**Database Fundamentals**

Information is what you get from a database. Data is what you put into a database (facts).

**Design 1 - Introduction to Database Technology**

Databases are everywhere and are important to many organizations. Data is used to maintain a business and for many businesses the data is their business. It does not matter what kind of an application you are building. Database skills are important and skills you learn in DMIT1508 could be used for year or decades from now. Fundamental database concepts have been tested for decades and are very solid.

**What are Databases?**

Databases are an organized collection of data. There are many kinds of databases. We will be using a type of database called a relational database. A relational database is where like information is stored in different tables that all relate to each other in some way. We will look more at what a relational database is and how it is designed later on.

**Why Databases?**

Databases help us to solve problems and work with data effectively and efficiently. In most businesses, there is likely data that needs to be stored. This could be data about customers, products, orders, employees, etc.  The data could be simply text, numeric, or more complex data like images.

Much of the data that we would need to store could be stored in a text file or a spreadsheet. Many business and organizations use spreadsheets without any trouble for their data. For small amounts of similar data this may work fine. The

problem some organizations encounter is that working with the data can be problematic. Consider these problems that can exists with storing data in other ways than a database.

**Size:** A little data tends to grow into a lot more

**Ease of Updating:** What if many people need to work with the data at the same time?

**Accuracy:** Is there something to prevent incorrect data being entered? Not usually in a spreadsheet.

**Security:** We often need to share data with others. Who gets to view certain data and what mechanisms are in place to ensure only authorized individuals or groups view certain data?

**Redundancy:** Is the data stored in only one location?  Having data stored in many different places leads to problems maintaining the data.

**Importance:** Data is important. It can be the entire business. It must be maintained properly.

**Summary**

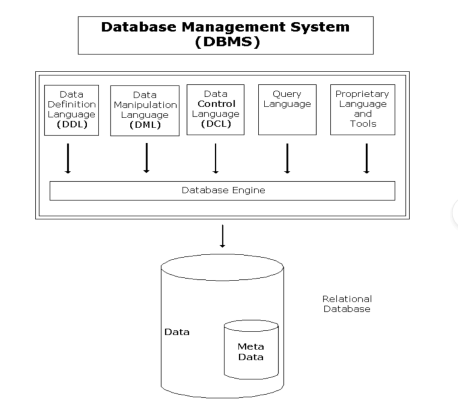
Itʼs not just about placing data somewhere. Itʼs about allowing it to grow and be maintainable. The data needs to be secure, accurate, and only available to those who should be able to access it.

Much of what is involved with a database is created behind the scenes. Users of the database only see how well it works. All the good stuff is invisible!

**Databases and DBMS**

Databases are what store the data. To effectively maintain the data we use a Database Management System (DBMS).  SQL Server, Oracle, Access, MySQL are examples of a DBMS. You would use the DBMS to manage your databases. The database is the data and the rules. The DBMS allows you to interact with your db.

Some organizations will use many different DMBSʼs. Each DBMS has its own particular strength.  Some of the most common relational DBMSʼs include SQL Server, Oracle, MySQL, and MS Access.

The Data Definition Language (DDL) facilitates the creation and manipulation of the meta data. It allows the developer to define the structure of the database (e.g. column definitions, constraints on the data).

The Data Manipulation Language (DML) facilitates the creation and manipulation of records stored in the database (the data).

The Data Control Language (DCL) facilitates the security of the data. It allows access to the data stored in the database to be limited to authorized users.

The Query Language facilitates retrieval of data from the database.

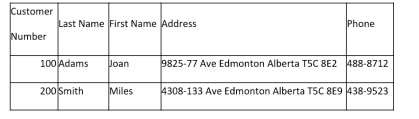
The proprietary language is a programming language used by developers to create programs that make up a system. For example, the Access DBMS provides Visual Basic for Applications (VBA) as the programming language that is used to create programs.

Most DBMS also provide other tools to perform development tasks. For example, a graphical user interface (GUI) used to define the meta data for a database.

The database engine is the core set of programs that make up a DBMS. These programs execute when a request is submitted to the DBMS and directly interact with the database itself to provide a service. For example, a user adds new data to the database. The user would execute an application program to perform this task. The application program would submit a DML statement to the DBMS requesting that new data be added to the database. The DML statement causes one or more programs in the database engine to execute and actually add the new data to the database.  It is important to note that developers and users do not interact directly with the database; instead they submit requests to the DBMS. The DBMS responds to requests from the various stakeholders in the system (e.g. users or developer) and performs the various tasks involved in creating, changing, and using a database.

**The Relational DBMS**

This course assumes that a relational DBMS is used. A relational DBMS uses tables to store data in the database. A table consists of rows and columns (similar to a spreadsheet). For example, information about customers would be stored in a table named Customer in the database as shown below:

Each column in the table represents one piece of information about a customer (e.g. the Phone column records the customers' phone number). Each row in the table represents a collection of information for one specific customer.

Data stored in a relational database is kept in a series of tables within the database. The tables are then related to each other based on the business rules of the organization. Some things to consider when designing relational database include:

How many tables are required to store the data in the database.●

Which columns (pieces of information) belong in each table within the

●

database.

How do the tables relate to each other (what are the business rules) ●

The topic of relational database design is discussed further in later lessons. There are many relational DBMS currently on the market. Microsoft SQL Server and Oracle are two of the more popular relational DBMS.

**The Database Design Process**

The database design process is a methodology used to design the structure of a database for an application. Prior to starting the database design process the designer must understand:

The purpose of the database (i.e. Why does one need the database?). ●

The data that will be stored in the database.

●

What functions the database will support (e.g. add a customer record). ●

The database designer develops this understanding by completing systems analysis and design tasks such as:

Talking to the users to identify requirements.

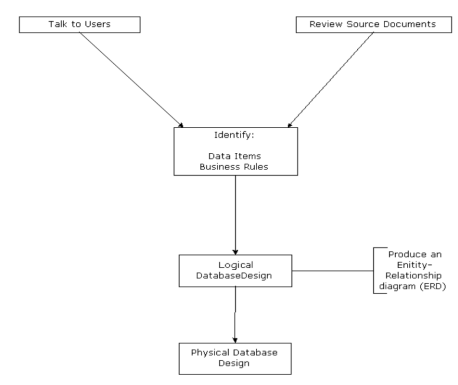
●

Reviewing source documents (e.g. reports, screen layouts, paper-based

●

business forms).

**The Database Design Process** The database design process is depicted below:

The first step is to talk to the users and to review source documents.  This allows the database designer to identify:

The data items that must be stored in the database.

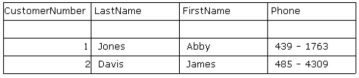
●

The business rules the application must adhere to/enforce. For example, for an

●

application that manages the issuing of driverʼs licenses, an applicant must be sixteen years of age in order to acquire a driverʼs license.  A good database design will account for the business rules enabling the DBMS software to enforce many of these rules.

The next step is to create a logical database design.  A logical database design represents the programmer's and the user's **perception** of the corporate data stored in the database. For example, when using a relational database management system, data is organized into a collection of tables (rows and columns) regardless of the actual arrangement of the data on the physical storage device (e.g. a hard-drive).  A Customer table is used to store data about customers as depicted below.

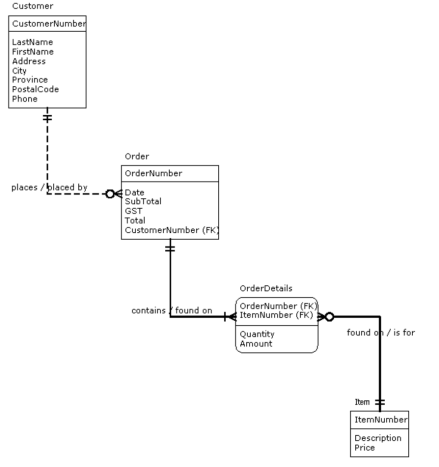


Each row in the table represents one specific customer.  Each column in the table represents one attribute or characteristic of a customer.

The **entity-relationship model** (a data model) and the **normalization process** are tools used to complete the logical design. The logical design shows the data the application requires and how the pieces of data inter-relate. It describes the business scenario in terms of **entities** and **relationships**. An entity represents a real world thing such as a customer or a course. A relationship represents a dependency between entities based on the business rules of the organization.  Consider an application to aid processing customer orders for merchandise. The following business rule may apply: many customers can place orders for merchandise but a specific order is placed by one customer.  This results in a one-to-many relationship between the customer entity and the order entity.

The logical design is depicted graphically as an entity-relationship diagram (ERD). Many notations (sets of symbols) have been developed to depict an ERD (e.g. Chen, Martin, IDEF1X). The IDEF1X notation is used for all ERDs in this course. IDEF1X was adopted as a standard for data modeling by the National Institute of Standards and Technology (NIST) in 1993.

A sample ERD for a database that records data about customers, the orders customers place and the items for sale is shown below.



The ERD shows four entities: Customer, Order, OrderDetails and Item, all represented by boxes. The lines connecting the entities represent relationships between the entities. Relationships reflect the business rules and procedures in place within an organization. When using a relational database management system each entity becomes a table in the database.

Normalization is a set of guidelines for creating the logical design for a relational database. Normalization helps database designers decide:

The number of entities/tables required within the database. ●

Which attributes/columns belong in which entities/tables.●

●

The normalization process helps minimize redundant data within the database and helps ensure the data remains consistent. An example of inconsistent data would be having two tables in a database that record customer information, and having conflicting data (e.g. phone number), for the same customer, stored in the two tables. Using the normalization process to fine tune the design produces database designs that minimize the occurrence of inconsistencies within the data.  The logical design process will be expanded in later lessons.

When the logical design is complete a **physical design** for the database is created. The physical design describes the physical structure of the data. The physical design describes how the logical design will be implemented. The physical design addresses issues that improve the performance: database optimization and tuning. Introductory topics related to physical design will be discussed in later lessons. Database optimization and tuning are beyond the scope of this course. Physical design is not discussed in detail within this course.  Physical design is included in the design process presented here for the sake of completeness.

Application development is an iterative process.  One task within the application can be in the testing phase while another is in the analysis and design phase. This iterative approach necessitates an iterative approach to database design. The database design process is best applied on a task-by- task basis. Another term is view.  A view is the data used to complete one task within the application (e.g. add a new customer). Applying the database design process to a task/view produces a database design to accommodate the task/view. The database is created and programs are written and tested. While program development proceeds with one task/view the database design process can be applied to another task/view. The design produced is merged with the database design produced from previous views. Gradually the database design is refined as more and more views are completed.

——————————————————————————————————————— ——————————————————————

**Theory 3 Entity Relationship Model**

Attributes describe entities.

Entities are attributes you can describe.

Composite attributes need to be broken down into atomic attributes.

Primary keys are only one. A foreign key is one to many; a foreign key must be attached to a primary key; a foreign key starts out at 0.

A foreign key is a pointer to your primary key basically.

**Entity-Relationship Model**

The logical database design process uses the entity-relationship model to represent the business scenario as a collection of entities and relationships. The entity-relationship diagram (ERD) documents the logical design and becomes a starting point for the physical design process, which defines the structure of the database. This lesson will discuss concepts related to the entity-relationship model.

The **Entity-Relationship** model is a tool that organizes and documents the logical design of a database. The entity-relationship model describes the data used by an application in terms of: **entities, attributes and relationships**.

**Entity**

An entity is a person, place, thing or concept that is used in the business context of the application and for which data is collected. Synonyms for entity include "entity class" and "entity type". Most people use the term entity.

Consider an application that maintains customer orders for merchandise purchased from a hardware store. A sample order is shown below.



The entities involved in this information include: Order, Customer and Item.

Entities can be physical things that you can touch (e.g. a customer) or conceptual things that do not physically exist (e.g. a course). It is important to distinguish between an **entity** and an **instance of an entity**. An instance of an entity is one specific occurrence of an entity. In the order example Customer is an entity. Joan Adams is an instance of the customer entity. An entity can have many instances (hopefully the hardware business is booming and there are many customers!).

**Attributes**

A piece of data that describes an entity is called an **attribute**. Synonyms for attribute include: element, property and field. In the order example, the customer entity has the following attributes:

● ●

Customer Number Last Name

● ● ● ●

First Name

Address

Phone Number

The values for all attributes describing an entity make up one instance of the entity. For example, the following values describe one specific customer.



**Types of Attributes**

There are two types of attributes: **atomic** and **composite**. An atomic attribute has a value that is in its simplest form. Atomic attributes cannot be subdivided into two or more other meaningful attributes. "Customer Number" is an example of an atomic attribute. Generally speaking, storing data in atomic form adds flexibility to your application.  A composite attribute is an attribute that can be decomposed into two or more atomic attributes. "Address" is an example of a composite attribute. Address can be broken down to street address, city, province, and postal code; all of which are atomic attributes.

Business rules play a role in deciding how to store data (atomic vs. composite). Consider the "Address" attribute. For a national company all addresses have the same format. Therefore, address is stored in its atomic form. For an international company addresses vary in format (Americans use a five digit zip code while Canadians use a six digit postal code). In this situation address would be stored in its composite form as a single attribute.

Attributes can be classified as **stored** or **derived**. A stored attribute has its value physically stored in the database. Order Number is an example of a stored attribute. A derived attribute can have its value calculated based on other data. Amount is an example of a derived attribute (amount can be calculated as Quantity \* Price). Derived attributes need not be physically stored in the database. During the implementation phase of application development a decision is made whether derived attributes should be stored in the database or not. If the decision is to physically store them, the database requires more space on the storage media. If the decision is not to physically store derived attributes, you save storage space, but you increase the resources used to execute the application. The application programs or the DBMS must now execute the instruction(s) to calculate the value

of the derived attribute each time it is used. Either method can be used in a fully functional application.

**Values an Attribute Can Assume**

An attribute has a set of legitimate values. **These values must "make sense" for the business situation**. Using the customer order example, the value of the Customer Number attribute must be a number greater than zero. An attribute's legitimate set of values is defined in terms of the **data type** of the attribute and the **domain** of the attribute. The domain is the set of values that an attribute can assume. The set of legitimate values for the Customer Number attribute is defined as a data type of integer and a value that is greater than zero. **Special Case: The Null Value**

Null is a special value that an attribute can assume. There are two fundamental ways to interpret the null value:

p.

The value is unknown.

Consider creating an instance of the customer entity. If the customer does not know their postal code the null value can be stored as the postal code for the new record. The customer can supply their postal code at a later date. The record can then be updated to reflect the correct postal code. q.

This attribute does not apply for this particular instance.

Consider an employee entity. One attribute could be the name of the employee's spouse. The null value would be recorded in the spousal name attribute for all records that represent a single employee.

The null value must be used with caution, as it propagates through expressions. This means that any expression using an attribute that contains the null value will result in null.

**Primary Key**

Many tasks within an application involve using one specific instance of an entity. For example, changing the price of a specific item in inventory. To accomplish this, the application must have a way of uniquely identifying each instance of the item entity. The **primary key** allows the DBMS to uniquely identify each instance of an entity. The primary key is an attribute, or group of attributes, that has a unique value for each instance of an entity. In the Customer Order example, the Item Number attribute uniquely identifies each item in inventory and can act as the primary key for the item entity.

Sometimes an entity does not have a naturally occurring attribute that can act as the primary key. In this case, the database designer arbitrarily makes up an attribute solely for the purpose of acting as the primary key. This is called a **technical key**. Customer Number is an example of a technical key. Customers do

not have numbers when they walk in the door to the business. The Customer Number attribute exists so the DBMS can identify a specific instance of the customer entity.

In some cases, a combination of two or more attributes acts as the primary key for an entity. This is termed a **concatenated key** or a **composite key**.

**The bottom line is that each entity must have a primary key that enables the DBMS to identify each instance of the entity.**

**Relationships**

A relationship represents a business association between two entities. Relationships are based on the business rules of the organization. All relationships are bi-directional in that they can be interpreted from the point of view of each entity. In the customer order example a relationship exists between the customer entity and the order entity. This reflects the fact that each time a customer makes a purchase an order is created. From the customer perspective, one customer can place many orders. From the order perspective, one and only one customer places an order.

**Types of Relationships**

Relationships are categorized based on how the instances of the entities are related to each other. The relating of instances between entities is called **cardinality**. The three major types of cardinality are:

●

One-to-One

Given an application that records parking stall assignments for employees,

○

the business rule states: **Each employee is assigned one, and only one parking stall**. Therefore, one instance of the employee entity can be related to one instance of the parking stall entity and one instance of the parking stall entity can be related to one instance of the employee entity.

●

One-to-Many

Using the customer order example, the business rule states: **one**

○

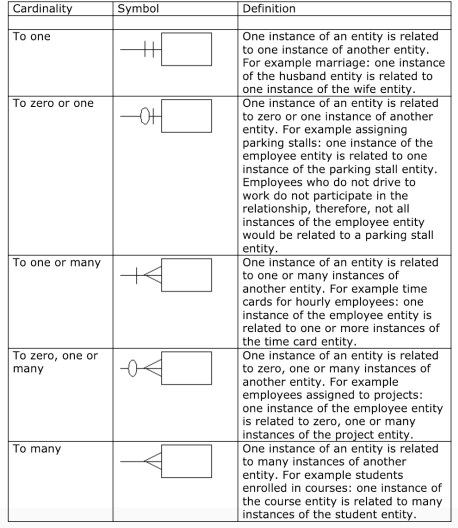
**customer can place many orders but one, and only one customer places an order**. Therefore, one instance of the customer entity can be related to many instances of the order entity and one instance of the order entity can be related to one and only one instance of the customer entity.

●

Many-to-Many

◦ Using the customer order example, the business rule states: **many items can be purchased on one specific order and one specific item can appear on many different orders**. Therefore, one instance of the order entity can be related to many instances of the item entity and one instance of the item entity can be related to many instances of the order entity.

The **cardinality** defines the number of instances of an entity that can be related to a single instance of another entity. Common cardinalities include:



**Defining Relationships**

A **foreign key** is the mechanism that defines a relationship between entities. **A foreign key is a primary key of one entity that appears as an attribute of another entity**. In each relationship one entity has a foreign key and the other entity has the related primary key. The entity with the foreign key is referred to as the **child** entity in the relationship. The entity with the primary key is referred to as the **parent** entity in the relationship. The instance in the child entity is said to reference or refer to the related instance in the parent entity. Foreign keys are used to define all relationships between entities. **The foreign key and related primary key must be the same data, however the two attributes do not need to have the same name.** The name of the foreign key attribute should be descriptive and "make sense" with the business scenario.  In some cases, the foreign key attribute will have the same name as the related primary key attribute in the parent entity; however, this is not always the case.

**The One-To-One Relationship**

In the one-to-one relationship one instance from an entity is related to at most one instance of another entity.

Consider the following scenario. A company consists of multiple departments (e.g. Research, Marketing etc). Company policy states that every department has a manager. A manager can manage one and only one department within the company. This business rule translates to a one-to-one relationship between the employee entity and the department entity. This relationship can be defined by:

Adding the DepartmentID attribute (primary key of the Department entity) as a

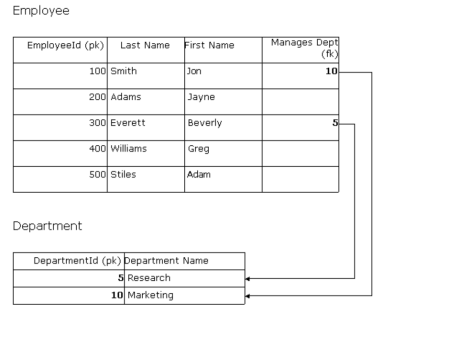
p.

foreign key, named "ManagesDept", to the Employee entity.

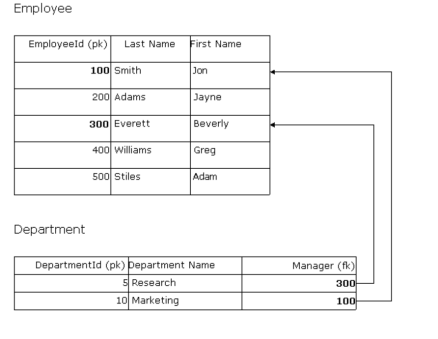
2. Adding the EmployeeID attribute (primary key of the Employee entity) as a

foreign key, named "Manager", to the Department entity.

Either method will work. Consider the first method, adding the DepartmentId to the Employee entity as a foreign key named Manages Dept.

This is not a good choice as many employees are not managers. This results in many instances in the Employee entity with the null value stored as the value for the Manages Dept attribute.

A better approach is to add the EmployeeId attribute to the Department entity as a foreign key named Manager.

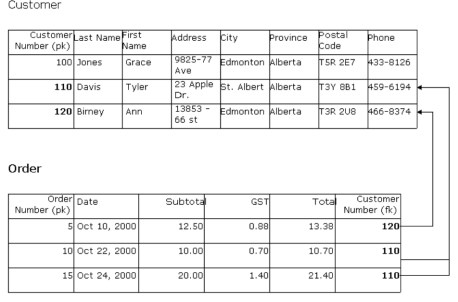
Every department has a manager so an attribute that holds the null value for many instances is not introduced.   In this implementation of the relationship the Employee entity acts as the parent while the Department entity acts as the child.

**The One-To-Many Relationship**

In the one-to-many relationship one instance from an entity is related to zero, one or many instances of another entity. Consider the following scenario. A retail sales company sells merchandise to customers. Each time a customer purchases goods an order is created. Over time a single customer can place many orders. Each

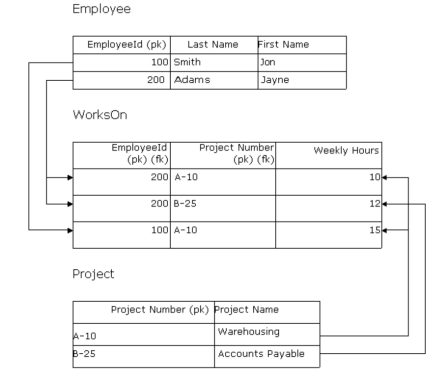
individual order is made out to one customer. A customer must register with the company prior to making their first purchase. These business rules translate to a one-to-many relationship between the customer entity and the order entity (one customer can have many orders and a specific order is placed by one customer).

The one-to-many relationship is defined by adding the primary key of the "one entity", as a foreign key, to the "many entity". The one entity (customer) is considered the parent entity and the many entity (order) is considered the child entity.

**The Many-To-Many Relationship**

The relational database does not facilitate a many-to-many relationship between two entities. Hence, many-to-many relationships need to be reduced to one-to many relationships. Creating a third entity referred to as an **associative entity** does this. The associative entity contains the primary key attributes of the two entities that share the many-to-many relationship as well as any other attributes that describe the association between the entities. Two one-to-many relationships are defined between the associative entity and the entities that share the many to-many relationship.

Consider a consulting company. The company develops many projects simultaneously. Multiple employees make up a project team. One employee can be assigned to multiple projects simultaneously. These business rules translate to a many-to-many relationship between the Project entity and the Employee entity.  The associative entity named **WorksOn** defines this relationship.

**The WorksOn entity contains the following attributes:**

EmployeeId (primary key of the Employee entity).

●

Project Number (primary key of the Project entity).

●

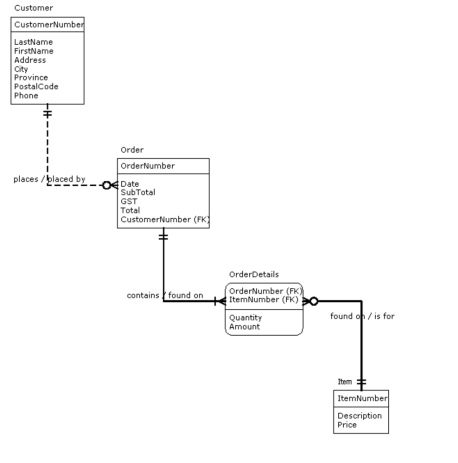
WeeklyHours (the hours per week the employee works on the project). ●

The primary key of the associative entity is always made up of two or more attributes. The primary keys from the entities that share the many-to-many relationship form the primary key for the associative entity. In this example, EmployeeId and ProjectNumber make up the primary key for the WorksOn entity.

A one-to-many relationship is defined between the Employee entity (parent) and the WorksOn entity (child). A one-to-many relationship is defined between the Project entity (parent) and the WorksOn entity (child).   Note that a single attribute (e.g. EmployeeId in the WorksOn entity) can act as both a primary key and a foreign key within the same entity.

**The Entity-Relationship Diagram (ERD)**

The IT industry has developed several notations that can be used to construct an ERD (e.g. Chen, Martin, IDEF1X). Each notation has its own set of symbols for representing entities, attributes, keys and relationships. This course will use the IDEF1X notation. An ERD for the Customer Order application is shown below.

A box represents an entity. If the box has square corners it is a base entity. If the box has rounded corners it is an associative entity. The name of the entity appears above the box. All attributes describing the entity are listed inside the box. Primary key attributes are listed first, and appear above the line. Attributes that do not act as the primary key for the entity appear below the line. Any attribute that acts as a foreign key are suffixed with "FK".

A line connecting two boxes represents a relationship between the two entities. The line has cardinality symbols that denote the type of relationship. A dashed line represents a **non-identifying** relationship. This means that the foreign key, in the child entity, does not act as part of the primary key for the entity. The relationship between the Customer entity and the Order entity is an example of a non identifying relationship. A solid line represents an **identifying** relationship. This means that the foreign key does act as part of the primary key in the child entity. The relationship between the Order entity and the OrderDetails entity is an example of an identifying relationship.  Relationships are labeled with a **verb phrase**. This allows the relationship to be read much like a sentence. The entities are the nouns and the relationship label is the verb. Relationships can be read in two directions:

From parent to child. ●

From child to parent. ●

For the relationship between the Customer entity and the Order entity, Customer is the parent and Order is the child. Reading from parent to child: Customer places Order. Reading from child to parent: Order placed by Customer.

——————————————————————————————————————— ——————————————————————

**Review:**

p.

ERD

p.

Entity Relationship Diagram

q.

Entity

An entity is a person, place, thing or concept that is used in the business

p.

context of the application and for which data is collected. Synonyms for entity include "entity class" and "entity type". Most people use the term entity.

s.

Attribute

A piece of data that describes an entity is called an **attribute**. Synonyms

p.

for attribute include: element, property and field.

t.

Atomic Attribute

An atomic attribute has a value that is in its simplest form. Atomic

p.

attributes cannot be subdivided into two or more other meaningful attributes.

u.

Composite Attribute

A composite attribute is an attribute that can be decomposed into two or

p.

more atomic attributes.

v.

Stored Attribute

p.

A stored attribute has its value physically stored in the database.

w.

Derived Attribute

p.

A derived attribute can have its value calculated based on other data.

x.

Primary Key

p.

The **primary key** allows the DBMS to uniquely identify each instance of an entity. The primary key is an attribute, or group of attributes, that has a unique value for each instance of an entity.

y.

Technical Key

p.

Sometimes an entity does not have a naturally occurring attribute that can act as the primary key. In this case, the database designer arbitrarily makes up an attribute solely for the purpose of acting as the primary key. This is called a **technical key**.

pz. pp.

Natural Key Foreign Key

p.

A **foreign key** is the mechanism that defines a relationship between entities. **A foreign key is a primary key of one entity that appears as an attribute of another entity**. In each relationship one entity has a foreign key and the other entity has the related primary key. The entity with the foreign key is referred to as the **child** entity in the relationship. The entity with the primary key is referred to as the **parent** entity in the relationship.

pq.

Composite / Concatenated Key

p.

In some cases, a combination of two or more attributes acts as the primary key for an entity. This is termed a **concatenated key** or a **composite key**.

ps.

Relationships

p. q. s.

One to One One to Many Many to Many

pt.

Cardinality

p.

Relationships are categorized based on how the instances of the entities are related to each other. The relating of instances between entities is called **cardinality**.

q.

The **cardinality** defines the number of instances of an entity that can be related to a single instance of another entity. Common cardinalities include:

p. q. s. t. u.

To One

To Zero or One

To One or Many

To Zero, One, or Many To Many

pu.

Ordinality

——————————————————————————————————————— ——————————————————————

**Normalization: What and Why**

When using a relational database management system the database is a collection of tables. A critical step in the database design process is deciding:

How many tables do we need to store the data for an application?

●

What data belongs in which table?

●

These questions have plagued database designers for many years. Initially we used intuition as our guide. We tended to group similar data items together in a table because it seemed to make sense to do so; for example all the data items describing an employee would be put in the employee table. Some designers seemed to have good intuition; they produced database designs that worked well resulting in applications that users were satisfied with. Other designers didnʼt have good intuition resulting in poor database designs and applications that users were not satisfied with.

The problem was there was no systematic method to help database designers decide how many tables there should be and what data items should be in which tables.

Codd developed a set of rules he called the rules of normalization, which we can apply when designing a RELATIONAL database. These rules guide us in deciding:

The number of tables there should be in a database.

●

The assignment of data items (attributes) to the various tables. ●

The rules of normalization have changed somewhat since Coddʼs initial version, but the normalization process remains the industry standard for designing a relational database.

A database design can be evaluated based on the following criteria:

Does the design minimize redundant data?

●

Does the design minimize the occurrence of anomalies within the data? ●

An anomaly is an inconsistency in the data stored in a database. There are three major types of anomalies: update, insert and delete.

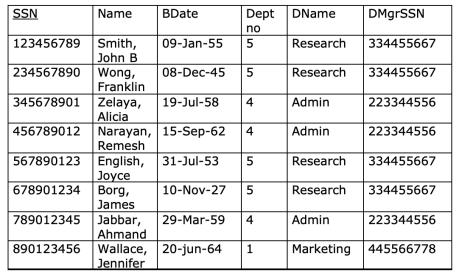
We will look at how the rules of normalization produce superior database designs in terms of these criteria.

**Minimizing Redundant Data**

One criterion for a good database design is that the design minimizes redundant data. Consider the following example.

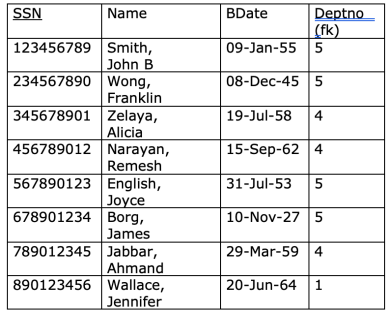
By intuition I could decide to create one table to store employee and department information as shown below.

EmpDept

Under this design the department name and the social security number of the department manager are repeated several times.

Applying the rules of normalization leads to the creation of two tables, one for the employee information and another for the department information. Data redundancy is reduced (not eliminated) in the normalized design.

Employee



Department



**Minimizing Update Anomalies**

An update anomaly occurs when we change the data in the database causing in an inconsistency in the data.

Suppose the manager of the research department retires and we hire a new manager. Under the intuitive design we must update four rows in the EmpDept table. If we update three correctly and forget to update the fourth row we have created an update anomaly in the database.

EmpDept



With the normalized design we need to update one row in the department table. No matter how we make the change we cannot get an anomaly in that all research department employees will have the same manager because the department is described by one row in the department table.

Employee

Department



**Minimizing Insert Anomalies**

An insert anomaly occurs when we add new data to the database causing an inconsistency in the data.

Suppose we hire a new employee named John Doe. Under the intuitive design we must know all the information about the department the employee works for in order to add the information about the new employee to the database. If the department related information is added incorrectly we have created an insert anomaly in the database.

EmpDept

With the normalized design we do not need to know or enter all the department

information in order to enter the new employee information. This minimizes the likelihood of an insertion causing an anomaly.

Employee



Department



Suppose the company expands and creates a new department. Under the intuitive design the new department information cannot be recorded in the database until the first employee is hired for the department (the employee social security number is the primary key for the EmpDept table). With the normalized design new departments can be added any time because the department information is kept in its own table.

**Minimizing Delete Anomalies**

A delete anomaly occurs when we delete data from the database causing in an inconsistency in the data.

Suppose we fire all employees in the research department with the intent of hiring better employees to replace them in the near future. Under the intuitive design when the last employee for the research department is

deleted we lose the information about the research department itself. Deleting employee information should not cause a loss of information about a department.

With the normalized design we are free to delete employee information without losing department information because each is stored in its own table.

By applying the rules of normalization to design a relational database we create a design that **minimizes** the amount of redundant data within the database and **minimizes** the likelihood of creating inconsistencies within the database during the life of the application.

——————————————————————————————————————— ——————————————————————

**Review:**

p.

What is normalization?

p. q.

Minimize redundant data. Minimize anomalies.

A critical step in the ERD database design process is deciding what? q.

p.

How many tables do we need to store the data for an application? q.

What data belongs in which table?

s.

Rules of Normalization:

The number of tables there should be in a database.

p.

q.

The assignment of data items (attributes) to the various tables.

t.

What is an anomaly?

An anomaly is an inconsistency in data. p.

How can a database be evaluated? u.

p.

Does the design minimize redundant data?

q.

Does the design minimize the occurrence of anomalies within data?

What are the different kind of anomalies in databases? v.

p. q. s.

Update Anomalies Insert Anomalies Delete Anomalies

How to minimize redundant data?w.

Applying the rules of normalization leads to the creation of multiple tables.

p.

Data redundancy is reduced (not eliminated) in the normalized design.

x.

How to minimize update anomalies?

With the normalized design we need to update one row in the department

p.

table. No matter how we make the change we cannot get an anomaly in that all research department employees will have the same manager because the department is described by one row in the department table.

How to minimize insert anomalies? y.

With the normalized design we do not need to know or enter all the

p.

department information in order to enter the new employee information. This minimizes the likelihood of an insertion causing an anomaly. With the normalized design new departments can be added any time

q.

because the department information is kept in its own table.

pz.

How to minimize delete anomalies?

With the normalized design we are free to delete employee information

p.

without losing department information because each is stored in its own table.

——————————————————————————————————————— ——————————————————————

**Normalization: How**

Large tables with many columns can be difficult to understand and increase redundant data within the database as well as increase the likelihood of anomalies occurring. The normalization process provides database designers with a systematic method of breaking down large tables into a group of related, smaller sized tables which:

Are easier to understand.

●

Minimize the amount of redundant data stored in the database.

●

Minimize the likelihood of anomalies occurring within the data. ●

The normalization process is based on a set of rules for designing relational databases enables and the database to be in one of five states:

First normal form (1NF)

●

Second normal form (2NF)

●

Third normal form (3NF)

●

Boyce-Codd normal form

●

Fourth normal form (4NF)

●

Domain-Key normal form●

Each normal form represents a set of rules. If a database is in a particular normal form, e.g. third normal form, then all of its tables comply with the rules for the normal form. First, second and third normal forms are based on E.F. Coddʼs initial set of rules for normalization. Over time additional normal forms have been developed to address problems, which were not addressed by Coddʼs initial forms. Boyce-Codd normal form is a revision of Coddʼs initial third normal form. Fourth normal form is an additional set of rules. Domain-Key normal form is a theoretical ultimate in relational database design proposed by Fagin in 1981. There is no defined method to consistently produce tables in Domain-Key normal form, so it remains a work in progress.

Each normal form enables you to create more tables with smaller number of columns in each table. As the number of tables in the database increases data redundancy and the likelihood of an anomaly occurring decreases. This makes the database easier to maintain. Unfortunately, as the number of tables rise the overhead increases. Performance can suffer, particularly if there are high volumes of rows in the tables. Normalization is a trade off between ease of maintenance and performance. Third normal form strikes the best balance (for most applications) between maintenance and performance requirements. Most relational databases are designed to comply with the rules of third normal form.

**Denormalization**

The term denormalization is used to describe the process where after you have completed a database design that meets the rules of normalization you adjust the design and introduce violations to a normal form.

This may be done to make the database more efficient or easier to use. We will introduce a final database design with the IQ School database that will contain a solution that is not fully normalized.

**Normal Forms Based on Keys and Business Rules**

Before looking at the specific rules of the normal forms we must understand that normal forms are based on two critical concepts.

First, a primary key uniquely identifies a row in a table. Every table has a primary key. If a mistake is made in selecting the primary key for a table, the normalization process will **not** produce a good database design.

The database design must reflect the business policies (rules) of the organization. The database designer must have a clear understanding of an organizationʼs business rules in order to apply the rules of normalization in a manner that produces a database design, which reflects the business policies of the organization. This is best demonstrated by an example.

Consider the database design for an assignment tracking application for a consulting firm. The database will record information about the firmʼs consultants, clients and which consultants are assigned to work with a given client. The firm may have a business policy that a client can be served by many different consultants based on a match between the type of service the client requests and the specialty of the consultant. The database design shown below would reflect this business policy.

The business policy results in a many to many relationship between clients and consultants (one consultant can serve many clients and a single client can receive services from many consultants). This causes the normalization process to create

the Assignment table with a primary key of ClientId, ConsultantId and StartDate; the combination of ClientId, ConsultantId and StartDate uniquely identifies an individual assignment.

If a different business policy were in place the normalization process would produce a different database design. Suppose the policy was that a client is permanently assigned to one consultant who will provide all services requested by the client. This results in a one-to-many relationship between consultants and clients (a client works with a single consultant but a single consultant can work with many clients). The following database design would emerge:



Note that the assignment table is no longer required to track which consultant is assigned to which client as there will only be one consultant working with a given client.

**The Rules of the Normal Forms**

The normalization process lets us systematically break down large tables to many

smaller ones. We typically apply the normalization process to each view (subset of data, which facilitates one specific task) in the application. This provides a database design for each individual view. The views are then merged to produce a single logical design for the entire database that accommodates the requirements of all the individual views.

We shall now work through an example view and apply the rules of first, second and third normal forms to produce a schema for the view in third normal form. We start with a source document of some sort, which describes the data required by the view, and use it to define a single table that contains all the data required by the view. We then apply the rules of normalization (first, second then third normal form) to break the single table into many smaller, related tables.

Example: Employee Workload

The employee workload is a display, which lets a manager determine the department an employee is assigned to within the company and the projects the employee is currently working on. This information helps the manager assign employees to new projects that are taken on by the company. A sample employee workload screen layout is shown below:

Business rules:

p.

An employee works for a single department in the company.

q.

A department can have many employees working for it simultaneously. s.

A single department develops a project.

t.

A department can develop many projects simultaneously.

u.

An employee can be assigned to a maximum of 4 projects simultaneously. v.

A project can have many employees working on it simultaneously. w.

Employee Id, Department Number and Project Number uniquely identify employees, departments and projects respectively.

Before applying the rules of normalization you should have a clear understanding of the entities, relationships and candidate keys in the view.

Entities: Employee, Department, Project

Relationships: 1. Employee : Department

N : 1

2. Department : Project

1 : N

3. Employee : Project

N : M

Candidate Keys: Employee Id, Department Number, Project Number

We can now apply the normalization process to produce the schema (database design) for the view.

**Step 1: Create the initial table**

The initial table is comprised of all data items in the view. The initial table MUST have a primary key that uniquely identifies each row in the table. If the view contains repeating groups of attributes they are listed in parenthesis.

The employee workload view has three candidate keys. We must select a primary key from the candidate keys. The view contains information about a specific employee; this points to the Employee Id as the primary key. We should consider all candidates though. The department number will not uniquely identify a row because a single department can have many employees assigned to it. The project number will not uniquely identify a row because many employees can work on a project. Each row in the table records information about a specific employee, the employee id will uniquely identify a row, so employee id **MUST** act as the primary key for the initial table.

A repeating group is one or more attributes that have multiple values within the view. The repeating group describes an entity that takes on the “many” role in a one-to-many or many-to-many relationship. When reviewing a source document, the repeating group will have multiple data values. A review of the screen layout shows the following attributes have multiple values and form a repeating group: Project Number, Project Name, Sponsoring Dept Name and Weekly Hours. These attributes all describe a project. Project shares a many-to-many relationship with Employee.

The initial table is shown below. A written notation is used for the initial table instead of a graphic representation of the schema (it is very difficult to depict a repeating group when using a graphic notation). The primary key attribute is underlined (EmployeeId). The repeating group of attributes appears within parenthesis.

Employee(Employee Id, Name, Department Number, Department Name, ( Project Number, Project Name, Sponsoring Department Name, Weekly Hours ) )

**Step 2: Apply the rules of 1NF**

1NF has two rules:

p.

A table MUST contain atomic attributes.

q.

A table cannot contain any repeating groups of attributes.

If composite attributes exist in the view they are replaced by two or more atomic attributes. This is tempered by the business situation. In some cases, it makes sense to store the data in composite form. Some designers overlook the atomic requirement.

If a repeating group of attributes exists in the view:

Remove the repeating group of attributes from the original table and

●

place them in a new table.

Duplicate the primary key of the original table and place in the new table

●

as a foreign key relating rows between the original and new tables. Designate the cardinality of the relationship between the new table

●

(child) and the original table (parent).

Designate a primary key for the new table.

●

Looking at the relationship between the entities involved helps set the primary key of the new table. A one-to-many relationship between the original tableʼs entity and the entity of the new table usually results in a single attribute primary key in the new table. A many-to-many relationship between the original tableʼs entity and the entity of the new table usually results in a composite primary key in the new table.

Reviewing the Employee Workload view, the following violations to 1NF exist:

p.

Name is a composite attribute.

q. Project Number, Project Name, Sponsoring Dept Name, and Weekly Hours form a repeating group.

The schema is altered to comply with the rules of 1NF.

The Name attribute, in the Employee table, is replaced with LastName and FirstName.

The repeating group of attributes is removed from the Employee table and placed in a new table named WorksOn. The EmployeeId attribute (primary key from the original table) is duplicated and placed in the WorksOn table where it acts as a foreign key relating rows in the Employee table (parent) with rows in the WorksOn table (child). Each row in the WorksOn table records information about a project that an employee is working on. The business rules state: One employee can work on a maximum of four projects. Therefore a one-to-many relationship exists between the Employee table and the WorksOn table. The cardinality of the relationship is marked as such. The last step is to designate a primary key for the WorksOn table. There is a many-to-many relationship between Employee and Project so we can expect a composite key. The candidate key attributes are considered:

EmployeeId and Project Number. Since one employee can work on multiple projects several rows within the WorksOn table can have the same EmployeeId. EmployeeId cannot uniquely identify a row in the WorksOn table. Since one project can be assigned to many employees several rows within the WorksOn table can have the same Project Number. Project Number cannot uniquely identify a row in the WorksOn table. The combination of EmployeeId and Project Number will uniquely identify a row in the WorksOn table and act as the primary key for the WorksOn table.

The database design, for the employee workload query view, is shown below.



The view schema complies with the rules of first normal form; the schema is said to be in first normal form.

The design enables many employees to be assigned to a single project and many projects to be assigned to a single employee (via the composite key in the Works On table). The database design is a partial implementation of the business rule **a single employee can be assigned to a maximum of 4 projects simultaneously**.

To fully implement this rule the logic of the program that adds a row to the WorksOn table will have to ensure there are a maximum of 4 rows for any given employee in the table.

Some business rules can be implemented via the DBMS in conjunction with the database design while others will need a combination of database design and program logic.

**Step 3: Apply the rules of 2NF**

2NF has one rule:

p.

All non-key attributes in a table must **fully depend** on the entire primary key of the table.

A violation to 2NF can only occur in a table with a composite (concatenated) primary key. When evaluating a database design for violations to 2NF any table with a single attribute primary key is ignored, as a violation to 2NF cannot occur in such a table.

To evaluate a database design for compliance with 2NF an understanding of the concept of dependence is needed. What is meant by the term fully depend? How do we know if a non-key column is fully dependent on its primary key?

Dependence has its roots in mathematics. A detailed discussion of these principles is beyond the scope of this lesson. A pragmatic definition of dependence follows.

A non-key attribute (neither a primary key or a foreign key) is fully dependent on a primary key attribute if:

For each value of the key attribute there can be one value for non-key

●

attribute.

This means that if you know the value of the key attribute you know the value of the non-key attribute (or you can figure it out).

Consider the following tables:



Does this database design comply with 2NF?

Since the Student table has a single attribute primary key there is no possibility of a violation to 2NF. The Marks table has a composite primary key so this table is evaluated further. The non-key attributes are CourseName and FinalMark. Is either of these attributes fully dependent on part of the primary key? If so, it is a violation to 2NF.

Each attribute acting as part of the primary key is evaluated.

If the value of StudentId (12345) is known: ●

Is the value of CourseName known? No, a student can enroll in many

○

courses so there is no way to know the value of CourseName. Therefore, CourseName is not fully dependent on StudentId.

Is the value of FinalMark known? No. Therefore, FinalMark is not fully

○

dependent on StudentId.

If the value of CourseId (BCS245) is known:

○

Is the value of CourseName known? Yes, if the CourseId is BCS245 the

○

CourseName can only be one value: Database Management. Therefore, CourseName is fully dependent on CourseId and not fully dependent on the primary key of the Marks table. This is a violation to 2NF.

Is the value of FinalMark known? No. Therefore, FinalMark is not fully

○

dependent on CourseId.

If the value of Term (2001-Fall) is known:

○

Is the value of FinalMark known? No. Therefore, FinalMark is not fully

○

dependent on Term.

What is FinalMark fully dependent on? A student can take a course multiple times in different terms and achieve a different mark each time the course is taken. Therefore, to know the value of FinalMark the StudentId, CourseId and Term must be known. Therefore, FinalMark is fully dependent on the primary key of the Marks table and complies with 2NF.

CourseName is fully dependent on CourseId, part of the primary key of the Marks table, and is a violation to 2NF. An attribute that violates 2NF is referred to as a **partial dependency**.

If a partial dependency exists:

Remove the partially dependent attribute from the original table and place

○

in a new table.

The primary key attribute(s) the partially dependent attribute is fully

○

dependent on, in the original table, are duplicated and placed in the new table.

The primary key attribute(s) the partially dependent attribute is fully

○

dependent on become foreign keys, in the original table, relating rows in the new table (parent) with rows in the original table (child). Designate the cardinality of the relationship between the new table

○

(parent) and the original table (child).

Designate a primary key for the new table.

○

To complete the student example:

Course Name is a partial dependency (fully dependent on CourseId). CourseName is removed from the Marks table and placed in a new table named Course. CourseId is duplicated and placed in the Course table. The primary key for the Course table is CourseId.

The database design, in 2NF, is shown below.

Returning to the Employee Workload view:



The following violations to 2NF exist:

ProjectName and SponsoringDepartmentName are both partial dependencies and are fully dependent on ProjectNumber.

Since both attributes are fully dependent on the same primary key attribute they are placed in a new table named Project. ProjectNumber is duplicated and placed in the Project table. ProjectNumber acts as a foreign key, in the WorksOn table, relating rows between the Project table (parent) and the WorksOn table (child). ProjectNumber is the primary key for the Project table.

The database design, for the employee workload query view, is shown below. 

The view schema complies with the rules of second normal form; the schema is said to be in second normal form.

WorksOn is an associative entity. The many-to-many relationship between the Employee entity and the Project entity is now fully implemented as two one-to-

many relationships (Employee to WorksOn and Project to WorksOn).

**Step 4: Apply the rules of 3NF**

3NF has one rule:

p.

A non-key attribute cannot be fully dependent on another non-key attribute.

A non-key attribute that is fully dependent on another non-key attribute is termed a **transitive dependency**.

A table must have two or more non-key attributes for a violation to 3NF to occur. If a transitive dependency exists:

Remove the transitively dependent attribute(s) and place in a new table.

●

Duplicate the non-key attribute that the violator was fully dependent on and

●

place in the new table.

The non-key attribute that the violator was fully dependent on acts as a

●

foreign key, in the original table, to relate the rows between the new table (parent) and the original table (child).

Designate the cardinality of the relationship between the new table (parent)

●

and the original table (child).

Designate a primary key for the new table.

●

In the Employee Workload view:

Department Name is fully dependent on Department Number in the Employee table. Department Name is a transitive dependency. Department Name is removed from the Employee table and placed in a new table named Department. Department Number is duplicated and placed in the Department table. Department Number acts as a foreign key in the Employee table relating rows in the Employee table (child) to rows in the Department table (parent). Department Number is the primary key of the Department Table.

The database design, for the employee workload view, is shown below.



This completes the schema for the Employee Workload view. The schema is in 3NF. Since we do not plan to move the design further than 3NF, the logical database design for the Employee Workload view is complete. This design can be used to create the necessary tables in the database. Each entity represents one table in the database.

A similar logical database design would be created for each view/task in the application. The designs would be merged to produce a single logical database design that accommodates the needs of all tasks within the application.

——————————————————————————————————————— ——————————————————————

**Review:**

**Normalization Process:**

p.

The normalization process provides database designers with a systematic method of breaking down large tables into a group of related, smaller sized tables which:

p.

Are easier to understand.

q.

Minimize the amount of redundant data stored in the database. s.

Minimize the likelihood of anomalies occurring within the data.

q.

The normalization process is based on a set of rules for designing relational databases enables and the database to be in one of five states:

First normal form (1NF)

○

Second normal form (2NF)

○

Third normal form (3NF)

○

Boyce-Codd normal form

○

Fourth normal form (4NF)

○

Domain-Key normal form ○

**Denormalization:**

s.

The term denormalization is used to describe the process where after you have completed a database design that meets the rules of normalization you adjust the design and introduce violations to a normal form.

This may be done to make the database more efficient or easier to use. We will introduce a final database design with the IQ School database that will contain a solution that is not fully normalized.

**The rules of Normal Forms**

t.

The normalization process lets us systematically break down large tables to many smaller ones. We typically apply the normalization process to each view (subset of data, which facilitates one specific task) in the application. This provides a database design for each individual view. The views are then merged to produce a single logical design for the entire database that accommodates the requirements of all the individual views.

p.

**Business Rules:**

p.

An employee works for a single department in the company. q.

A department can have many employees working for it simultaneously. s.

A single department develops a project.

t.

A department can develop many projects simultaneously. u.

An employee can be assigned to a maximum of 4 projects simultaneously.

v.

A project can have many employees working on it simultaneously. w.

Employee Id, Department Number and Project Number uniquely identify employees, departments and projects respectively.

w.

Before applying the rules of normalization you should have a clear understanding of the entities, relationships and candidate keys in the view.

**Names are almost never unique.**

u.

We can now apply the normalization process to produce the schema (database design) for the view.

p.

Step 1: Create initial table (0NF Zero Normal Form)

p.

The initial table is comprised of all data items in the view. The initial table **MUST** have a primary key that uniquely identifies each row in the table. If the view contains repeating groups of attributes they are listed in parenthesis.

q.

Step 2: Apply the rules of 1NF

p.

1NF has two rules:

p.

A table **MUST** contain atomic attributes.

q.

A table cannot contain any repeating groups of attributes.

s.

Step 3: Apply the rules of 2NF

p.

2NF has one rule:

p.

All non-key attributes in a table must **FULLY DEPEND** on the entire primary key of the table.

t.

Step 4: Apply the rules of 3NF

p.

3NF has one rule:

p.

A non-key attribute cannot be fully dependent on another non-key attribute.

p.

What is a **transitive dependency**?

q.

It is unique and not apart of the primary key.

s.

A non-key attribute that is fully dependent on another non key attribute is termed a **transitive dependency**.

q.

A table must have two or more non-key attributes for a violation to 3NF to occur.

v.

Candidate keys are unique keys.

w.

One primary key per table. (Composite Keys allow more than one Primary Key)

——————————————————————————————————————— ——————————————————————

***Database Management: Normalization Exercise Suite***

******

The ***Emergency Service & Product*** (ESP) Company is a very small parts and service company. It currently uses a crude paper based system (3 by 5 cards, paper journal, etc) to keep track of its customers, inventory, suppliers, payments and invoices.

The company is growing rapidly and has found that it is unable to keep pace with the volume of data it must record about its business activities. It has decided that it wishes to automate some portions of its business.

You have been hired to assist the company in analyzing its business activities and to design a set of relational tables that they can use to develop a database system.

The company will provide you with copies of all their paper based forms (invoices, orders, batch records, etc). You will use these over several weeks to analyze the various forms and produce **third normal form (3NF)** relational **entity relationship diagrams (ERD)** for each **view** of their data. As a last step in the process, you will **merge** all the views to produce **one integrated ERD** and **fully detailed relational entities.**

**Customer Details View**

ESP keeps detailed information about each customer so that they can be contacted when needed or when a product needs to be sent in the mail. This information is kept on 3x5 cards as it is fairly static in size and doesnʼt change very often. The cards look as follows:



Customer numbers are unique and are never reused by the company. A card is created for each new customer that buys from the company.

Customer Orders View

ESP has preprinted order forms that they fill in by hand when a customer buys some of their product. The customer details are copied from the customer card onto the order form as well as details about the product ordered. Each order form has a preprinted sequential order number that is unique. The top portion of the forms looks as follows:



The forms are two part with carbon in between. The customer gets the top copy and ESP keeps the other. The ESP copy is filed by order number into a binder.

Because prices change on a fairly regular basis ESP would like to keep the following information:

The current price for each item in inventory

●

The price each item sold for at the time the item was sold ●

You are to take the information provided above and prepare a set of 3NF relational entities and produce a relational 3NF ERD for each view. Merge the two view schemas to produce a single set of tables (conceptual level schema) that will satisfy the requirements of both views.

——————————————————————————————————————— ——————————————————————

**Review:**

p.

What is an ESP?

The ***Emergency Service & Product*** (ESP) Company is a very small parts

p.

and service company. It currently uses a crude paper based system (3 by 5 cards, paper journal, etc) to keep track of its customers, inventory, suppliers, payments and invoices.

What does an ESP contain?

q.

p. q.

Customer Details View Customer Orders View

——————————————————————————————————————— ——————————————————————

***Database Management: Normalization Exercise Suite 2 ***

The ***Emergency Service & Product*** (ESP) Company is a very small parts and service company. It currently uses a crude paper based system (3 by 5 cards, paper journal, etc) to keep track of its customers, inventory, suppliers, payments

and invoices.

The company is growing rapidly and has found that it is unable to keep pace with the volume of data it must record about its business activities. It has decided that it wishes to automate some portions of its business.

You have been hired to assist the company in analyzing its business activities and to design a set of relational tables that they can use to develop a database system.

The company will provide you with copies of all their paper based forms (invoices, orders, deposit batch records, etc). You will use these over several weeks to analyze the various forms and produce **third normal form (3NF)** relational **entity relationship diagrams (ERD)** for each **view** of their data. As a last step in the process, you will **merge** all the views to produce **one integrated ERD** and **fully detailed relational entities.**

***Customer Payments Against Orders***

ESP allows its customers to make payments against their orders in several ways. These include:

Full payment at the time of order

●

Partial payment at the time of order with final payment(s) at a later time

●

No payment at time of order with final payment(s) at a later time

●

Payments can be made by cash, cheque, Visa, America Express, etc ●

ESP keeps a payment log to keep track of all payments made against an order. Each order has its own log sheet. The log looks similar to the following:

Note that a customer payment is applied against a single order.

When a payment is made, ESP goes through the following process (worst case scenario – where a cheque arrives in the mail with no indication of an order

number on it):

Customer name is searched in the customer file to determine their customer

●

number. The address on the cheque is used to confirm the customer. The Orders File and the Payments Log are manually searched by customer

●

number to find any order(s) that have an outstanding balance. If this is the first payment for this order, then a new Payment Log form is filled

●

in with customer name, number, etc.

The payment details are entered and the balance owing is updated.

●

The Deposit Batch # is entered on the log when the payment is deposited into

●

the bank so that the payments can be reconciled with the deposit on the bank statement.

ESP has indicated that there are never any over payments made and refunds or cancellations are never issued.

ESP has also indicated that they would like the new system to only allow specific payment types so they can monitor usage of each type.

You are to take the information provided above and prepare a set of **3NF** relational entities and produce a relational **3NF ERD.**

*——————————————————————————————— ——————————————————*

***Primary key is always one and foreign key is always many. If it can be derived using a formula do not store the = xxxxxx***

*——————————————————————————————— ——————————————————*

***Database Management: Normalization Exercise Suite 3***

******

**Inventory Control**

ESP keeps track of its inventory using a paper based ledger system where purchases from suppliers are recorded for each item that they have in stock.

The system uses a form that looks as follows:

Item Numbers are unique and never reused. The Inventory Control Sheet records details of each item as well as all orders of that particular item from a supplier. ESP would like to be able to keep track of how many of each item they have in

stock on an on-going basis so that they can use JIT (Just In Time) ordering rather than ordering when they run out. They would like the new system to keep an in stock count for each item as well as a re-order value. If the in-stock count is less than or equal to the re-order level the item will be ordered from a supplier.

**Purchase Orders**

When items are ordered from suppliers a Purchase Order is created. A sample Purchase Order is shown below.



Purchase Order (PO) numbers are unique and never reused. The same supplier always has the same supplier number (which is also unique) and the supplier details are hand-written on the PO every time.

The item number is the number assigned by ESP from the Inventory Control Sheet and is the same regardless of supplier the item is purchased from. The Supplierʼs Item Number and Description are specific to the supplier and may vary on different Purchase Orders. The cost of the item depends on the supplier and may vary between different purchase orders for the same supplier.

You are to take the information provided above and prepare a set of **3NF** relational entities and produce a relational **3NF ERD** for each view.

Merge the schemas for the Inventory Control view and the Purchase Orders view.

Merge the schema for the two views in ESP # 3 with the schema for the previous views to produce a single set of tables that will support the needs of all the ESP views completed to date (Customer Details, Customer Orders, Customer Payments, Inventory Control and Purchase Orders).

*——————————————————————————————— ——————————————————*

***Database Management: Normalization Exercise Suite 4 Payments to Suppliers for Purchase Orders***

ESP pays the full amount of each purchase order when it is due. ESP may pay for more than one purchase order with one cheque if they are all from the same supplier. The cheque is always written for the full and correct amount of all the purchase orders being paid for. ESP keeps a register of its cheques that looks as follows:

Each cheque number is unique and numbers are never used over again.

You are to take the information about the Cheque Register, provided above, and prepare a set of **3NF** relational entities and produce a relational **3NF ERD.**

*——————————————————————————————— ——————————————————*

Structured Query Language = SQL

Microsoft SQL Server Management Studio 2017

MySQL Express 2017

F5 is hotkey for Execute in Microsoft SQL Server Management Studio DMIT-SQL1

Ctrl-Shift-R to refresh

**Identity is only used for Primary Keys**

**Do not put spaces in table or column names**

**If youʼre using an identity you should store it as an integer**

**Anything you do separate it with a “go”**

dbo is database owner

New Query

use DMIT1508SectionA04\_bmanke2

go

Create Table Customers

(

CustomerNumber int IDENTITY(1,1),

FirstName varchar(30),

LastName varchar(30),

PhoneNumber decimal(11,0)

)

**Constraint names are only 126 characters long**

**When dropping a table go from children to parents**

*——————————————————————————————— ——————————————————*

**Constraints**

with { CHECK | NOCHECK }

{ check | nocheck }

Dropping a column deletes it from the table and all data associated with it If a table is not null you need a default

sp\_help TableName

Can have up to 249 indexes

Can have up to 16 columns in index

Creating a Unique Index is useful when you have multiple datas unique to the original data

Cannot drop an index for a primary key

*——————————————————————————————— ——————————————————*

**Queries**

**Do not use \* in select**

is null

is not null

“between” or “not between”

Do not use between for strings

in finds values

**The order of which select works:**

Select

From

Where

Group By

Having

single quotes are for strings

double quotes are for column headings

**DO NOT USE SINGLE QUOTES FOR COLUMN HEADINGS**

Select Mark “Mark”

From MickeyMouseGrade

is not LIKE

LIKE %S means it ends in S

LIKE %S% means it has to have an S

LIKE %S\_% means it has to have an S and one character after it anywhere

% looks for the character

\_ looks for any character

*——————————————————————————————— ——————————————————*

**Aggregates**

**AVG:** The AVG function returns the average of a column of values. It can only be used on numeric columns and records containing null are ignored and not included in the average calculation.

**SUM:** The SUM function returns the sum of a column of values. It can only be used on numeric columns and records containing null are ignored and not included in the sum calculation.

**MIN:** The MIN function returns the minimum value from a column of values. Can be used with columns that have a a numeric, date or character datatype. Records with a null value are ignored.

**MAX:** The MAX function returns the maximum value from a column of values. Can be used with columns that have a a numeric, date or character datatype. Records with a null value are ignored.

**COUNT**: The COUNT function returns the number of non-null values from a column of values OR returns the number of records that match the where criteria.

*——————————————————————————————— ——————————————————*

**String Functions**

**Length:** LEN - Returns the length

**LEFT**: LEFT - Returns the left side column of data, takes 2 arguments data and how many digits you want

**RIGHT:** RIGHT - Returns the right side column of data, takes 2 arguments data and how many digits you want

**Substring:** Returns characters from a specific spot with the count of the second number.

**Reverse:** Returns the string in reverse

**Upper or Lower:** Returns the string in upper or lowercase

**LTRIM or RTRIM:** Trims the spaces/whitespace from the column data or expression

*——————————————————————————————— ——————————————————*

**Group By**

The GROUP BY clause is used in conjunction with aggregate functions to provide subtotal calculations of the aggregate function. We could easily return the average mark from the grade table. What if we wanted the average mark for each course? For each student? For each course for each student? To do this we would use GROUP BY.

Donʼt group by the whole primary key or else you wonʼt be getting the proper info.

*——————————————————————————————— ——————————————————*

**Order By**

The Order By clause is used to return results in a certain order. The results can be sorted by one column or more than one column in either descending (DESC) or ascending (ASC) order. When no order is specified, ASC is the default.

*——————————————————————————————— ——————————————————*

**Having**

Where clause for aggregate functions. Do not use a where clause for aggregate functions. This is what the having clause is for.

Has a dependency on Group By.

having avg(mark) >= 80

*——————————————————————————————— ——————————————————*

**Inner Join**

**Old Style Inner Join**

from table alias,

table alias,

table alias,

where alias.attribute = alias.attribute and

alias.attribute = alias.attribute

**Use first two letters of table name for table-alias.**

**New Style Inner Join**

select table-alias.attribute,

table-alias.attribute,

table-alias.attribute,

table-alias.attribute

from table table-alias

inner join table table-alias

on alias.key = alias.key

inner join table table-alias

on alias.key = alias.key

*——————————————————————————————— ——————————————————*

**Unions**

*——————————————————————————————— ——————————————————*

**Sub-Queries**

Use in instead of an = operator if you want to return multiple values

*——————————————————————————————— ——————————————————*

**Insert/Update/Delete**

Can only insert/update/delete one table at a time

You either delete the whole record or donʼt delete it at all

*——————————————————————————————— ——————————————————*

**Stored Procedures**

Is the stored procedure above an internal or external procedure? How do you

●

know? External or internal table

Assigning values returned from a query

●

<> is not equal

●

**Flow Control:**

**Declaring a variable**

Declare @variablename datatype

**Assigning a LITERAL value to a single variable**

Declare @variablename datatype

Set @variablename = datatype

**Assigning a LITERAL value to many variables**

Declare @firstname varchar(40), @lastname varchar(40)

Select @firstname = ‘Homerʼ,

@lastname = ‘Simpsonʼ

**Assigning a value returned from a query**

declare @firstname varchar(40)

Select @firstname = StaffFirstName from Staff where StaffID = 1

**Assigning values to more than a single variable by select more than one column in a query**

declare @firstname varchar(40), @lastname varchar(40)

select @firstname = StaffFirstName, @lastname = StaffLastName from Staff where StaffID = 1

**Simple:**

Exec sp\_helptext procedure name

Retrieves the source code for a stored procedure

●

If the stored procedure call is the first statement in a batch you can omit the

●

EXEC statement

Create Procedure PR\_Name

As

SQL Statements

Return

**Transactions:**

**Logical Unit of Work (LUW):**

An SQL Transaction is a structure that ensures that a complete LUW succeeds or fails.

Example: A LUW could be used to register for a course, or transfer funds from

●

one bank account to another. The key is understanding that each of these LUW consists of more than one step. To register for a course a record is added to the grade table and the students balance is updated with the cost of the course(s). When you transfer funds from one bank account to another you subtract funds from one account and add funds to another.

Using SQL Server Transactions means all the operations to complete a LUW work

or they all fail.

What happened when you write a single statement to insert/update/delete a record(s) into a table and it fails? It all fails correct? That is because, SQL Server automatically wraps up a single DML statement (even it affects multiple records) in a transaction to ensure it all works or it all fails.

Begin Transaction — Start a Transaction

●

Select \* from grade — See the data in the grade table BEFORE the delete

●

Delete Grade — Delete all the records from the grade table

●

Rollback Transaction — Rollback the Transaction

●

Select \* from grade — See the data in the grade table after the delete and

●

rollback

——————————————

● ● ● ● ●

Begin Transaction Select \* from grade Delete grade

Commit Transaction Select \* from grade