**Database Management System (DBMS)**

* ■ Basically, a system for managing data
* ■ DBMS contains information about a particular enterprise (application)

● Collection of interrelated data

● Set of programs to access the data

● An environment that is both *convenient* and *efficient* to use

**■ Drawbacks of using file systems to store data:**

● Data redundancy and inconsistency

! Multiple file formats, duplication of information in different files

● Difficulty in accessing data

! Need to write a new program to carry out each new task

● Data isolation — multiple files and formats

● Integrity problems

! Integrity constraints (e.g. account balance > 0) become “buried” in program code rather than being stated explicitly

! Hard to add new constraints or change existing ones

* ■ Schema – the logical structure of the database

● Example: The database consists of information about a set of customers and accounts and the relationship between them)

● Analogous to type information of a variable in a program

● Physical schema: database design at the physical level

● Logical schema: database design at the logical level

* ■ Instance – the actual content of the database at a particular point in time

● Analogous to the current values of a set of variables

**Attribute Types**

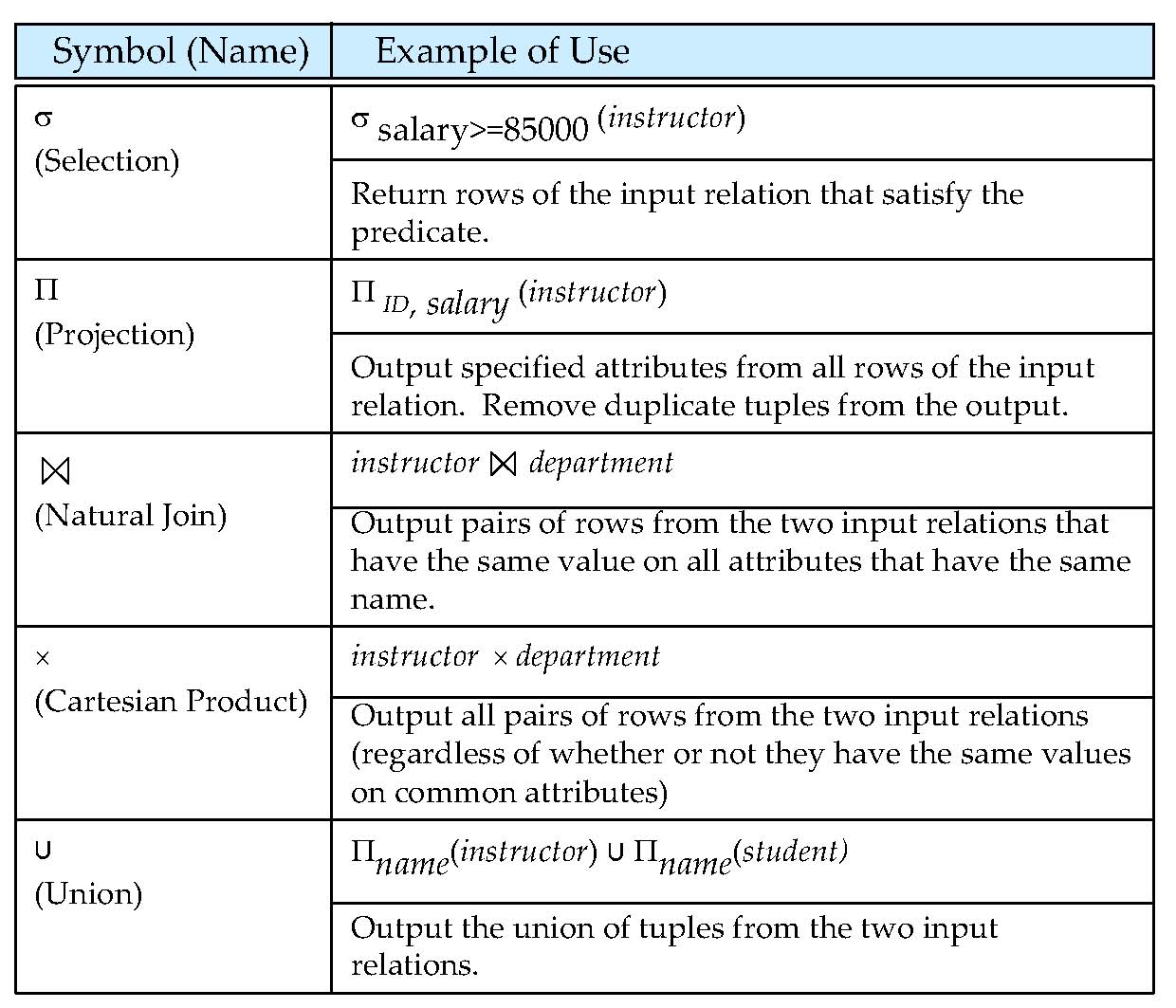
■ The set of allowed values for each attribute is called the **domain** of the attribute

■ Attribute values are (normally) required to be **atomic**; that is, indivisible

* ■ The special value ***null*** is a member of every domain
* ■ The current values (**relation instance**) of a relation are specified by a table
* ■ An element ***t*** of ***r*** is a *tuple*, represented by a *row* in a table

**Keys**

* ■ Let K ⊆ R, R set of attributes of a relation schema
* ■ *K* is a **superkey** of *R* if values for *K* are sufficient to identify a unique tuple of each possible relation *r(R)*
* ● Example: {*ID*} and {ID,name} are both superkeys of *instructor.*
* ■ Superkey *K* is a **candidate key** if *K* is minimal
* ■ One of the candidate keys is selected to be the **primary key**.
* ■ **Foreign key** constraint: Value in one relation must appear in another
* ● **Referencing** relation ● **Referenced** relation



**Outer Join**

* ■ An extension of the join operation that avoids loss of information.
* ■ Computes the join and then adds tuples form one relation that does not match tuples in the other relation to the result of the join.
* ■ Uses *null* values:
* ● *null* signifies that the value is unknown or does not exist
* ● All comparisons involving *null* are (roughly speaking) **false** by definition.

**Null Values**

* ■ The result of any arithmetic expression involving *null* is *null.*
* ■ Aggregate functions simply ignore null values (as in SQL)
* ■ Comparisons with null values return the special truth value: *unknown*
* ● If *false* was used instead of *unknown*, then *not (A < 5)*  would not be equivalent to *A >= 5*
* ■ Three-valued logic using the truth value *unknown*:
* ● OR: (*unknown* **or** *true*) = *true*, (*unknown* **or** *false*) = *unknown*,
* (*unknown* **or** *unknown*) *= unknown*
* ● AND: (*true* **and** *unknown*) *= unknown,* (*false* **and** *unknown*) *= false,*  (*unknown* **and** *unknown*) *= unknown*
* ● NOT*:* (**not** *unknown*) *= unknown*

● In SQL “*P* **is unknown**” evaluates to true if predicate *P* evaluates to *unknown*

■ Result of select predicate is treated as *false* if it evaluates to *unknown*

**create table**

**create table** *instructor* ( *ID* **char**(5),

*name* **varchar**(20) **not null,**

*dept\_name* **varchar**(20),

*salary* **numeric**(8,2),

**primary key** (*ID*),

**foreign key** *(dept\_name*) **references** *department)*

//primary key declaration on an attribute automatically ensures not null

● **insert into** *instructor* **values** (‘10211’, ’Smith’, ’Biology’, 66000);

* ● **alter table** *r* **add** *A D*
* ! where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A.*
* ! All tuples in the relation are assigned *null* as the value for the new attribute.
* ● **alter table** *r* **drop** *A*
* ! where *A* is the name of an attribute of relation *r*
* ! Dropping of attributes not supported by many databases.

**The select Clause**

* ■ To force the elimination of duplicates, insert the keyword **distinct** after select**.**

**select distinct** *dept\_name*

**from** *instructor*

■ The keyword **all** specifies that duplicates not be removed.

**select all** *dept\_name*

**from** *instructor*

■ An asterisk in the select clause denotes“all attributes”

**select** \*

**from** *instructor*

**The from Clause**

■ The **from** clause lists the relations involved in the query

● Corresponds to the Cartesian product operation of the relational algebra.

**select**\*

**from** *instructor, teaches*

● generates every possible instructor – teaches pair, with all attributes from both relations.

**The where Clause**

■ The **where** clause specifies conditions that the result must satisfy

● Corresponds to the selection predicate of the relational algebra.

To find all instructors in Comp. Sci. dept with salary > 80000

**select** *name* **from** *instructor*

**where** *dept\_name =* ‘Comp. Sci.' **and** *salary* > 80000

Comparison results can be combined using logical connectives **and, or,** and **not.**

**The Rename Operation**

■ The SQL allows renaming relations and attributes using the **as** clause:

*old-name* **as** *new-name*

* ● **select** *ID, name, salary/12* **as** *monthly\_salary*

**from** *instructor*

* ■ Find the names of all instructors who have a higher salary than some instructor in ‘Comp. Sci’.

● **select distinct** *T. name***from** *instructor* **as** *T, instructor* **as** *S*

**where** *T.salary > S.salary* **and** *S.dept\_name =* ‘*Comp. Sci.*’

**String Operations**

● percent (%). The % character matches any substring.

● underscore (\_). The \_ character matches any character.

Find the names of all instructors whose name includes the substring “dar”.

**select** *name* **from** *instructor*

**where** *name* **like '**%dar%'

Match the string “100 %”

**like** ‘100 \%' **escape '**\'

**Ordering the Display of Tuples**

* ■ List in alphabetic order the names of all instructors
* **select distinct** *name*
* **from** *instructor*
* **order by** *name*
* ■ We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
* ● Example: **order by** *name* **desc**
* ■ Can sort on multiple attributes
* ● Example: **order by** *dept\_name, name*

**Aggregate Functions and Operations Aggregation function**

**avg**: average value

**min**: minimum value

**max**: maximum value

**sum**: sum of values

**count**: number of values

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

■ What if collection has only null values?

● count returns 0  ● all other aggregates return null

■ Find the average salary of instructors in each department

● **select***dept\_name*, **avg**(*salary*) **as** avg\_salary

**from** *instructor*

**group by** *dept\_name*;

* ■ Attributes in **select** clause outside of aggregate functions must appear in **group by** list

■ predicates in the **having** clause are applied after the formation of groups, whereas predicates in the **where** clause are applied before forming groups

**select** *dept\_name*, **avg** (*salary*) **from** *instructor*

**group by** *dept\_name*

**having avg** (*salary*) > 42000;

**Test for Empty Relations**

■ The **exists** construct returns the value **true** if the argument subquery is nonempty.

●Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester

**select** *course\_id*

**from** *section* **as** *S*

**where** *semester* = ’Fall’ **and** *year*= 2009 **and**

**exists** (**select** \*  **from** *section* **as** *T*

**where** *semester* = ’Spring’ **and** *year*= 2010 **and** *S*.*course\_id*=*T*.*course\_id*);

■ The **unique** construct tests whether a subquery has any duplicate tuples in its result.

■ Find all courses that were offered at most once in 2009

**select** *T*.*course\_id*

**from** *course* **as** *T*

**where unique** (**select** *R*.*course\_id*

**from** *section* **as** *R*

**where** *T*.*course\_id*= *R*.*course\_id* **and** *R*.*year* = 2009);

**With Clause**

■ The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs.

■ Find all departments with the maximum budget

**with** *max\_budget* (*value*) **as**

(**select max**(*budget*)

**from** *department*)

**select** *budget*

**from** *department*, *max\_budget*

**where** *department*.*budget* = *max\_budget.value*;

■ Note: often easy to output max, but hard to output who has the max.

■ **Deletion**

Delete all instructors:

**delete from** *instructor*

Delete all instructors from the Finance department:

**delete from** *instructor*

**where** *dept\_name*=‘Finance’;

Delete all tuples in the *instructor* relation for those instructors associated with a department located in the Watson building.

**delete from** *instructor*

**where** *dept name* **in** (**select** *dept name*

**from** *department*

**where** *building* =‘Watson’);

**Insertion**

Add a new tuple to *course*

**insert into** *course*

**values** (’CS-437’, ’Database Systems’, ’Comp. Sci.’, 4);

or equivalently

**insert into** *course* (*title, course\_id*, *dept\_name*, *credits*)

**values** (’Database Systems’, ’CS-437‘, ’Comp. Sci.’, 4);

■ Add all instructors to the *student* relation with tot\_creds set to 0

**insert into** *student*

**select** *ID, name, dept\_name, 0*

**from** *instructor*

■ **Update**

Increase salaries of instructors whose salary is over $100,000 by 3%, and all others receive a 5% raise

●**update** *instructor*

**set** *salary* = *salary* \* 1.03

**where** *salary* > 100000;

**update** *instructor*

**set** *salary* = *salary* \* 1.05

**where** *salary* <= 100000;

**Types of Indexes**

**Bitmap Index**

A bitmap index is a special kind of indexing that stores the bulk of its data as bit arrays (bitmaps) and answers most queries by performing [bitwise logical operations](https://en.wikipedia.org/wiki/Bitwise_operation) on these bitmaps. The most commonly used indexes, such as [B+ trees](https://en.wikipedia.org/wiki/B%2B_tree), are most efficient if the values they index do not repeat or repeat a small number of times.

In contrast, the bitmap index is designed for cases where the values of a variable repeat very frequently. For example, the sex field in a customer database usually contains at most three distinct values: male, female or unknown (not recorded). For such variables, the bitmap index can have a significant performance advantage over the commonly used trees.

**Dense Index**

A dense index in [databases](https://en.wikipedia.org/wiki/Database) is a [file](https://en.wikipedia.org/wiki/Computer_file) with pairs of keys and [pointers](https://en.wikipedia.org/wiki/Pointer_(computer_programming)) for every [record](https://en.wikipedia.org/wiki/Record_(computer_science)) in the data file. Every key in this file is associated with a particular pointer to *a record* in the sorted data file. In clustered indices with duplicate keys, the dense index points *to the first record* with that key.

**Sparse Index**

A sparse index in databases is a file with pairs of keys and pointers for every [block](https://en.wikipedia.org/wiki/Block_(data_storage)) in the data file. Every key in this file is associated with a particular pointer *to the block* in the sorted data file. In clustered indices with duplicate keys, the sparse index points *to the lowest search key* in each block.

**Reverse Index**

A reverse key index reverses the key value before entering it in the index. E.g., the value 24538 becomes 83542 in the index. Reversing the key value is particularly useful for indexing data such as sequence numbers, where new key values monotonically increase.