

# Inf 523 Fundamentals of Information Technology: Databases (Fall 2011)

Jagdish S. Gangolly  
Informatics  
CCI  
SUNY Albany

September 16, 2011

- ▶ Entity-Relationship Diagram (ERD)
  - ▶ Entity
    - ▶ Weak entities
    - ▶ Strong entities
    - ▶ Dependent entities
  - ▶ Relationship
    - ▶ Degree of a relationship
    - ▶ Cardinality of a relationship
    - ▶ Recursive relationship
  - ▶ Attributes
  - ▶ Identifier
  
- ▶ ERD with crows foot notation
  
- ▶ Uniform Modeling Language (UML)

# Entity-Relationship Diagrams: Entity

## ▶ Entity

- ▶ Entity is a set that consists of a number of *instances*, each *named* by using *identifiers*. An identifier of an entity may be a single attribute or a set of attributes (composite identifier).
- ▶ Weak entities are entities that have no independent existence of their own, but exist by borrowing their identities from other entities (strong entities)
- ▶ Strong entities are entities that have an existence of their own
- ▶ If a weak entity also has its own identifying attribute(s) in addition to the identifiers of other entities, it is called an ID-Dependent entity

# Entity-Relationship Diagrams: Relationship

- ▶ Degree of a relationship is the number of entities participating in that relationship. If two entities participate in a relationship, it is called a *binary relationship*. If three entities participate, it is called a *ternary relationship* and so on. In general, if  $n$  entities participate in a relationship it is called an  *$n$ -ary relationship*.
- ▶ Cardinality of a relationship is the number of entity instances that can occur in a relationship

- ▶ Maximum and Minimum Cardinality (use of hash signs and ovals in ERDs)
- ▶ To normalize, all N:N relationships are eliminating by replacing them with an entity with 1:N relationships on either side
- ▶ Procedure for normalization with ERDs:
  - ▶ STEP 1: Draw ERD
  - ▶ STEP 2: If there are any relationships with degree greater than two, convert such relationships into binary relationships
  - ▶ STEP 3: Replace all N:N relationships by entities (you can name them by hyphenating the names of entities on either side) which have only 1:N relationships on either side, and include among the attributes of such entities the keys of the adjoining entities

# ERD with Crow's Foot Notation

- ▶ Relationships between strong entities are shown in dashed lines and are called *non-identifying* relationships since the entities do not borrow foreign keys to establish their identities
- ▶ Crows foot indicates "many", single hash mark indicates "one" (or "mandatory"), circle indicates "zero"
- ▶ The symbol closest to the entity indicates maximum cardinality, the other symbol indicates minimum cardinality
- ▶ The relationships between weak entities and the entities on which they depend for their identities, called identifying relationships, are marked by solid lines
- ▶ It is possible to change an ID dependent entity into a non-ID dependent entity by coding such weak entity to remove the ID dependence. An example is a vehicle ID number (which encodes the manufacturer as well as model year). In this case, the relationship is shown in dashed lines

# Functional Dependency

Functional dependence is a relationship between attributes in which given the value of one attribute, the value of another attribute is determined.

**An Example:** Consider the following relation:

```
EMPLOYEE( Social_Security_No, Date_of_Birth, Date_of_Hire, ... )
```

Since social security number uniquely identifies an employee, if you know an employee's social security number his/her date of birth and date of hire are already determined. In other words, social security number functionally determines the date of birth and date of hire. This can be written as:

*Social\_Security\_No* → *Date\_of\_Birth*

*Social\_Security\_No* → *Date\_of\_Hire*

# Functional Dependency

## Another Example:

**SalesOrder**(SalesOrderNumber, CustomerName, Address, ItemOrdered, QuantityOrdered)

The functional dependencies in this relation are:

*SalesOrderNumber* → *CustomerName*

*SalesOrderNumber* → *Address*

*SalesOrderNumber* → *ItemOrdered*

*SalesOrderNumber* → *QuantityOrdered*

*CustomerName* → *Address* (Assuming a customer can have only one address)

**Remark:** Here, a *non-key* attribute *address* is functionally dependent on another *non-key* attribute *CustomerName*

## Yet Another Example:

**Student**(*StudentName*, *Studentemail*, *AdvisorName*, *Advisoremail*,  
*Department*, *DepartmentAdmin*)

The functional dependencies in the above relations are:

*StudentName* → *Studentemail*

*StudentName* → *AdvisorName*

*StudentName* → *Advisoremail* (assuming a student can have only one advisor)

*StudentName* → *Department* (assuming a student can register in only one department)

*StudentName* → *DepartmentAdmin* (assuming a student can register in only one department)

*AdvisorName* → *Advisoremail*

*Department* → *DepartmentAdmin*

# Functional Dependency

- ▶ The relationships between attributes expressed in functional dependencies are exploited in relational database design to minimise duplication of data and eliminating deletion, update and insertion anomalies.
- ▶ Functional dependencies reflect the business rules that exist in organisations. Therefore, considering them in database design prevents conflicts between the database and business operations.
- ▶ Functional dependencies are also very useful in the identification of candidate keys, since to be a key, a set of attributes must functionally determine all other remaining attributes in that relation.

# Reasoning with Functional Dependencies

Armstrong Axioms: These axioms are a complete set of inference rules for reasoning about functional dependencies:

**Reflexivity** If  $\{B_1, B_2, B_3, \dots, B_m\} \subseteq \{A_1, A_2, A_3, \dots, A_n\}$ , then  $A_1A_2 \dots A_n \rightarrow B_1B_2 \dots B_m$ . These are called *trivial functional dependencies*

**Augmentation** If  $A_1A_2 \dots A_n \rightarrow B_1B_2 \dots B_m$ , then  $A_1A_2 \dots A_nC_1C_2 \dots C_k \rightarrow B_1B_2 \dots B_mC_1C_2 \dots C_k$  ie., if you append the same excess baggage on both ends of the arrow, the functional dependency still holds

**Transitivity** If  $A_1A_2 \dots A_n \rightarrow B_1B_2B_3 \dots B_m$  and  $B_1B_2B_3 \dots B_m \rightarrow C_1C_2 \dots C_k$ , then  $A_1A_2 \dots A_n \rightarrow C_1C_2 \dots C_k$ .

# Relation Keys

- ▶ **Key:** *Key of a relation is an attribute or a set of attributes that uniquely identifies a row.* Since each row in a relation pertains to an individual entity, if you know the values of the key attributes of such entity, then the values of the remaining non-key attributes for that entity are already determined. In other words, the non-key attributes in a relation are functionally dependent on the key.

## An Example:

The *social security number* can be the key of an *Employee* relation given below because no two employees can have the same social security number. The relation key is underlined.

**Employee**(SocialSecurityNo., FullName, Address, eMailAddress)

# Relation Keys

- ▶ It is possible that in a given situation, FullName of the employees can also be a key because no two employees currently share FullName. However, merely that there is a possibility that two employees can share a FullName suggests that it is not a good idea to consider it a key.
- ▶ **Candidate Keys:** A relation can have more than one key. For example, in the above Employee relation, if an employee can not have more than one email address, then eMailAddress can also be a key. In that case, FullName and eMailAddress are both called *Candidate keys*.

# Relation Keys

- ▶ **Primary Keys:** The database designer must pick one of the candidate keys to be the *Primary key*. Database Management Systems such as Oracle or DB2 maintain indexes to the tables based on such primary keys. This makes retrieval of information from databases faster.
- ▶ **Composite Keys:** It is not necessary that the keys consist of just one attribute. It may be composed of a set of attributes, in which case it is called a *Composite key*. For example, in the *GradeRoster* example below, the key is composed of *StudentIDNumber, CourseNumber*. Sometimes the only key of a relation may consist of all the attributes of the relation.

**GradeRoster**(*StudentIDNumber, CourseNumber*, *Grade*)

# Relation Keys

- ▶ **Surrogate Keys:** Sometimes the Database Management System assigns the primary key. Such keys are called *surrogate keys*. A common example is the *sales order* that companies prepare when they receive an order from a customer. DBMS assigns sales orders unique consecutive primary keys usually called *salesOrderNumber*. See the relation below:

**SalesOrder**(*SalesOrderNumber*, *CustomerName*, *Address*, *ItemOrdered*, *QuantityOrdered*)

Assigning such consecutive surrogate keys makes it possible to find out if all sales orders have been processed.

Surrogate keys are also used when the key natural to a relation is awkward to use. For example, for the property records the street address is the natural key, but a surrogate key such as PropertyID may be used.

# Relation Keys

- ▶ *Foreign Keys*: Foreign keys are non-key attributes of a relation that are primary keys of a *foreign* (ie., different) relation. For example, in the *Delivery* example below, *DeliveryNo.* is the surrogate key, and the attributes *OrderNo.*, *CustomerNo.*, *TruckNo.*, *SalesmanNo.* are keys of foreign relations *Order*, *Customer*, *Truck*, and *Salesman* respectively.

**Delivery**(*DeliveryNo.*, *OrderNo.*, *CustomerNo.*, *TruckNo.*, *SalesmanNo.*)

Foreign keys enable us to represent *relationships* in databases. They also enable us to ensure referential integrity of the database. For example, in the delivery example, referential integrity would entail not permitting in the database a delivery to a customer who does not exist in the database, delivery of a non-existent order, delivery by a non-existent salesman, or delivery by a non-existent truck.

# Use of Armstrong axioms to identify candidate keys

$$R(A, B, C, D) \quad (1)$$

with the set of functional dependencies given by

$$F = A \rightarrow B, A \rightarrow E, B \rightarrow D, C \rightarrow A, D \rightarrow C \quad (2)$$

The candidate keys for the relation R are A and C:

**A is a Candidate key:** (i)  $A \rightarrow B$ , (given); (ii)  $A \rightarrow C$ , (using transitivity twice)  $A \rightarrow B$ ,  $B \rightarrow D$ , and  $D \rightarrow C$ ; (iii)  $A \rightarrow D$ , (by transitivity):  $A \rightarrow B$ , and

$B \rightarrow D$ ; (iv)  $A \rightarrow E$ , (given)

**C is a Candidate key:** (i)  $C \rightarrow A$ ; (ii)  $C \rightarrow B$  (by transitivity)  $C \rightarrow A$ , and  $A \rightarrow B$ ; (iii)  $C \rightarrow D$  (using transitivity twice)  $C \rightarrow A$ ,  $A \rightarrow B$ , and  $B \rightarrow D$ ; (iv)  $C \rightarrow E$  (given)

## **Anomalies Caused by Poor Database Design**

When a table contains information on more than one entity certain anomalies, which make the maintenance of the database difficult, can occur. Care must be exercised in the design of the databases to avoid such anomalies.

- ▶ Insertion Anomalies
- ▶ Deletion Anomalies
- ▶ Update Anomalies

You can not enter facts about an entity unless you have information about another entity.

## **An Example:**

*You can not insert a new admitted student unless you know who her advisor is.*

Deletion of facts about one entity can delete facts about another entity.

## **An Example:**

*If an advisor has only one advisee, deleting that advisee will delete the advisor entirely from the database.*

Updating the database for a fact may require scanning the entire table.

## **An Example:**

*If an advisor's email address changes, you will need to scan the entire Student table and replace the email for each occurrence of that advisor.*

# Characteristics of a Good Database

- ▶ Insertion of a row in a table does not have any side-effects on other entities (anomaly)
- ▶ Modifications require replacing at most one row (tuple) in a table
- ▶ Insertion of a row in a table does not refer to a non-existent entity (referential integrity)

# Database Design: Normalization of Databases

Database normalization avoids unnecessary duplication of data, and leads to smaller relations. If the tables in the database are not normalized, the following anomalies can occur

- ▶ Insertion Anomaly adding new rows forces user to create duplicate data
- ▶ Deletion Anomaly deleting rows may cause a loss of data that would be needed for other future rows
- ▶ Modification Anomaly changing data in a row forces changes to other rows because of duplication

**General rule of thumb:** *A table should not pertain to more than one entity type*

# Database Anomalies: An Example

EMPLOYEE2

<u>EmpID</u>	Name	DeptName	Salary	CourseTitle	DateCompleted
100	Margaret Simpson	Marketing	48,000	SPSS	6/19/201X
100	Margaret Simpson	Marketing	48,000	Surveys	10/7/201X
140	Alan Beeton	Accounting	52,000	Tax Acc	12/8/201X
110	Chris Lucero	Info Systems	43,000	Visual Basic	1/12/201X
110	Chris Lucero	Info Systems	43,000	C++	4/22/201X
190	Lorenzo Davis	Finance	55,000		
150	Susan Martin	Marketing	42,000	SPSS	6/19/201X
150	Susan Martin	Marketing	42,000	Java	8/12/201X

Question—Is this a relation?

Answer—Yes: Unique rows and no multivalued attributes

Question—What's the primary key?

Answer—Composite: EmpID, CourseTitle

Figure: Database Anomalies

©Prentice Hall, 2011

# Database Anomalies: An Example

- ▶ **Insertion** cant enter a new employee without having the employee take a class
- ▶ **Deletion** if we remove employee 140, we lose information about the existence of a Tax Acc class
- ▶ **Modification** giving a salary increase to employee 100 forces us to update multiple records

Why do these anomalies exist?

Because there are two themes (entity types) in this one relation. This results in data duplication and an unnecessary dependency between the entities

©Prentice Hall, 2011

# Functional Dependency and Relation Key

**Functional Dependency** If the value of an attribute (or set of attributes)  $A$  determines the value of another attribute (or set of attributes)  $B$ , then we say that  $A$  *functionally determines*  $B$ , or that  $B$  is *functionally dependent* on  $A$

**Relation key** An attribute (or a set of attributes) of a relation is said to be a *relation key* if all other remaining attributes of the relation are functionally dependent on such an attribute (or set of attributes)

# Functional Dependence and Relation Key: Example 1

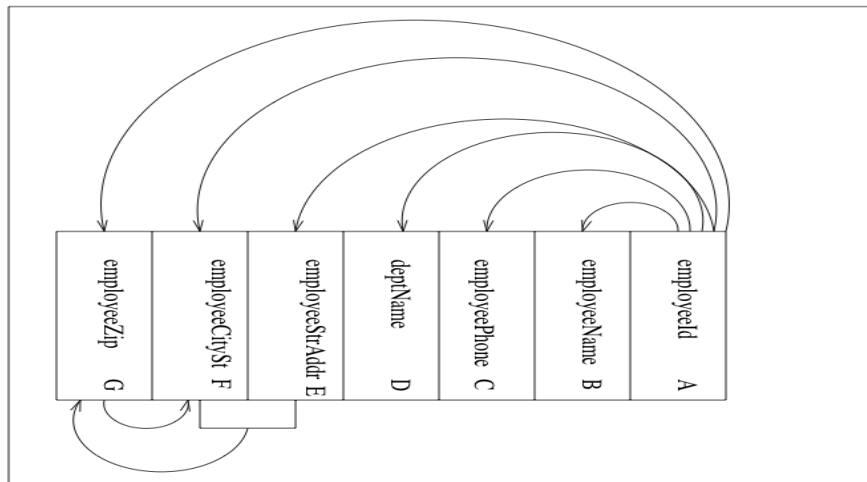


Figure: Employee

# Functional Dependence and Relation Key: An extended Invoice example

XYZ CORP.				
Customer Name Customer Address Customer Order Reference B/L Reference			Invoice Number Invoice Date Invoice Terms	
Item No.	Item Desc.	Item Quantity	Item Price	Item Amount
Invoice Total				

# Database Normalisation: An extended example

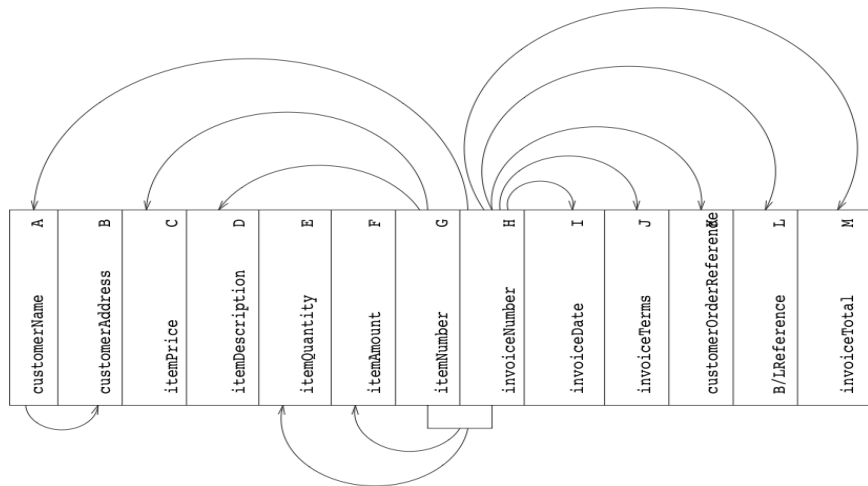


Figure: Un-normalised Database Relation

## NON-NORMALISED RELATION

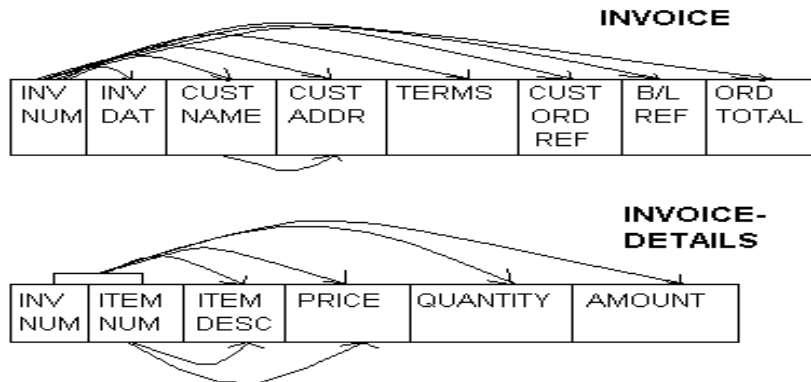
CUST NAME	CUST ADDR	INV NUM	INV DAT	TERMS	CUST ORD REF	B/L REF	ORD TOTL	ITEM NO.	ITEM DESC	PRICE	QUANT	AMOUNT

Repeated group of attributes

Figure: Un-normalised Database Relation

# Database Normalisation: An extended example

The 1NF relations are derived from non-normalized relations by splitting (decomposing) such relations so as to eliminate repeated groups. For example, we can split the above invoice relation into two relations (Invoice and Invoice Details) as done below:



## Second Normal Form

A relation is said to be in the Second Normal Form if it is in the First normal Form and all non-key attributes are functionally dependent on the WHOLE key. Now let us see if the two relations in the above figure are in 2NF.

- ▶ INVOICE RELATION: INVOICE-NUM is the relation key since all other attributes in the relation INVOICE are functionally dependent on it. Since all such non-key attributes are functionally dependent on the whole key (consisting of the single attribute INVOICE\_NUM), the relation INVOICE is in 2NF.

## Second Normal Form

- ▶ INVOICE DETAILS RELATION: Since ITEM-DESCR, PRICE, QUANTITY, and AMOUNT are all functionally dependent on the set of attributes INV-NUM, ITEM-NUM, this set is the relation key. However, this relation is NOT in the Second Normal Form since the non-key attributes ITEM\_DESCR and PRICE are functionally dependent on a part of the relation key ( ITEM\_NUM) rather than the whole relation key. We can derive the Second Normal Form relations for INVOICE\_DETAILS by splitting this relation into two by forming a separate relation with the offending attributes of ITEM\_DESCR and PRICE.

The resultant 2NF relations for INVOICE\_DETAILS are given in the figure on the next slide

# Database Normalisation: An extended example

Second Normal Form Decomposition of INVOICE-DETAILS Relation

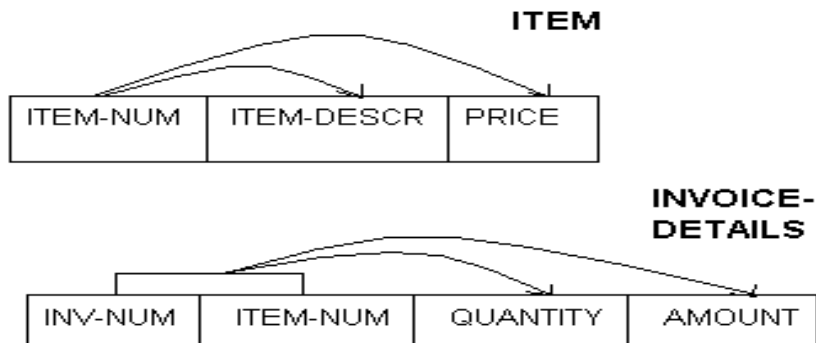


Figure: Second Normal Form Decomposition of INVOICE-DETAILS Relation

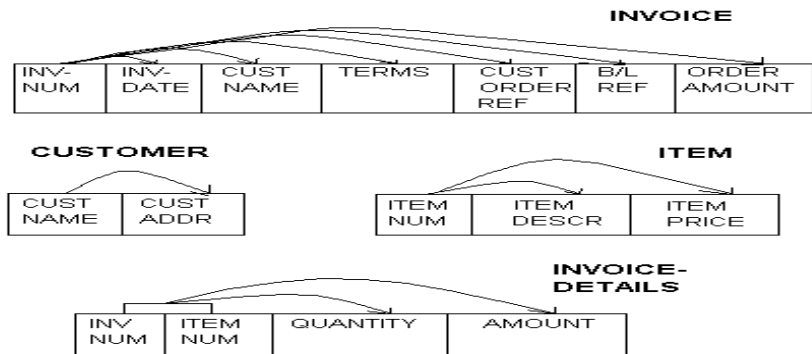
## Database Normalisation: An extended example

A relation is said to be in the Third Normal Form if it is in the Second Normal Form, and in addition each non-key attribute is functionally dependent on the whole key and nothing but the key.

The INVOICE relation above is in the Second Normal Form, but is NOT in the Third Normal Form since the attribute CUST\_ADDR is functionally dependent on the non-key attribute CUST\_NAME. To convert it to the Third Normal Form, we again split the INVOICE relation into INVOICE and CUSTOMER relations by removing the offending attributes from INVOICE and creating the CUSTOMER relation. The final 3NF relations for the INVOICE example are on the next slide

# Database Normalisation: Third Normal Form Relations

## Second Normal Form Decomposition of INVOICE-DETAILS Relation



### Third Normal Form Relations for the Invoice Example

Figure: Third Normal Form Decomposition of Invoice Relation

# Structured Query Language

- ▶ Not a complete programming language
- ▶ It is a data sub-language
- ▶ Much of the Data Manipulation Language (DML) part of SQL is a *declarative language*, ie., you *instruct* the computer *what* you want, and not *how* to find it
- ▶ To build complete systems, you must embed SQL code in an *imperative* language such as *Java*, *C*, or *C++*. Imperative languages *instruct* the computer *how* to find answers
- ▶ SQL standards
  - ▶ ANSI SQL 1986, 1989, 1992 (SQL-92)
  - ▶ 1999 ANSI SQL-3 (added object-oriented features). 2003, 2006
  - ▶ 2008 SQL:2008 (added support for XML)
  - ▶ NOT ALL STANDARD SQL SYNTAX WORKS IN MS-ACCESS. Access also defaults to SQL-89, but you can change the options to change the default to SQL-1992 (SQL Server Compatible Syntax)

# Structured Query Language

- ▶ QBE (Query By Example), a graphical interface for querying, based on logic (relational calculus)
- ▶ SQL (Structured Query Language), a command line and text oriented query language (based on relational algebra)
- ▶ A good idea to know both, though as a beginner it is a good idea to be able to use SQL
- ▶ SQL consists of two sub-languages
  - ▶ Data Definition Language (DDL): used to create database tables, specify keys, data types for attributes, etc.
  - ▶ Data Manipulation Language (DML): used to populate tables (insert, delete, update)

# Data Definition Language

- ▶ **CREATE TABLE:**

```
CREATE TABLE TableName (  
  ColumnName DataType optionalConstraint,  
  ColumnName DataType optionalConstraint,  
  ... .. ..., ) ;
```

- ▶ **optional constraints:** PRIMARY KEY, NOT NULL, NULL, and UNIQUE which is not supported by MS-Access SQL-89)
- ▶ **Allowable data types:** AutoNumber, Text (max length: 256 characters), Memo (up to 2GB data and RTF support), Number (up to 16 bytes of data), Date/Time (including Auto calendar feature), Currency (with precision to four decimal places), Yes/No (Boolean data values), OLE object (up to 2GB, to store graphs, documents, etc.), Hyperlink (max size limit: 1GB), Attachment ( used to store images, spreadsheet files, documents, charts and other types of supported files. New feature in Access 2007) *See Figure 3-5 in the textbook.*

# Data Definition Language

- ▶ For details of all data types supported see  
[http://msdn.microsoft.com/en-us/library/ms714540\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms714540(VS.85).aspx)  
or  
[http://www.databasedev.co.uk/access2007\\_data\\_types.html#auto\\_calendar](http://www.databasedev.co.uk/access2007_data_types.html#auto_calendar)

## An Example:

```
CREATE TABLE EMPLOYEE (  
    EmployeeNumber    Int                NOT NULL IDENTITY (1, 1),  
    FirstName         Char(25)           NOT NULL,  
    LastName          Char(25)           NOT NULL,  
    Department        Char(35)           NOT NULL DEFAULT 'Human Resources',  
    Phone             Char(12)           NULL,  
    Email             VarChar(100)       NOT NULL UNIQUE,  
    CONSTRAINT        EMPLOYEE_PK        PRIMARY KEY (EmployeeNumber),  
    CONSTRAINT        EMP_DEPART_FK      FOREIGN KEY (Department),  
    REFERENCES DEPARTMENT (DepartmentName),  
    ON UPDATE CASCADE  
);
```

## Another Example:

```
CREATE TABLE ASSIGNMENT (  
    ProjectID          Int          NOT NULL,  
    EmployeeNumber    Int          NOT NULL,  
    HoursWorked       Numeric (6, 2) NULL,  
    CONSTRAINT        ASSIGNMENT_PK PRIMARY KEY (ProjectID, EmployeeNumber),  
    CONSTRAINT        ASSIGN_PROJ_FK FOREIGN KEY (ProjectID)  
                        REFERENCES PROJECT (ProjectID)  
                        ON UPDATE NO ACTION  
                        ON DELETE CASCADE,  
    CONSTRAINT        ASSIGN_EMP_FK FOREIGN KEY (EmployeeNumber)  
                        REFERENCES EMPLOYEE (EmployeeNumber)  
                        ON UPDATE NO ACTION  
                        ON DELETE NO ACTION  
);
```

# Structured Query Language

- ▶ Business rules are implemented in DDL part of SQL as CONSTRAINTs
- ▶ Most constraints deal with the treatment of UPDATEs and DELETEs
- ▶ Surrogate keys are implemented in SQLServer through specification of IDENTITY (m,n) where m is the starting sequence number assigned to the first row, and n is the increment.
- ▶ Implementation of surrogate keys varies from vendor to vendor

# Data Manipulation Language (DML)

- ▶ SQL provides SELECT, INSERT, and DELETE statements
- ▶ SELECT statement is the work horse of DML and is the fundamental statement for database querying
- ▶ INSERT and DELETE statements are available for updating the database

# Data Manipulation Language: INSERT

- ▶ Syntax:  
INSERT INTO *<TableName>*  
(VALUES *<Tuple to be inserted>* );
- ▶ Integer and numeric values must NOT be enclosed in single quotes
- ▶ Char, Varchar, and DateTime values must be enclosed in single quotes
- ▶ SQL is fussy about single quotes. Directional quotes will produce errors

# Data Manipulation Language: INSERT

If some column data is missing:

- ▶ Syntax:

```
INSERT INTO <TableName>  
(<List of field names to be inserted>)  
(VALUES <Tuple to be inserted> );
```

- ▶ Order of the field (column) names must match the order of values, but the order of the field (column) names need not match the order in the table
- ▶ values must be provided for all fields (columns) that can not have NULL values
- ▶ NULL values will be inserted for missing field names

# Data Manipulation Language: SELECT

- ▶ We will proceed from very simple to complex queries. Our objective is to master the syntax
- ▶ A Simple Query:  
SELECT *⟨Comma separated Column names⟩*  
FROM *⟨Table name⟩*  
WHERE *⟨Condition⟩*;
- ▶ SELECT tells us which column data must be returned by the query
- ▶ FROM specifies the table to be searched for the data
- ▶ WHERE specifies the conditions to be met if the data item is to be returned by the query

# Data Manipulation Language: SELECT

- ▶ You can select every column from a table by specifying \* to be the *Comma separated Column names*
- ▶ The results of every SQL query is a *Table*
- ▶ However, the result is *NOT* a relation since it can contain duplicate rows
- ▶ You can eliminate such duplicate rows by using the keyword **DUPLICATE**
- ▶ An example  

```
SELECT DISTINCT Department  
FROM Project  
WHERE MaxHours > 135;
```

# Data Manipulation Language: SELECT WHERE

- ▶ You can connect any number of conditions by using keywords AND and OR. For example,

```
WHERE Department = 'Accounting' OR Phone =  
360-285-8410';
```

- ▶ You can You can specify that the column value should belong to a set of values. For example,

```
WHERE Department IN ('Accounting', 'Finance, Marketing');  
or
```

```
WHERE Department NOT IN ('Accounting', 'Finance,  
Marketing');
```

# Data Manipulation Language: SELECT WHERE

- ▶ You can specify ranges by using comparison operators such as =, >, <, >=, <=, <> and the keyword BETWEEN. For example,

```
WHERE EmployeeNumber BETWEEN 2 AND 5;
```

which is equivalent to

```
WHERE EmployeeNumber >= 2 AND EmployeeNumber <= 5;
```

- ▶ You can search by pattern by using wildcard characters \_ and %. \_ stands for one unspecified character and % stands for one or more unspecified characters as in the following

```
WHERE Phone LIKE '360-237-5_67'; or
```

```
WHERE Phone LIKE '360-237-5%';
```

# Data Manipulation Language: SELECT WHERE

- ▶ You can search for NULL values by using the keyword IS NULL as in

```
WHERE Phone IS NULL
```