DATABASE ESSENTIALS

Daniel Gaudreault

CONTENTS

CHAPTER 1 - BASIC DATABASE CONCEPTS

WHAT IS A DATABASE?

A database is an organized or structured collection of data. It contains the data necessary for some purpose or purposes. The data is arranged to allow a set of activities to occur, as well as ensuring the data can be accessed and altered in an efficient manner.

There are several common types of databases; each type of database has its own data model (how the data is structured). They include; Flat Model, Hierarchical Model, Relational Model and Network Model.

THE FLAT MODEL DATABASE

In a flat model database, there is a two-dimensional (flat structure) array of data. For instance, there is one column of information and within this column it is assumed that each data item will be related to the other. For instance, a flat model database includes only zip codes. Within the database, there will only be one column and each new row within that one column will be a new zip code.

THE HIERARCHICAL MODEL DATABASE

The hierarchical model database resembles a tree like structure, such as how Microsoft Windows organizes folders and files. In a hierarchical model database, each upward link is nested in order to keep data organized in a particular order on a same level list. For instance, a hierarchal database of sales, may list each days sales as a separate file. Within this nested file are all of the sales (same types of data) for the day.

THE NETWORK MODEL

In a network model, the defining feature is that a record is stored with a link to other records - in effect networked. These networks (or sometimes referred to as pointers) can be a variety of different types of information such as node numbers or even a disk address.

THE RELATIONAL MODEL

The relational model is the most popular type of database and an extremely powerful tool, not only to store information, but to access it as well. Relational databases are organized as tables. The beauty of a table is that the information can be accessed or added without reorganizing the tables. A table can have many records and each record can have many fields.

Tables are sometimes called a relation. For instance, a company can have a database called customer orders, within this database will be several different tables or relations all relating to customer orders. Tables can include customer information (name, address, contact, info, customer number, etc.) and other tables (relations) such as orders that the customer previously bought (this can include item number, item description, payment amount, payment method, etc.). It should be noted that every record (group of fields) in a relational database has its own primary key. A primary key is a unique field that makes it easy to identify a record.

Relational databases use a programming interface called SQL or Standard Query Language. SQL is currently used on practically all relational databases. Relational databases are extremely easy to customize to fit almost any kind of data storage. You can easily create relations for items that you sell, employees that work for your company, etc.

WHAT IS A DATABASE MANAGEMENT SYSTEM?

A database management system is software that manages data in an organized or structured way.

This definition follows very logically from the definition of a database given above. Common features offered by most database management systems include:

- A way to define containers (typically tables and columns) to hold data.
- Commands and other features allowing the user to search for and sort the data.
- Commands allowing the user to insert new data, update and delete existing data.
- A GUI builder, or other feature allowing the user to create forms to view the data
- A report generator allowing data to be printed
- Enforcement of relational integrity and business rule constraints
- Management of simultaneous data access among multiple users

WHAT IS A RELATIONAL DATABASE?

Early attempts to create computer databases were closely tied with the physical storage of data on disk. This resulted in onerous requirements on the database developer and user in order to get the desired results. For example, to search for records on disk, it was necessary to write code to sequentially load data using physical disk addresses.

The relational database model was developed to solve that problem. Fundamentally, the relational database model was created to separate the design and use of databases from the storage of data on disk. This concept is known as data independence. The relational database model views data in table form.

Part

Part Number	Description
27	Umbrella Stand
32	Spittoon
48	Buggy Whip

In relational terms, this collection of data is known as a table. Thus, the relational database model views data in terms of collections of data in tabular form. A relational database is simply a database in which the user perceives that all data contained within the system is in the form of tables. Notice that the physical storage of records is not relevant. The only relevant view of the data is the user's view. How the data is stored on disk is immaterial for our purposes.

BASIC RELATIONAL DATABASE TERMINOLOGY

So far, this text contained some terms which we will now take the time to define formally.

TABLE

A table is a two-dimensional collection of data. That is, it consists of columns and rows. Look again at the table shown previously. It looks a lot like a spreadsheet, doesn't it?

But there is a unique feature which distinguishes a table from a spreadsheet. A table is non-positional. This has two aspects. First, this means that the order in which the columns are presented can be switched around at will. Similarly, the order in which rows are displayed can be changed at any time. All of this can be done without changing the table at all. (While this can be done with a spreadsheet, it cannot be done without changing the spreadsheet.)

Tables are the foundation of every Relational Database Management System. Every database consists of one or more tables, which store the database's data, metadata and information. Each table has its own unique name and consists of columns and rows.

The database table columns (called also table fields) have their own unique names and have a pre-defined data types. Table columns can have various attributes defining the column functionality (the column is a primary key, there is an index defined on the column, the column has certain default value, etc.).

While table columns describe the data types, the table rows contain the actual data for the columns.

Tables tend to come in 2 different varieties:

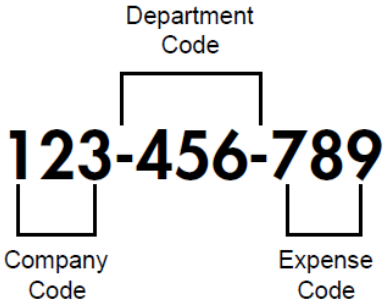
- Base or Regular Tables – standard tables can contain information
- Reference or Lookup Tables – which are used to describe a list of values that may appear in another table. Often these are the source of foreign keys. Also, when you are filling out a computerized form, online or otherwise, the fields that are drop downs are probably coming from a reference table.

COLUMN

The definition of column follows from that of table given above. A column is simply an item of data which is stored in every row in a table. Columns are sometime called attributes since they contain something about the table which the database is intended to store.

Columns should contain only one item of data in each row. Stated another way, a column should not contain more than one item of data in a single row. Technically, this is referred to as a composite column. An example of this is a typical accounting code as shown on the diagram to the right for a column called Account Number.

Avoid this type of column in your databases whenever possible. It may make accountants smile, but it makes your databases difficult to work with. Either you must redundantly store the department code data in another column, or a search to retrieve accounts for a specific department will require a contains type search. Both alternatives are messy.



There are other problems which arise when one of the components of the composite column changes. For example, suppose that an expense code is transferred from one department to another. This would theoretically require that you change the value in the Account Number column as well. That may not be possible, especially if this column is being used to identify each row in the table. The alternative is to leave the data in the composite column unchanged, and update other columns containing the components of the composite column. If you do that, your database is now inconsistent.

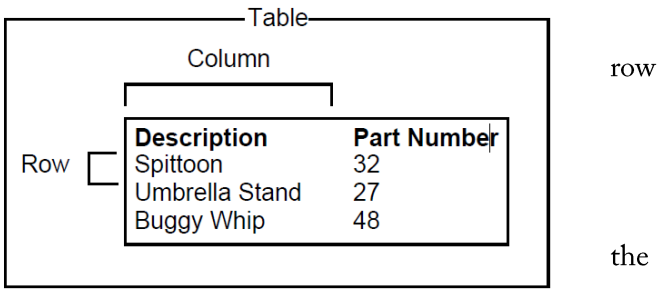
A better approach is to split the composite column up into three individual columns, as shown on the following example:

Company Code	Department Code	Expense Code
123	456	7890

Each of these three columns now contains one, and only one, item of data. The quality of a column which contains one, and only one, data value for each row is called atomic. When you design a database, make your columns as atomic as possible.

ROW

Like columns, the definition of row follows from the definition of table given above. A row is simply a single instance of whatever is stored in a table. Every row contains every column in the table, even if the value in any given column is undefined. The diagram on the right summarizes what we have discussed on tables, columns and rows.



PRIMARY KEY

The concept of a primary key is basic to the relational database model. The relational database model states that every row in a table must be unique. Therefore, there must be either a single column or a combination of columns which uniquely identifies each and every row in the table. This column or combination of columns is referred to as the primary key.

Thus, the definition of a primary key is the column or combination of columns which can be used to uniquely identify one row from any other row in the table. Now consider three tables instead of just one:

Part

Description	Part Number
Spittoon	32
Umbrella Stand	27
Buggy Whip	48

Warehouse

Warehouse Code	Warehouse City
A	Buffalo
B	Hong Kong
C	New York

Stock

Part Number	Warehouse Code	Stock Quantity
27	A	10
27	B	15
27	C	7
32	B	25
48	A	8
48	B	12

The primary key of [Part] is [Part]Part Number. This column uniquely identifies each and every row in this table. Similarly, the primary key of [Warehouse] is [Warehouse]Warehouse Code.

When we get to [Stock], the issue is not so obvious. There is not any single column which uniquely identifies every row of this table.

The answer is that [Stock] has a multi-column primary key. The primary key of [Stock] consists of two columns: [Stock]Part Number and [Stock]Warehouse Code. A primary key, like [Part]Part Number, which contains only one column is referred to as a simple key. A primary key which contains more than one column, like the combination of [Stock]Part Number and [Stock]Warehouse Code, is referred to as a composite key.

Another aspect of a primary key is permanence. Permanence indicates that the value in the primary key will never change over the life of a row. For example, examine the following table:

Person

Last Name	First Name	SSN
Williams	Doris	458-38-6214
Worth	Karla	155-78-1648
Vernon	Mark	788-45-3546
Jones	Marc	258-45-3577

While the combination of [Person]Last Name and [Person]First Name uniquely identifies each row in this table, there is a problem with the permanence of these columns. Assume that Mark Vernon and Karla Worth get married, and they decide to hyphenate their last names. Suddenly the values stored in the table become:

Person

Last Name	First Name	SSN
Williams	Doris	458-38-6214
Worth-Vernon	Karla	155-78-1648
Worth-Vernon	Mark	788-45-3546
Jones	Marc	258-45-3577

That might seem fine, just looking at this table alone, but what if you used these columns to relate to other tables? (See the section below on relations.) Now suddenly, you must change all the values stored in the related tables. But wait! That might affect other tables related to the related tables. This process can go on endlessly. This points out another obvious fact. In a composite key, all columns must be unchanging. Any change to any of the columns in the primary key results in a change to the primary key, and that is not allowed.

The obvious solution is to make sure that all columns in the primary key never change during the life of a row. Looking at the [Person] table you might choose the [Person]SSN column as the primary key. (This is very common.) But there is a problem with this approach as well. Obviously, a primary key must be defined for every row in the table. (It must be mandatory.) Assume that Karla and Mark have a baby whom they name Jason. You now wish to enter a row for Jason, but during the first two years of his life, Jason doesn't have a value for [Person]SSN, leading to the following:

Person

Last Name	First Name SSN	SSN
Williams	Doris	458-38-6214
Worth-Vernon	Karla	155-78-1648
Worth-Vernon	Mark	788-45-3546
Jones	Marc	258-45-3577
Worth-Vernon	Jason	

This is unacceptable. There is no way to uniquely identify Jason's row. (By the way, this also implies that every column in a composite key must be mandatory.) For this reason, you decide to create a unique column to serve as the primary key, as shown:

Person

Last Name	First Name	SSN	Person Number
Williams	Doris	458-38-6214	1
Worth-Vernon	Karla	155-78-1648	2
Worth-Vernon	Mark	788-45-3546	3
Jones	Marc	258-45-3577	4
Worth-Vernon	Jason	5	

A primary key (like [Person]SSN if it works) which is already contained in the table, and is mandatory, permanent and uniquely identifies every row is referred to as a natural key. A primary key which you manufacture yourself (such as [Person]Person Number, as shown above) is referred to as a surrogate key. Natural keys are rare, but they do exist.

One other thing about primary keys. All columns in a table which are not in the primary key is called non-key columns. For example, in the above table, [Person]Last Name, [Person]First Name and [Person]SSN are non-key columns.

RELATIONS

Closely related to the concept of primary key is that of a relation. Take another look at the [Stock] table:

Stock

Part Number	Warehouse Code	Stock Quantity
27	A	10
27	B	15
27	C	7
32	B	25
48	A	8
48	B	12

As indicated above, the combination of [Stock]Part Number and [Stock]Warehouse Code is the primary key for this table. Now assume that you wish to see the [Part]Description of each part which is contained in the [Stock] table.

By creating a relation, you can now produce the following view:

Stock/Part Description View

Part Number	Description from Part	Warehouse Code	Quantity
27	Umbrella Stand	A	10
27	Umbrella Stand	B	15
27	Umbrella Stand	C	7
32	Spittoon	B	25
48	Buggy Whip	A	8
48	Buggy Whip	B	12

Notice that the values for Description are being pulled from the [Part] table. That is to say, you are not redundantly storing the Description in every row in the [Stock] table. You can do this because the values stored in [Stock]Part Number match the values stored in [Part]Part Number. Notice, for example that the value for Umbrella Stand (Part Number 27) matches in both [Stock] and [Part]. This points out several things about the relational database model:

- The relation database model states that links between tables should be performed only by matching data values stored in columns. In the example above, [Stock]Part Number is being used to match to [Part]Part Number.
- This feature allows two tables to be related. A relation is simply the link between two tables both of which contain matching data in columns. (In this guide, we will use the terms related, relation and relationship interchangeably. They all refer to a link between two tables as described in this section.)
- A relation is two-way. You should be able to navigate to the matching row or rows of either table from a row of the other table. For example, you should be able to get the [Part] row for any given [Stock] row, and you should be able to get the matching [Stock] rows for any given [Part] row.
- Relations come in three flavors: many-to-many, one-to-many or one-to-one.
- A one-to-many relation is shown like this: 1:M.
- A many-to-many relation is shown like this: M:M.
- A one-to-one relation is shown like this: 1:1.
- A table on the one side of a 1:M relation is called the one table. A table on the many side of a 1:M relation is called the many table.

M:M relations can be shown during the design phase, but a M:M relation must eventually be resolved into two 1:M relations using a third linking table (More on this later.) A foreign key is a column or set of columns which is being used to match values in a relation and is not the primary key of its table. In a correctly designed database, a foreign key will always be on the many side of a 1:M relation. Also, the matching set of columns of the one table will always be the primary key of the one table. If either of these statements is not true, your design is wrong.

Look again at our example using the [Stock] table. The relation between the [Part] table and the [Stock] table is 1:M. That a part can be in stock in more than one warehouse, but a [Stock] row can only refer to one part. In this situation, the matching column of the many table, i.e. [Stock]Part Number, is the foreign key of the relation.

Notice that the matching column of the one table, i.e. [Part]Part Number, is the primary key of [Part]. The reason why the matching set of columns on the one side of a 1:M relation is always the primary key of the one table is simple: The primary key is the only way to reliably identify the desired row in the one table. A primary key is guaranteed to be permanent, mandatory and unique. This means that the primary key is always the best way to identify a row in a table.

Also, notice that [Stock]Part Number is not the primary key of [Stock]. (While Stock]Part Number participates in the primary key of [Stock], it is not by itself the primary key of [Stock].)

EQUIVALENT TERMINOLOGY

One of the most difficult aspects of database design is keeping the terminology straight. Below is a table that display the equivalent terms for the same things at the various stages in database design and development.

Conceptual Model	Relational Model	Physical Model
Entity Type	Relation	Table
Entity Instance	Tuple	Row
Attribute	Attribute	Column
Allowable Values	Domain	Data Type
Property Value	Element	Column Value
Identifier	Prime Key	Primary Key

REVIEW

Please take the time to fill in these review questions to help cement your understanding of these concepts.

1. What is a database?

   _____

2. Relational databases are organized how?

   _____

3. The separation of the design from the storage of data is know as what?

   _____

4. A table is a what?

   _____

5. The most important difference between a table and a spreadsheet is that a table is _____.
6. At what level is naming uniqueness applied within a database design?
   a. For tables

      _____

   b. For columns

      _____

7. Columns are also known as _____
8. What is a row?

   _____

9. What purpose does a primary key serve?

   _____

CHAPTER 2 – DATABASE MODELING CONCEPTS

WHAT IS DATA MODELING?

Data modeling is the act of exploring data-oriented structures. Like other modeling artifacts data models can be used for a variety of purposes, from high-level conceptual models to physical data models. From the point of view of an object-oriented developer data modeling is conceptually similar to class modeling. With data modeling, you identify entity types whereas with class modeling you identify classes. Data attributes are assigned to entity types just as you would assign attributes and operations to classes. There are associations between entities, similar to the associations between classes – relationships, inheritance, composition, and aggregation are all applicable concepts in data modeling.

Traditional data modeling is different from class modeling because it focuses solely on data – class models allow you to explore both the behavior and data aspects of your domain, with a data model you can only explore data issues. Because of this focus data modelers tend to be much better at getting the data "right" than object modelers. However, some people will model database methods (stored procedures, stored functions, and triggers) when they are physical data modeling.

Although the focus of this chapter is data modeling, there are often alternatives to data-oriented artifacts (never forget Agile Modeling's Multiple Models principle). For example, when it comes to conceptual modeling ORM diagrams aren't your only option – In addition to LDMs it is quite common for people to create UML class diagrams and even Class Responsibility Collaborator (CRC) cards instead. In fact, my experience is that CRC cards are superior to ORM diagrams because it is very easy to get project stakeholders actively involved in the creation of the model. Instead of a traditional, analyst-led drawing session you can instead facilitate stakeholders through the creation of CRC cards.

HOW ARE DATA MODELS USED IN PRACTICE?

Although methodology issues are covered later, we need to discuss how data models can be used in practice to better understand them. You are likely to see three basic styles of data model:

- Conceptual data models. These models, sometimes called domain models, are typically used to explore domain concepts with project stakeholders. On Agile teams, high-level conceptual models are often created as part of your initial requirements envisioning efforts as they are used to explore the high-level static business structures and concepts. On traditional teams, conceptual data models are often created as the precursor to LDMs or as alternatives to LDMs.

- Logical data models (LDMs). LDMs are used to explore the domain concepts, and their relationships, of your problem domain. This could be done for the scope of a single project or for your entire enterprise. LDMs depict the logical entity types, typically referred to simply as entity types, the data attributes describing those entities, and the relationships between the entities. LDMs are rarely used on Agile projects although often are on traditional projects (where they rarely seem to add much value in practice).
- Physical data models (PDMs). PDMs are used to design the internal schema of a database, depicting the data tables, the data columns of those tables, and the relationships between the tables. PDMs often prove to be useful on both Agile and traditional projects and thus the focus of this chapter is on physical modeling.

The table below compares the different features:

Feature	*Conceptual*	*Logical*	*Physical*
Entity Names	✓	✓	
Entity Relationships	✓	✓	
Attributes		✓	
Primary Keys		✓	✓
Foreign Keys		✓	✓
Table Names			✓
Column Names			✓
Column Data Types			✓

Below we show the conceptual, logical, and physical versions of a single data model.

CONCEPTUAL DATA MODEL

A conceptual data model identifies the highest-level relationships between the different entities. Features of conceptual data model include:

- Includes the important entities and the relationships among them.
- No attribute is specified.
- No primary key is specified.

LOGICAL DATA MODEL

A logical data model describes the data in as much detail as possible, without regard to how they will be physical implemented in the database. Features of a logical data model include:

- Includes all entities and relationships among them.
- All attributes for each entity are specified.
- The primary key for each entity is specified.
- Foreign keys (keys identifying the relationship between different entities) are specified.
- Normalization occurs at this level.

The steps for designing the logical data model are as follows:

1. Specify primary keys for all entities.
2. Find the relationships between different entities.
3. Find all attributes for each entity.
4. Resolve many-to-many relationships.
5. Normalization.

PHYSICAL DATA MODEL

Physical data model represents how the model will be built in the database. A physical database model shows all table structures, including column name, column data type, column constraints, primary key, foreign key, and relationships between tables. Features of a physical data model include:

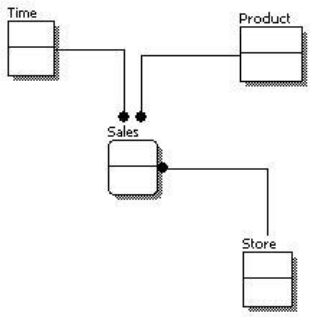
- Specification all tables and columns.
- Foreign keys are used to identify relationships between tables.
- Denormalization may occur based on user requirements.
- Physical considerations may cause the physical data model to be quite different from the logical data model.
- Physical data model will be different for different RDBMS. For example, data type for a column may be different between MySQL and SQL Server.

The steps for physical data model design are as follows:

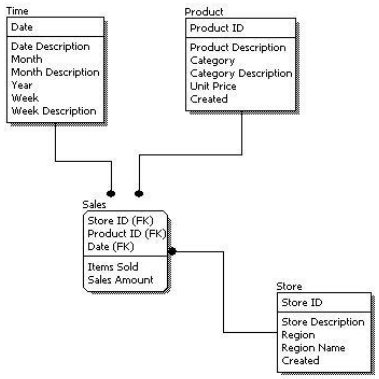
1. Convert entities into tables.
2. Convert relationships into foreign keys.
3. Convert attributes into columns.

4. Modify the physical data model based on physical constraints / requirements.

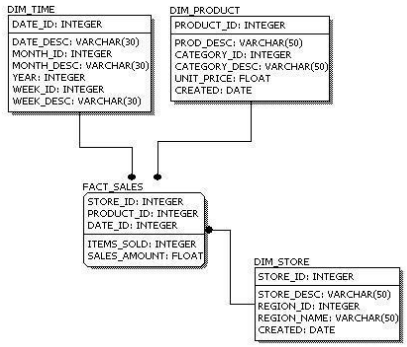
Conceptual Model Design



Logical Model Design



Physical Model Design



We can see that the complexity increases from conceptual to logical to physical. Therefore, we always first start with the conceptual data model (so we understand at high level what are the different entities in our data and how they relate to one another), then move on to the logical data model (so we understand the details of our data without worrying about how they will be implemented), and finally the physical data model (so we know exactly how to implement our data model in the database of choice). In a data warehousing project, sometimes the conceptual data model and the logical data model are considered as a single deliverable.

CONCEPTUAL AND LOGICAL TERMINOLOGY

In database design, there is various terminology that is used. Below we will define the various term involved in conceptual and logical modeling.

ENTITY TYPES / ENTITY

An entity type, also simply called entity (not exactly accurate terminology, but very common in practice), is similar conceptually to object-orientation's concept of a class – an entity type represents a collection of similar objects. An entity type could represent a collection of people, places, things, events, or concepts. Examples of entities in an order entry system would include Customer, Address, Order, Item, and Tax. If you were class modeling you would expect to discover classes with the exact same names. However, the

difference between a class and an entity type is that classes have both data and behavior whereas entity types just have data.

Ideally an entity should be normal, the data modeling world's version of cohesive. A normal entity depicts one concept, just like a cohesive class models one concept. For example, customer and order are clearly two different concepts; therefore, it makes sense to model them as separate entities. An entity should also be an object that will have many instances in the database that is composed of multiple attributes.

Here are some specific definitions.

- **Entity type / Entity** – a person, a place, an object, an event, or a concept entity that share common properties or characteristics in the user environment about which the organization wishes to maintain data
- **Entity instance** – A single occurrence of an entity type i.e. Entity would be student, Entity Instance is a single student.

STRONG ENTITY VS. WEAK ENTITY

STRONG ENTITY

The Strong Entity is the one whose existence does not depend on the existence of any other entity in a schema. It is denoted by a single rectangle. A strong entity always has the primary key in the set of attributes that describes the strong entity. It indicates that each entity in a strong entity set can be uniquely identified.

A strong entity holds the relationship with the weak entity via an Identifying Relationship.

WEAK ENTITY

A Weak entity is the one that depends on its owner entity i.e. a strong entity for its existence. A weak entity is denoted by the double rectangle. Weak entity does not have the primary key instead it has a partial key that uniquely discriminates the weak entities. The primary key of a weak entity is a composite key formed from the primary key of the strong entity and partial key of the weak entity.

The collection of similar weak entities is called Weak Entity Set. The relationship between a weak entity and a strong entity is always denoted with an Identifying Relationship.

ATTRIBUTES

Each entity type will have one or more data attributes. For example, in Figure 1 you saw that the Customer entity has attributes such as First Name and Surname and in Figure 2 that the TCUSTOMER table had corresponding data columns CUST_FIRST_NAME and CUST_SURNAME (a column is the implementation of a data attribute within a relational database).

Attributes should also be cohesive from the point of view of your domain, something that is often a judgment call. – in Figure 1 we decided that we wanted to model the fact that people had both first and last names instead of just a name (e.g. “Scott" and “Ambler" vs. “Scott Ambler") whereas we did not distinguish between the sections of an American zip code (e.g. 90210-1234-5678). Getting the level of detail right can have a significant impact on your development and maintenance efforts. Refactoring a single data column into several columns can be difficult, database refactoring is described in detail in Database Refactoring, although over-specifying an attribute (e.g. having three attributes for zip code when you only needed one) can result in overbuilding your system and hence you incur greater development and maintenance costs than you actually needed.

There are several types of attributes that you have to take into account when creating a design. These include:

REQUIRED VERSUS OPTIONAL ATTRIBUTES

A required attribute is an attribute that is absolutely necessary to define an entity. For example, a required attribute may include a person’s name or social insurance number. On the other hand, an optional attribute is an attribute that helps to define the entity, but is not needed for all business cases. An example would be a fax number or an email address.

SIMPLE VERSUS COMPOSITE ATTRIBUTE

A simple attribute is a regular atomic piece of data. It is not made up of anything else but a single piece of information. An example would be a date of birth or a student number.

A composite attribute is an attribute made up of multiple pieces. The best example of this is an address. It is made up from a street address, city, province, postal code. When modeling at the conceptual and logical level, it is fine to simply refer to this as a single attribute. However, when creating the physical diagram, you should break down this attribute into its component pieces.

SINGLE-VALUED VERSUS MULTIVALUED ATTRIBUTE

A single valued attribute is the same as a simple attribute. It is an attribute that contains a single value.

A multivalued attribute contains a list of values. This is also known as a repeating group of values. An example of this could be a list of skills an employee has or the products in an order. Multi valued attributes will be broken down into its own table during the normalization process.

STORED VERSUS DERIVED ATTRIBUTES

A stored attribute is an attribute that actually contains data. Examples of this would be a date of birth, the quantity being purchased or the selling price of an item.

Alternatively, a derived attribute is an attribute whose value can be calculated using other attributes. These are often not included in the physical design. The only time that these would be included would be for performance gains or auditing purposes.

Examples of derived attributes would include:

- Age – can be calculated by substracting date of birth from today's date
- Line total – can be calculated by multiplying the quantity with the sell price
- Order total – can be calculated by adding up all of the line totals

IDENTIFIER ATTRIBUTES

Identifier attributes are attributes that can be used to unique identify a single instance of an entity. An identifier can be made up of one or more attributes. If it is made up of more than one attribute, it is known as a composite identifier.

Before being formalized into a primary key, identifiers are known as candidate keys.

KEYS

There are several types of keys in database design. These include the following.

CANDIDATE KEYS

A candidate key is an attribute that could be an identifier since it satisfies the requirements for being an identifier.

PRIMARY KEYS

Primary keys are the candidate keys that are chosen as the key that will actually be used by the DBMS to uniquely identify each row in an entity.

FOREIGN KEYS

Foreign keys are attributes that is a key of one or more entities other than the one in which it appears. Essentially, they are an attribute whose value is found in the primary key of another entity.

SURROGATE KEYS

Surrogate keys are columns that have a unique DBMS assigned identifier that has been added to a table to be the primary key. The values of the surrogate key are assigned automatically by the DBMS each time a row is added to the entity and the values never change. Often these keys are numeric and are ideal as primary keys as they are completely isolated from the outside world. Real world examples of this would include taking a number to stand in line or a lap clicker.

RELATIONSHIPS

A relationship works by matching data in key columns — usually columns with the same name in both tables. In most cases, the relationship matches the primary key from one table, which provides a unique identifier for each row, with an entry in the foreign key in the other table. There are three types of relationships between tables. The type of relationship that is created depends on how the related columns are defined.

- One-to-Many Relationships
- Many-to-Many Relationships
- One-to-One Relationships

TYPES OF RELATIONSHIPS

ONE-TO-MANY RELATIONSHIPS

A one-to-many relationship is the most common type of relationship. In this type of relationship, a row in table A can have many matching rows in table B, but a row in table B can have only one matching row in table A. For example, the publishers and titles tables have a one-to-many relationship: each publisher produces many titles, but each title comes from only one publisher.

A one-to-many relationship is created if only one of the related columns is a primary key or has a unique constraint.

The primary key side of a one-to-many relationship is denoted by a key symbol. The foreign key side of a relationship is denoted by an infinity symbol.

MANY-TO-MANY RELATIONSHIPS

In a many-to-many relationship, a row in table A can have many matching rows in table B, and vice versa. You create such a relationship by defining a third table, called a junction table, whose primary key consists of the foreign keys from both table A and table B. For example, the authors table and the titles table have a many-to-many relationship that is defined by a one-to-many relationship from each of these tables to the titleauthors table. The primary key of the titleauthors table is the combination of the au_id column (the authors table's primary key) and the title_id column (the titles table's primary key).

ONE-TO-ONE RELATIONSHIPS

In a one-to-one relationship, a row in table A can have no more than one matching row in table B, and vice versa. A one-to-one relationship is created if both of the related columns are primary keys or have unique constraints.

This type of relationship is not common because most information related in this way would be all in one table. You might use a one-to-one relationship to:

- Divide a table with many columns.
- Isolate part of a table for security reasons.
- Store data that is short-lived and could be easily deleted by simply deleting the table.
- Store information that applies only to a subset of the main table.

The primary key side of a one-to-one relationship is denoted by a key symbol. The foreign key side is also denoted by a key symbol.

CARDINALITY AND OPTIONALITY

You also need to identify the cardinality and optionality(modality) of a relationship (the UML combines the concepts of optionality and cardinality into the single concept of multiplicity).

Cardinality represents the concept of “how many" whereas optionality represents the concept of “whether you must have something." For example, it is not enough to know that customers place orders. How

many orders can a customer place? None, one, or several? Furthermore, relationships are two-way streets: not only do customers place orders, but orders are placed by customers. This leads to questions like: how many customers can be enrolled in any given order and is it possible to have an order with no customer involved?

Optionality (modality) refers to the minimum number of times an instance in one entity can be associated with an instance in the related entity.

Cardinality can be 1 or Many and the symbol is placed on the outside ends of the relationship line, closest to the entity, Modality can be 1 or 0 and the symbol is placed on the inside, next to the cardinality symbol. For a cardinality of 1 a straight line is drawn. For a cardinality of Many a foot with three toes is drawn. For a modality of 1 a straight line is drawn. For a modality of 0 a circle is drawn.

Cardinality and modality are indicated at both ends of the relationship line. Once this has been done, the relationships are read as being 1 to 1 (1:1), 1 to many (1:M), or many to many (M:M).

	zero or more
	1 or more
	1 and only 1 (exactly 1)
	zero or 1

REVIEW

Please take the time to fill in these review questions to help cement your understanding of these concepts.

1. These models, sometimes called domain models, are typically used to explore domain concepts with project stakeholders.

   _____

2. Logical data models depict what?

   _____

3. Which models are used to design the internal schema of a database?

   _____

4. In the conceptual data model are primary keys defined?
   Yes / No
5. Normalization occurs at what diagramming level?

   _____

6. A normal entity depicts how many concepts?

   _____

7. Should an entity only ever contain one entity instance?
   Yes / No
8. What type of entity depends on it's owner?

   _____

9. A attribute made up of multiple parts is know as a?

   _____

10. Multi-valued attributes are lists of values. Should they be broken down into their own tables as part of the design process?
    Yes / No

11. An attribute whose value can be derived mathematically from another attribute is known as?

    _____

12. What kind of attribute is used to identify a single instance of an entity type?

    _____

13. Surrogate keys are based on real world data?
    Yes / No
14. The most common type of relationship is?

    _____

CHAPTER 3 - NORMALIZATION

INTRODUCTION

Database normalization is process used to organize a database into tables and columns. The idea is that a table should be about a specific topic and that only those columns which support that topic are included. For example, a spreadsheet containing information about sales people and customers serves several purposes:

- Identify sales people in your organization
- List all customers your company calls upon to sell product
- Identify which sales people call on specific customers.

By limiting a table to one purpose you reduce the number of duplicate data that is contained within your database, which helps eliminate some issues stemming from database modifications. To assist in achieving these objectives, some rules for database table organization have been developed. The stages of organization are called normal forms; there are three normal forms most databases adhere to using. As tables satisfy each successive normalization form, they become less prone to database modification anomalies and more focused toward a sole purpose or topic. Before we move on be sure you understand the definition of a database table.

REASONS FOR NORMALIZATION

There are three main reasons to normalize a database. The first is to minimize duplicate data, the second is to minimize or avoid data modification issues, and the third is to simplify queries. As we go through the various states of normalization we'll discuss how each form addresses these issues, but to start, let's look at some data which hasn't been normalized and discuss some potential pitfalls. Once these are understood, I think you'll better appreciate the reason to normalize the data. Consider the following table:

SalesStaff						
EmployeeID	**SalesPerson**	**SalesOffice**	**OfficeNumber**	**Customer1**	**Customer2**	**Customer3**
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
1004	John Hunt	New York	212-555-1212	Dell	HP	Apple
1005	Martin Hap	Chicago	312-555-1212	Boeing		

Note: The primary key columns are underlined

The first thing to notice is this table serves many purposes including:

- Identifying the organization's salespeople

- Listing the sales offices and phone numbers
- Associating a salesperson with an sales office
- Showing each salesperson's customers

As a DBA this raises a red flag. In general, I like to see tables that have one purpose. Having the table serve many purposes introduces many of the challenges; namely, data duplication, data update issues, and increased effort to query data.

DATA DUPLICATION AND MODIFICATION ANOMALIES

Notice that for each SalesPerson we have listed both the SalesOffice and OfficeNumber. This information is duplicated for each SalesPerson. Duplicated information presents two problems:

- It increases storage and decrease performance.
- It becomes more difficult to maintain data changes.

For example:

- Consider if we move the Chicago office to Evanston, IL. To properly reflect this in our table, we need to update the entries for all the SalesPersons currently in Chicago. Our table is a small example, but you can see if it were larger, that potentially this could involve hundreds of updates.
- Also consider what would happen if John Hunt quits. If we remove his entry, then we lose the information for New York.

These situations are modification anomalies. There are three modification anomalies that can occur:

INSERT ANOMALY

There are facts we cannot record until we know information for the entire row. In our example we cannot record a new sales office until we also know the sales person. Why? Because in order to create the record, we need provide a primary key. In our case this is the EmployeeID.

EmployeeID	SalesPerson	SalesOffice	OfficeNumber	Customer1	Customer2	Customer3
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
1004	John Hunt	New York	212-555-1212	Dell	HP	Apple
1005	Martin Hap	Chicago	312-555-1212	Boeing		
???	???	Atlanta	312-555-1212			

UPDATE ANOMALY

The same information is recorded in multiple rows. For instance if the office number changes, then there are multiple updates that need to be made. If these updates are not successfully completed across all rows, then an inconsistency occurs.

EmployeeID	SalesPerson	SalesOffice	OfficeNumber	Customer1	Customer2	Customer3
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
1004	John Hunt	New York	212-555-1212	Dell	HP	Apple
1005	Martin Hap	Chicago	312-555-1212	Boeing		

DELETION ANOMALY

Deletion of a row can cause more than one set of facts to be removed. For instance, if John Hunt retires, then deleting that row cause use to lose information about the New York office.

EmployeeID	SalesPerson	SalesOffice	OfficeNumber	Customer1	Customer2	Customer3
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
~~1004~~	~~John Hunt~~	~~New York~~	~~212-555-1212~~	~~Dell~~	~~HP~~	~~Apple~~
1005	Martin Hap	Chicago	312-555-1212	Boeing		

SEARCH AND SORT ISSUES

The last reason we'll consider is making it easier to search and sort your data. In the SalesStaff table if you want to search for a specific customer such as Ford, you would have to write a query like

```
SELECT SalesOffice
FROM SalesStaff
WHERE Customer1 = 'Ford' OR
      Customer2 = 'Ford' OR
      Customer3 = 'Ford'
```

Clearly if the customer were somehow in one column our query would be simpler. Also, consider if you want to run a query and sort by customer. The way the table is currently defined, this isn't possible, unless you use three separate queries with a UNION. These anomalies can be eliminated or reduced by properly separating the data into different tables, to house the data in tables which serve a single purpose. The process to do this is called normalization, and the various stages you can achieve are called the normal forms.

DEFINITION OF NORMALIZATION

There are three common forms of normalization: 1st, 2nd, and 3rd normal form. There are several additional forms, such as BCNF, but I consider those advanced, and not too necessary to learn in the beginning. The forms are progressive, meaning that to qualify for 3rd normal form a table must first satisfy the rules for 2nd normal form, and 2nd normal form must adhere to those for 1st normal form. Before we discuss the various forms and rules in details, let's summarize the various forms:

- **First Normal Form** – The information is stored in a relational table and each column contains atomic values, and there are not repeating groups of columns.
- **Second Normal Form** – The table is in first normal form and all the columns depend on the table's primary key.
- **Third Normal Form** – the table is in second normal form and all of its columns are not transitively dependent on the primary key

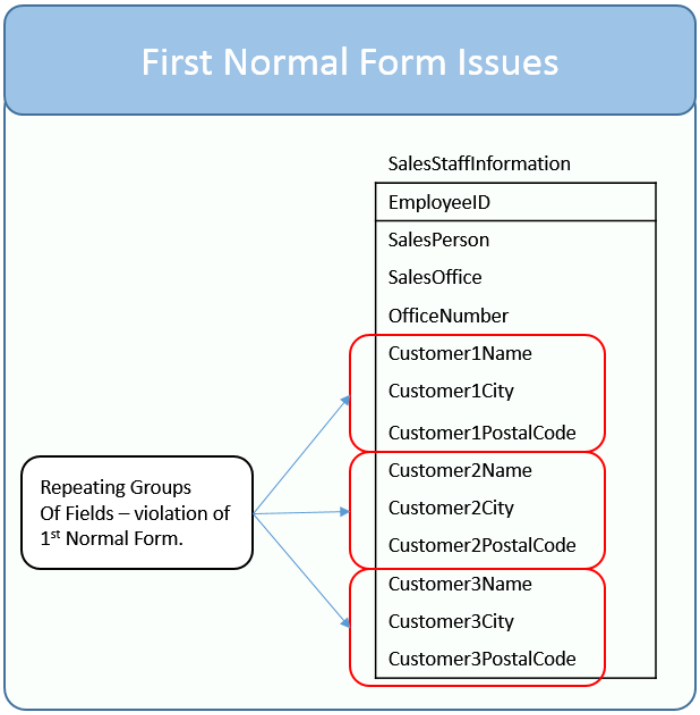
FIRST NORMAL FORM

The first steps to making a proper SQL table is to ensure the information is in first normal form. Once a table is in first normal form it is easier to search, filter, and sort the information. The rules to satisfy 1st normal form are:

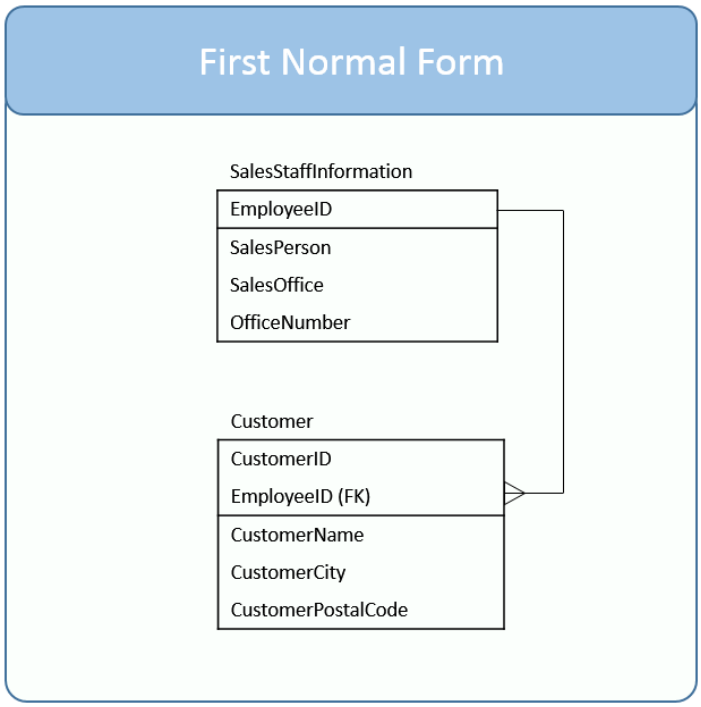
- That the data is in a database table. The table stores information in rows and columns where one or more columns, called the primary key, uniquely identify each row.
- Each column contains atomic values, and there are not repeating groups of columns.

Tables in first normal form cannot contain sub columns. That is, if you are listing several cities, you cannot list them in one column and separate them with a semi-colon.

When a value is atomic, the values cannot be further subdivided. For example, the value "Chicago" is atomic; whereas "Chicago; Los Angeles; New York" is not. Related to this requirement is the concept that a table should not contain repeating groups of columns such as Customer1Name, Customer2Name, and Customer3Name.



Our example table is transformed to first normal form by placing the repeating customer related columns into their own table. This is shown below:



The repeating groups of columns now become separate rows in the Customer table linked by the EmployeeID foreign key. As mentioned in the lesson on Data Modeling, a foreign key is a value which

matches back to another table's primary key. In this case, the customer table contains the corresponding EmployeeID for the SalesStaffInformation row. Here is our data in first normal form.

SalesStaffInformation			
EmployeeID	**SalesPerson**	**SalesOffice**	**OfficeNumber**
1003	Mary Smith	Chicago	312-555-1212
1004	John Hunt	New York	212-555-1212
1005	Martin Hap	Chicago	312-555-1212

Note: Primary Key: EmployeeID

Customer				
CustomerID	**EmployeeID**	**CustomerName**	**CustomerCity**	**PostalCode**
C1000	1003	Ford	Dearborn	48123
C1010	1003	GM	Detroit	48213
C1020	1004	Dell	Austin	78720
C1030	1004	HP	Palo Alto	94303
C1040	1004	Apple	Cupertino	95014
C1050	1005	Boeing	Chicago	60601

This design is superior to our original table in several ways:

1. The original design limited each SalesStaffInformation entry to three customers. In the new design, the number of customers associated to each design is practically unlimited.
2. It was nearly impossible to Sort the original data by Customer. You could, if you used the UNION statement, but it would be cumbersome. Now, it is simple to sort customers.
3. The same holds true for filtering on the customer table. It is much easier to filter on one customer name related column than three.
4. The insert and deletion anomalies for Customer have been eliminated. You can delete all the customer for a SalesPerson without having to delete the entire SalesStaffInformaiton row.

Modification anomalies remain in both tables, but these are fixed once we reorganize them as 2nd normal form.

SECOND NORMAL FORM

I like to think the reason we place tables in 2nd normal form is to narrow them to a single purpose. Doing so bring's clarity to the database design, makes it easier for us to describe and use a table, and tends to eliminate modification anomalies.

This stems from the primary key identifying the main topic at hand, such as identifying buildings, employees, or classes, and the columns, serving to add meaning through descriptive attributes.

An EmployeeID isn't much on its own, but add a name, height, hair color and age, and now you're starting to describe a real person.

2NF – SECOND NORMAL FORM DEFINITION

A table is in Second Normal Form if:

- The table is in 1st normal form, and
- All the non-key columns are dependent on the table's primary key.

We already know about the 1st normal form, but what about the second requirement? Let me try to explain.

The primary key provides a means to uniquely identify each row in a table. When we talk about columns depending on the primary key, we mean, that in order to find a particular value, such as what color is Kris' hair, you would first have to know the primary key, such as an EmployeeID, to look up the answer.

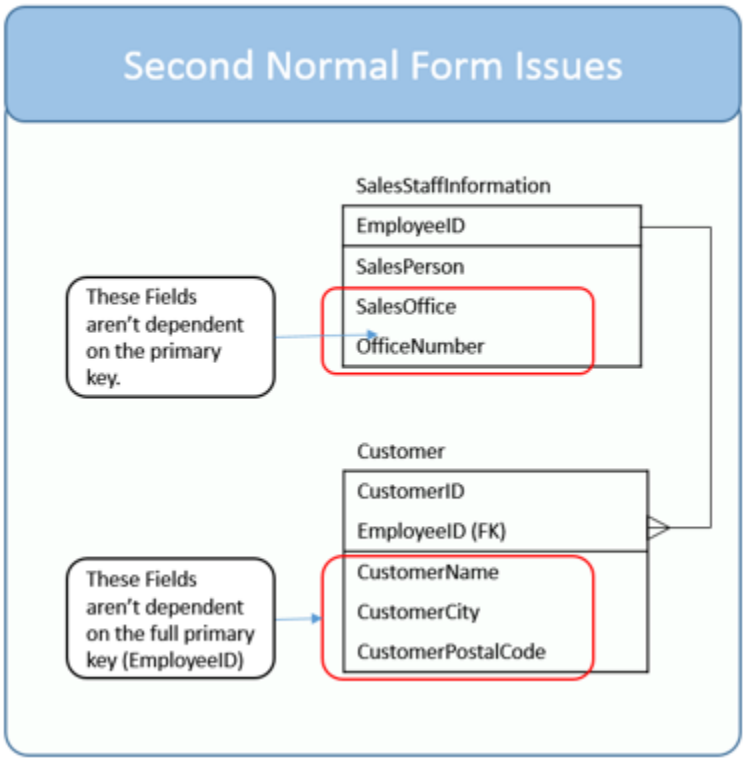
Once you identify a table's purpose, then look at each of the table's columns and ask yourself, "Does this column serve to describe what the primary key identifies?"

- If you answer "yes," then the column is dependent on the primary key and belongs in the table.
- If you answer "no," then the column should be moved different table.

When all the columns relate to the primary key, they naturally share a common purpose, such as describing an employee. That is why I say that when a table is in second normal form, it has a single purpose, such as storing employee information.

So far, we have taken our example to the first normal form, and it has several issues.

The first issue is the SalesStaffInformation table has two columns which aren't dependent on the EmployeeID. Though they are used to describe which office the SalesPerson is based out of, the SalesOffice and OfficeNumber columns themselves don't serve to describe who the employee is.



The second issue is that there are several attributes which don't completely rely on the entire Customer table primary key. For a given customer, it doesn't make sense that you should have to know both the CustomerID and EmployeeID to find the customer.

It stands to reason you should only need to know the CustomerID. Given this, the Customer table isn't in 2nd normal form as there are columns that aren't dependent on the full primary key. They should be moved to another table.

These issues are identified on the right in red.

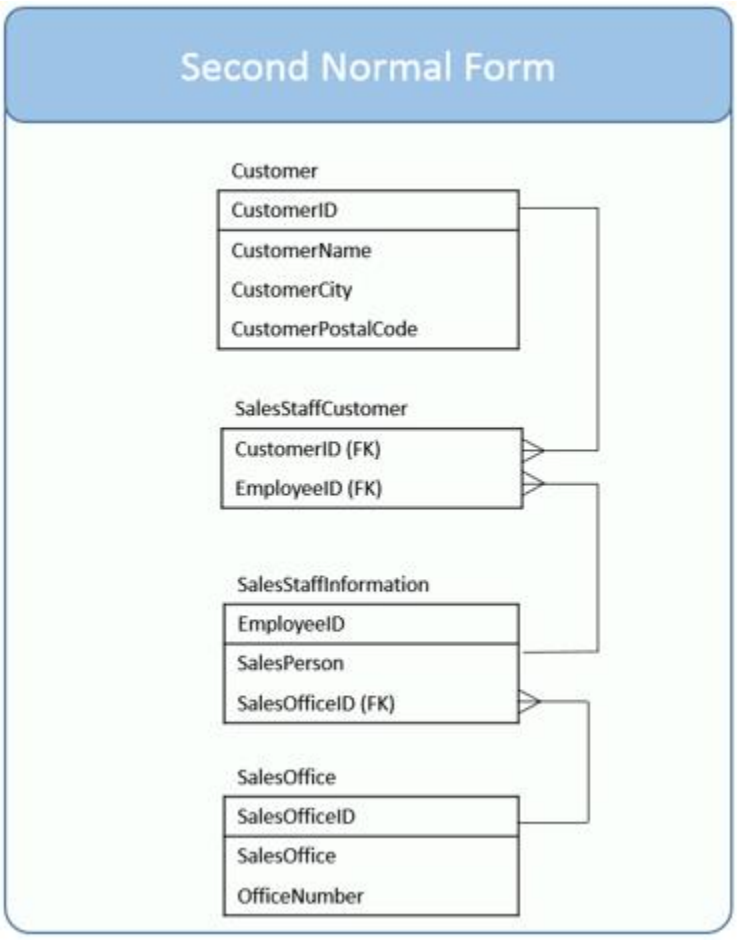
FIX THE MODEL TO 2NF STANDARDS

Since the columns identified in red aren't completely dependent on the table's primary key, it stands to reason they belong elsewhere. In both cases, the columns are moved to new tables.

In the case of SalesOffice and OfficeNumber, a SalesOffice was created. A foreign key was then added to SalesStaffInformaiton so we can still describe in which office a sales person is based.

The changes to make Customer a second normal form table are a little trickier. Rather than move the offending columns CustomerName, CustomerCity, and CustomerPostalCode to new table, recognize that the issue is EmployeeID! The three columns don't depend on this part of the key. Really this table is trying to serve two purposes:

- To indicate which customers are called upon by each employee
- To identify customers and their locations.



For the moment remove EmployeeID from the table. Now the table's purpose is clear, it is to identify and describe each customer.

Now let's create a table named SalesStaffCustomer to describe which customers a sales person calls upon. This table has two columns CustomerID and EmployeeID. Together, they form a primary key. Separately, they are foreign keys to the Customer and SalesStaffInformation tables respectively.

With these changes made the data model, in second normal form, is shown to the right.

To better visualize this, here are the tables with data.

Customer			
CustomerID	**CustomerName**	**CustomerCity**	**CustomerPostalCode**
C1000	Ford	Dearborn	48123
C1010	GM	Detroit	48213
C1020	Dell	Austin	78720
C1030	HP	Palo Alto	94303
C1040	Apple	Cupertino	95014
C1050	Boeing	Chicago	60601

As you review the data in the tables notice that the redundancy is mostly eliminated. Also, see if you can find any update, insert, or deletion anomalies. Those too are gone. You can now eliminate all the sales people, yet retain customer records. Also, if all the SalesOffices close, it doesn't mean you have to delete the records containing sales people.

SalesStaffCustomer	
CustomerID	**EmployeeID**
C1000	1003
C1010	1003
C1020	1004
C1030	1004
C1040	1004
C1050	1005

The SalesStaffCustomer table is a strange one. It's just all keys! This type of table is called an associative table. An associative table is useful when you need to model a many-to-many relationship.

Each column is a foreign key. If you look at the data model you'll notice that there is a one to many relationship to this table from SalesStaffInformation and another from Customer. In effect the table allows you to bridge the two tables together.

SalesStaffInformation		
EmployeeID	**SalesPerson**	**SalesOffice**
1003	Mary Smith	S10
1004	John Hunt	S20
1005	Martin Hap	S10

SalesOffice		
SalesOfficeID	**SalesOffice**	**OfficeNumber**
S10	Chicago	312-555-1212
S20	New York	212-555-1212

For all practical purposes this is a pretty workable database. Three out of the four tables are even in third normal form, but there is one table which still has a minor issue, preventing it from being so.

THIRD NORMAL FORM

Once a table is in second normal form, we are guaranteed that every column is dependent on the primary key, or as I like to say, the table serves a single purpose. But what about relationships among the columns? Could there be dependencies between columns that could cause an inconsistency?

A table containing both columns for an employee's age and birth date is spelling trouble, there lurks an opportunity for a data inconsistency!

How are these addressed? By the third normal form.

3NF – THIRD NORMAL FORM DEFINITION

A table is in third normal form if:

- A table is in 2nd normal form.
- It contains only columns that are non-transitively dependent on the primary key

Wow! That's a mouthful. What does non-transitively dependent mean? Let's break it down.

TRANSITIVE

When something is transitive, then a meaning or relationship is the same in the middle as it is across the whole. If it helps think of the prefix trans as meaning "across." When something is transitive, then if something applies from the beginning to the end, it also applies from the middle to the end.

Since ten is greater than five, and five is greater than three, you can infer that ten is greater than three.

In this case, the greater than comparison is transitive. In general, if A is greater than B, and B is greater than C, then it follows that A is greater than C.

If you're having a hard time wrapping your head around "transitive" I think for our purpose it is safe to think "through" as we'll be reviewing to see how one column in a table may be related to others, through a second column.

DEPENDENCE

An object has a dependence on another object when it relies upon it. In the case of databases, when we say that a column has a dependence on another column, we mean that the value can be derived from the other. For example, my age is dependent on my birthday. Dependence also plays an important role in the definition of the second normal form.

TRANSITIVE DEPENDENCE

Now let's put the two words together to formulate a meaning for transitive dependence that we can understand and use for database columns.

I think it is simplest to think of transitive dependence to mean a column's value relies upon another column through a second intermediate column.

Consider three columns: AuthorNationality, Author, and Book. Column values for AuthorNationality and Author rely on the Book; once the book is known, you can find out the Author or AuthorNationality. But also notice that the AuthorNationality relies upon Author. That is, once you know the Author, you can determine their nationality. In this sense then, the AuthorNationality relies upon Book, via Author. This is a transitive dependence.

This can be generalized as being three columns: A, B and PK. If the value of A relies on PK, and B relies on PK, and A also relies on B, then you can say that A relies on PK though B. That is A is transitively dependent on PK.

Let's look at some examples to understand further.

Primary Key (PK)	Column A	Column B	Transitive Dependence?
PersonID	FirstName	LastName	No, In Western cultures a person's last name is based on their father's LastName, whereas their FirstName is given to them.
PersonID	BodyMassIndex	IsOverweight	Yes, BMI over 25 is considered overweight.It wouldn't make sense to have the value IsOverweight be true when the BodyMassIndex was < 25.
PersonID	Weight	Sex	No:There is no direct link between the weight of a person and their sex.
VehicleID	Model	Manufacturer	Yes:Manufacturers make specific models. For instance, Ford creates the Fiesta; whereas, Toyota manufacturers the Camry.

To be non-transitively dependent, then, means that all the columns are dependent on the primary key (a criterion for 2nd normal form) and no other columns in the table.

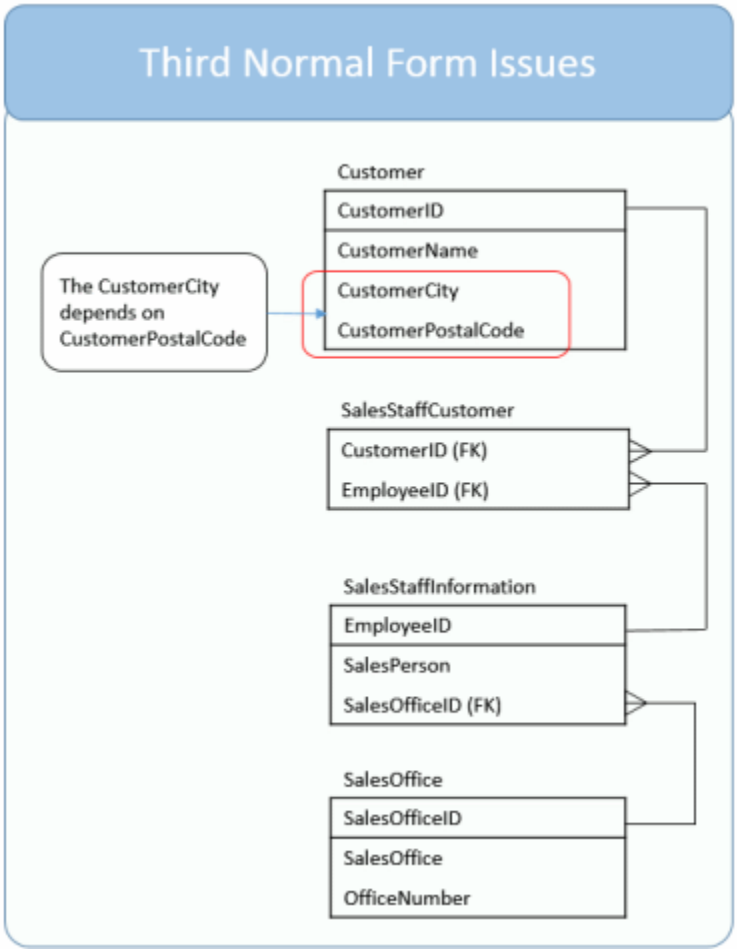
ISSUES WITH OUR EXAMPLE DATA MODEL

Let's review what we have done so far with our database. You'll see that I've found one transitive dependency:

CustomerCity relies on CustomerPostalCode which relies on CustomerID

A postal code applies to one city. Although all the columns are dependent on the primary key, CustomerID, there is an opportunity for an update anomaly as you could update the CustomerPostalCode without making a corresponding update to the CustomerCity.

We've identified this issue in red.



FIX THE MODEL TO 3NF STANDARDS

For our model to be in third normal form, we need to remove the transitive dependencies. As we stated our dependency is:

CustomerCity relies on CustomerPostalCode which relies on CustomerID

It is OK that CustomerPostalCode relies on CustomerID; however, we break 3NF by including CustomerCity in the table. To fix this we'll create a new table, PostalCode, which includes PostalCode as the primary key and City as its sole column.

The CustomerPostalCode remains in the customer table. The CustomerPostalCode can then be designated a foreign key. In this way, through the relation, the city and postal code is still known for each customer. In addition, we've eliminated the update anomaly.

REVIEW

Please take the time to fill in these review questions to help cement your understanding of these concepts.

1. What are the 3 primary reasons to normalize a database design?

    a. _____

    b. _____

    c. _____

2. The situation when you cannot insert a row without knowing the values for the entire row is known as what kind of anomaly?

    _____

3. The situation when same information is recorded in multiple rows is known as what kind of anomaly?

    _____

4. The situation when you lose unique information when a row is removed from the database is known as what kind of anomaly?

    _____

5. The **First Normal Form** states:

    _____

    _____

6. The **Second Normal Form Form** states:

_____

_____

7. The **Third Normal Form Form** states:

_____

_____

CHAPTER 4 – MODELING DATA

It is critical for an application developer to have a grasp of the fundamentals of data modeling so they can not only read data models but also work effectively with DBAs who are responsible for the data-oriented aspects of your project. Your goal reading this section is not to learn how to become a data modeler, instead it is simply to gain an appreciation of what is involved.

The following tasks are performed in an iterative manner:

1. Identify entity types
2. Identify attributes
3. Apply naming conventions
4. Identify relationships
5. Apply data model patterns
6. Assign keys
7. Normalize to reduce data redundancy
8. Denormalize to improve performance

IDENTIFY ENTITY TYPES

An entity type, also simply called entity (not exactly accurate terminology, but very common in practice), is similar conceptually to object-orientation's concept of a class – an entity type represents a collection of similar objects. An entity type could represent a collection of people, places, things, events, or concepts. Examples of entities in an order entry system would include Customer, Address, Order, Item, and Tax. If you were class modeling you would expect to discover classes with the exact same names. However, the difference between a class and an entity type is that classes have both data and behavior whereas entity types just have data.

Ideally an entity should be normal, the data modeling world's version of cohesive. A normal entity depicts one concept, just like a cohesive class models one concept. For example, customer and order are clearly two different concepts; therefore, it makes sense to model them as separate entities.

IDENTIFY ATTRIBUTES

Each entity type will have one or more data attributes. Attributes should also be cohesive from the point of view of your domain, something that is often a judgment call. For example, we could decide that we wanted to model the fact that people had both first and last names instead of just a name (e.g. "Scott" and

"Ambler" vs. "Scott Ambler") whereas we did not distinguish between the sections of an American zip code (e.g. 90210-1234-5678). Getting the level of detail right can have a significant impact on your development and maintenance efforts. Refactoring a single data column into several columns can be difficult, database refactoring is described in detail in Database Refactoring, although over-specifying an attribute (e.g. having three attributes for zip code when you only needed one) can result in overbuilding your system and hence you incur greater development and maintenance costs than you actually needed.

APPLY DATA NAMING CONVENTIONS

Your organization should have standards and guidelines applicable to data modeling, something you should be able to obtain from your enterprise administrators (if they don't exist you should lobby to have some put in place). These guidelines should include naming conventions for both logical and physical modeling, the logical naming conventions should be focused on human readability whereas the physical naming conventions will reflect technical considerations.

Developers should agree to and follow a common set of modeling standards on a software project. Just like there is value in following common coding conventions, clean code that follows your chosen coding guidelines is easier to understand and evolve than code that doesn't, there is similar value in following common modeling conventions.

The basic rules are as follows:

1. Any naming standard is better than no standard.
2. There is no "one true" standard, we all have our preferences
3. If there is standard already in place, use it. Don't create another standard or muddy the existing standards.

IDENTIFY RELATIONSHIPS

In the real-world entities have relationships with other entities. For example, customers PLACE orders, customers LIVE AT addresses, and line items ARE PART OF orders. Place, live at, and are part of are all terms that define relationships between entities. The relationships between entities are conceptually identical to the relationships (associations) between objects.

You also need to identify the cardinality and optionality of a relationship (the UML combines the concepts of optionality and cardinality into the single concept of multiplicity). Cardinality represents the concept of "how many" whereas optionality represents the concept of "whether you must have something." For example, it is not enough to know that customers place orders. How many orders can a

customer place? None, one, or several? Furthermore, relationships are two-way streets: not only do customers place orders, but orders are placed by customers. This leads to questions like: how many customers can be enrolled in any given order and is it possible to have an order with no customer involved? Figure 5 shows that customers place zero or more orders and that any given order is placed by one customer and one customer only. It also shows that a customer lives at one or more addresses and that any given address has zero or more customers living at it.

APPLY DATA MODEL PATTERNS

Some data modelers will apply common data model patterns. Data model patterns are conceptually closest to analysis patterns because they describe solutions to common domain issues.

ASSIGN KEYS

There are two fundamental strategies for assigning keys to tables. First, you could assign a natural key which is one or more existing data attributes that are unique to the business concept. Second, you could introduce a new column, called a surrogate key, which is a key that has no business meaning

NORMALIZE TO REDUCE DATA REDUNDANCY

Data normalization is a process in which data attributes within a data model are organized to increase the cohesion of entity types. In other words, the goal of data normalization is to reduce and even eliminate data redundancy, an important consideration for application developers because it is incredibly difficult to stores objects in a relational database that maintains the same information in several places. With respect to terminology, a data schema is considered to be at the level of normalization of its least normalized entity type. For example, if all your entity types are at second normal form (2NF) or higher then we say that your data schema is at 2NF.

Why data normalization? The advantage of having a highly-normalized data schema is that information is stored in one place and one place only, reducing the possibility of inconsistent data. Furthermore, highly-normalized data schemas in general are closer conceptually to object-oriented schemas because the object-oriented goals of promoting high cohesion and loose coupling between classes results in similar solutions (at least from a data point of view). This generally makes it easier to map your objects to your data schema. Unfortunately, normalization usually comes at a performance cost.

DENORMALIZE TO IMPROVE PERFORMANCE

Normalized data schemas, when put into production, often suffer from performance problems. This makes sense – the rules of data normalization focus on reducing data redundancy, not on improving performance of data access. An important part of data modeling is to denormalize portions of your data schema to improve database access times.

WHERE TO START

When starting to develop a database several steps need to be taken into account. Assuming you are aware of the target database server, you will need to decide which design tools you will implement. A diagramming tool is an important first step. There are many open source and commercial products availe including, but not limited to, pgDesigner, MySQL Workbench and Sybase PowerArchitect.

The next step would involve determining the best approach to actually create the structure. The following section highlights two of the more common situations.

IDENTIFYING DATA STRUCTURES FROM A REAL-WORLD DOCUMENT

Identifying data structures from a real-world document tends to be the easier route to creating a database structure. Normally, you would take an existing form and a marker and start circling all the pieces that may or may not be data. A more refined approach is to use different colored markers to identify different field types. You would start by identifying the entities and then following up with the attributes.

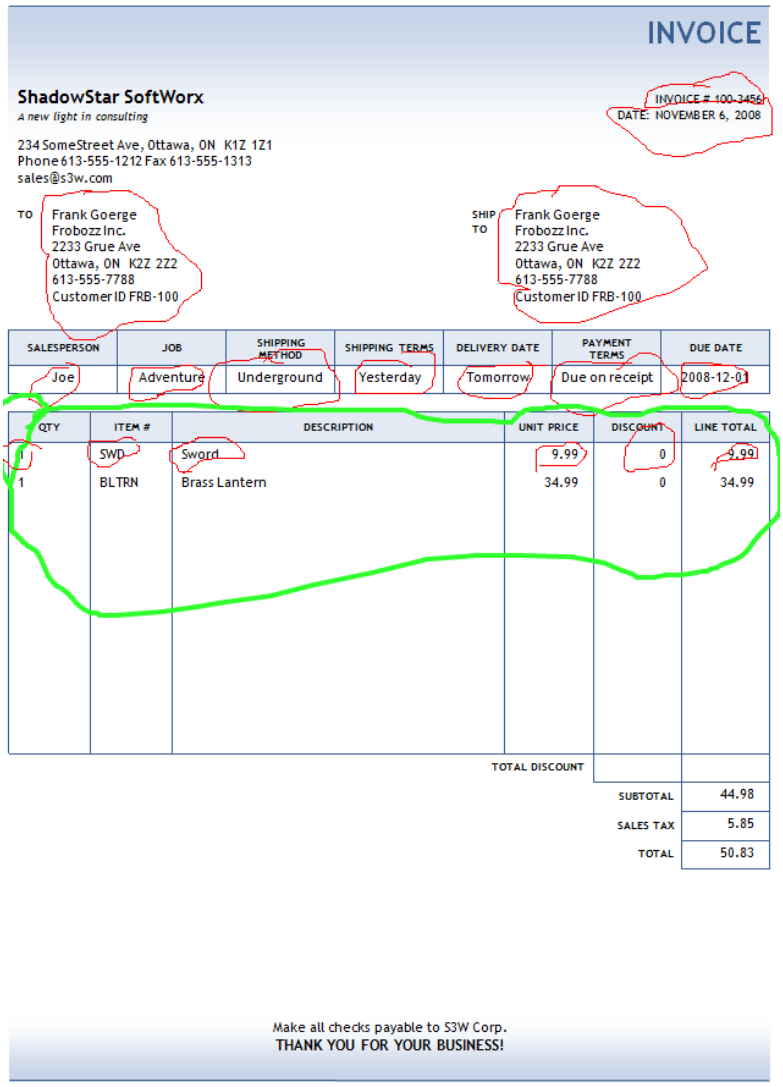
In the following form, an invoice, all the fields have been identified. The singlet pieces of data are identified in red and since an invoice is a master detail type of form, the details area is circled in green. The master record in this case is the invoice as a whole and the details/child record is the line items.

Each of the items circled in red can be broken down into individual items. For example:

- The invoice number
- The date of the invoice
- The sales person
- Etc…

The section circled in green contains line items. The line items are repeated more than once, thus it is a details area of a master detail form. Each detail is made up of several fields. These include:

- Quantity
- Item Number
- Description
- Etc…



INVOICE

ShadowStar SoftWorx
A new light in consulting

234 SomeStreet Ave, Ottawa, ON K1Z 1Z1
Phone 613-555-1212 Fax 613-555-1313
sales@s3w.com

INVOICE # 100-3456
DATE: NOVEMBER 6, 2008

TO
Frank Goerge
Frobozz Inc.
2233 Grue Ave
Ottawa, ON K2Z 2Z2
613-555-7788
Customer ID FRB-100

SHIP TO
Frank Goerge
Frobozz Inc.
2233 Grue Ave
Ottawa, ON K2Z 2Z2
613-555-7788
Customer ID FRB-100

SALESPERSON	JOB	SHIPPING METHOD	SHIPPING TERMS	DELIVERY DATE	PAYMENT TERMS	DUE DATE
Joe	Adventure	Underground	Yesterday	Tomorrow	Due on receipt	2008-12-01

QTY	ITEM #	DESCRIPTION	UNIT PRICE	DISCOUNT	LINE TOTAL
1	SWD	Sword	9.99	0	9.99
1	BLTRN	Brass Lantern	34.99	0	34.99

TOTAL DISCOUNT	
SUBTOTAL	44.98
SALES TAX	5.85
TOTAL	50.83

Make all checks payable to S3W Corp.
THANK YOU FOR YOUR BUSINESS!

In other words each invoice can contain multiple line items. We have now identified 2 entities with multiple fields in each. So for this form we will require at least 1 "entities" form and 2 "fields" forms. However, with further analysis you can find additional tables and fields.

IDENTIFYING DATA STRUCTURES FROM A VAGUE SPECIFICATION

Often you are asked to create a database from a vague verbal description. This can be a very daunting for beginners and inexperienced designers. To help define what needs to be created, there are several "tools" you can employ.

INTERVIEWS

Interviewing is your initial source of information. You should ask your customer for as much detail as possible. Unfortunately, the questions themselves vary from customer to customer. However, there are a few that are common. These include:

- Do you plan to support multiple users?
- Do you plan to have different privilege levels for your end users?
- When defining "objects", try to ask if these "objects" can have a finite number of values. For example, order status and article publish state would be finite lists.
- Try to get as many details as possible for each component of the project.

WHITEBOARD BRAINSTORMING

Whiteboard brainstorming involves taking what your interview produced and exploding them. You can use a whiteboard, pencil and paper or even a mind mapping tool to do this. The usual process includes:

- List a heading for each component. Try to think about as many different components as possible. Bad ideas can be thrown out later.
- For each component, list all tables/entities contained therein.
- For each entity, list all the fields you can think of.
- If a field is a finite list, mark it as so with a star or in a different color.
- Once a field is identified as being a finite list, create an entity for it and list it fields.
- Walk away and have a coffee in another room, or go for a walk in the park. In other words, stop thinking about it.
- Go back and look at the brainstorming session. Does anything look strange, did you think of anything while you weren't thinking about it? Make your revisions and move on to the forms.

RESEARCH

Research usually involves looking at similar products and comparing your brainstorming ideas with what you see in competing products and services.

Once you have defined a general outline of the project and its entities and fields, you would fill out the entities and fields forms.

MAPPING DATA STRUCTURES

MAPPING YOUR DATA

Once you start mapping your data structures, you will need to start assigning data types to them. As straight forward as that sounds, there are a few things that go on at this point. The first is proper field design. The second step is assigning data types to a field and finally, you will need to identify relations between entities.

DATABASE DATA TYPES

Data type	Description
CHARACTER(n)	Character string. Fixed-length n
VARCHAR(n) or CHARACTER VARYING(n)	Character string. Variable length. Maximum length n
BOOLEAN	Stores TRUE or FALSE values
INTEGER(p)	Integer numerical (no decimal). Precision p
SMALLINT	Integer numerical (no decimal). Precision 5
INTEGER	Integer numerical (no decimal). Precision 10
BIGINT	Integer numerical (no decimal). Precision 19
DECIMAL(p,s) **NUMERIC(p,s)**	Exact numerical, precision p, scale s. Example: decimal(5,2) is a number that has 3 digits before the decimal and 2 digits after the decimal
FLOAT(p)	Approximate numerical, mantissa precision p. A floating number in base 10 exponential notation. The size argument for this type consists of a single number specifying the minimum precision
REAL	Approximate numerical, mantissa precision 7

FLOAT	Approximate numerical, mantissa precision 16
DOUBLE PRECISION	Approximate numerical, mantissa precision 16
DATE	Stores year, month, and day values
TIME	Stores hour, minute, and second values
TIMESTAMP	Stores year, month, day, hour, minute, and second values
INTERVAL	Composed of a number of integer fields, representing a period of time, depending on the type of interval

FIELD DESIGN

Field design involves naming a field and assigning it a data type. There are several guidelines you can follow when designing your fields. You should always:

- Make your field names meaningful.
- Decide on a naming convention and stick to it throughout the design.
- Try to make an informed guess as the maximum size the data will be.
- Name your relations in an appropriate manner.

Also, you should try to design your database as close to the ANSI standards as possible because all database systems support ANSI SQL naming conventions and data types. You should try and avoid database system specific items as much as possible. Some basic rules to remember are:

- Keep all names lower case
- Do not use spaces in any field, table or database names. Use underscores instead.
- Stick to basic data types wherever possible.

Sometimes you don't have a choice but use database system specific "extensions" to the ANSI standards. Then include storing GIS, special or network data. In this case, make sure you document where and why you used these data types.

MAPPING DATA TYPES TO FIELDS

When mapping data types to a field, you should try and plan for the future. Changes are always easier in the beginning than later in the design process.

Mapping data types to fields is usually a pretty straightforward process. You would look at some sample data, when available, and simply start identifying it properties.

Things you need to ask as you crawl through the data:

- How long does the field need to be?
- Do I need to plan for slightly larger data?
- Is this data text, numeric or a date or time of some sort?
- When looking at numbers:
- Does the number contain decimal places?
- How many decimal places of precision do I need to worry about?
- Can the number be negative?
- How big can this number get?
- When looking at dates and times, do you need to store both the date and time?

Of course, not all these questions apply to all fields and experience will guide you into making good decisions and speed up the identification process. Once you've identified the properties of your fields, name them and then give them a data type on your database design forms.

As a general guide, you should allow for:

- Addresses: at least 2 address fields with at least 100 characters in length for each.
- Phone numbers:
- at least 10 digits for North American unformatted phone numbers
- If you plan to store formatted numbers allow at least 13 characters.
- You should allow for at least 6 digits for extension numbers
- E-mail address should get at least 100 characters.
- Postal codes should get at least 10 digits if dealing with any outside Canada
- Any body of text longer than 5 words should get a full TEXT(LONGTEXT in MySQL) field.
- Dollar values should have at least 5 digits of precision. You can always round the values in you code later.

IDENTIFYING RELATIONS

When identifying relations, you should always keep an eye out for the "gotchas". These are fields that you think may not need to be controlled but end up require extensive control at a later date.

Lists such as order status or publication category are examples of "relation" targets. When identifying your relations, consider that sometimes some data structures can relate to each other more than once.

For example, the invoice form in the Identifying Data Structures chapter can contain multiple relations to an employee table. These relations could include:

- Who created the order?
- Who was the sale person involved?
- Who shipped the order?
- Who billed the order?

At this point, we've identified at least 4 relations just for a single order. Now comes the tricky part. How are you going to create 4 employee ID fields in 1 table? You are not allowed to have more than 1 field with same name in the same table. Normally you would, either prepend or append the field name with the function of the field.

For example:

- Who created the order? entered_by_employee_id or employee_id_creator
- Who was the saleperson involved? salesrep_employee_id or employee_id_salesrep
- Who shipped the order? shipped_by_employee_id or employee_id_shipped_order
- Who billed the order? order_billed_by_employee_id or employee_id_billed_order

Please note that the above examples are just that example and may not reflect the best naming conventions once could ask for.

RESOLVING MANY TO MANY RELATIONSHIPS

There are 2 kinds of resolutions for many to many relationships. These are:

- An associative entity. These are also known as linking table. They are made up of 2 columns.
- An entity with attributes. This is a straight up table that connects 2 or more tables.

ASSOCIATIVE ENTITIES

In figure 1, you have an example of an associative entity.

It allows for a relationship between part and company. Unfortunately, that is all it is good for. It is a very limited data structure that assumes to normally have no extra data stored. Due to modern regulatory rules and law, often significantly more data needs to be stored.

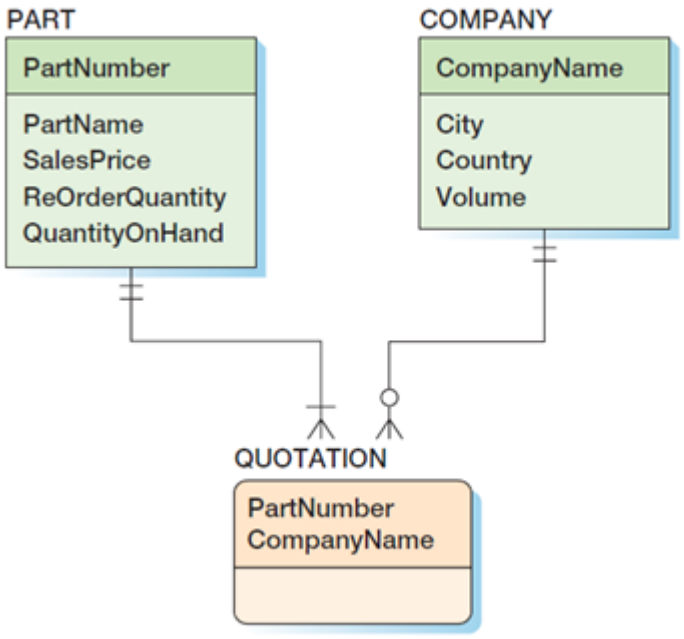


Figure 1

ENTITIES WITH ATTRIBUTES

In Figure 2, then, the relationships between PART and QUOTATION and between COMPANY and QUOTATION are both identifying. This fact is shown in Figure 1 by the solid, non-dashed line that represents these relationships.

As with all identifying relationships, the parent entities are required. Thus, the minimum cardinality from QUOTATION to PART is one, and the minimum cardinality from QUOTATION to COMPANY also is one. The minimum cardinality in the opposite direction is determined by business requirements. Here, a PART must have a QUOTATION, but a COMPANY need not have a QUOTATION.

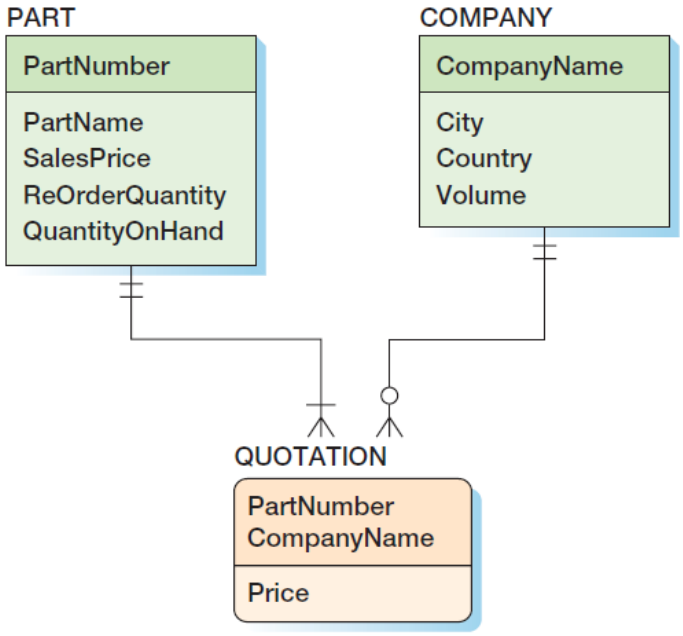


Figure 2

NATURAL VS. SURROGATE KEYS

Natural versus Surrogate keys is a topic of some debate in the database industry. Although Natural keys have "meaning", they can be difficult to manage due to data collection and potential data entry issues. Natural keys often end up being compound keys. Compound keys make programming both the application and the database calls somewhat more complex.

Surrogate keys suffer from a different problem. They have no meaning outside of the database, thus the values are "alien" to most workflows. Surrogate keys have some very specific advantage.

- They are automatically generated and thus are not prove to data entry errors
- They are often numeric. Numeric keys are easier to index and tend to have much faster retrieval rates.
- They are easier to search against. Surrogate keys are normally single field.
- They allow for a simpler naming convention.

In recent years, it has become preferable to use Surrogate keys due to reasons listed above.

CHAPTER 5 – SQL PRIMER

While there are plenty of good tutorials for SQL on the Internet, I didn't find anything that set the up the historical context and "assumed understanding" of SQL very well, so I decided to write a short primer on the topic. Here's what we'll cover:

- What is SQL?
- The SQL programmng language
- Databases and SQL
- Schemas and queries

WHAT IS SQL?

At this point, SQL, which stands for Structured Query Language, is several different things, and at any moment someone saying "SQL" (which is variously pronounced "sequel" or "ess-que-ell") might mean any or several of them. I can think of at least three distinct things which people refer to as SQL in common usage:

- The SQL standard, a massive document produced by the ANSI standards committee, which can be yours for the low, low price of only $60! Useless, especially since it hasn't really changed in 30 years.
- The programming language described by the SQL standard, which people use to talk to databases.
- The various databases (Oracle, Microsoft, MySQL, PostgreSQL, etc.) that IMPLEMENT the SQL programming language.

Most of the time, we'll be interested in the second meaning of the word -- the SQL programming language. However, it's important to realize that this programming language is a sort of platonic ideal of what SQL is; few database programs are entirely compliant with the SQL standard. However, this almost always doesn't matter and most of the time SQL commands that work in one database will work in another.

THE SQL PROGRAMMING LANGUAGE

SQL is a DECLARATIVE programming language intended to express RELATIONAL ALGEBRA queries in a syntax similar to NATURAL LANGUAGE. I'll go over each of these bullet points below:

- SQL is, unusually among programming languages, DECLARATIVE [1]. When using SQL, you issue commands (declare things) and let the database itself figure out how to actually do what you said. You don't run around telling the computer HOW to find the data you want -- you just specify what you want and let the database bring it to you. Idiomatic SQL is almost always declarative; if you find yourself getting too bogged down in how to get your data, you're probably doing something wrong.
- SQL is a natural-language papering over the mathematics of relational algebra. In typically helpful form, Wikipedia explains that relational algebra is "AN OFFSHOOT OF FIRST-ORDER LOGIC AND OF ALGEBRA OF SETS CONCERNED WITH OPERATIONS OVER FINITARY RELATIONS, USUALLY MADE MORE CONVENIENT TO WORK WITH BY IDENTIFYING THE COMPONENTS OF A TUPLE BY A NAME (CALLED ATTRIBUTE) RATHER THAN BY A NUMERIC COLUMN INDEX, WHICH IS CALLED A RELATION IN DATABASE TERMINOLOGY." Right. What this is saying is that the there is a well-understood mathematical model behind what SQL is doing. Relational algebra is really very easy to understand once you've done a little SQL.
- SQL is a NATURAL LANGUAGE programming language. This means that the designers of the language tried hard to make SQL queries look like English. Don't be fooled: SQL is not English. SQL is a full-on programming language that is parsed and interpreted by single-minded programs incapable of overlooking unclosed quote marks or misplaced commas. I have found SQL consoles to be particularly bad at identifying the root cause of syntax errors. You just have to deal with it.

We'll go over the basics of how to actually do stuff in SQL momentarily. However, notice that in the above bullet points I repeatedly referenced the SQL 'consoles' and 'databases' which answer user queries. Before diving into the language itself, we should briefly talk about what these programs are.

DATABASES AND SQL

SQL, the language, is implemented by various long-running programs we usually refer to as SQL databases. A SQL database is simply a program that accepts some variety of SQL commands, manipulates some state internal to the program, and spits output back to the user. That's it.

At a conceptual level, databases are simple things. However, modern programming relies so heavily on databases that these programs have reached an extraordinary level of maturity. For example, modern SQL

databases can ensure that data safety and consistency is maintained even in the event of sudden power failure, disk corruption [2], etc, even while serving hundreds of simultaneous commands.

SQL databases are the workhorses of most of the modern Internet. Many websites essentially perform what developers refer to as 'CRUD' operations: they Create, Read, Update or Delete various data according to business rules. SQL databases excel in this role, and for that reason they are among the most used pieces of technology on the Internet. However, SQL databases do have limits. Most, by design, attempt to protect the safety and consistency of data at all costs. For this reason, there are usually strong upper limits on the effective scale of a SQL database. Google search does not and cannot run on SQL; on the other hand, most companies do not operate at Google's scale.

There are two basic flavors of SQL databases: open-source and closed-source. Open source databases are free to use and are typically developed by companies looking to sell support to users of the database. Popular open source databases include MySQL, PostgreSQL [3] and SQLite. Closed source databases are not free to use and typically users of closed source databases have much closer relationships with the database development company than users of open source databases. Popular closed source databases are Microsoft SQL Server and Oracle.

There are a few cultural connotations around which database companies use. Smaller companies and startups, especially tech-savvy companies, will usually use an open source product because they are cheaper. Larger companies will often closed source databases because of the easier access to support and advice from the database vendor. However, these lines are somewhat blurry. Salesforce runs almost entirely on Oracle, but also maintains one of the largest Postgres instances in the world. Google's core data services are home-grown systems that do not implement SQL at all, but it runs a number of auxiliary services on a home-grown implementation of MySQL. Facebook is also known to rely heavily on MySQL.

SCHEMAS AND QUERIES

Now we'll briefly touch on what day-to-day work in SQL looks like. This is not intended to be a tutorial (since so many have been written already) as much as a preview.

SQL is oriented around the creation and querying of TABLES that store data in an organized way. A table is essentially a name, a list of columns (each of which also has a data type, such as "string" or "numeric"), and possibly some constraints on the table (such as "start_date must be before end_date").

Taken together, the tables in a database are called the SCHEMA. A database schema is typically designed up front and then slowly evolves over time; changing a schema once it has been created can be very tricky.

The individual entries in a table are typically called relations, rows, or entries. They are essentially "just data" -- they are the core elements that the rest of SQL is concerned with manipulating. Here are a couple SQL queries involving these elements; note that often SQL programers will capitalize the words that are part of the SQL syntax and leave lower-case the specifics of the query. However, this is optional; SQL is not case-sensitive.

```
CREATE TABLE users (
    id          INTEGER,
    name        TEXT,
    join_date   DATE
);
```

This would create a 'users' table, where each user can have an id, a name and a join date. If we wanted to add some data to the table we could do so:

```
INSERT INTO users (id, name, join_date) VALUES (1, 'Lionel', '2013-12-
29');
INSERT INTO USERS (id, name, join_date) VALUES (2, 'Fred', '2013-12-
30');
INSERT INTO USERS (id, name, join_date) VALUES (3, 'Jane', '2013-12-
29'), (5, 'Dan', '2013-11-20');
```

(The last query would insert two users at once.)

We could then issue a query on the table:

```
SELECT id, name FROM users WHERE join_date = '2013-12-29';
```

This would return something like this:

```
id, name

1, 'Lionel'

3, 'Jane'
```

But it could also return this:

```
id, name

3, 'Jane'

1, 'Lionel'
```

That's because I haven't declared an order on the data. This is the first lesson in about SQL: it does what you ask for and nothing more. Because I didn't ask for an order, no order is guaranteed. I can fix this easily enough though:

```
SELECT id, name FROM users WHERE join_date = '2013-12-29' ORDER BY id;
```

This will always return Jane first, because she has the largest id in the set returned.

Conclusion

That's about it for this primer. SQL is a big and complicated topic, but hopefully now you have all the CONTEXT you need to dive into the nuts and bolts of the queries and problems you need to solve.

REVIEW

Please take the time to fill in these review questions to help cement your understanding of these concepts.

CHAPTER 6 – SQL 1

DML

SQL provides you with some commands that allow you to create and alter your database structure. These commands are known as the Database Definition Language or DDL.

CREATING TABLES

CREATE TABLE *tablename*

(

fieldname data type [CONTRAINTS][,]

[fieldname data type [CONTRAINTS][,]]

);

CONSTRAINTS

- PRIMARY KEY – creates the field as a primary key
- NOT NULL – Require a value for the field. i.e. the field can't be empty
- AUTO_INCREMENT – Automatically increments a integer field. Usually used for unique

FOREIGN KEYS

REFERENCES <tablename>(<fieldname>) - Creates a table reference to the defined table and field.

EXAMPLE

```
CREATE TABLE login
(
user_id INT PRIMARY KEY AUTO_INCREMENT,
username VARCHAR(50) NOT NULL,
password VARCHAR(50) NOT NULL,
full_name VARCHAR(100),
user_type_id INT REFERENCES user_types(user_type_id)
);
```

ALTERING TABLES

RENAME

ALTER TABLE <TABLENAME> RENAME TO <NEW TABLENAME>;

ADD COLUMN

ALTER TABLE <TABLENAME> ADD COLUMN <COLUMNAME> <DATATYPE>;

RENAME/CHANGE DATA TYPE

ALTER TABLE <TABLENAME> CHANGE COLUMN <COLUMNNAME> <NEW COLUMNNAME>

<DATATYPE>;

DROP COLUMN

ALTER TABLE <TABLENAME> DROP COLUMN <COLUMNNAME>;

NOTES

When renaming/changing a column you can:

- rename a column by putting in a new column name and keeping the same data type definition
- change the data type definition by using the same field name as the new column name
- change both the name and the data type definition

EXAMPLES

```
ALTER TABLE login RENAME TO employees;
ALTER TABLE employees ADD COLUMN last_login datetime;
ALTER TABLE employees CHANGE COLUMN full_name employee_name
VARCHAR(100);
ALTER TABLE employees CHANGE COLUMN employee_name employee_name
VARCHAR(150);
ALTER TABLE employees CHANGE COLUMN employee_name fill_name
VARCHAR(100);
ALTER TABLE employees DROP COLUMN last_login;
```

DROPPING TABLES

DROP <tablename>;

EXAMPLE

```
DROP employees;
```

DML

DML is the subset of the SQL language used to create, update, display and delete data from a database.

SELECTING DATA

The SELECT statement is used to retrieve data from the database. The most basic version of the SELECT statement returns all columns and all rows the SELECT statement is explained in further details in the next chapter.

SELECT * FROM <TABLENAME>;

EXAMPLE

```
SELECT * FROM account_type;
```

ADDING DATA

INSERT INTO <tablename>
(
  <fieldname> [,
  <fieldname> [,]
  …]
)
VALUES
(
  <value>[,
 value [,]
  …]
);

NOTES

When inserting data, character, date/time and binary fields must be single quoted. Numeric values are not.

Also, you must have the same number of values being inserted as there are columns listed.

EXAMPLE

```
INSERT INTO employees
(
   username,
   password,
   full_name
)
VALUES
(
   'johnd',
   'mypass',
   'John Doe'
);
```

UPDATING DATA

UPDATE <tablename> SET <fieldname>=<value> [WHERE <WHERE CLAUSE>];

EXAMPLES

```
UPDATE accounts SET phone='615551212'
WHERE email='danielg@dodgeit.com';

UPDATE accounts
     SET phone='615551234',
          fullname='Dan G'
     WHERE phone='615551212';
```

DELETING DATA

DELETE FROM <tablename> [WHERE <WHERE CLAUSE>];

EXAMPLE

```
DELETE from account_type where description='user';
```

CHATPER 7 - SQL 2

INTRODUCTION

The most common purpose of the SQL language is to search and retrieve data.

WORKING WITH COLUMNS

SELECT SPECIFIC COLUMNS AND ALL ROWS

SELECT <COL1>[],[<COL2>,] FROM <TABLENAME>;

EXAMPLE

```
SELECT description FROM account_type;
```

SELECT SPECIFIC COLUMNS AND SPECIFIC ROWS

SELECT <COL1>,[<COL2>,] FROM <TABLENAME> WHERE <WHERE CLAUSE>;

EXAMPLES

```
SELECT * from employees where full_name='administrator';
SELECT last_login FROM employees WHERE username='johnd';
SELECT username FROM employees where last_login >= 'Oct 1, 2007';
```

FILTERING RECORDS

To filter your results you would use the WHERE clause syntax. It uses a predicate format normally in the format of <FIELD> <OPERATOR> <CONDITION>. Operators include, but not limited to:

=	equals
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
<>	not equal to
IN	specifies a list of values to match against i.e. (2,3,4) or ('dan', 'dave', 'frank')

You can have multiple WHERE clauses by separating then with AND or OR. You can also group WHERE clauses with brackets. i.e. **WHERE (name ='dan' OR name ='dave') and last_login =' Oct 1, 2007';**

PATTERN MATCHING

If you need to find a portion of a string you can use pattern matching in the form of the LIKE statement. It has 2 wild card characters.

The first is % (percent). This character allows you to match multiple characters as a wild character. The other character is _ (underscore). It allows you to match against a single character.

Examples:

- **name like 'dan%'** will match any name starting with dan
- **name like '%dan%'** will match any name containing with dan
- **name like '%dan'** will match any name ending with dan
- **name like 'd_n'** will match dan, don, din.

A more powerful search method is the SIMILAR TO syntax. Not all servers implement it so be aware of some of the limitations you may face. The following is from the PostgreSQL documentation.

string SIMILAR TO pattern [ESCAPE escape-character]

string NOT SIMILAR TO pattern [ESCAPE escape-character]

The SIMILAR TO operator returns true or false depending on whether its pattern matches the given string. It is similar to LIKE, except that it interprets the pattern using the SQL standard's definition of a regular expression. SQL regular expressions are a curious cross between LIKE notation and common regular expression notation.

Like LIKE, the SIMILAR TO operator succeeds only if its pattern matches the entire string; this is unlike common regular expression behavior where the pattern can match any part of the string. Also like LIKE, SIMILAR TO uses _ and % as wildcard characters denoting any single character and any string, respectively (these are comparable to . and .* in POSIX regular expressions).

In addition to these facilities borrowed from LIKE, SIMILAR TO supports these pattern-matching metacharacters borrowed from POSIX regular expressions:

- | denotes alternation (either of two alternatives).
- denotes repetition of the previous item zero or more times.
- + denotes repetition of the previous item one or more times.
- ? denotes repetition of the previous item zero or one time.
- {m} denotes repetition of the previous item exactly m times.
- {m,} denotes repetition of the previous item m or more times.
- {m,n} denotes repetition of the previous item at least m and not more than n times.
- Parentheses () can be used to group items into a single logical item.
- A bracket expression [...] specifies a character class, just as in POSIX regular expressions.

Notice that the period (.) is not a metacharacter for SIMILAR TO.

As with LIKE, a backslash disables the special meaning of any of these metacharacters; or a different escape character can be specified with ESCAPE. Some examples:

'abc' SIMILAR TO 'abc' *true*

'abc' SIMILAR TO 'a' *false*

'abc' SIMILAR TO '%(b|d)%' *true*

'abc' SIMILAR TO '(b|c)%' *false*

ALIASES

At times, queries can get rather complicated and long. In order to help make sense of the larger queries, SQL allows you to "rename" columns and tables. The syntax for this is:

COLUMN NAME ALIAS

The syntax is:

```
SELECT column AS column_alias FROM table
```

TABLE NAME ALIAS

The syntax is:

```
SELECT column FROM table AS table_alias
```

EXAMPLE

The query:

```
SELECT last_user_login_timestamp, username FROM user_login_logs
```

Could be rewritten as:

```
SELECT last_user_login_timestamp as last_login, username FROM
user_login_logs as ull WHERE ull.last_user_login_timestamp >=
'01/01/2008'
```

ORDERING RESULTS

There are times you will want to change the order in which your query returns the data. SQL provides a method call the ORDER BY clause. The basic syntax is as follows:

```
SELECT field1[,field2][,field3…] FROM <TABLENAME>
[WHERE Clause]
ORDER BY ordering_field [ASC/DESC], [ordering_field2 [ASC/DESC]];
```

The ORDER BY clause supports sorting either ascending or descending by adding the suffix to the ordering field. ASC is for sorting in an ascending manner. This means that it would sort from smallest to largest or A to Z. DESC is for sorting in reverse order. When sorting dates, ASC means oldest to newest.

EXAMPLE

```
SELECT username,last_login_timestamp, first_name, last_name FROM
employee
ORDER by last_login desc,last_name asc;
```

In plain English, the above query will return the username, last time an employee logged in and their name. It will be sorted by most recent login and then by last name.

AGGREGATES AND HAVING

Another powerful feature or SQL is the ability to summarize our data to determine trends or produce top-level reports. An important thing to remember is that when ou are using aggregates, you may often need to group the results so that they actually make sense. Also, if you are returning anything else other than the aggregate in the field list, you must include that column in the group by clause.

All of our example queries will use the products table described below.

last_name	quantity	unit_price	continent
Jacob	21	4.52	North America
Wiggum	192	3.99	North America
Johnson	87	4.49	Africa
Smith	842	2.99	North America
Marks	48	3.48	Africa
Linea	9	7.85	North America
Jonas	638	3.29	Europe

SUM

The SUM function is used within a SELECT statement and, predictably, returns the summation of a series of values. If the widget project manager wanted to know the total number of widgets sold to date, we could use the following query:

```
SELECT SUM(quantity) AS Total
FROM products
```

Our results would appear as:

```
Total
-----------
1837
```

AVG

The AVG (average) function works in a similar manner to provide the mathematical average of a series of values. Let's try a slightly more complicated task this time. We'd like to find out the average dollar amount of all orders placed on the North American continent.

Note that we'll have to multiply the quantity column by the unit_price column to compute the dollar amount of each order. Here's what our query will look like:

```
SELECT AVG(unit_price * quantity) As AveragePrice
FROM products
WHERE continent = "North America"
```

And the results:

```
AveragePrice
--------------------
862.3075
```

COUNT

SQL provides the COUNT function to retrieve the number of records in a table that meet given criteria. We can use the COUNT(*) syntax alone to retrieve the number of rows in a table. Alternatively, a WHERE clause can be included to restrict the counting to specific records.

For example, suppose our Widgets product manager would like to know how many orders our company processed that requested over 100 widgets.

Here's the SQL query:

```
SELECT COUNT(*) AS 'Number of Large Orders'
FROM products
WHERE quantity > 100
```

And the results:

```
Number of Large Orders
----------------------
3
```

The COUNT function also allows for the use of the DISTINCT keyword and an expression to count the number of times a unique value for the expression appears in the target data. Similarly, the ALL keyword returns the total number of times the expression is satisfied, without worrying about unique values.

First, let's take a look at the use of the ALL keyword:

```
SELECT COUNT(ALL continent) As 'Number of continents'
FROM products
```

And the result set:

```
Number of continents
--------------------
7
```

Obviously, this is not the desired results. If you recall, all of our orders came from North America, Africa and Europe. Let's try the DISTINCT keyword instead:

```
SELECT COUNT(DISTINCT continent) As 'Number of continents'
FROM products
```

And the output:

```
Number of continents
--------------------
3
```

That's more like it!

MAX

The MAX() function returns the largest value in a given data series. We can provide the function with a field name to return the largest value for a given field in a table. MAX() can also be used with expressions and GROUP BY clauses for enhanced functionality.

Once again, we'll use the products example table for this query. We could use the following query to find the order with the largest total dollar value:

```
SELECT MAX(quantity * unit_price)As 'Largest Order'
FROM products
```

Our results would look like this:

```
Largest Order
---------------------
2517.58
```

MIN

The MIN() function functions in the same manner, but returns the minimum value for the expression. Let's try a slightly more complicated example utilizing the MIN() function. Let's retrieve information on the smallest widget order placed on each continent. This requires the use of the MIN() function on a computed value and a GROUP BY clause to summarize data by continent.

Here's the SQL:

```
SELECT continent, MIN(quantity * unit_price) AS 'Smallest Order'
FROM products
GROUP BY continent
```

And our result set:

```
continent     Smallest Order
------------- --------------------
Africa        167.04
Europe        2099.02
North America 70.65
```

HAVING

The HAVING clause enables you to specify conditions that filter which group results appear in the final results.

The WHERE clause places conditions on the selected columns, whereas the HAVING clause places conditions on groups created by the GROUP BY clause.

Syntax:

The HAVING clause must follow the GROUP BY clause in a query and must also precede the ORDER BY clause if used. The following is the syntax of the SELECT statement, including the HAVING clause:

SELECT column1, column2
FROM table1
WHERE [conditions]
GROUP BY column1, column2
HAVING [conditions]
ORDER BY column1, column2

Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is the example, which would display record for which similar age count would be more than or equal to 2:

```
SELECT ID, NAME, AGE, ADDRESS, SALARY
FROM CUSTOMERS
GROUP BY age
HAVING COUNT(age) >= 2;
```

This would produce the following result:

```
+----+--------+-----+---------+---------+
| ID | NAME   | AGE | ADDRESS | SALARY  |
+----+--------+-----+---------+---------+
|  2 | Khilan |  25 | Delhi   | 1500.00 |
+----+--------+-----+---------+---------+
```

CHAPTER 8 - SQL 3

INTRODUCTION

Often when dealing with databases, it becomes necessary to query multiple tables. When pulling data from multiple sources, there are a few methods you can use. They include JOINS, Subqueries and Set operations. This chapter deals with JOINS and Subqueries.

JOINS

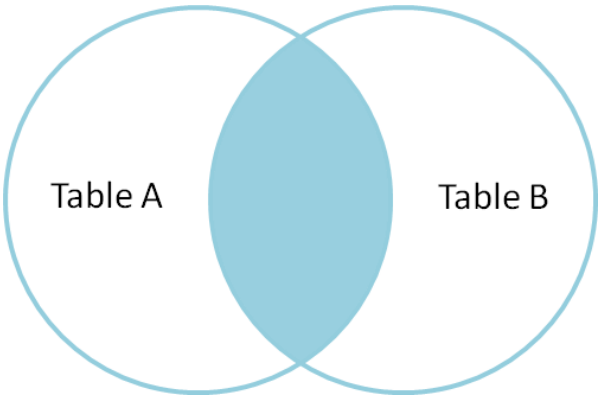
For the discussion on joins the following tables are used.

```
Table A          Table B
id name        id  name
-- ----        --  ----
1  Pirate      1   Rutabaga
2  Monkey      2   Pirate
3  Ninja       3   Darth Vader
4  Spaghetti   4   Ninja
```

Let's join these tables by the name field in a few different ways and see if we can get a conceptual match to those nifty Venn diagrams.

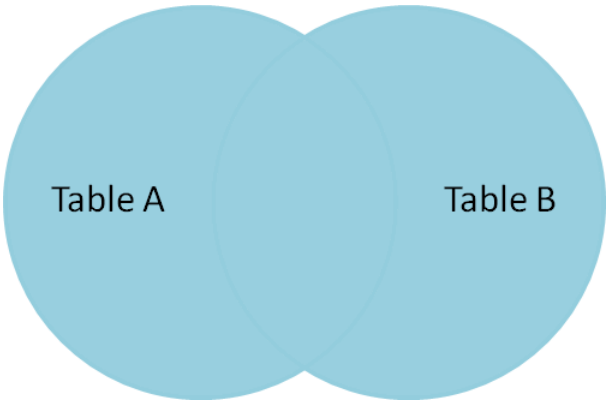
Inner join produces only the set of records that match in both Table A and Table B.

```
SELECT * FROM TableA
INNER JOIN TableB
ON TableA.name = TableB.name

id  name        id   name
--  ----        --   ----
1   Pirate      2    Pirate
3   Ninja       4    Ninja
```



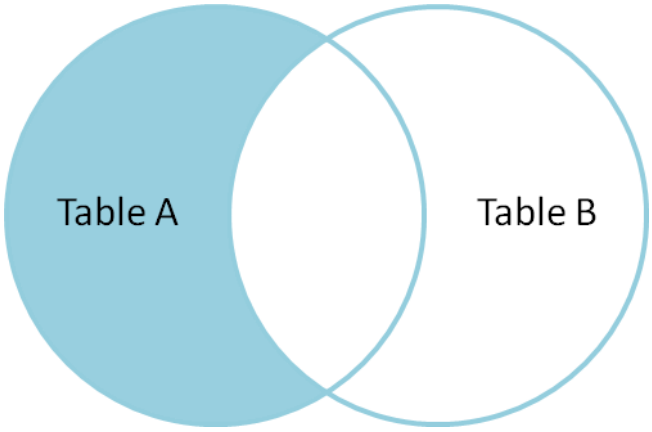
Full outer join produces the set of all records in Table A and Table B, with matching records from both sides where available. If there is no match, the missing side will contain null.

```
SELECT * FROM TableA
FULL OUTER JOIN TableB
ON TableA.name = TableB.name

id    name        id    name
--    ----        --    ----
1     Pirate      2     Pirate
2     Monkey      null  null
3     Ninja       4     Ninja
4     Spaghetti   null  null
null  null        1     Rutabaga
null  null        3     Darth Vader
```



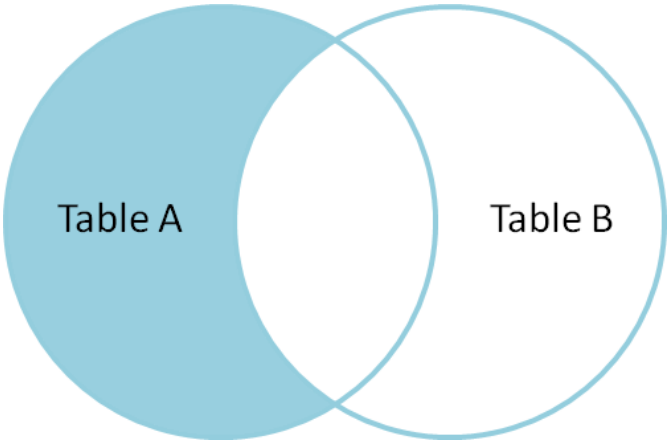
Left outer join produces a complete set of records from Table A, with the matching records (where availe) in Table B. If there is no match, the right side will contain null.

```
SELECT * FROM TableA
LEFT OUTER JOIN TableB
ON TableA.name = TableB.name

id  name        id    name
--  ----        --    ----
1   Pirate      2     Pirate
2   Monkey      null  null
3   Ninja       4     Ninja
4   Spaghetti   null  null
```



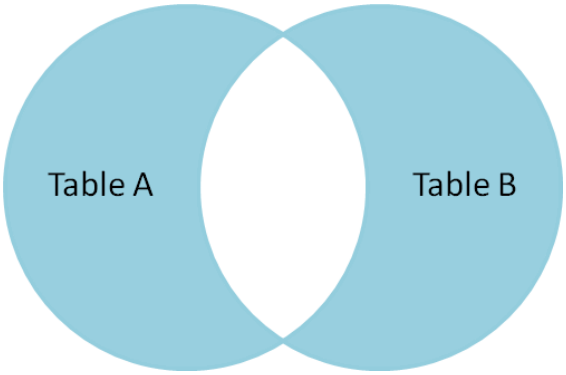
To produce the set of records only in Table A, but not in Table B, we perform the same left outer join, then **exclude the records we don't want from the right side via a where clause.**

```
SELECT * FROM TableA
LEFT OUTER JOIN TableB
ON TableA.name = TableB.name
WHERE TableB.id IS null

id  name        id    name
--  ----        --    ----
2   Monkey      null  null
4   Spaghetti   null  null
```



To produce the set of records unique to Table A and Table B, we perform the same full outer join, then **exclude the records we don't want from both sides via a where clause.**

```
SELECT * FROM TableA
FULL OUTER JOIN TableB
ON TableA.name = TableB.name
WHERE TableA.id IS null
OR TableB.id IS null

id    name       id    name
--    ----       --    ----
2     Monkey     null  null
4     Spaghetti  null  null
null  null       1     Rutabaga
null  null       3     Darth Vader
```



You'll generally use the OUTER JOIN form that asks for all the rows from one table or result set and any matching rows from a second table or result set. To do this, you specify either a LEFT OUTER JOIN or a RIGHT OUTER JOIN.

What's the difference between LEFT and RIGHT? When you begin building queries using OUTER JOIN, the SQL Standard considers the first table you name as the one on the "left," and the second table as the one on the "right." So, if you want all the rows from the first table and any matching rows from the second table, you'll use a LEFT OUTER JOIN. Conversely, if you want all the rows from the second table and any matching rows from the first table, you'll specify a RIGHT OUTER JOIN.

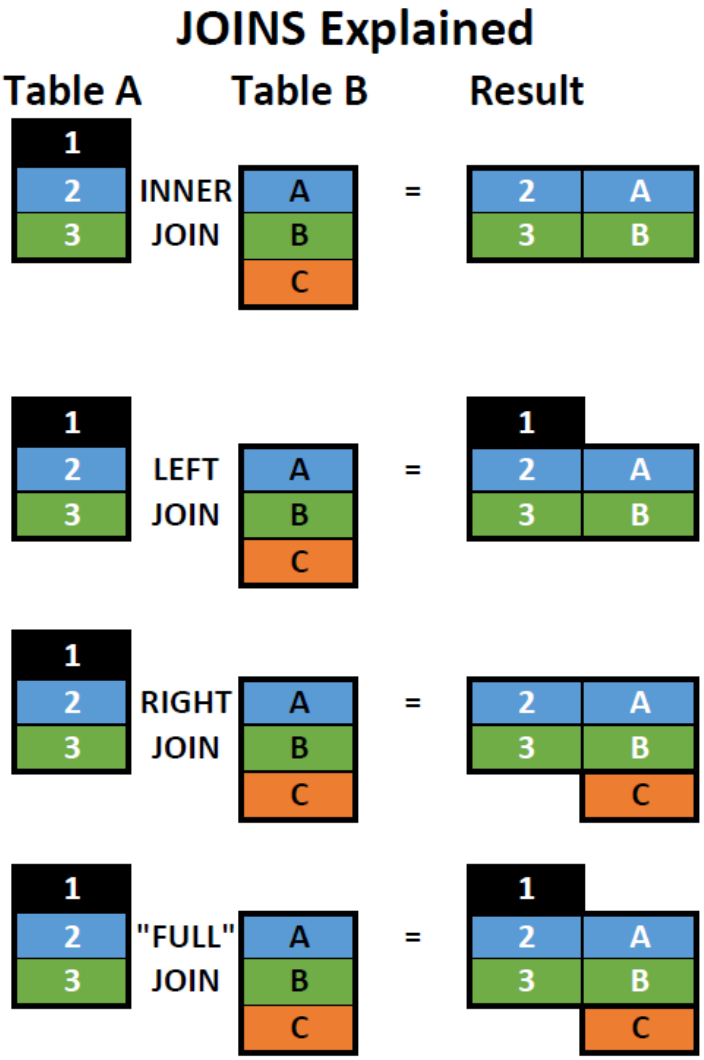
There's also a cartesian product or **cross join**, which as far as I can tell, can't be expressed as a Venn diagram:

SELECT * FROM TableA

CROSS JOIN TableB

This joins "everything to everything", resulting in 4 x 4 = 16 rows, far more than we had in the original sets. If you do the math, you can see why this is a *very* dangerous join to run against large tables.

Alternatively, to help visualize what joins actually do, you can refer to the diagram on the right.



SUBQUERIES

A subquery is a SELECT statement that is nested within another SQL statement. A subquery SELECT statement if executed independently of the SQL statement, in which it is nested, will return a result set. Meaning a subquery SELECT statement can standalone and is not depended on the statement in which

it is nested. A subquery SELECT statement can return any number of values, and can be found in, the column list of a SELECT statement, a FROM, GROUP BY, HAVING, and/or ORDER BY clauses of a SQL statement. A Subquery can also be used as a parameter to a function call. Basically a subquery can be used anywhere an expression can be used.

USE OF A SUBQUERY IN THE COLUMN LIST OF A SELECT STATEMENT

Suppose you would like to see the last OrderID and the OrderDate for the last order that was shipped to Paris. Along with that information, say you would also like to see the OrderDate for the last order shipped regardless of the ShipCity. In addition to this, you would also like to calculate the difference in days between the two different OrderDates. Here is my SQL SELECT statement to accomplish this:

```
select top 1 orderid,
       convert(char(10), orderdate,121) last_paris_order,
       (select convert(char(10),max(orderdate),121)
          from orders) last_orderdate,
       datediff(dd,orderdate,(select max(orderdate)
                                from orders)) day_diff
  from orders
 where shipcity = 'paris'
order by orderdate desc
```

The above code contains two subqueries. The first subquery gets the OrderDate for the last order shipped regardless of ShipCity, and the second subquery calculates the number of days between the two different OrderDates. Here we used the first subquery to return a column value in the final result set. The second subquery was used as a parameter in a function call. This subquery passed the "max(OrderDate)" date to the DATEDIFF function.

USE OF A SUBQUERY IN THE WHERE CLAUSE

A subquery can be used to control the records returned from a SELECT by controlling which records pass the conditions of a WHERE clause. In this case the results of the subquery would be used on one side of a WHERE clause condition. Here is an example:

```
select distinct country
 from customers
where country not in (select distinct country
                        from suppliers)
```

Here we have returned a list of countries where customers live, but there is no supplier located in that country. We suppose if you where trying to provide better delivery time to customers, then you might target these countries to look for additional suppliers.

Suppose a company would like to do some targeted marketing. This targeted marketing would contact customers in the country with the fewest number of orders. It is hoped that this targeted marketing will increase the overall sales in the targeted country. Here is an example that uses a subquery to return the customer contact information for the country with the fewest number of orders:

```
select country,
       companyname,
       contactname,
       contacttitle,
       phone
from customers
where country = (select top 1 country
                 from customers c
                 join orders o
                   on c.customerid = o.customerid
                 group by country
                 order by count(*))
```

Here we have written a subquery that joins the Customer and Orders Tables to determine the total number of orders for each country. The subquery uses the "TOP 1" clause to return the country with the fewest number of orders. The country with the fewest number of orders is then used in the WHERE clause to determine which Customer Information will be displayed.

USE OF A SUBQUERY IN THE FROM CLAUSE

The FROM clause normally identifies the tables used in the SQL statement. You can think of each of the tables identified in the FROM clause as a set of records. Well, a subquery is just a set of records, and therefore can be used in the FROM clause just like a table. Here is an example where a subquery is used in the FROM clause of a SELECT statement:

```
select au_lname,
       au_fname,
       title from (select au_lname, au_fname, au_id
                   from authors
                   where state = 'ca') as a
             join titleauthor ta on a.au_id = ta.au_id
             join titles t on ta.title_id = t.title_id
```

Here we have used a subquery to select only the author record information, if the author's record has a state column equal to "CA." We have named the set returned from this subquery with a table alias of "A". WeI can then use this alias elsewhere in the SQL statement to refer to the columns from the subquery by prefixing them with an "A", as we did in the "ON" clause of the "JOIN" criteria. Sometimes using a subquery in the FROM clause reduces the size of the set that needs to be joined. Reducing the number of records that have to be joined enhances the performance of joining rows, and therefore speeds up the overall execution of a query.

USE OF A SUBQUERY IN THE HAVING CLAUSE

In the following example, we used a subquery to find the number of books a publisher has published where the publisher is not located in the state of California. To accomplish this we used a subquery in a HAVING clause. Here is the code:

```
select pub_name,
       count(*) bookcnt
  from titles t
  join publishers p  on t.pub_id = p.pub_id
group by pub_name
having p.pub_name in (select pub_name
                      from publishers
                      where state <> 'ca')
```

Here the subquery returns the pub_name values for all publishers that have a state value not equal to "CA." The HAVING condition then checks to see if the pub_name is in the set returned by my subquery.

CORRELATED SUBQUERIES

A correlated subquery is a SELECT statement nested inside another SQL statement, which contains a reference to one or more columns in the outer query. Therefore, the correlated subquery can be said to be dependent on the outer query. This is the main difference between a correlated subquery and just a plain subquery. A plain subquery is not dependent on the outer query, can be run independently of the outer query, and will return a result set. A correlated subquery, since it is dependent on the outer query will return a syntax errors if it is run by itself.

A correlated subquery will be executed many times while processing the SQL statement that contains the correlated subquery. The correlated subquery will be run once for each candidate row selected by the outer query. The outer query columns, referenced in the correlated subquery, are replaced with values from the

candidate row prior to each execution. Depending on the results of the execution of the correlated subquery, it will determine if the row of the outer query is returned in the final result set.

USING A CORRELATED SUBQUERY IN A WHERE CLAUSE

Suppose you want a report of all "OrderID's" where the customer did not purchase more than 10% of the average quantity sold for a given product. This way you could review these orders, and possibly contact the customers, to help determine if there was a reason for the low quantity order. A correlated subquery in a WHERE clause can help you produce this report. Here is a SELECT statement that produces the desired list of "OrderID's":

```
select distinct orderid
  from orderdetails od
 where quantity > (select avg(quantity) * .1
                     from orderdetails
                    where od.productid = productid)
```

The correlated subquery in the above command is contained within the parenthesis following the greater than sign in the WHERE clause above. Here you can see this correlated subquery contains a reference to "OD.ProductID". This reference compares the outer query's "ProductID" with the inner query's "ProductID". When this query is executed, the SQL engine will execute the inner query, the correlated subquery, for each "[Order Details]" record. This inner query will calculate the average "Quantity" for the particular "ProductID" for the candidate row being processed in the outer query. This correlated subquery determines if the inner query returns a value that meets the condition of the WHERE clause. If it does, the row identified by the outer query is placed in the record set that will be returned from the complete SQL SELECT statement.

The code below is another example that uses a correlated subquery in the WHERE clause to display the top two customers, based on the dollar amount associated with their orders, per region. You might want to perform a query like this so you can reward these customers, since they buy the most per region.

```
select c1.companyname,
       c1.contactname,
       c1.address,
       c1.city,
       c1.country,
       c1.postalcode
  from customers c1
 where c1.customerid in (select top 2 c2.customerid
                         from orderdetails od
                              join orders o on od.orderid = o.orderid
                              join customers c2 on o.customerid =
c2.customerid
                        where c2.region = c1.region
                        group by c2.region, c2.customerid
                        order by sum(od.unitprice * od.quantity * (1 -
od.discount)) desc)
order by c1.region
```

Here you can see the inner query is a correlated subquery because it references "**C1**", which is the table alias for the "Customers" table in the outer query. This inner query uses the "Region" value to calculate the top two customers for the region associated with the row being processed from the outer query. If the "CustomerID" of the outer query is one of the top two customers, then the record is placed in the record set to be returned.

CORRELATED SUBQUERY IN THE HAVING CLAUSE

Say your organizations wants to run a yearlong incentive program to increase revenue. Therefore, they advertise to your customers that if each order they place, during the year, is over $750 you will provide them a rebate at the end of the year at the rate of $75 per order they place. Below is an example of how to calculate the rebate amount. This example uses a correlated subquery in the HAVING clause to identify the customers that qualify to receive the rebate.

```
select c.customerid,
       count(*) * 75 rebate
  from customers c
  join orders o on c.customerid = o.customerid
 where datepart(yy,orderdate) = '1998'
 group by c.customerid
having 750 < all(select sum(unitprice * quantity * (1 - discount))
                   from orders o
                   join order_details od on o.orderid = od.orderid
                  where o.customerid = c.customerid
                    and datepart(yy,o.orderdate) = '1998'
                  group by o.orderid)
```

By reviewing this query, you can see the correlated query in the HAVING clause to calculate the total order amount for each customer order. We use the "CustomerID" from the outer query and the year of the order "Datepart(yy,OrderDate)", to help identify the Order records associated with each customer, that were placed the year '1998'. For these associated records I am calculating the total order amount, for each order, by summing up all the "[Order Details]" records, using the following formula: sum(UnitPrice * Quantity * (1-Discount)). If each and every order for a customer, for year 1998 has a total dollar amount greater than 750, I then calculate the Rebate amount in the outer query using this formula "Count(*) * 75 ".

The database server's query engine will only execute the inner correlated subquery in the HAVING clause for those customer records identified in the outer query, or basically only those customer that placed orders in "1998".

CHAPTER 9 - SQL 4

INTRODUCTION

SQL supports few Set operations to be performed on table data. These are used to get meaningful results from data, under different special conditions. The common set operators are:

- UNION
- INTERSECT
- MINUX/EXCEPT

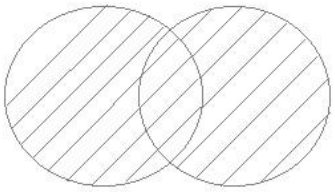
SET OPERATORS

The examples will use the following data:

First Table			Second Table	
ID	**Name**		**ID**	**Name**
1	abhi		2	adam
2	adam		3	Chester

UNION

UNION is used to combine the results of two or more Select statements. However it will eliminate duplicate rows from its result set. In case of union, number of columns and datatype must be same in both the tables.



EXAMPLE OF UNION

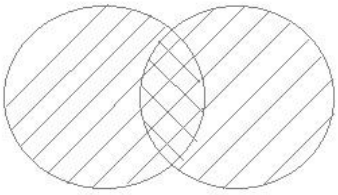
Union SQL query will be,

```
select * from First
UNION
select * from second
```

The result table will look like,

ID	NAME
1	abhi
2	adam
3	Chester

UNION ALL



This operation is similar to Union. But it also shows the duplicate rows.

EXAMPLE OF UNION ALL

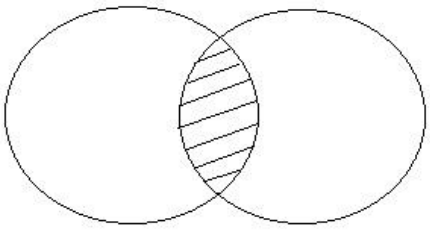
Union All query will be like,

```
select * from First
UNION ALL
select * from second
```

The result table will look like,

ID	NAME
1	abhi
2	adam
2	adam
3	Chester

INTERSECT



Intersect operation is used to combine two SELECT statements, but it only retuns the records which are common from both SELECT statements. In case of **Intersect** the number of columns and datatype must be same. MySQL does not support INTERSECT operator.

EXAMPLE OF INTERSECT

The **First** table,

Intersect query will be,

```
select * from First
INTERSECT
select * from second
```

The result table will look like

ID	NAME
2	adam

MINUS/EXCEPT

Minus operation combines result of two Select statements and return only those result which belongs to first set of result. MySQL does not support Minus/Except operator.

EXAMPLE OF MINUS/EXCEPT

Minus query will be,

```
select * from First
EXCEPT
select * from second
```

The result table will look like,

ID	NAME
1	abhi

CHAPTER 10 - FUNCTIONS AND TRIGGERS

INTRODUCTION

A lot of people in the database and programming professions are vehemently in favor or opposed to the use of stored procedures and the like in databases. They will argue that all access to the database should go thru stored procedures because it is more secure and shields applications from changing logic. The other side will vehemently argue that you should avoid this because not all databases support this and the way each supports it is different so your code is less portable.

BEST PRACTICES VS. BEST APPROACH TO ACHIEVE THE GOAL

Before dismissing one approach over another, it is important to map out what you are trying to achieve and then determine how using one approach aligns with your objectives.

You will often hear the term **Best Practices** used in Application Architecture. The danger of such a term is that it gives one a sense that there is no need to question a practice because it is always best. Just follow Best Practices outlined by an authority in the subject matter and things will magically fall into place. Rather than just focus on Best Practices, I would like to propose that one should think about what they are trying to achieve and why one approach is better than another to get there.

WHAT TO CONSIDER?

In this section, we will talk about considerations on a very elemental functional level rather than a more macro application level. We think it is important to consider each functional task of an application separately rather than thinking of an application as a whole. Example you may have one part of an application that needs to talk to a mainframe database or run scheduled tasks, and given that it doesn't make sense to drive your whole application structure based on the need of this single functionality.

- Does this function need to work in various kinds of databases?
- Is this function used in multiple parts of an application or applications?
- Does this function require many arguments and return one set of values in form of single table or scalar value?
- Does this function require few arguments passed to the database, but require lots of data from the database to arrive at results?

- Is the function data intensive or processor intensive? E.g. is it an encryption function or one to get results of a complex query - is it more SQL intensive or more procedural intensive
- Are the parameters passed into the function consistently the same or can they vary? E.g. a complex search form will not always need to pass the same parameters, but a simple one will just have a single search field and always pass just that.
- Does the function require page long batches of SQL statements or is it a one line SQL statement?
- Are the fields of the result set always the same or do they need to vary. For example do you always find yourself joining the same set of tables varying slightly by the need for different fields or subsets of data?

DATABASE OBJECTS

I always find it amusing that when people talk about database logic they are very focused on stored procedures and its almost like nothing else exists. Makes me wonder if these people have ever worked with modern databases. Stored procedures are one of the oldest methods of encapsulating database logic, but they are not the only method availe. Many relational databases nowadays have views, constraints, referential integrity with cascading update, delete, stored functions, triggers and the like. These are extremely powerful tools when used appropriately.

In the next couple of sections we'll cover stored procedures and these other kinds of database objects and detail the strengths and weaknesses of each for encapsulating logic. We will give a rating of 0-5 for each feature 0 meaning the feature is non-existent, 5 meaning this object is one of the best suited objects for implementing this kind of task.

STORED PROCEDURES

Stored procedures are one of numerous mechanisms of encapsulating database logic in the database. They are similar to regular programming language procedures in that they take arguments, do something, and sometimes return results and sometimes even change the values of the arguments they take when arguments are declared as output parameters. You will find that they are very similar to stored functions in that they can return data; however stored procedures can not be used in queries. Since stored procedures have the mechanism of taking arguments declared as OUTPUT they can in theory return more than one output.

Feature	Rating
Works in various kinds of databases	3 (many databases such as DB II, Oracle, SQL Server, MySQL 5, PostGreSQL, FireBird support them). There are also a lot that don't e.g. MySQL < 5.0, MS Access (although parameterized queries serve a similar role)
Can be called by multiple applications and interfaces	4 (generally they can be called, but the use of OUTPUT arguments is not always usable)
Can take an undefined number of arguments	2 (note most databases allow to define optional arguments, but this can become very unwieldy to maintain if there are a lot because you end up duplicating logic even within the stored procedure so is generally avoided)
Reusability within the database	3 (you can not reuse them in views, rarely in stored functions and other stored procedures unless the stored procedure using it does not require a return value or result query). This varies slightly from DBMS to DBMS.
Can be used to change data in a table without giving rights to a user to change table directly	4 In general true for most DBMSs that support them.
Can return varying number of fields given different arguments.	3 –again in theory it can, but very hard to maintain since you would often be duplicating logic to say return one field in one situation and other set of fields in another situation or update a field when the field is passed in as an argument. Note that in many databases such as for example SQL Server and Oracle, one can return multiple result sets with a stored procedure, but the receiving end needs to be able to do a next result set call and know the sequence in which the result sets are being sent.

Long stretches of SQL easy to read and maintain	5 (one of the great strengths of stored procedures is that you can have long transactions of sql statements and conditional loops which can be all committed at once or rolled back as a unit. This also saves on network traffic.

STORED FUNCTIONS

Stored Functions are very similar to stored procedures except in 3 major ways.

Unlike stored procedures, they can be used in views, stored procedures, and other stored functions.

In many databases they are prohibited from changing data or have ddl/dml limitations. Note for databases such as PostGreSQL this is not true since the line between a stored function and a stored procedure is very greyed

They generally cannot take output arguments (placeholders) that are then passed back out with changed values.

Feature	**Rating**
Works in various kinds of databases	3 (many databases such as DB II, Oracle, SQL Server support them, MySQL 5, PostGreSQL). There are also a lot that don't e.g. MySQL < 5.0, MS Access
Can be called by multiple applications and interfaces	4 (generally they can be called, but the use of OUTPUT arguments is not always usable)
Can take an undefined number of arguments	2 (note most databases allow to define optional arguments, but this can become very unwieldy to maintain if there are a lot because you end up duplicating logic even within the stored function so is generally avoided)
Reusability within the database	5 (you can reuse them in views, in other stored functions and stored procedures). This varies slightly from DBMS to DBMS.

Can be used to change data in a table without giving rights to a user to change table directly	3 Many databases do not allow changing of data in stored functions except temp table data, but those that do in general support this.
Can return varying number of fields given different arguments.	4 –For databases such as SQL Server, PostgreSQL, DB 2, Oracle that allow return tables and sets, you can selectively pick fields you want from within a query. So although the function always outputs the same number of fields, you can selectively use only some similar to what you can do with views. This is not true for scalar functions (MySQL 5.1- only supports scalar functions).
Long stretches of SQL easy to read	5 - yes - you can do fairly intensive multi-line processing which in the end returns one value or table to the user.

TRIGGERS AND RULES

Triggers are objects generally tied to a table or view that run code based on certain events such as inserting data, before inserting data, updating/deleting data and before these events happen.

Triggers can be very great things and very dangerous things. Dangerous in the sense that they are tricky to debug, but powerful because no update to a table with a trigger can easily escape the trigger.

They are useful for making sure certain events always happen when data is inserted or updated - e.g. set complex default values of fields, inserting logging records into other tables.

Triggers are especially useful for one particular situation and that is for implementing "instead of" logic. For example, as we said earlier, many views involving more than one table are not updateable. However, in DBMS such as PostgreSQL, you can define a rule on a view that occurs when someone tries to update or insert into the view and will occur instead of the insert. The rule can be fairly complex and can layout how the tables should be updated in such a situation. MS SQL Server and SQLite let you do something similar with INSTEAD OF triggers. Note the term Rule is a little confusing in DBMS because they mean quite different things. In Microsoft SQL Server for example a Rule is an obsolete construct that was used to define constraints on tables. In PostgreSQL a Rule is very similar to a trigger except that it does not get triggered per row event and is defined without need of a handling function.

Feature	Rating
Works in various kinds of databases	2 (many databases such as DB II, Oracle, SQL Server support them, MySQL 5, PostGreSQL,). There are lots that don't e.g. MySQL < 5.0, MySQL 5 limited, MS Access
Can be called by multiple applications and interfaces	5 (it just happens behind the scenes. No application can escape them)
Can take an undefined number of arguments	0 (does not accept any arguments)
Reusability within the database	2 - No and Yes. Some database servers allow for code abstraction in triggers
Can be used to change data in a table without giving rights to a user to change table directly.	4 In general yes for databases that support them
Can return varying number of fields given different arguments.	0 –Triggers are strictly for updating data
Long stretches of SQL easy to read. A trigger can often be defined with an administrative designer or using a color coded sql editor so is fairly easy to read	5