|  |  |  |
| --- | --- | --- |
| \*Data\*  * Fact/information that can be stored and recorded  \*Data Base\*  * Collection of data that contains all the information to operate a function  \*Data Base Management System\*  * Software Package designed to store and manage Data Base(s)  \*System Catalogue\* \*Data Dictionary\*  * Stores metadata about the Database   + Format   + Data type  \*Relational DBMS\*  * Linked DBMS via special key to normalize * database  \*Instance\*  * Content of Database * Restricted by **schema structuring.**  \*Schema\*  * Description of the **Structure** of the data in Database   + What tables exist   + Attributes, types, format  \*Integrity Constraint\* \*Data Integrity\*  * Restriction of possible instance/values that goes against logical structure/schema * Constantly checked when data is manipulated  \*Data Design\*  * **Decision of schema** that will be used for the data * **Decisions on what information** is necessary and in what **format**  \*Data Definition Language\* \*DDL\*  * Used to define Schema. * Allows one to tell DBMS what tables exist and what structure they have.  \*Data Control Language\* \*DCL\*  * Commands that control a database   + Administration   + privileges   + users | \*Data Manipulation Language\* \*DML\*  * Used to Access Data * Commands to:   + Query   + Update   + Insert   + Delete  \*SQL\*  * Combination of DDL and DML that is used to run commands on BDMS  \*Declarative Command\*  * Demands “what is needed” rather than “how to retrieve”  \*View\*  * Describes how a user sees the data  \*Logical Schema\*  * Defines the **structure of data** as it is shared among all users  \*Physical Schema\*  * Describes the **files and indexes** used for **storage on disk**  \*Data Independence\*  * **Applications** not affected by changes in **data structure**  \*Logical data independence\*  * Protection from changes in **Logical Schema**   + Adding extra column  \*Physical Data Independence\*  * Protection from changes in **physical structure and location**  \*Entity\*  * Person, place, object about which you want to gather and store data  \*Entity Type\* \*Entity Set\*  * Collection of entities that share common properties or characteristics * Described by a set of Attributes   + Students, courses, accounts  \*Attribute\*  * Description of one aspect of an entity type | |
| \*Domain\*  * Acceptable values within Data Integrity constraints  \*Key\*  * **Minimal set of attributes** that uniquely identify an **entity** in a set  \*Entity Schema\*  * Schema for entities  \*Data Model\*  * Collection of concepts for describing data   + Structure of data   + Operations on the data   + Constraints on the data  \*Relation\*  * Named, two-dimensional table of data * Must be at least 1NF * Stems from mathematical concept of “sets”  \*Relation Schema\*  * Specifies name of relation as well as it’s attributes and data types  \*Relation instance\*  * Set of tuples for a schema (table)  \*Relational Algebra\*  * Defines some basic operators that can express declarative commands  \*Functional Dependency\*  * When the value of one attribute determines the value of another attribute * If a pair of tuple has the same value in the candidate attribute, they share the same value in the non-candidate attribute.  \*Data Minimalism\*  * The theory of storing as little as possible to make sure hackers wouldn’t have access to information even if they break in  \*Presentation Logic\*  * how data should be presented   + GUI  \*Processing Logic\*  * How data should be processed   + Procedures, functions and programs | | \*Data Management\*  * How data should be managed   + DBMS  \*Interactive SQL\*  * SQL Statement input from terminal  \*Non-Interactive SQL\*  * SQL statements are included in an application program written in a host language   \*Key\*   * Allows identification of any row in a table | |

|  |  |
| --- | --- |
| \*Data Design Processes\*  1. Produce a conceptual or semantic model 2. Translate them into Relational schema 3. Evaluate the schema for quality 4. Constantly improve as needed  \*Level of Abstraction in BDMS\*  \*Roles with DBMS\*\*End User\*  * People who do something that advances the organization’s purpose   + Off-line users who receive **reports**   + Parametric users who execute **pre-written applications**   + Ad hoc user who **explore data**  \*Application Programmer\*  * IT professionals who produce the application that end users run   + Need understanding on how to create an application with access to data on DBMS  \*DB Admin\* \*DBA\*  * Responsible for management of effective and efficient use of resources in providing access to data   + Design logical/physical schema   + Handle security and authorization   + Data availability, crash recovery   + Database tuning  \*DBMS Vendor Staff\*  * DBMS software staff who help DBMS interactions from customer’s perspective   + Operational support staff   + Sales support staff   + Training staff   + DBMS implementors   + Tools Development | \*Database can exist without a DBMS\*  * Data will be stored in a file and directly accessed by programs  \*Files vs DBMS\*\*DBMS adv\*  * Data definition occurs **once** * Declarative Queries * Data Integrity * Single point of data * Security management * Concurrent access * System crash recovery * Quick application development * Absence of Data redundancy  \*Database Design Sequence\*  1. Requirement analysis    * What to store    * What application to build    * What operations to optimize 2. Conceptual design    * Develop high level description of data about how user think of the data 3. Logical Design    * Conceptual design > database schema 4. Physical Design    * Logical Design > physical schema  \*Null values\*  * Allows better Aggregate methods   + Instead of confusion due to -1 * Causes complications in the definition of many operations  \*Data Base Security\*  * Security   + Restricting accessing **without** permission * Integrity   + Restricting modification **without** permission * Availability   + Allowing modification and access to those **with** access  \*Database Access Control\*  * Mandatory Access Control (**Authentication**)   + **User Profile/log in** * Discretionary Access Control (**Authorization**)   + **Access Rights or privileges** |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| \*Relational Algebra\*Set Operations  |  |  | | --- | --- | | Selection | Select a subset of rows from relation  **WHERE, HAVING** | | Projection | Deletes unwanted columns from relation  **SELECT** |  SQL declarative statements  |  |  | | --- | --- | | Union  **R1 UNION R2** | Tuples in relation 1 OR relation 2 | | Intersection  **R1 INTERSECTION R2** | Tuples in relation 1 AND relation 2 | | Difference  **R1 EXCEPT R2** | Tuples in relation 1 NOT relation 2 |  Join Operations  |  |  | | --- | --- | | Cross-Product  X | Allows us to fully combine two relations  **GROUP BY** | | Join  |⪥|(attribute 1 = attribute 2) | Combine matching tuples  **FROM, WHERE** | |  | |  Rename  |  |  | | --- | --- | | Rename ρnew name | Allows renaming | |  | |  Relational Division  |  |  | | --- | --- | | R/S | R is the big set  S is the subset we want to satisfy | | \*Data Modeling\*Conceptual modelling  * Focuses on **abstract representation** of the **key entity types** in the business domain of interest, **their attributes and their**   **inter-relationships**   * ERD  Logical Modeling  * More detailed description of each attribute, relationships and key values * **Functional dependencies, Schema**  Physical Domain  * Detail of physical storage of the data including indexes |

name( grade > 85 Student)

Find names of all the student with grade over 85

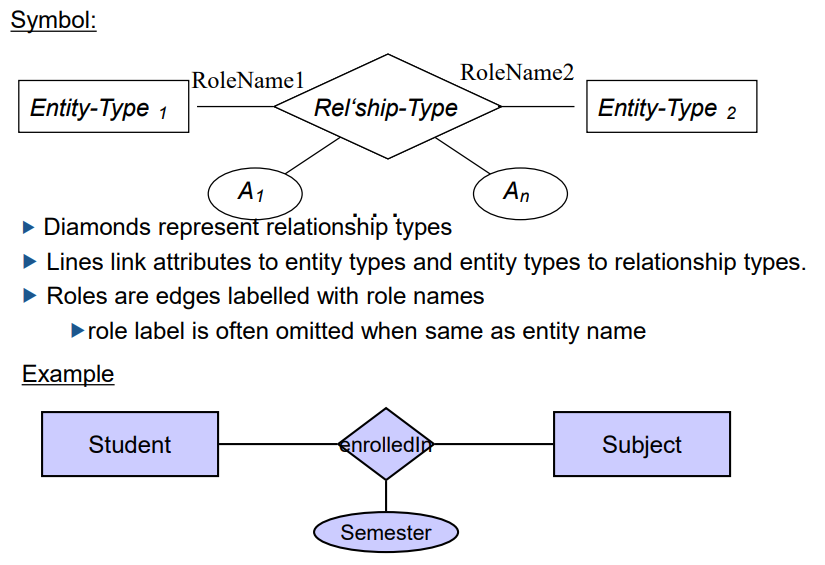
# Conceptual Data Model

Aim: specification of database schema

## \*Conceptual design\*

* Technique for understanding and capturing business **information requirements graphically**
* Depiction of association among **different entity** within business or information system

It does not define **how it is** **implemented** but it focuses on **inter-relational links**

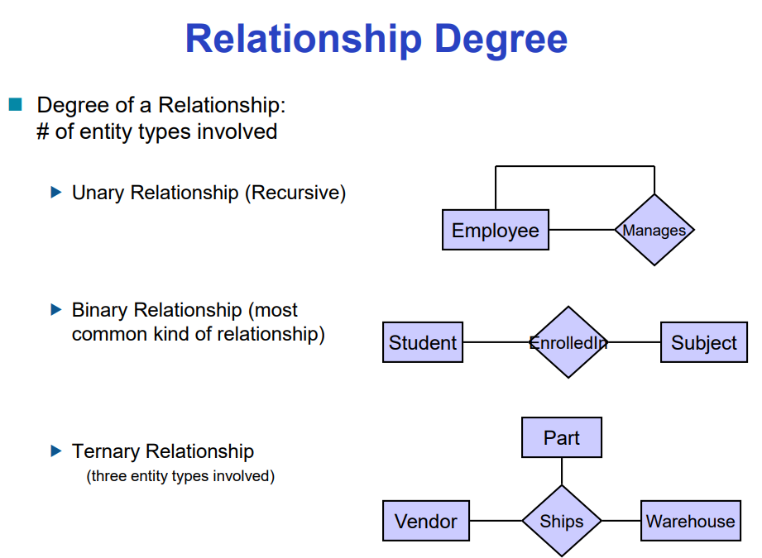
**Model & database independent**

# \*Entity Relationship Model\*

* Graphical representation of
  + what data needs to be contained in the system
  + association among different categories/entity

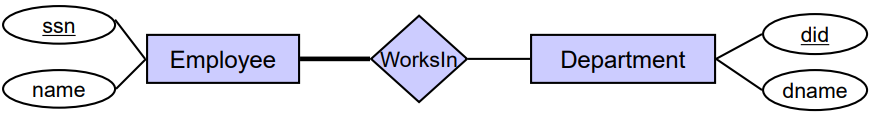
## \*ERD\*

There are three objects that govern ERD

* Entity type : Rectangle
* Attribute : Ellipses
* Relationship : Diamond

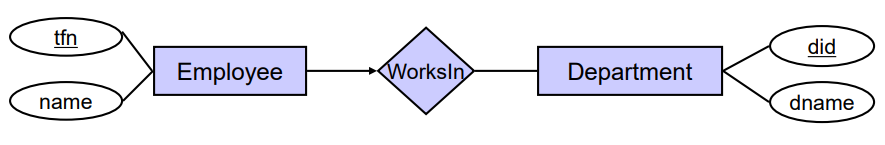
## \*Multiplicity / Constraints\*

## \*Key Constraint \*

* AT MOST ONE
* Many-to-one
* **Thin arrow**

|  |  |
| --- | --- |
| \*Participation constraint\*  * AT LEAST ONE   + **Total participation**   + **Partial participation** * Depicted by a **thick line**    \*Participation and key constraint\*  * EXACTLY ONE * Depicted by **thick arrow** | \*Cardinality Constraints\*  * Restrictions on participation domain * **Number of participation** restriction   Employee works at **1~3** department  Department has **0~** employees \*Weak Entity\*  * When existence of an entity **depends** on existence of another * Entity that doesn’t have individual **primary key** but holds **parent’s primary key** as their **primary key** * Showed by double rectangle * **Discriminator** is a attribute that distinguishes among weak entities |

## Key Constraint

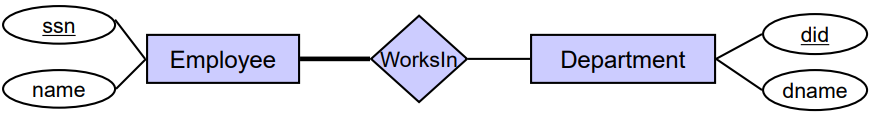


CREATE TABLE **employee(**Tfn, name**)**

CREATE TABLE **department(**did, dname, employee references Employee(tfn)**)**

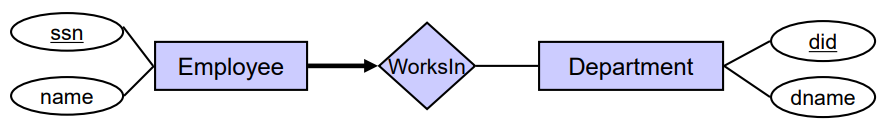
* Department has values of employee
* Employee does not have any information of department

## Participation Constraint

****

CREATE TABLE **employee(**SSN, name, **)**

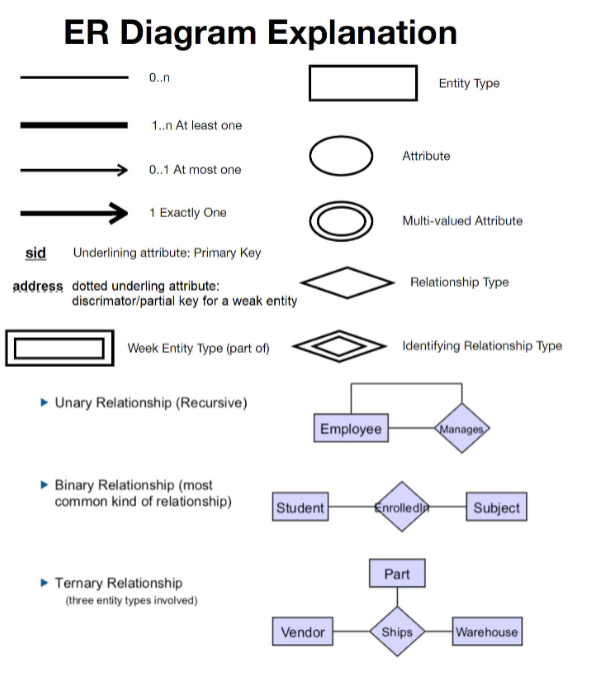
## Participation and key constraint

****

CREATE TABLE **employee(**Tfn, name,did references DEPARTMENT(did)**)**

CREATE TABLE **department(**did, dname**)**

* Store did in employee

****

### #Rows = Cardinality

### #Columns = Degree of relation

# \*Conversion from ERD to SQL\*

Types of Attributes

|  |  |  |
| --- | --- | --- |
| \*Simple Attribute\* | \*Composite Attribute\* | \*Multi-value attribute\* |
| Simple Attribute for an entity | When attribute has attributes on themselves | When attribute holds more than one values |
| Directly onto the relation | Flattened out and added as a separate attribute columns | Separated into a table with a foreign key taken from superior entity |

## \*Weak entity\*

* Becomes a separate relation with foreign key taken from superior entity **NOT NULL**
* An entity type that does not have a primary key
* The primary key of a weak entity type is formed by the primary key of the strong entity types on which the weak entity type is existence dependent

## \*Multiplicity\*

* Many to many
  + Make a relation as a separate table
* One to many
  + Put foreign key with **NOT NULL** constraint
* One to One
  + Put it as an attribute value, or put both **foreign and primary** as **UNIQUE**

# OOP implementations

## \*Specialization\*

When you want to refer to the Sub-class

## \*Generalization\*

When you want to refer to the Superclass

## \*ISA Constraints\*

Overlap Constraints

## \*Disjoint\*

* + Entity can belong to **only one** lower level entity set

## \*Overlapping\* default

* + entity can belong to **more than one** lower level entity set

Covering Constraints

## \*Total Covering\*

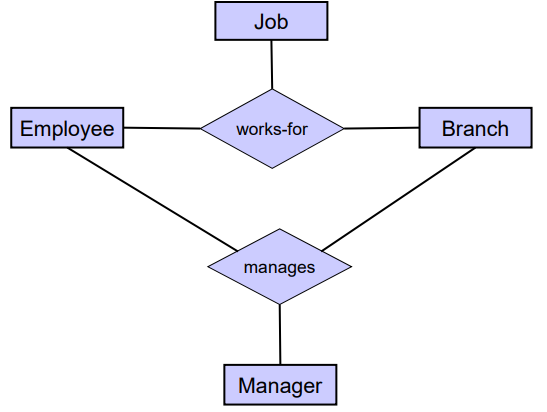
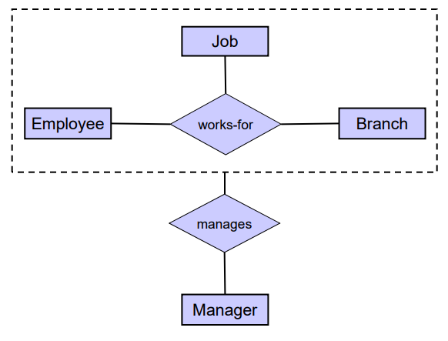
* + Entity must belong to one of the lower level entity set

## \*Partial Covering\*

* + Entity does not need to belong to the lower level entity set

## \*Relational Aggregation\*

Aggregation allows any part of a relationship to become an **abstract entity** for the purpose of participating in another relationship



## \*Conversion from ISA Hierarchy to SQL\*

**Super class’s primary key** also becomes **subclass’s primary key**, (foreign key)

# **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

**SQL Statements** are not **case sensitive**; HOWEVER, the naming conventions of attributes and tables are **case sensitive** and does not permit ‘- ‘characters

# \*The order of Query Clause Evaluation\*

## \*Order of operation\*

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**

## \*SELECT\*

* **\*DISTINCT\*** 
  + **SELECT DISTINCT** country…
  + To force eliminate any duplicates while search
* **Simple arithmetic operations** 
  + **SELECT** age\*2, year/5, age + 4-2 …
* **\*AS\*** 
  + **SELECT** old\_name **AS** new\_name
  + Renaming attribute names
* **COALESCE** (attr\_name, default)
* **\*AGGREGATE FUNCTIONS\***
  + must use **DISTINCT** in addition to aggregate over **sets**
  + ignores **NULL** values.

|  |  |
| --- | --- |
| **AVG()** | Average Value |
| **MIN()** | Minimum Value |
| **MAX()** | Maximum Value |
| **SUM()** | Sum of Values |
| **COUNT()** | Number of values |

## \*FROM\*

* Lists the relations involved in the query

|  |  |
| --- | --- |
| Available join types | Join condition |
| * **INNER JOIN** * **LEFT OUTER JOIN** * **NATURAL JOIN** * **RIGHT OUTER JOIN** * **FULL OUTER JOIN** | * **NATURAL** * **ON** <CONDITION> * **USING** <ATTRIBUTE LIST> |

## \*GROUP BY\*

* Aggregation done **via specific values of attributes**
* ‘partition’ a relation into groups
* EVERY attribute that is not in AGGREGATE needs to be in Group by

## \*HAVING\*

* Conditions for GROUP BY
* Different from where in a sense that, it conditions the **group**

## \*WHERE\*

* Specifies conditions that the result must satisfy
* **Comparison operator** =,>,<,>=,<=,!=
* **AND, OR, NOT**
* **BETWEEN … AND … WHERE** mark **BETWEEN** 75 **AND** 100
* **LIKE …** **WHERE** uos\_Code **LIKE** ‘COMP%’
  + Wild cards
    - ‘%’ : Matching **any number of characters**
    - ‘\_’ : Matching **single character**
* **Concatenation** **‘||’** **WHERE** unikey = name **||** number
* **UPPER(), LOWER() WHERE UPPER(**name**) =** ‘JOSH’
* **CHAR\_LENGTH() WHERE CHAR\_LENGTH(**name**)** = 4
* **IS** **(NOT) NULL** **WHERE** NAME **IS NOT NULL**

## \*ORDER BY\* “…**ORDER BY** name **DESC**, age **ASC”**

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

# TABLE CREATION AND OPERATIONS

|  |  |  |
| --- | --- | --- |
| \*Create\* table **CREATE TABLE** name(  Attribute\_name data\_type constraints  ) | \*Delete\* Table **DROP TABLE** name   * Schema and instances are deleted | Change Existing schema/ \*Alter\* **ALTER TABLE** name  **ADD COLUMN**  **ADD CONSTRAINT** |

# PostgreSQL \*constraints\*

|  |  |  |  |
| --- | --- | --- | --- |
| **\*NOT NULL\*** | | | Set it so that no value in a given column can be null |
| **\*PRIMARY KEY\*** | **Primay Key (attr1, attr2)** | | * Unique, Not null values by default. * Each legal tuple has unique value of primary key |
| **\*FOREIGN KEY \*** | Attribute\_name data\_type **REFERENCES** Table(attribute)  NULLABLE | | Enable a **dependant relation** to refer to it’s **parent relation** |
| **\*UNIQUE\*** | **UNIQUE(attr1,attr2)** | | Make it so that the value in the attribute(pair) is unique in the table |
| **\*CASCADE\*** | **ON UPDATE**  **ON DELETE** | **CASCADE**  **DEFAULT**  **DELETE** | When foreign key gets changed, the content of the referenced value will be reacting accordingl |

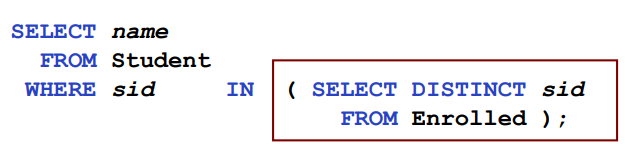
## PostgreSQL \*datatypes\*

|  |  |
| --- | --- |
| SMALLINT  INTEGER  BIGINT | Integer value |
| DECIMAL(p,q)  NUMERIC(p,q) | Fixed-point numbers with precision p and q decimal places |
| FLOAT(p)  REAL  DOUBLE PERCISION | Floating point number with precision p |
| CHAR(q)  VARCHAR(q)  CLOB(q) | Binary string of size r |
| BLOC(r) | Binary string of size r |
| DATE | Date |
| TIME |  |
| TIMESTAMP |  |

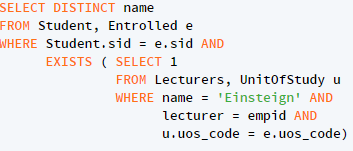
# \*Instance Modification\*

|  |  |  |
| --- | --- | --- |
| \*Insert\* to table \* **INSERT INTO** table\_name (list\_attr) **VALUES** (values) | \*Update\* **UPDATE** table\_name  **SET** attr\_name = value  **WHERE** condition | \*Delete\* **DELETE FROM** table  **WHERE** Condition |
| **INSERT INTO** Student(sid,name) **VALUES** (480222279,”Kim”) | **UPDATE** Student  **SET** sid = 480222279  **WHERE** lname = ‘Kim’ | **DELETE FROM** Student  **WHERE** name = ‘Kim’ |

## \*Nested Subqueries\*

When **SELECT-FROM-WHERE** statement is nested within another query

### \*Noncorrelated subqueries\*

* Does not **depend** on **data** from the **outer query**
* **Executes once** for the **entire outer query**

### \*Correlated Subqueries\*

* Makes use of **data from the** **outer query**
* **Executes once** for **each row** of the **outer query**
* Can use the **EXISTS** operator

## \*In vs Exists\*

**\*IN\***

* Compares a **value v** with a **set of values V** and evaluates **TRUE** if **v** is **one** of the **element in V**

**\*EXISTS \*(correlated subquery)**

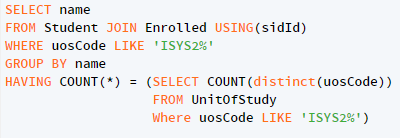
* Used to check whether the result of a correlated nested query is **EMPTY** or not

## Subquery operators

|  |  |  |
| --- | --- | --- |
| (NOT) EXISTS | Tests whether a set is empty or not |  |
| UNIQUE | Tests whether a subquery has any duplicate tuples |  |
| ALL | Tests whether a predicate is true for the whole set |  |
| SOME | Tests whether some comparison holds for at least one set element |  |

# \*Divisions\*

* “For all set”
* Condition that must be met across all set
  + *“Write an SQL query that finds the student(s) that have taken every ISYS subject in second year”*

  
Manipulation of Comparison to do DIVISONS

|  |  |
| --- | --- |
| Checking for   * Empty set (NOT EXISTS(set)) * Set membership (value IN set) | SET Operations   * Set UNION * Set INTERSECTION * Set EXCEPT |

# Access Control \*Grant\* \*Revoke\*

|  |  |
| --- | --- |
| **GRANT** privilege\_list  **ON** table\_name  **TO** user\_list  **{WITH GRANT OPTION}** | **REVOKE** privilege\_list  **ON** table\_name  **FROM** user\_list |
| **PRIVILEGE LIST**   * SELECT * UPDATE * INSERT * DELETE * REFERENCE | |

# \*VIEW\*

|  |  |
| --- | --- |
| 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 VIEW** name  **AS** <query> |
| **CREATE VIEW** ageStudent **AS**  **SELECT** sid, name  **FROM** Student |

## \*View Updates\*/ \*View Insert \*

**UPDATES** or **INSERTS** will be possible **with access** however, attributes not in **VIEW** will either be **NULL** or **DEFAULT VALUES**

## 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

|  |  |
| --- | --- |
| Problem with View  * Updates will cause null or default value insert * New tuples will not be visible to view **unless they have insertion access** * Because View may be ambiguous in relation definition | Role Based Authorization **CREATE ROLE** manager  **GREANT SELECT,INSERT ON** Student **TO** manager  **GRANT** manager **TO** shari  **REVOKE** manager **FROM** 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**

# **Integrity constraints**

## \*Static Integrity Constraint\*

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

\*Semantic constraint\* vs \*Structural Constraint\*

|  |  |
| --- | --- |
| Semantic   * When you want to verify what is inside the attributes | Structural   * When you want to control how the attributes are constructed and structured |

## Dynamic Integrity Constraint

* Predicated on Database **state changes**
  + Triggers

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Integrity constraint type | Definition | | | | |
| \*Domain Constraint\* | Fields must be of right data domain | | DEFAULT  NOT NULL  NULL | | |
| User defined domains | | **CREATE DOMAIN** domain\_name **data\_type** | | |
| **CREATE DOMAIN** Dollars **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 Key & referential integrity\* | **Must** refer to an **existing** parent entity | | | | |
| * **ON DELETE** * **ON UPDATE** | | | * **NO ACTION** * **CASCADE** * **SET NULL** * **SET DEFAULT** | |
| Constraint checking across multiple attribute simultaneously | **CONSTRAINT** constraint\_name **CHECK** (semantic-condition)   * Semantic-condition could be SQL as well. | | | | |
| DEFERRING CONSTRAINT CHECKING \*Differ\* \*Differable\* | Deferring refers to the ability to delay constraint checking until after transaction is complete | **\*NOT DEFERRABLE\*** | | | **\*DEFERRABLE\*** |
| Every modification will check the constraints **immediately** | | | **INITIALLY DEFERRED**   * Wait until transaction end, but allow dynamic change later |
| **INITIALLY IMMEDIATE**   * Check immediate, but allow dynamic change later |

## Modification to constraints

### 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

## Dynamic Constraint Integrity

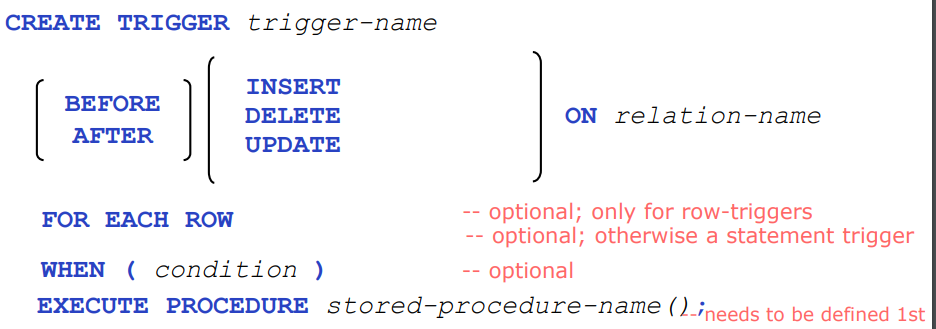
### \*Triggers\*

Statement that is executed automatically if specified modification occurs to DBMS

## Basics

**ON** event **IF** precondition **THEN** condition

|  |  |
| --- | --- |
| **BEFORE TRIGGER** | **AFTER TRIGGER** |
| Checking **Integrity** | **Integrity maintenance** and **update propagation** |

****

## Trigger graduality

* Management of how many times a trigger should be called

### \*Row-Level granularity\*

* + - Change of a single row is an **event**
    - Called multiple times per Statement

### \*Statement level granularity\* (default)

* + - Each statement that can change multiple rows is an **event**
    - Called Once per statement

# Database Application Development

### Client Side DB Application Development

Different level of \*SQL interface\*

|  |  |
| --- | --- |
| \*Statement level interface\* | \*Call level interface\* (\*Python\*) |
| * Embedded SQL * Application program is a mixture of host language statements and SQL statements | * Create special API to call SQL * SQL statements are passed as arguments to host language |

## \*Connect\*ing to Database

* Sessions start when **Connection is created**
* Conn = psycoph2.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.

## Static and Dynamic SQL

|  |  |
| --- | --- |
| \*Static SQL\* | \*Dynamic SQL\* |
| When the whole SQL is constructed prior to running  Useful when SQL is known before   * No SQL injection possible | * Application constructs SQL statements at run time * Python is a interpreted language who constructs at run time   + Prone to SQL injection |
| * Executed more efficiently | Can be advantageous because   * Flexible * General |

## \*Parameterized SQL\*

**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**

## \*Impedance Mismatch\*, \*Buffer Mismatch\*

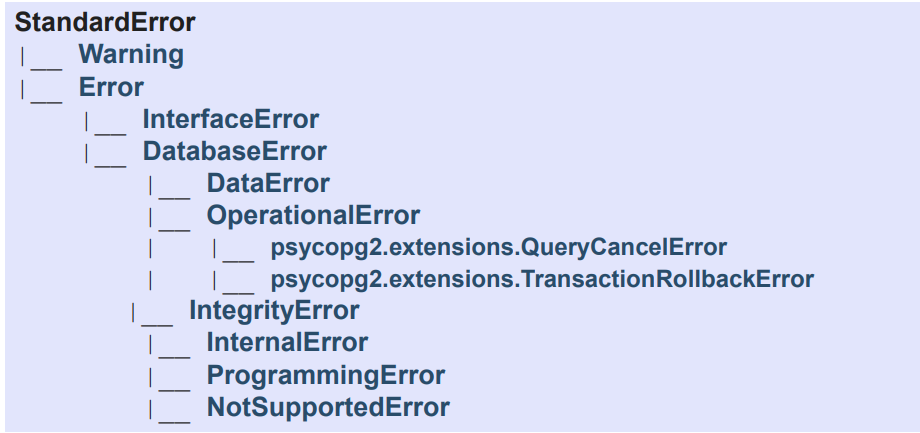
* This is a problem that happens when SQL deals with **tables with arbitrary** size while **host** **language** program deals with **fixed sized buffers**
* This is fixed by concept of **cursor** or **pointers**
* **Cursor points** to the **result set**
  + **Result set** is the set of rows produced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

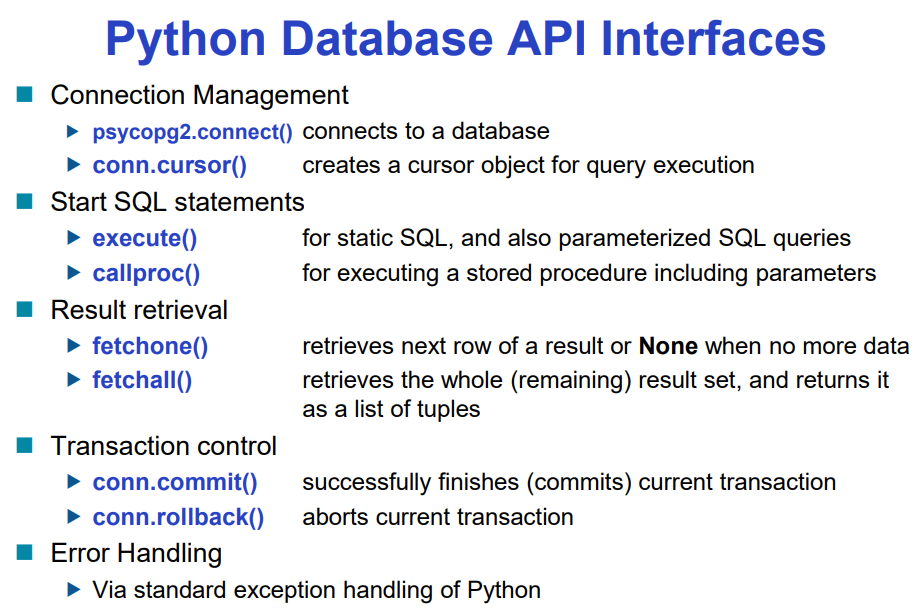
# Server side DB Application Development

* Stored Procedures

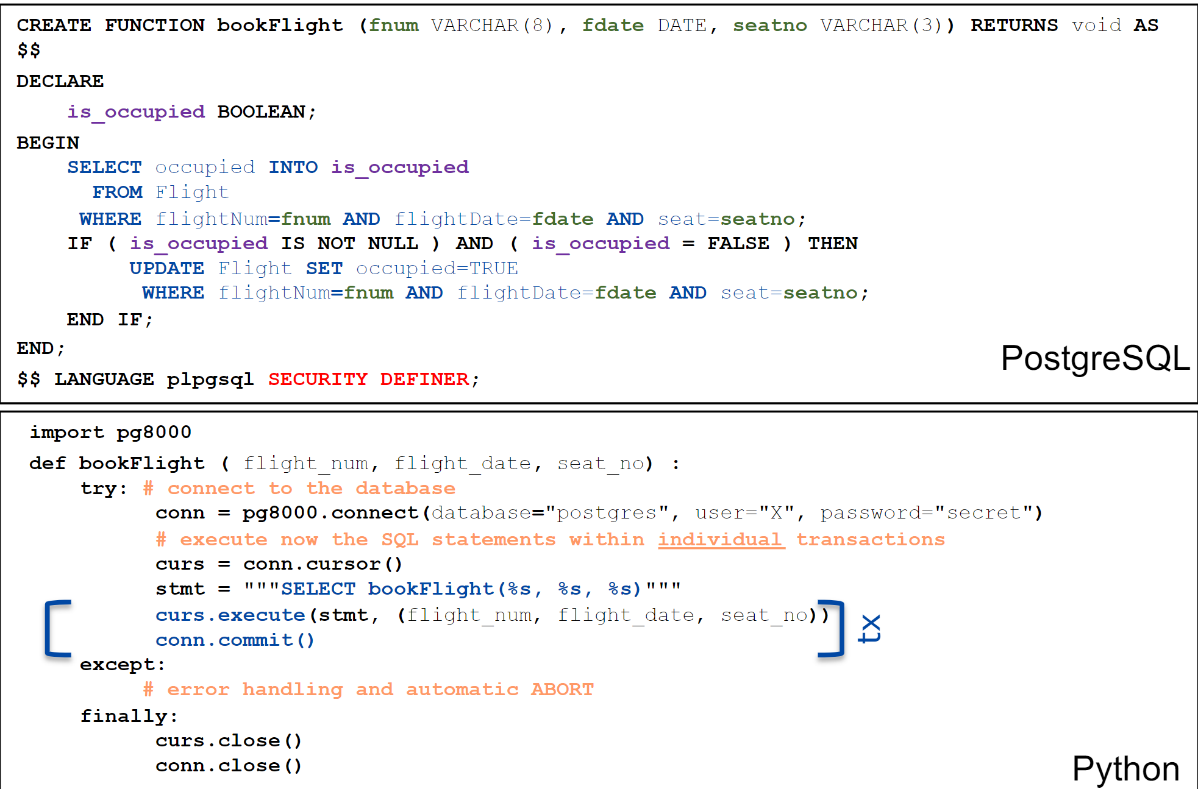


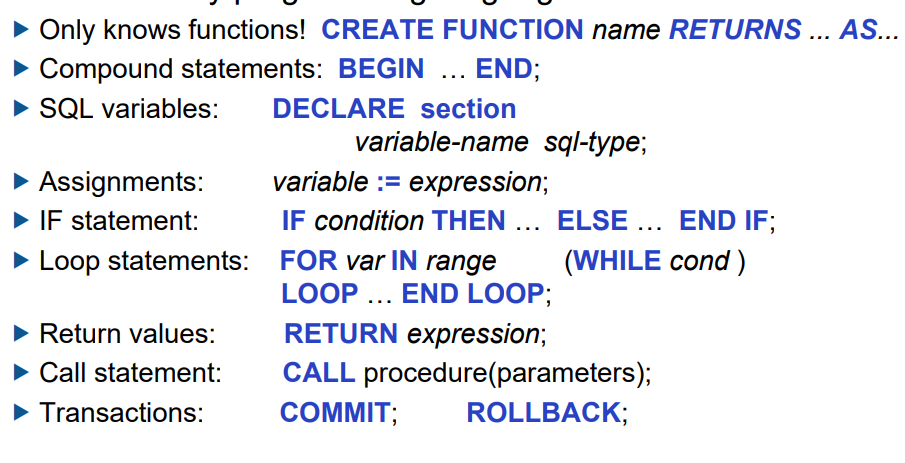
## Python Database API Interface

|  |  |
| --- | --- |
| **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 |

****

# Stored procedures





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

$$

DECLARE

BEGINE

SELECT …

FROM….

END

$$ **LANGUAGE** plpgsql

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

# Aggregation for OLAP

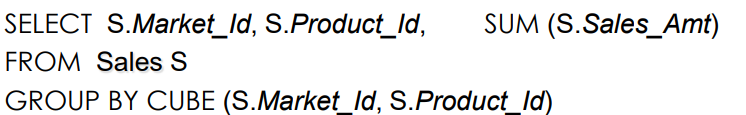
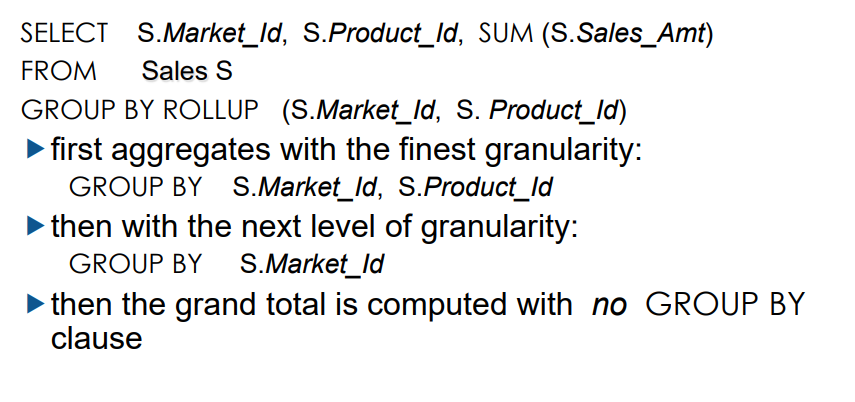
### \*Pivoting\*

* When we view data as a multi-dimensional cube and group on a subset of attributes
  + **GROUP BY**

### \*Slicing\*

* When we use **WHERE** to specify a specific instance.

## Cubing and power of **GROUP BY** analysis, \*Cube\*

* To be able to analyse the whole data you need to **GROUP BY**
  + Entire table with all attributes
  + Each relationship attributes
  + Individual attribute values.
* This is simplified by doing **CUBE**
  + **GROUP BY CUBE(**attr1, attr2…)

## \*ROLLUP\*

* It is similar to **CUBE** but then it only is focused with **attribute 1** FROM the parameter

## \*Window\*

* Ordered group of tuples around each tuples of tables

## \*Indexing\*

|  |  |
| --- | --- |
| **CREATE (UNIQUE) INDEX** name **ON** relation\_name(attribute\_list)   * This creates an index on primary keys * **UNIQUE** is to make it so search key is not repeated | **DROP INDEX** index\_name |

|  |  |
| --- | --- |
| \*Clustering Index\* | \*Un-clustered Index\* |
| When **index entries** and **pointed row in table** is **ordered in the same way** | **Index entries** and **pointed rows** are **not in the same way** |
| * There can only be one clustering index on a table | * There can be many un-clustered index in a table |
| * Created by the **DBMS** when **table is created** | * Created by the **User** when doing **CREATE INDEX** |
| **Adv**   * You want to find the **range of search key values** * When **result order** matters | **Adv**   * Index is **inserted** more than **searched** |

# Evaluation of DB Design

## The condition of **adequacy**

* The design should allow representing all the important facts about the Database

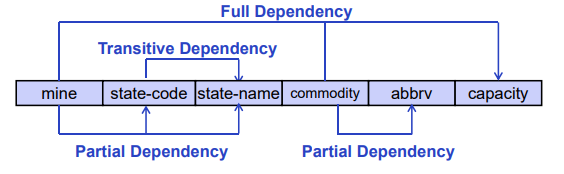
## Condition of **Reduced Redundancy**/ Normalization

* Removal of wrong processes by **removing redundancies** and **creating separate tables**

## Aftereffect of **REDUNDANCY**

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

# Type of \*functional dependencies\*

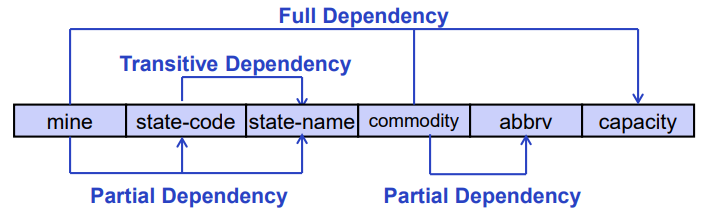
****

|  |  |  |
| --- | --- | --- |
| \*Partial Dependency\* | \*Transitive Dependency\* | \*Trivial Dependency\* |
| When the attribute is determined only by a part of candidate key | When a dependency has MIDDLE MAN  When non-key value determines an attribute | When key pair is used to determine itself |
| Mine -> state-code | Mine -> State-code -> state-name | (Mine, Commodity) -> Mine |

|  |  |
| --- | --- |
| \*Candidate key\* | \*Super Key\* |
| **Minimal set of attributes** with ability to uniquely identify a row | **Any number of attribute** with ability to uniquely identify a row |
| **Candidate key** can be **Super key**, but **Super key** is not always a **Candidate Key** | |
| **Differences** between **candidate key** and **primary key** is that **primary key** cannot be NULL while **candidate key**’s attributes can be NULL | |

## \*Normalization\*

|  |  |  |
| --- | --- | --- |
| **Type of Normalized form** | **Process** | **Result** |
| **1NF** | Removal of **Multivalued attributes** |  |
| **2NF** | Removal of **Partial Dependencies** | All non-key attributes are fully functionally dependent on primary key |
| **3NF** | Removal of **Transitive Dependencies** | No functional dependencies between Non-primary key attributes |
| **Boyce-Codd normal Form** | Remove instances when **non-key** attribute determines **key attribute** |  |

****

|  |  |  |
| --- | --- | --- |
| \*1NF\* | \*2NF\* | \*3NF\* |
| The table is already at 1FN because it does not have any multivalued attributes. |  |  |

## Table Decomposition

* An act of breaking down table into many components so that
  + Each new **Relation Schema** contains a **subset of the attributes** from **original table**
  + All the dependencies are still present

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

## Method to find keys from Functional Dependencies

### \*Attribute closure\*/ The chase

* + Determining the **close of attributes (X+)**
    - This represents all the attributes accessible by X
    - Find the first that **cannot be accessed** by any **Attribute**
    - Then find **another attribute** that **cannot** be accessed by that **initial attribute**
    - Keep **listing attributes you have access to** and keep finding elements that **you cannot access yet** until you have **all the attributes**
    - That is the result

### Closure of Functional Dependencies / Armstrong Axioms

* + **Reflexivity rule**
  + **Augmentation rule**
  + **Transitivity rule**

## Minimal cover

* Simplify the functional dependencies

|  |  |  |
| --- | --- | --- |
| Method to find keys from Functional Dependencies | | |
| Attribute closure/ The chase | * + Determining the **close of attributes (X+)**     - Find the first that **cannot be accessed** by any **Attribute**     - Then find **another attribute** that **cannot** be accessed by that **initial attribute**     - Keep **listing attributes you have access to** and keep finding elements that **you cannot access yet** until you have **all the attributes**     - That is the result | |
| Closure of Functional Dependencies / Armstrong Axioms | Reflexivity Rule |  |
| Augmentation Rule |  |
| Transitivity Rule |  |
| Union Rule |  |
| Decomposition Rule |  |
| Pseudo-Transitivity Rule |  |

# Transaction

Collection of **one or more operations** on **one or more databases** which reflects a **single real world transaction**

## Transaction Properties ACID

* Atomic
* Consistent
* Isolated
* Durable

|  |  |
| --- | --- |
| **\*COMMIT\*** | **\*ROLLBACK\*** |
| * When you want to store changes made by the transaction * **Request**, meaning system might reject it with reason | * When you want to **cancel/abort transaction process** and **revert** tot eh **most recent stable state before transaction** beginning   + User’s change of mind   + Explicit program calls for **error handling**   + Integrity constraints   + System crash |

**Auto-commit**

* When commit is done after EACH SQL COMMAND

**Python DB-API** is **explicit mode** meaning you need to call **commit()** to store and complete transactions

Transaction is considered **consistent if**

* All **Static Integrity** is satisfied
* No **Dynamic constraints** have been violated
* **New State** satisfies specification of **transaction**

## Transaction Checking

|  |  |
| --- | --- |
| Automatically | Manually |
| * CHECK * ASSERTION * TRIGGER * Not always desirable since unnecessary checking might result in slower processing time | * Perform checks in application code only when it is needed * Difficult to maintain as transactions are **modified/added** |

## Deferrable Integrity constraints

* When you can withhold the Integrity constraint checking until after transaction is complete

## Incomplete Transaction, System crash recovery

* When System crash happens **during transaction** the process should clean up and call **rollback** responsively to revert transaction.
* Each item holds the value of the last committed transaction for this **rollback functionality**

## Concurrency

* To allow transactions to be **MULTI-THREAD SAFE**

### \*Serializability\*

* + Process of finding elements that do not require **changes to internal state of database** and only locking the database to those processes **that updates**
  + Performance drop due to bottleneck

## \*Logging\*, \*Log\*

* **Append only** collection of entries showing all the changes to the data that happened in order as they happened
* they are used to **abort and rollback** Transactions as well for **crash recovery** to figure out the **last stable state to revert to**
* Requires **extra** **storage**
* Possible delay in **transaction commit** as it needs to **store information**

## 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**
* Due to the number of information transfer, it has very long delay

## \*Locking \*

* Blocking transactions if another transaction already has a lock
* Risk of **dreadlock**

**\*ISOLATION\***

What information should Transactions read?

|  |  |  |
| --- | --- | --- |
| Transactions to read **Committed data** | Transactions to read **uncommitted data** | Transaction to have **repeatable read** |
| * Data being read is always clean and stable * It can be inconsistent because commit could be taken place **as reading is done** | * Data being read can be **dirty** as transaction is not complete | * Unless the transaction updates the record, the data is always the same |

|  |  |
| --- | --- |
| Potential Error with SQL implemented Application   * Connection issue * SQL syntax error * NULL value error * Log in error * Empty Result | Error Messages Should Never Be Visible To Users   * Internal Logics * Internal Relations * Database Vulnerability * Potential to SQL Injection Attacks |

Protection Against SQL Injections

|  |  |
| --- | --- |
| Anonymous Parameter | Named Parameter |
| Cur.execute(“SELECT name  FROM Student  WHERE sid = %s”,(stdid,)) | Cur.execute(“  SELECT name  FROM student  WHERE sid = %{sid}s“,  {‘sid’:stdid}  ) |

This is so that any input from user will be checked in the function cur.execute() function and it’s built in security checking functionality

# \*Stored procedure\*

* SQL functions that are stored in DBMS and users have access to **calling** the function.
  + Logic is not visible even to the application layer
* **CALL** functionname()

|  |  |
| --- | --- |
| Pro | Con |
| * Security   + Access privileges on stored procedures and not any of the databases that the logic requires * Efficiency   + Instead of sending all the SQL commands via online you only send the request for one procedure call. * Correctness   + Avoid pitfall of wrong transaction boundaries by controlled function calls * Improved Maintainability * Central Code-base for all applications’ * Reduced Data transfer * DBMS centric Security * Access to all tables. | * Good design needed * Difficult to debug * Separate language to learn * Only meant for functions that happen at regular basis |

# \*Password Security\*

* Never store Password as a string value in SQL

|  |  |
| --- | --- |
| Hash Function | Salting |
| * **Hashing** is a method in which you **feed** **given** **text** **field** **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   + Faster access   + Less sensitive data transmitted through internet | Salting is a method of **inserting a random text** into the **password** to the inputted text field value into another value that would result in a **more complex process** for the **hash function.**  Merge of functions to make hash value more complex. |

# Data Encryption methods

* Data Minimalism
* Strong Encryption
* Consult expert > implement from scratch

# Data Structures

|  |  |  |
| --- | --- | --- |
| \*Unstructured\* | \*Semi-Structured\* | \*Structured\* |
| * Data that cannot be easily organized into rows and columns * Difficult to search, manage and analyze   + Text document   + Video   + Audio Files | Mix of both types of data that can have both structured and unstructured section in data   * Emails   + Sender, receiver CC     - structured   + Body of email     - unstructured | Data that is structured in an interactive format with **rows and column** and can be mapped into predefined **schema**   * rational table * excel spreadsheet |

Method of analysis

|  |  |
| --- | --- |
| Unstructured | Structured |
| 1. First get the data 2. try to understand the format of the content 3. read data definition 4. interpret the data 5. implement data fact | 1. Schema First 2. reading data 3. interpret the data 4. implement data facts |

# Ontology

|  |  |
| --- | --- |
| * Method to construct and define a vocabulary for **unstructured data** * Explicit and formal specification of information in a domain   + Concept   + Properties   + Attributes   + Constraints on properties and attributes   + Individual instances | *“formal, explicit specification of shared conceptualization”*   * Formal   + Machine readable and processable * Explicit   + Something in concrete form * Conceptualization   + Defines an abstract model describing a field of knowledge or 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**
* Create Vocabulary for describing domain knowledge and provide explicit specification of intended meanings
* Creation of **Knowledge** **graph**

# Knowledge Graph

* **Formal description** of a **certain knowledg**e that can be **accessed and explored by machine**
* Large integrated ontologies that represent the knowledge by **modeling the objects in the world and their relationships**
* GOOGLE SEARCH ENGINE IS A KNOWLWEDGE GRAPH

# Semantic Web

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

|  |  |
| --- | --- |
| \*OLAP\* | \*OLTP\* |
| * 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 | * Maintains a Database that is accurate model of real world enterprise * Constructed to support **day to day tasks** and **constant update on DBMS**   + Short **simple transactions**   + Relatively **frequent updates**   + Only accesses a **fraction of the databases** |
| Complex SQL Queries and Views in order to **create and produce an analytical report** based on the data and facts |  |
| Updated **periodically**   * Updating OLAP takes a large sum of computer power | Updated **Every time** |
| When you Aggregate or analyses from the data | When you query or add/insert/update to the data |

\*Data Warehouse\*

* Special server where OLAP and Data Minding operations are done and stored
  + Subject oriented, integrated, time variant, non-updatable collections of data
  + Can accommodate **large sets of data**
  + **Allows OLAP and Data Mining** to be operated **independently** from main **OLAP server**

## Issues with Data Warehouse

|  |  |
| --- | --- |
| Semantic Integration | When getting data from **multiple sources,** must eliminate **mismatching data** or **accommodate different data integrity** |
| Heterogenous Sources | Must **access data** from variety of **source format and repository** |
| Load, refresh, purge | Must load data, periodically refresh and purge old data   * Due to the volume of data, these takes take up a lot of computer power   + Data integrity checking and consistency translations |
| Metadata management | Must keep track of 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**

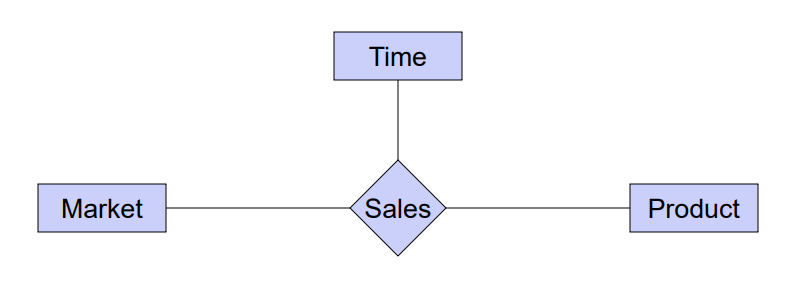
|  |  |
| --- | --- |
| Relational OLAP | Multi-Dimensional OLAP |
| * Data is stored in relational database | * One **BIG ASS TABLE** that holds all the informations * NON-RELATIONAL |

## \*Fact Table\* (Multi-dimensional Table for Multi-dimensional OLAP)

* Table that covers all the tables relevant to the Business Process
* 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 has **fact table at the center** with radiating entities as **different dimensions** that hold **primary-foreign key relationship**
  + 1 Central **fact table**
  + N Dimensions table with **foreign key** relationships **from the fact table**



|  |  |  |
| --- | --- | --- |
| Primary Storage | Secondary Storage | Tertiary Storage |
| RAM or Cache   * Volatile * Fast | Hard disk   * Magnetic tape, optical storage * Non-volatile * Slow | Hard disk   * Offline storage * Big servers |
| Currently used data | Main database | Historic data |

# File organizational method

* Heap Files
  + Search algorithm is **Linear Complexity**
  + Advantageous when getting **all the data at once**
* Sorted Files
  + Can implement Binary search
    - Log(n)
  + Maintaining Sorted order is difficult when doing **Insert, Remove or update**
  + Best if records must be retrieved **in some order, or range**
* Indexed
  + No form of structured order but access is easily looked by **index pointers**
    - Ordered Index
      * When search keys are stored in some order
    - Hash Index
      * Search keys are distributed uniformly across “buckets” using **hash function**

# Downside of Indexing

* Additional I/O to manage
* Index must be updated when table is modified
  + This can be costly, dependant on Index structure