**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

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

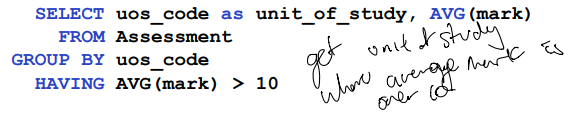
* **UNIQUE**
  + List attributes that comprise a candidate key
* **DISTINCT SELECT DISTINCT** country…
  + To force eliminate any duplicates while search
* **Simple arithmetic operations SELECT** age\*2, year/5, age + 4-2 …
* **AS SELECT** old\_name **AS** new\_name
  + Renaming attribute names
* **AGGREGATE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

|  |  |
| --- | --- |
| **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 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. |
| **FOREIGN KEY** | Attribute\_name data\_type **REFERENCES** Table(attribute) | | 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 |

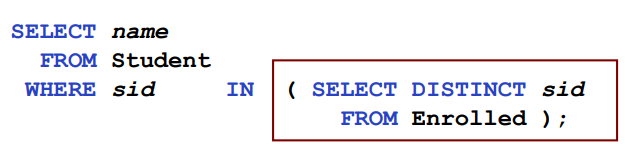
## PostgreSQL datatypes

|  |  |
| --- | --- |
| 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 |  |

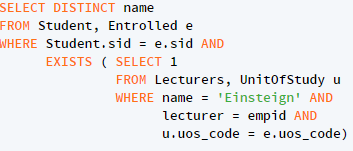
# Instance Modification

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

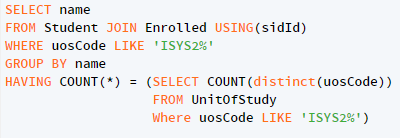
* Used to check whether the result of a correlated nested query is **EMPTY** or not

## 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 DIFFERENCE |

# Access Control

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

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

## View Updates/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
  + Assertion

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** | 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**

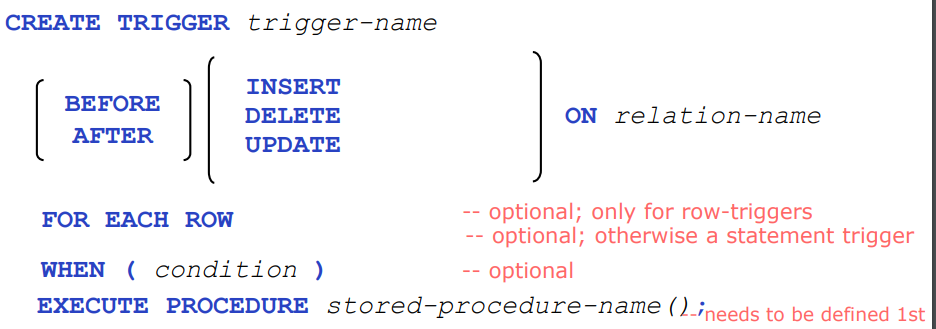
* Constraint modification
  + **ADD CONSTRAINT** constraint-name new-constraint
  + **DROP CONSTRAINT** constraint-name
  + **RENAME CONSTRAINT** old name **TO** new-name
  + **ALTER COLUMN** attribute-name **SET** domain-constraint

## Dynamic Constraint Integrity

### Triggers

Statement that is executed automatically if specified modification occurs to DBMS

|  |  |
| --- | --- |
| 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
    - Events are statements
    - Called Once per statement

# Database Application Development

### Client Side DB Application Development

|  |  |
| --- | --- |
| 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 |

## Connecting 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 |

## 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

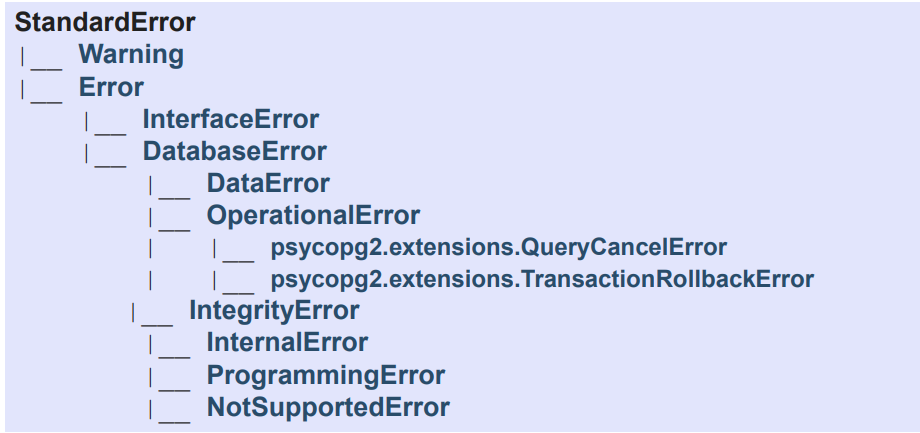
* This is a problem that happens when SQL deals with **tables with arbitrary** size while **host** **language** program deals with **fixed sized buffers**
* This is fixed by concept of **cursor** or **pointers**
* **Cursor points** to the **result set**
  + **Result set** is the set of rows produced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

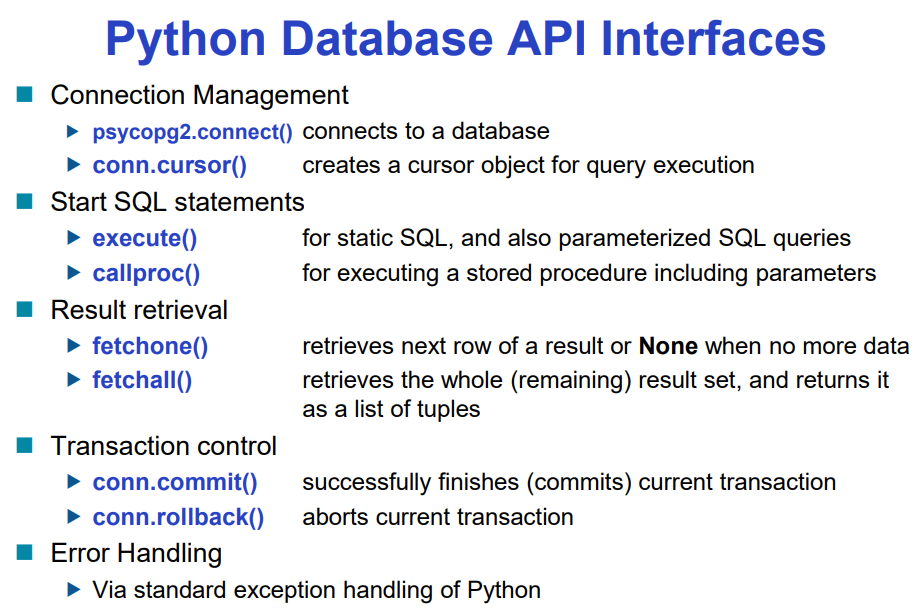
# Server side DB Application Development

* Stored Prodecures

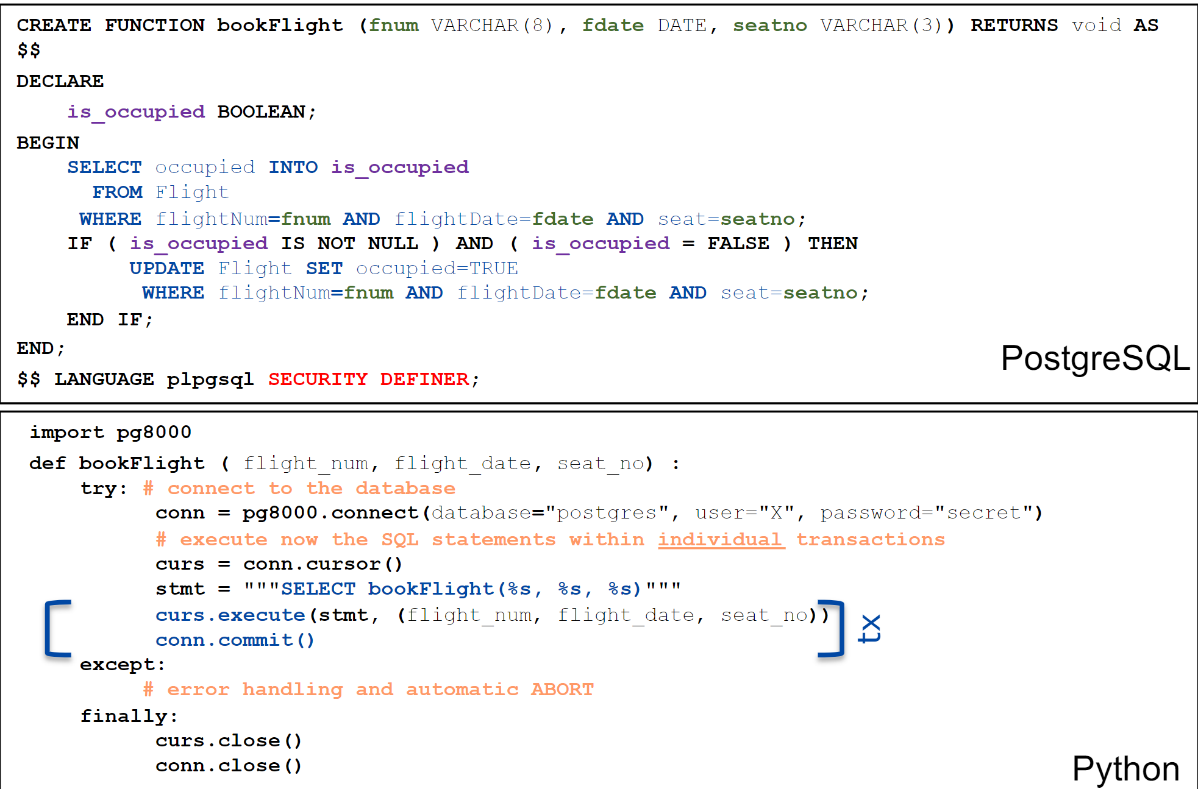


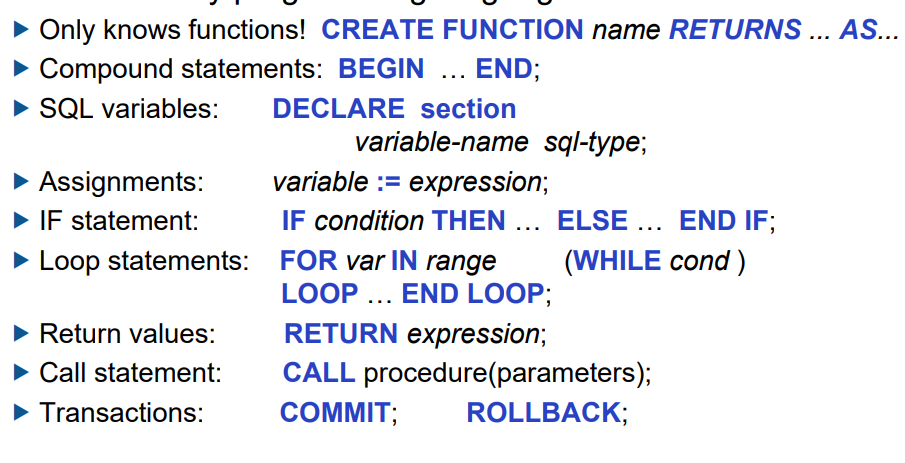
## Python Database API Interface

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

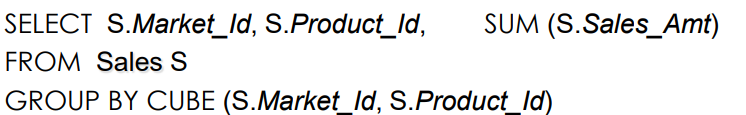
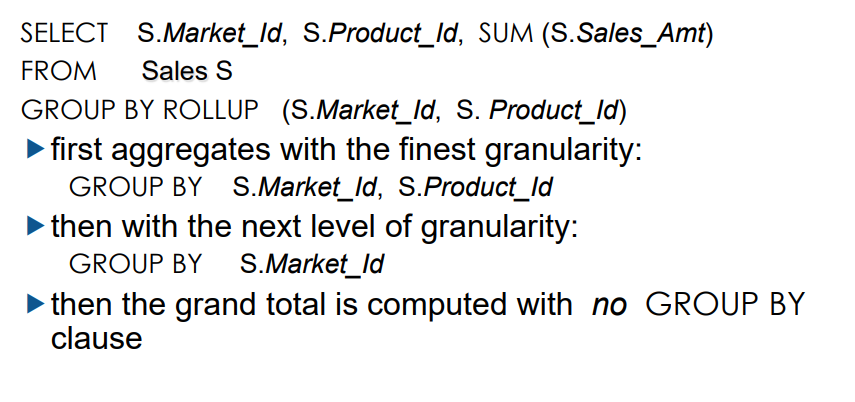
### Pivoting

* 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

* 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** |