimensions table with **foreign key** relationships **from the fact table**

**PIVOTING**

* When we call **group by** to specify a **subset of the axis**

SELECT \*

FROM HELLO <- This is said to be pivoting against Frequency

GROUP BY FREQUENCY

**SLICING**

* When we call **WHERE** to call a **specific row**

SELECT NAME

FROM STUDENT <- this is said to be slicing through the NAME attribute

WHERE NAME = ‘JOSHUA’

**CUBE OPERATION**

* Process that allows **PIVOTING** to be done across **attribute values** with regards to different Instances

SELECT **S.Market\_id, S.Product\_Id**

FROM Sales S

GROUP BY **CUBE (**S.Market\_Id, S.Product\_Id**)**

**ROLL UP**

* Similar to a **CUBE** operation but creates a subsets moving from right to left

SELECT S.Market\_Id, S.Product\_Id, SUM(S.Sales\_AMT)

FROM Sales S

GROUP BY ROLLUP(S.Market\_Id, S.Product\_Id)

**CUBE VS ROLLUP**

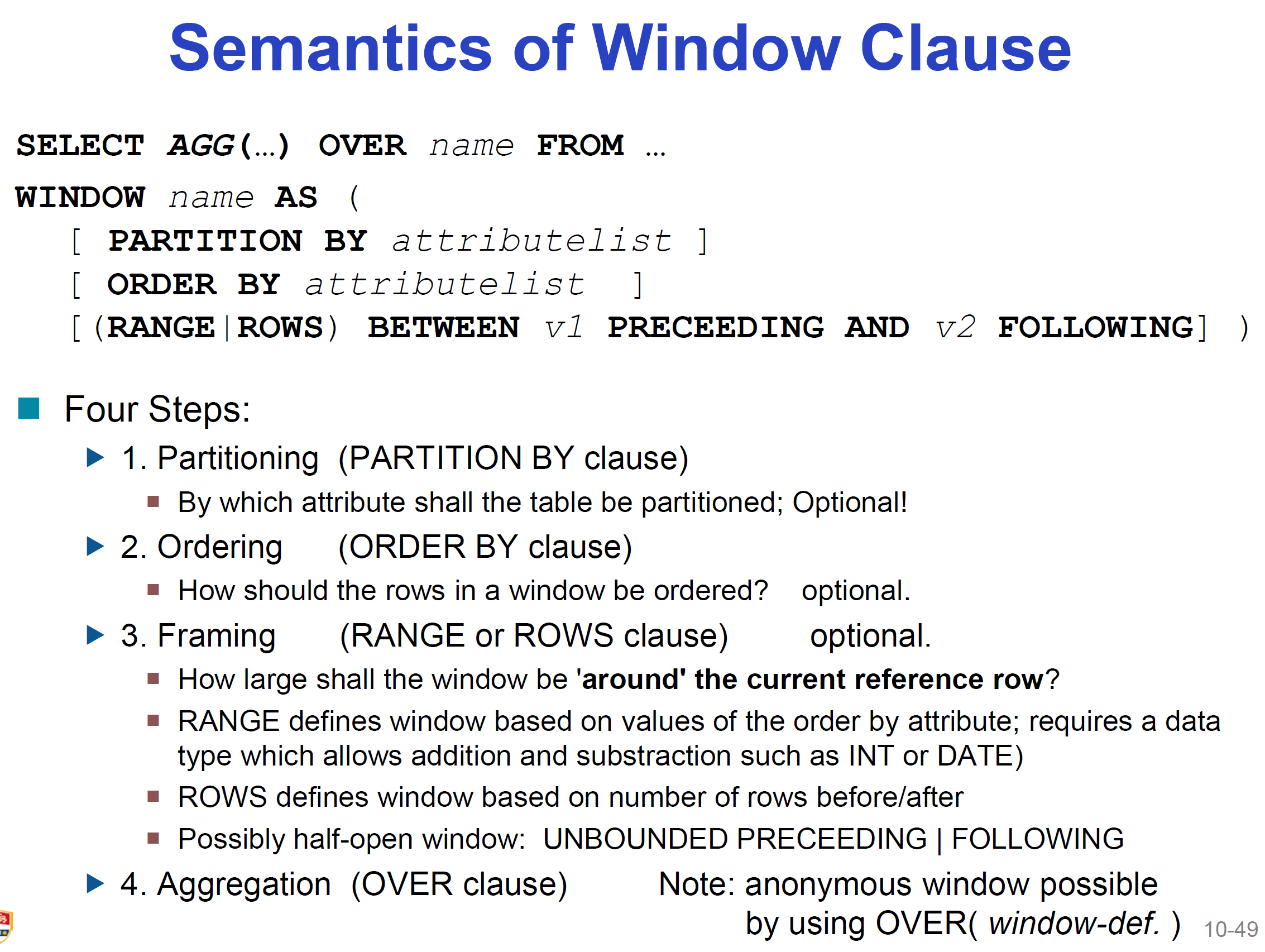
Cube

* First aggregate with all the attributes **combined**
* Then aggregate with all the attributes **individually**
* Then Grand total **with no group by**

**WINDOW**

* Ordered group of tuples around each tuple of a table
  + SQL statement for Window can specify
    - Order
    - Width
  + Tuples can be aggregated using the standard Aggregate functions (SUM, AVG, COUNT)

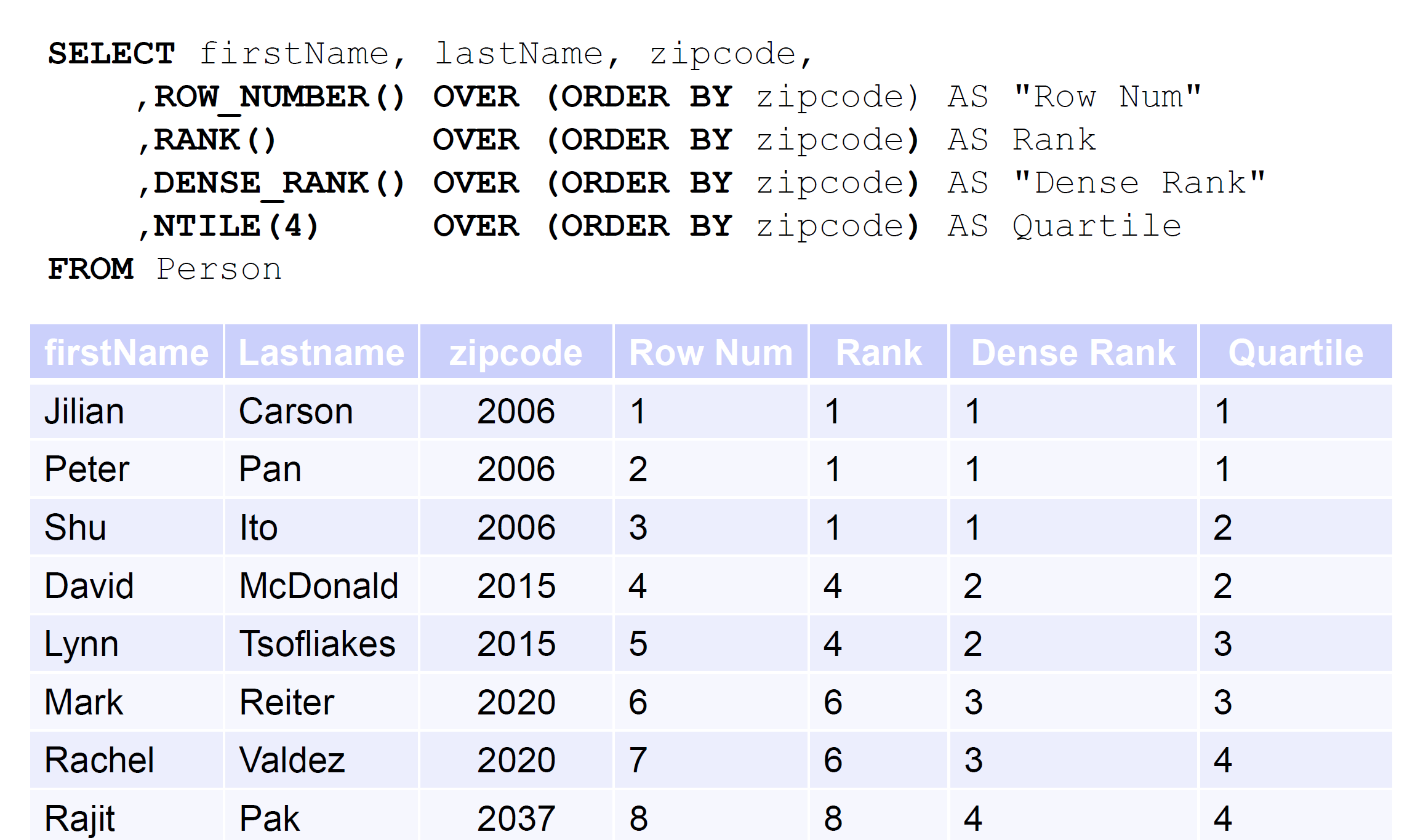
|  |  |
| --- | --- |
| SELECT sid, uos\_code, mark,  AVG(mark) OVER(PARTITION BY uos\_code)  FROM Enrolled | SELECT sid, uos\_code, mark, AVG(mark) OVER w  FROM Enrolled  WINDOW w as (PARTITION BY **uos\_code)** |

****

**RANKS**

We can combine Window and Rank to either order, bucket or split data by a specific attribute

RANK(), Dense\_Rank(), NTile(), ROW\_Number()



Week 11: Storage, Indexing

**Storage Hierarchy**

* **Primary storage**
  + **RAM or Cache** 
    - Volatile but fast
* **Secondary Storage**
  + Hard disk or solid state drive
  + Non-volatile but not that fast
* **Tertiary storage**
  + Magnetic tape, optical storage
  + Offline storage
  + Slowest but non-voletile

**Primary** for currently used data

**Secondary** for main database

**Tertiary** for old versions of the data

File organizational methods

* **Heap files**
  + Search algorithm is in **Linear complexity**
* **Sorted files**
  + Search algorithm is on log(n) complexity with binary search
  + However maintaining the sorted order is difficult as **INSERT, REMOVE, UPDATE** will cause troubles

during both methods above, they leave an empty space when a data row is deleted resulting in an awkard sized area,

* **Indexes** 
  + Entities are stored everywhere but can be accessed easily by looking into **index pointers**
    - **ORDERED INDEX**
      * search keys are stored in sorted order
    - **HASH INDEX**
      * search keys are distributed uniformly across “buckers” using a “hash function”

CREATE INDEX name ON relation\_name (attributelist)

CREATE **UNIQUE** INDEX will indirectly specify and enforce the condition that the search key is a **candidate key**

**DROP INDEX** index name

Disadvantages of Index

* Additional I/O to manage
* Index dataset must be updated when table is modified

**Index management**

* **Clustering index**
  + When **index entries** and **pointed rows in a table** is **ordered the same way**
    - There can only be one clustering index on a table
    - **CREATE TABLE** statement generally **creates clustered index** on **primary key**
* **Un-clustered Index**
  + **Index entries** and **pointed rows** are **not in the same order**
  + There can be many un-clustered index in a table
  + Index created by **CREATE INDEX** is usually **un-clustered**

**Clustering index** is better than **Un-Clustered index when**

* You want to find the **range of search key values**
* **Order of index** matters in **searching**

**Unclustering index** is better than **clustered index** when

* The index is **inserted** more than **searched**

**Index** is called **primary index** when index contains actual data rows, else it is called **secondary index**

An **index** over a **candidate key** is called **unique index** (no duplicate entries)

When **search key** has multiple fields, it is called **multi-attribute**, else it is called **single attribute**

**Given**

**CREATE TABLE** Library(

Callno char(20) PRIMARY KEY,

Title VarChar(255)

)

**CREATE INDEX TitleCatelog on Library(title)**

This **index** called **TitleCatelog** is a **secondary** **Index** while the **initial** **index** called **Library** is a **primary index**

Type of index designs

* Tree based index
  + Binary tree
* Hash based index
  + Fast for equality searches
* Special index
  + Other types of disk based index structures

**Covering index**

* An index that contains all attributes required to answer a given SQL query
* Multi-attribute index

In process of choosing index types you should consider

* **The most important queries in tern.** 
  + **If** the most used query uses several attributes, maybe go for **covering index**
  + If **range** query is used often go for **binary index**