INTRODUCTION TO DATABASES

A guide to the basics of design and database communications

ABSTRACT

A low key and basic introduction to understanding database structures, design and usage. A hands on focus that will give students practical knowledge on how to create and use standard database structures.

Daniel Gaudreault

CST8215

Contents

Introduction 1
Database Structures 3
Introduction 3
What is a database? 3
Types of Databases 3
The Flat Model Database 3
The Hierarchical Model Database 3
The Network Model 3
The Relational Model 4
Parts of a database 4
Tables 4
Keys 5
Relations 5
One-to-Many Relationships 5
Many-to-Many Relationships 5
One-to-One Relationships 5
Database Design Concepts 7
Introduction 7
Types of data structures 7
Singlet 7
Reference 7
Master-detail 7
Database Design & Analysis forms 7
Entities form 7
Fields form 8
Identifying data structures from a real world document 9
Identifying data structures from a vague specification 11
Interviews 11
Whiteboard Brainstorming 11
Research 11
Mapping Data Structures 13

Introduction 13
Database Data Types........ 13
Character/text data........ 13
Numeric data........ 13
Date data........ 14
Boolean data 15
Field Design 15
Mapping data types to fields 16
Identifying relations 17
Normalization........ 19
Introduction 19
Goals of normalization 19
Normal Forms 19
First Normal Form 19
Second Normal Form 19
Third Normal Form........ 20
Result of normalization 20
One-to-one relations........ 20
Many-to-many relations 20
Denormalization........ 20
Diagramming........ 23
Introduction 23
Naming Conventions........ 23
Styles of diagrams 23
Basic Entity Relationship Diagram 23
Physical Entity Relationship Diagram........ 25
Crow's foot notation 25
Introduction to SQL........ 27
Introduction 27
DDL........ 29
Introduction 29
Create Table Syntax 29
Constraints 29

Identifiers (MySQL specific syntax) ... 29
Example ... 29
Alter table syntax ... 29
Rename ... 29
Add Column ... 29
Rename/Change Data Type ... 30
Drop Column ... 30
Notes ... 30
Examples ... 30
Drop table Syntax ... 30
Example ... 30
DML ... 31
Introduction ... 31
How to “SELECT” some data ... 31
Select all columns and all rows ... 31
Example ... 31
Select specific columns and all rows ... 31
Example ... 31
Select specific columns and specific rows ... 31
Examples ... 31
WHERE clause syntax ... 31
How to “INSERT” some data ... 32
Notes ... 32
Example ... 32
How to “UPDATE” some data ... 33
Examples ... 33
How to “DELETE” some data ... 33
Example ... 33
Joins, Ordering and Aggregates ... 35
Introduction ... 35
Joins ... 35
Example ... 35
Inner join ... 35

Left, Right & Full join 35

Ordering Results 35

Example 36

Aliases 36

Column Name Alias 36

Table Name Alias 36

Example 36

Aggregates 36

SUM 37

AVG 37

COUNT 38

MAX 39

MIN 39

Views 41

Introduction 41

Creating views 41

Using Views 41

Subqueries, Unions, Intersects and Excepts 43

Introduction 43

Subqueries 43

Unions 43

Intersects 43

Excepts 43

User Defined Functions and Triggers 45

Introduction 45

User defined functions 45

Function Example 45

Triggers 46

Events 46

Insert 46

Update 46

Delete 46

Time frames 46

Before 46
After 46
Trigger languages 46
MySQL 46
PostGreSQL 46
Oracle 47
Sybase/MS-SQL server 47
Creating triggers in MySQL 47
Trigger Example 47
Specialty Queries 49
Introduction 49
Finding and deleting duplicate rows 49
How to find duplicated rows 49
Why can't you use a `WHERE` clause? 50
How to delete duplicate rows 50
How to find duplicates in multiple columns 51
Queries that don't work 52
Some correct solutions 53
Updating rows based on a join 55
MySQL UPDATE JOIN syntax 55
MySQL UPDATE JOIN examples 56
MySQL UPDATE JOIN example with INNER JOIN clause 57
MySQL UPDATE JOIN example with LEFT JOIN 57
Appendix A 61

Introduction

Databases are the backbone of the IT industry. From running your contact list on your cell phone right up to indexing the search data for your favorite search engine, databases are everywhere.

The basic concepts of database design and querying are fairly straight forward as long as you learn a few basic rules and caveats. This text will help guide you through the various stages of database design and querying. It is designed to be a companion document to go with the lectures presented to you. You will find all the labs and assignments detailed within these pages as well as all the basic reference items that you will need to succeed in your introduction to databases.

Database Structures

Introduction

What is a database?

In its simplest form, a database is a collection of information. When referring to computers, a database is an application that provides a method to store, maintain and retrieve information. In most database systems, the data is stored in tables and may or may not have relationships between tables.

On a Windows PC, Excel is often used as a basic database. Microsoft Access and FileMaker are examples of consumer grade database systems. These applications are good for smaller quantities of data.

However, when larger set of data are involved, a full blown database server is then required. A database server is an application that runs as a "service" on most computer platforms such as Windows or Mac. A service is a program that runs in the background and provides a service of some sort to the operating service. Examples of a database server include Oracle, Microsoft SQL server and MySQL.

Types of Databases

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 program 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.

Parts of a database

Tables

The foundation of every Relational Database Management System is a database object called table. Every database consists of one or more tables, which store the database's data/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:

- Regular Tables – which stores collections of data
- Reference Tables – which are used to describe a list a values that may appear in another table

Keys

A database key is an attribute utilized to sort and/or identify data in some manner. Each table has a primary key which uniquely identifies records. Foreign keys are utilized to cross-reference data between relational tables.

Relations

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

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.

Database Design Concepts

Introduction

The first step in designing a database is to learn how to identify various data structures. There are different kinds of data structures and forms used when identifying and planning a database. This chapter describes some of the more common data structures and database design and analysis form.

Using design and analysis forms help you develop the critical thinking skills that are required when starting to plan any size database project whether it is a small project or a large scale enterprise solution.

Types of data structures

There are 3 common kinds of data structures that most database designers come across when developing a database. They are:

Singlet

The singlet is a single field that contains discreet information such as a name or a postal code. A singlet can be uniquely identified and named. It normally contains a single value at a time and is normally typed to a specific data type.

Reference

A reference is a field that can contain a single value. However, this value is normal selected from a fixed list. Example reference fields could include a country list or a series of statuses.

The fixed list is normally another table. This type of table is often referred to as being a reference table.

Master-detail

A master detail is a series of fields that are related to a single master record. An example of this includes line items on an invoice or on a receipt.

Database Design & Analysis forms

When doing database analysis, there are 2 common forms used to design. They are the entities form and the fields form. See Appendix A for the forms.

Entities form

The Entities form is used to define tables and their relationships. It is made up of:

- A database name. This can be a temporary name, a general description or a “code” name.

- The revision of the document. Sometimes, it is necessary to make multiple revisions of a given design and it can be useful to help track where you have been thinking about.
- The date the form was filled out.
- A list of entities. An entity is another word for a table. It is made up of:
  - Name. The name of the entity
  - Entity type. This is usually singlet or reference.
  - Relates to. If it is a details table, it usually contains the name of the master record.

    Normally, every entities form has a matching Fields form in order to help describe the database in better detail.

Fields form

The fields form is used to define the fields in an entity. An entity is better commonly known as a table. It is made up of:

- A database name. This database name usually corresponds to the database name on the entities form.
- The revision of the document. Sometimes, it is necessary to make multiple revisions of a given design document. It can be useful to help track what you have been thinking about.
- The date the form was filled out.
- A list of field properties. These include:
  - Field name. The name of the field.
  - Data type. This contains what kind of data will be going into it.
  - Field Type. This usually contains one of the following items:
    - Singlet. Contains a discreet piece of data
    - Reference. Contains a matching field from another table. Also known as a foreign key.
    - Primary Key. Identifies the record as being a unique identifying record.
  - Relates to. When the field is a foreign key, it identifies the entity/table and field that this field is related to.

Normally, a binder is used to hold the project with section tabs to identify each entity and field form collection.

Of course, in modern times, a shared wiki will do this job much more efficiently and flexibly.

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. Once you have identified all the data structures, you start filling out a form for each entity type. Once you have identified all the entities, you then start filling out a fields form for each entity.

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:

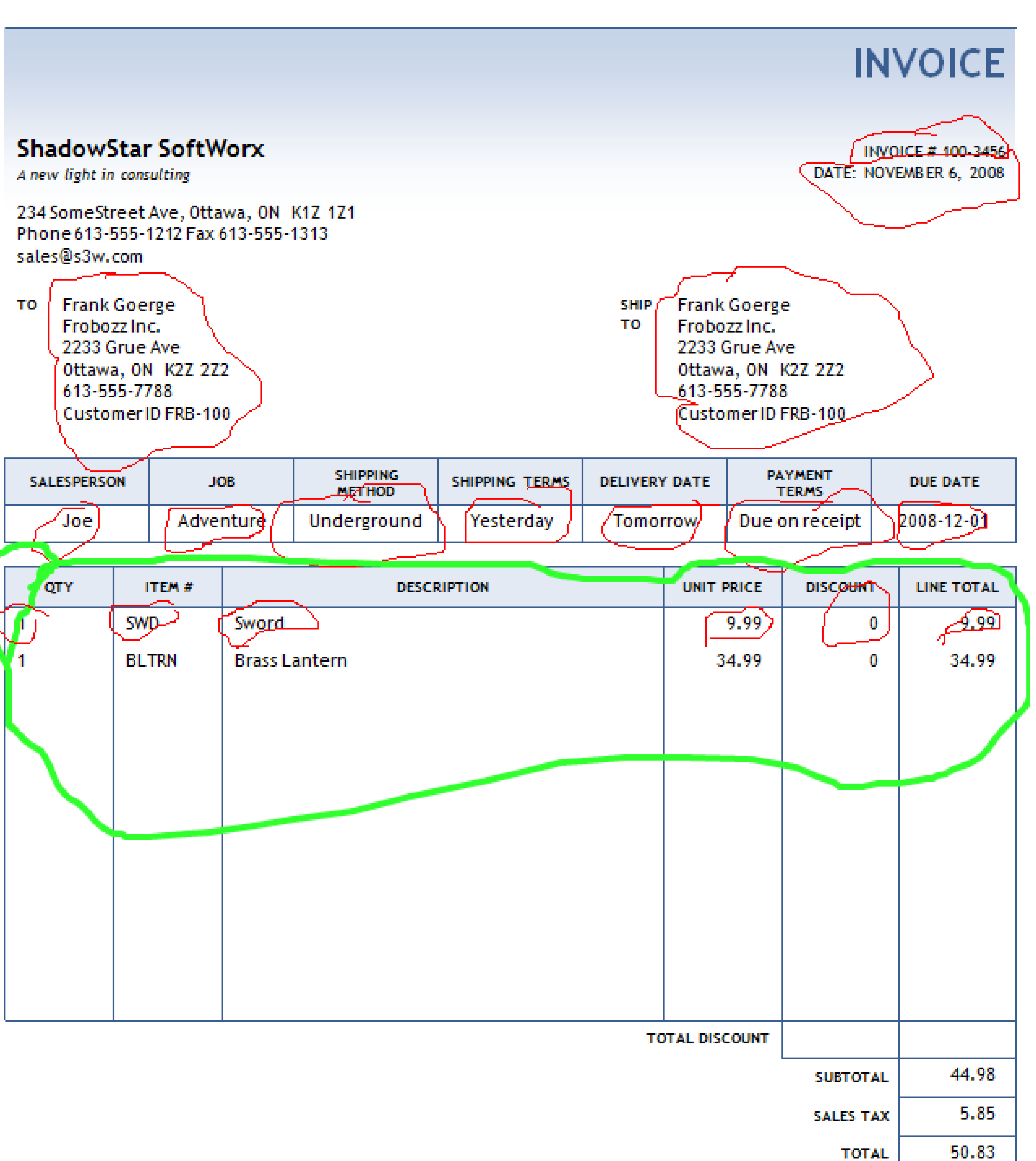
- The invoice number
- The date of the invoice
- The sales person
- Etc…

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…

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.



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!

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. In order 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:

1. List a heading for each component. Try to think about as many different components as possible. Bad ideas can be thrown out later.
2. For each component, list all tables/entities contained therein.
3. For each entity, list all the fields you can think of.
4. If a field is a finite list, mark it as so with a star or in a different color.
5. Once a field is identified as being a finite list, create an entity for it and list it fields.
6. Walk away and have a coffee in another room, or go for a walk in the park. In other words, stop thinking about it.
7. 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

Introduction

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

Character/text data

There are 3 common data types when dealing with character data. These are:

- CHAR(L) – this is used to store fixed length strings such as postal codes or province/state abbreviations. This field defines the maximum length a string can by. L determines the length of the field up to 255 characters.
- VARCHAR(L) – this is used to store variable length strings. L determines the length of the field up to 255 characters.
- TEXT – this is used to store huge amounts of text. The maximum amount you can store in this field type varies from database system to database system. However, for the most part, the TEXT data type will hold enormous amounts of data, in the upwards of hundreds of megabytes of text.

Numeric data

There are many data type to store numeric data.

- SMALLINT – A small integer. The signed range is -32768 to 32767. The unsigned range is 0 to 65535.
- MEDIUMINT – A medium-size integer. The signed range is -8388608 to 8388607. The unsigned range is 0 to 16777215.
- INT, INTEGER – A normal-size integer. The signed range is: -2147483648 to 2147483647. The unsigned range is 0 to 4294967295.
- BIGINT – A large integer. The signed range is -9223372036854775808 to 9223372036854775807. The unsigned range is 0 to 18446744073709551615.
- FLOAT(precision) – floating-point number. Precision can be <=24 for a single-precision floating-point number and between 25 and 53 for a double-precision floating-point number. These types are like the

FLOAT and DOUBLE types described immediately below. FLOAT(X) has the same range as the corresponding FLOAT and DOUBLE types, but the display size and number of decimals are undefined.

- FLOAT(M,D) – A small (single-precision) floating-point number. Allowable values are -3.402823466E+38 to -1.175494351E-38, 0, and 1.175494351E-38 to 3.402823466E+38. If UNSIGNED is specified, negative values are disallowed. The M is the display width and D is the number of decimals. FLOAT without arguments or FLOAT(X) where X <= 24 stands for a single-precision floating-point number.
- DOUBLE(M,D) – A normal-size (double-precision) floating-point number. Allowable values are -1.7976931348623157E+308 to -2.2250738585072014E-308, 0, and 2.2250738585072014E-308 to 1.7976931348623157E+308. If UNSIGNED is specified, negative values are disallowed. The M is the display width and D is the number of decimals. DOUBLE without arguments or FLOAT(X) where 25 <= X <= 53 stands for a double-precision floating-point number.
- DOUBLE PRECISION(M,D), REAL(M,D), DECIMAL(M[,D]) – An unpacked floating-point number. Behaves like a CHAR column: ``unpacked'' means the number is stored as a string, using one character for each digit of the value. The decimal point and, for negative numbers, the `-' sign, are not counted in M (but space for these is reserved). If D is 0, values will have no decimal point or fractional part. The maximum range of DECIMAL values is the same as for DOUBLE, but the actual range for a given DECIMAL column may be constrained by the choice of M and D. If UNSIGNED is specified, negative values are disallowed. If D is omitted, the default is 0. If M is omitted, the default is 10

Date data

There are several data types to allow a database designer to store dates and times. Some do only dates, others do only time and still others allow you to store the entirety of a date time value.

- DATE – a date. The supported range is '1000-01-01' to '9999-12-31'. MySQL displays DATE values in 'YYYY-MM-DD' format, but allows you to assign values to DATE columns using either strings or numbers
- DATETIME – A date and time combination. The supported range is '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. MySQL displays

DATETIME values in 'YYYY-MM-DD HH:MM:SS' format, but allows you to assign values to DATETIME columns using either strings or numbers

- TIMESTAMP – A timestamp. The range is '1970-01-01 00:00:00' to sometime in the year 2037.
- TIME – A time. The range is '-838:59:59' to '838:59:59'. MySQL displays TIME values in 'HH:MM:SS' format, but allows you to assign values to TIME columns using either strings or numbers
- YEAR – A year in 2- or 4-digit format (default is 4-digit). The allowable values are 1901 to 2155, 0000 in the 4-digit year format, and 1970-2069 if you use the 2-digit format (70-69).

Boolean data

Binary data is stored using either the BOOL data type or TINYINT. MySQL uses TINYINT to "emulate" a Boolean data type. Other database systems handle Booleans just fine. The TINYINT is a very small integer. The signed range is -128 to 127. The unsigned range is 0 to 255.

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 on 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, take into account 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.

Normalization

Introduction

The goal of normalization is to remove the following conditions from a database design:

- Duplication of data.
- Inconsistent data.
- Ambiguous data.

Goals of normalization

When normalizing a database you should achieve four goals:

1. Arranging data into logical groups that describe a small part of the whole database design
2. Minimizing the amount of duplicated data
3. Building a database in which can be accessed and manipulated the data in a quick and efficient manner that does not compromise the integrity of the data storage
4. Organizing the data such that modifications only need to occur in one place.

Normal Forms

There are six stages to Normalization but the first three - known as the 1st, 2nd, and 3rd Normal Forms - are adequate for most database designs.

First Normal Form

- There is a Primary Key that uniquely identifies each record.
- There are no repeating fields - you don't have a series of similar fields named 'Item1', 'Item2', 'Item3', etc.
- Remove any such fields into a separate table and include a foreign key that refers back to the matching record in the parent table.

Second Normal Form

- The table is in 1st Normal Form and all fields depend on the entire primary key. This happens automatically if you are using a simple primary key based on a single field.
- If you are using a composite key made up of two or more fields then you have to check that no other field relies on just one of them.

- Remove any such fields into a separate table, together with that part of the primary key on which they depend

Third Normal Form

- The table in 2nd Normal Form and no fields depend on anything but the primary key.
- Remove any such fields to another table, leaving the key in the original table.

Result of normalization

Normalization starts with all the information in a single flat table. Normalization will give you a number of smaller tables that hold the same information in a simpler and unambiguous structure.

One-to-one relations

Normalization should never generate two tables with a one-to-one relationship between them. There is no real reason to split the fields of a single into two table; but you might want to split a table into two for practical reasons such as:

- reduce the number of fields in a table and meet some limit in the programming language.
- store large and rarely-used fields in a separate table so that you do not waste time transferring and processing them during routine operations.
- separate sensitive information from commonly-used fields so that the tables can be stored and backed-up separately.

Many-to-many relations

A many-to-many relationship cannot be implemented directly in most database servers and has to be represented by an intermediate table that holds the links between the two tables.

Sometimes this intermediate table will just hold these two foreign keys. However, you should always create a primary key in the intermediate table so that a unique record can be identified. Sometimes more information will be held, for example a discount offered on a product when ordered by a specific customer.

It is important to remember that normalization will not tell you how best to implement the link between these tables. This is something that the database designer needs to determine.

Denormalization

Normalized tables store data as efficiently and as safely as possible without duplication but you may need to denormalize in order to:

- make reporting easier for non-technical staff.
- create a Data Warehouse.
- improve the speed of execution.
- reduce the length of an SQL statement and comply with language restrictions.

Diagramming

Introduction

Database diagramming allows you to create a visual representation of the structure of a database. It can be a very useful tool to help you visualize your database.

Naming Conventions

In the past, naming conventions for database objects was a loose and free kind of thing. Each company and database designer created their own. Some of the standards are enforced by the underlying database server itself also.

However, with the advent of many popular programing frameworks, a few common naming standard have started to finally evolve. We can thank CakePHP, Ruby on Rails and many others for the sanity that is finally coming to database naming conventions.

The rules are as follows:

- All database objects are lower case. No exceptions.
- Primary key is always called: **id**.
- Primary key is designed to be unique (which it should be since it is a PK). However, automatic value creation is normally strongly encouraged. Auto increment, sequences and GUID style fields are all acceptable.
- Primary descriptor fields are normally called: **name**
- Table names are plural
- Field names are singular
- Foreign key names are the singular form of the parent table with id appended. For example: **customer_id** would refer to the primary key (**id**) of the **customers** table.

Of course, there are exceptions to the rule when it comes to naming conventions. Some field names, such as counters, can be pluralized as needed.

Styles of diagrams

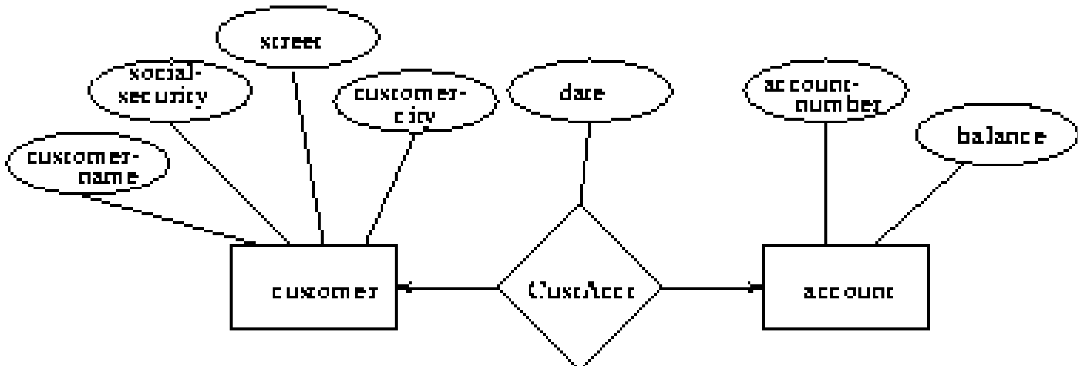
There are many different styles of database diagrams. The most common is the Entity Relationship Diagram. Of course, nothing is ever simple when it comes down to computers. There are a large number of “styles” of ERD diagrams. Describe below are the 2 most common diagram types.

Basic Entity Relationship Diagram

The basic ERD allows you to express the logical structure of the database. It is made up of 4 components:

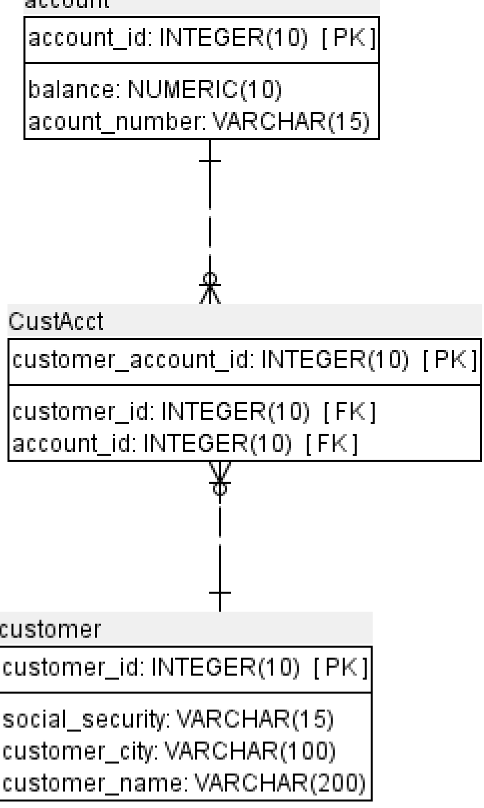
- rectangles that represent entity sets.
- ellipses represent attributes.
- diamonds represent relationship sets.
- lines link attributes to entity sets and entity sets to relationship sets.

This one is useful when planning a large scale project. Below is a sample that maps a customer to a company account.



Physical Entity Relationship Diagram

A more useful diagram is the physical ERD. This diagram contains a full description of all the entities and their attributes (fields). This diagram normally lists all the fields, their data types and key state.



Crow's foot notation

There are several notation methods available for use when diagramming a database. However, one of the most popular is the crow's foot notation. The crow's foot notation allows you to indicate cardinality and modality of a relationship.

Entity – A person, place or thing about which we want to collect and store multiple instances of data. It has a name, which is a noun, and attributes which describe the data we are interested in storing. It also has an identifier, which uniquely identifies one instance of an entity. The attribute which acts as the identifier is marked with an asterisk.

Relationship – Illustrates an association between two entities. It has a name which is a verb. It also has cardinality and modality.

Cardinality and **Modality** are the indicators of the business rules around a relationship. Cardinality refers to the maximum number of times an instance in one entity can be associated with instances in the related entity. 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.

>O————	zero or more

>+———	1 or more
‖———	1 and only 1 (exactly 1)
+O———	zero or 1

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).

Introduction to SQL

Introduction

Over the years, several languages have emerged to allow humans to communicate with database systems. However, SQL has emerged to be the most popular choice. SQL stands for "Structured Query Language". It is often spoken by its acronym letters S-Q-L or spoken acronym "seequal".

The SQL language is broken down into 3 sections:

- DDL or Database definition language. This portion of the language is used to define and create databases and their relationships.
- DML or Data Manipulation language. This portion of the language is used to add, modify and find records.
- DCL or Data Control language. This portion of the language concerns itself with managing and controlling access and permissions to the database.

SQL is best learned by example. The following chapters will explain the basic commands. When reading this section, please note the following information:

- Identifiers such as tables and fieldnames as contained in < >.
- Optional constructs as contained in square brackets [ ].

DDL

Introduction

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.

Create Table Syntax

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

Identifiers (MySQL specific syntax)

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)
);
```

Alter table syntax

Rename

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

Add Column

ALTER TABLE <TABLENAME> ADD COLUMN <COLUMNNAME> <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;
```

Drop table Syntax

DROP <tablename>;

Example

```
DROP employees;
```

DML

Introduction

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

How to "SELECT" some data

Select all columns and all rows

SELECT * FROM <TABLENAME>;

Example

```
SELECT * FROM account_type;
```

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';
```

WHERE clause syntax

FIELDNAME OPERATOR CONDITION

Operators include

- = - 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')
- LIKE – allows wild card matching. Wild card character is **%. 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

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';**

How to "INSERT" some data

Here is the syntax of a basic insert statement.

```
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.

Example

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

How to “UPDATE” some 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';
```

How to “DELETE” some data

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

Example

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

Joins, Ordering and Aggregates

Introduction

In this chapter, we will be covering the following topics:

- Querying multiple tables at once
- Ordering your results
- Aggregate functions.

Joins

Querying data from multiple tables at once requires a technique known as Joining. The syntax below shows an inner join.

```
SELECT * FROM <TABLENAME>
JOIN <TABLENAME2> on (TABLENAME.FIELD=TABLENAME2.FIELD)
[JOIN2]
[WHERE Clause];
```

Example

```
SELECT username FROM employees
JOIN account_types on
(employees.account_type_id=account_types.account_type_id)
WHERE account_type='Manager';
```

In plain English, this query will give me all the usernames who are managers.

There are several types on joins. The most common are:

Inner join

An inner join requires each record in the two joined tables to have a matching record. An inner join essentially combines the records from two tables (A and B) based on a given join clause.

Left, Right & Full join

Also known as outer joins, Left, Right & Full joins do not require each record in the two joined tables to have a matching record. The joined table retains each record—even if no other matching record exists. The terms left join, right join, and full join, depend on which table(s) one retains the rows from (left, right, or both).

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.

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'
```

Aggregates

Another powerful feature or SQL is the ability to summarize our data to determine trends or produce top-level reports. 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

Views

Introduction

Views are a mechanism provided by RDBMS servers that enable database developers to abstract the underlying database structure in order to simplify and, sometimes, secure database queries.

Creating views

Views are created using DDL commands. In order to create a view, you define it using the CREATE VIEW command followed by an SQL statement. The syntax is as follows:

CREATE VIEW <VIEWNAME> AS
<SQL STATEMENT>

For example:

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

If you need to update a view, some database servers allow you to actually replace the view definition. However, the safest approach is to drop and recreate the view. To drop a view, use the DROP VIEW command.

DROP VIEW <viewname>

Using Views

Views can be used in a variety of ways. One of the most basic is to simplify complex queries. For example, enabling a developer to call a simple SELECT * FROM some_view is much better than having them issue complex multi join queries.

The fact that you can use joins with a view, just like a regular table, gives you some small performance advantages as well as reducing the possibility of causing errors down the road.

Subqueries, Unions, Intersects and Excepts

Introduction

Subqueries, unions, intersects and excepts are specialized queries that allow you to merge data from multiple tables in a manner similar to JOINS, except it allows for pulling dissimilar data into a unified result set.

Subqueries

Subqueries is a method that uses the results of one query to modify the data that is used by another query.

Unions

Unions are used to merge the unique records from 2 different data sources.

Intersects

Intersects are used to find the data that exists in both datasources.

Excepts

Excepts are used to find the data that is unique to a single table based on the data from another table. Essentially, this is the opposite of an intersect.

User Defined Functions and Triggers

Introduction

This chapter covers some of the more advanced concepts when it comes to database design and implementation. User defined functions and triggers are used to add procedural functionality to a database.

User defined functions

User defined functions are used to add functionality to a database that is not normally part and parcel with whichever database server you are using.

In MySQL, there are 3 kinds of user defined functions. The first two involve creating C based programs that are compiled and then added to the MySQL instance.

The third method involves creating stored routines. A stored routine is a set of SQL statements that can be stored in the server. Once this has been done, clients don't need to keep reissuing the individual statements but can refer to the stored routine instead.

For complete MySQL syntax refer to:
http://dev.mysql.com/doc/refman/5.1/en/create-procedure.html
http://dev.mysql.com/doc/refman/5.1/en/sql-syntax-compound-statements.html

Function Example

```
USE `test`;
DROP function IF EXISTS `add_random_seed`;

DELIMITER $$
USE `test`$$
CREATE FUNCTION `add_random_seed` (originalString VARCHAR(50))
RETURNS VARCHAR(55)
BEGIN
    DECLARE randString char(10);
    DECLARE returnString VARCHAR(55);
    SET randString = concat(
        char(round(rand()*25)+97),
        char(round(rand()*25)+97),
        char(round(rand()*25)+97),
        char(round(rand()*25)+97),
        char(round(rand()*25)+97)
    );
    SET  returnString = concat(originalString,randString);
    RETURN returnString;
END$$

DELIMITER ;
```

Triggers

Triggers are used to execute code when certain specified events occurs.

Events

There are 3 events with 2 time frames each for a total of 6 events.

Insert

This is triggered when data is being added to the specified table

Update

This event is triggered when data is being altered in the specified table

Delete

This event is triggered when data is being removed from a specified table

Time frames

Before

This time frame occurs before the change, whether insert, update or delete, is committed to the table.

After

This time frame occurs after the change, whether insert, update or delete, is committed to the table.

Trigger languages

Each database server has its own language and method for defining triggers.

MySQL

MySQL has a fairly robust language that can be extended with compiled functions that are created externally. Trigger code is contained directly in the trigger definition.

PostGreSQL

PostGreSQL is one of the most flexible database servers when it comes to triggers. Trigger code is contained in a function that is called when a trigger is fired. Although this make creating triggers a little more complicated, it also allows for a maximum amount of flexibility. Triggers do not need to be recreated when functionality changes, instead, just the underlying function needs changing. Also PostGreSQL provides the most languages for creating triggers. These languages include, but aren't limited to:

- Perl (trusted and untrusted)
- Python (trusted and untrusted)
- TCL
- PL/SQL which is about 90% compatible with Oracles PL/SQL
- SQL

- Java

Oracle

Oracle uses PL/SQL which is a very mature trigger language that many other database servers use as a template for their own languages (MySQL ad PostGreSQL).

Sybase/MS-SQL server

Both Sybase and MS-SQL server use a language call Transact-SQL (T-SQL) because of their shared heritage. This is also a very mature language that allows for complex functionality. MS-SQL Server has also started allowing triggers to be written using C#.

Creating triggers in MySQL

Creating triggers is similar to creating procedures and functions. See the following for full syntax:

http://dev.mysql.com/doc/refman/5.1/en/create-trigger.html

http://dev.mysql.com/doc/refman/5.1/en/trigger-syntax.html

Trigger Example

```
DELIMITER $$

DROP TRIGGER `update_data `$$

CREATE TRIGGER `update_data` AFTER UPDATE on `data_table`
FOR EACH ROW
BEGIN
    IF (NEW.field1 != OLD.field1) THEN
        INSERT INTO data_tracking set old_value = OLD.field1, new_value =
NEW.field1, field = "field1";
    END IF;
END$$

DELIMITER ;
```

Specialty Queries

Introduction

When it comes to searching for data, sometimes there is some queries that are real head scratchers. Listed below are some example specialty queries that are always a pain.

Finding and deleting duplicate rows

Every time I have had to delete duplicate rows, I have often had to refer to one article or other. Here is one of the better ones I have found.

This article is courtesy of XAPRB at *http://www.xaprb.com/blog/2006/10/09/how-to-find-duplicate-rows-with-sql*

This article shows how to find duplicated rows in a database table. This is a very common beginner question. The basic technique is straightforward. I'll also show some variations, such as how to find "duplicates in two columns" (a recent question on the #mysql IRC channel).

How to find duplicated rows

The first step is to define what exactly makes a row a duplicate of another row. Most of the time this is easy: they have the same value in some column. I'll take this as a working definition for this article, but you may need to alter the queries below if your notion of "duplicate" is more complicated.

For this article, I'll use this sample data:

```
create table test(id int not null primary key, day date not null);

insert into test(id, day) values(1, '2006-10-08');
insert into test(id, day) values(2, '2006-10-08');
insert into test(id, day) values(3, '2006-10-09');

select * from test;
+----+------------+
| id | day        |
+----+------------+
|  1 | 2006-10-08 |
|  2 | 2006-10-08 |
|  3 | 2006-10-09 |
+----+------------+
```

The first two rows have the same value in the `day` column, so if I consider those to be duplicates, here's a query to find them. The query uses a `GROUP BY` clause to put all the rows with the same `day` value into one "group" and then count the size of the group:

```
select day, count(*) from test GROUP BY day;
+------------+----------+
| day        | count(*) |
+------------+----------+
| 2006-10-08 |        2 |
| 2006-10-09 |        1 |
+------------+----------+
```

The duplicated rows have a count greater than one. If you only want to see rows that are duplicated, you need to use a `HAVING` clause (not a `WHERE` clause), like this:

```
select day, count(*) from test group by day HAVING count(*) > 1;
+------------+----------+
| day        | count(*) |
+------------+----------+
| 2006-10-08 |        2 |
+------------+----------+
```

This is the basic technique: group by the column that contains duplicates, and show only those groups having more than one row.

Why can't you use a `WHERE` clause?

A `WHERE` clause filters the rows *before* they are grouped together. A `HAVING` clause filters them *after* grouping. That's why you can't use a `WHERE` clause in the above query.

How to delete duplicate rows

A related question is how to delete the 'duplicate' rows once you find them. A common task when cleaning up bad data is to delete all but one of the duplicates, so you can put proper indexes and primary keys on the table, and prevent duplicates from getting into the table again.

Again, the first thing to do is make sure your definition is clear. Exactly which row do you want to keep? The 'first' one? The one with the largest value of some column? For this article, I'll assume you want to keep the 'first' row — the one with the smallest value of the `id` column. That means you want to delete every other row.

Probably the easiest way to do this is with a temporary table. Especially in MySQL, there are some restrictions about selecting from a table and updating it in the same query. You can get around these, as I explain in my article How to select from an update target in MySQL, but I'll just avoid these complications and use a temporary table.

The exact definition of the task is to **delete every row that has a duplicate, except the row with the minimal value of `id` for that group**. So you need to find not only the rows where there's more than one in the group, you also need to find **the row you want to keep**. You can do that with the `MIN()` function. Here are some queries to create the temporary table and find the data you need to do the `DELETE`:

```
create temporary table to_delete (day date not null, min_id int not null);

insert into to_delete(day, min_id)
   select day, MIN(id) from test group by day having count(*) > 1;

select * from to_delete;
+------------+--------+
| day        | min_id |
+------------+--------+
| 2006-10-08 |      1 |
+------------+--------+
```

Now that you have this data, you can proceed to delete the 'bad' rows. There are many ways to do this, and some are better than others (see my article about many-to-one problems in SQL), but again I'll avoid the finer points and just show you a standard syntax that ought to work in any RDBMS that supports subqueries:

```
delete from test
   where exists(
      select * from to_delete
      where to_delete.day = test.day and to_delete.min_id <> test.id
   )
```

If your RDBMS does not support subqueries, or if it's more efficient, you may wish to do a multi-table delete. The syntax for this varies between systems, so you need to consult your system's documentation. You may also need to do all of this in a transaction to avoid other users changing the data while you're working, if that's a concern.

How to find duplicates in multiple columns

Someone recently asked a question similar to this on the #mysql IRC channel:

I have a table with columns `b` and `c` that links two other tables `b` and `c`, and I want to find all rows that have duplicates in either `b` or `c`.

It was difficult to understand exactly what this meant, but after some conversation I grasped it: the person wanted to be able to put unique indexes on columns `b` and `c` separately.

It's pretty easy to find rows with duplicate values in one or the other column, as I showed you above: just group by that column and count the group size. And it's easy to find entire rows that are exact duplicates of other rows: just group by as many columns as you need. But it's harder to identify rows that have either a duplicated `b` value or a duplicated `c` value. Take the following sample table, which is roughly what the person described:

```
create table a_b_c(
   a int not null primary key auto_increment,
   b int,
   c int
);

insert into a_b_c(b,c) values (1, 1);
insert into a_b_c(b,c) values (1, 2);
insert into a_b_c(b,c) values (1, 3);
insert into a_b_c(b,c) values (2, 1);
insert into a_b_c(b,c) values (2, 2);
insert into a_b_c(b,c) values (2, 3);
insert into a_b_c(b,c) values (3, 1);
insert into a_b_c(b,c) values (3, 2);
insert into a_b_c(b,c) values (3, 3);
```

Now, you can easily see there are some 'duplicate' rows in this table, but no two rows actually have the same tuple `{b, c}`. That's why this is a bit more difficult to solve.

Queries that don't work

If you group by two columns together, you'll get various results depending on how you group and count. This is where the IRC user was getting stumped. Sometimes queries would find some duplicates, but not others. Here are some of the things this person tried:

```
select b, c, count(*) from a_b_c
group by b, c
having count(distinct b > 1)
   or count(distinct c > 1);
```

This query returns every row in the table, with a `COUNT(*)` of 1, which seems to be wrong behavior, but it's actually not. Why? Because the `> 1` is inside the `COUNT()`. It's pretty easy to miss, but this query is actually the same as

```
select b, c, count(*) from a_b_c
group by b, c
having count(1)
   or count(1);
```

Why? Because `(b > 1)` is a boolean expression. That's not what you want at all. You want

```
select b, c, count(*) from a_b_c
group by b, c
having count(distinct b) > 1
   or count(distinct c) > 1;
```

This returns zero rows, of course, because there are no duplicate {b, c} tuples. The person tried many other combinations of HAVING clauses and ORs and ANDs, grouping by one column and counting the other, and so forth:

```
select b, count(*) from a_b_c group by b having count(distinct c) > 1;
+------+----------+
| b    | count(*) |
+------+----------+
|    1 |        3 |
|    2 |        3 |
|    3 |        3 |
+------+----------+
```

Nothing found all the duplicates, though. What I think made it most frustrating is that it partially worked, making the person think it was almost the right query… perhaps just another variation would get it…

In fact, it's **impossible** to do with this type of simple `GROUP BY` query. Why is this? It's because when you group by one column, you distribute like values of the *other* column across multiple groups. You can see this visually by ordering by those columns, which is what grouping does. First, order by column `b` and see how they are grouped:

a	b	c
7	1	1
8	1	2
9	1	3
10	2	1
11	2	2
12	2	3
13	3	1
14	3	2
15	3	3

When you order (group) by column `b`, the duplicate values in column `c` are distributed into different groups, so you can't count them with `COUNT(DISTINCT c)` as the person was trying to do. Aggregate functions such as `COUNT()` only operate within a group, and have no access to rows that are placed in other groups. Similarly, when you order by `c`, the duplicate values in column `b` are distributed into different groups. It is not possible to make this query do what's desired.

Some correct solutions

Probably the simplest solution is to *find the duplicates for each column separately* and `UNION` them together, like this:

```
select b as value, count(*) as cnt, 'b' as what_col
 from a_b_c group by b having count(*) > 1
 union
 select c as value, count(*) as cnt, 'c' as what_col
 from a_b_c group by c having count(*) > 1;
+-------+-----+----------+
| value | cnt | what_col |
+-------+-----+----------+
|     1 |   3 | b        |
|     2 |   3 | b        |
|     3 |   3 | b        |
|     1 |   3 | c        |
|     2 |   3 | c        |
|     3 |   3 | c        |
+-------+-----+----------+
```

The `what_col` column in the output indicates what column the duplicate value was found in. Another approach is to use subqueries:

```
select a, b, c from a_b_c
 where b in (select b from a_b_c group by b having count(*) > 1)
    or c in (select c from a_b_c group by c having count(*) > 1);
+----+------+------+
| a  | b    | c    |
+----+------+------+
|  7 |    1 |    1 |
|  8 |    1 |    2 |
|  9 |    1 |    3 |
| 10 |    2 |    1 |
| 11 |    2 |    2 |
| 12 |    2 |    3 |
| 13 |    3 |    1 |
| 14 |    3 |    2 |
| 15 |    3 |    3 |
+----+------+------+
```

This is probably much less efficient than the `UNION` approach, and will show every duplicated row, not just the values that are duplicated. Still another approach is to do self-joins against grouped subqueries in the `FROM`clause. This is more complicated to write correctly, but might be necessary for some complex data, or for efficiency:

```
select a, a_b_c.b, a_b_c.c
from a_b_c
   left outer join (
      select b from a_b_c group by b having count(*) > 1
   ) as b on a_b_c.b = b.b
   left outer join (
      select c from a_b_c group by c having count(*) > 1
   ) as c on a_b_c.c = c.c
where b.b is not null or c.c is not null
```

Any of these queries will do, and I'm sure there are other ways too. If you can use `UNION`, it's probably the easiest.

Updating rows based on a join

Another difficult situation is when you must update data in one table based on the contents of a different table. Once again, I often find myself having to look up how to do this and I found the following tutorial which is very good.

This tutorial is found at: http://www.mysqltutorial.org/mysql-update-join/

Summary: in this tutorial, you will learn how to use **MySQL UPDATE JOIN** statement to perform cross-table update. We will show you step by step how to use `INNER JOIN` clause and `LEFT JOIN` clause with the `UPDATE` statement.

MySQL UPDATE JOIN syntax

You often use `JOIN` clauses to query records in a table that have (in case of INNER JOIN) or do not have (in case of LEFT JOIN) corresponding records in another table. In MySQL, you can use the `JOIN` clauses in the UPDATE statement to perform cross-table update.

The syntax of the MySQL `UPDATE JOIN` is as follows:

```
UPDATE T1, T2,
[INNER JOIN | LEFT JOIN] T1 ON T1.C1 = T2. C1
SET T1.C2 = T2.C2,
   T2.C3 = expr
WHERE condition
```

Let's examine the MySQL `UPDATE JOIN` syntax in greater detail:

- First, you specify the main table (`T1`) and the table that you want the main table to join to (`T2`) after the `UPDATE` clause. Notice that you must specify at least one table after the `UPDATE` clause. The data in the table that is not specified after the `UPDATE` clause is not updated.
- Second, you specify a kind of join you want to use i.e., either `INNER JOIN` or `LEFT JOIN` and a join condition. Notice that the `JOIN` clause must appear right after the `UPDATE` clause.
- Third, you assign new values to the columns in T1 and/or T2 tables that you want to update.
- Fourth, the condition in the `WHERE` clause allows you to limit the rows to update.

If you follow the UPDATE statement tutorial, you notice that there is another way to update data cross-table using the following syntax:

```
UPDATE T1, T2
SET T1.c2 = T2.c2,
     T2.c3 = expr
WHERE T1.c1 = T2.c1 AND condition
```

This `UPDATE` statement works the same as `UPDATE JOIN` with implicit `INNER JOIN` clause. It means you can rewrite the above statement as follows:

```
UPDATE T1,T2
INNER JOIN T2 ON T1.C1 = T2.C1
SET T1.C2 = T2.C2,
    T2.C3 = expr
WHERE condition
```

Let's take a look at some examples of using the `UPDATE JOIN` statement to having a better understanding.

MySQL UPDATE JOIN examples

We are going to use a new sample database in these examples. The sample database contains 2 tables:

- `employees` table stores employee data with employee id, name, performance and salary.
- `merits` table stores performance and merit's percentage.

The SQL script for creating and loading data in this sample database is as follows:

```
CREATE DATABASE IF NOT EXISTS empdb;
  -- create tables
CREATE TABLE merits (
  performance INT(11) NOT NULL,
  percentage FLOAT NOT NULL,
  PRIMARY KEY (performance)
);

CREATE TABLE employees (
  emp_id INT(11) NOT NULL AUTO_INCREMENT,
  emp_name VARCHAR(255) NOT NULL,
  performance INT(11) DEFAULT NULL,
  salary FLOAT DEFAULT NULL,
  PRIMARY KEY (emp_id),
  CONSTRAINT fk_performance
  FOREIGN KEY(performance)
  REFERENCES merits(performance)
);
-- insert data for merits table
INSERT INTO merits(performance,percentage)
VALUES(1,0),
        (2,0.01),
        (3,0.03),
        (4,0.05),
        (5,0.08);
-- insert data for employees table
INSERT INTO employees(emp_name,performance,salary)
VALUES('Mary Doe', 1, 50000),
     ('Cindy Smith', 3, 65000),
     ('Sue Greenspan', 4, 75000),
     ('Grace Dell', 5, 125000),
     ('Nancy Johnson', 3, 85000),
     ('John Doe', 2, 45000),
     ('Lily Bush', 3, 55000);
```

MySQL UPDATE JOIN example with INNER JOIN clause

Suppose you want to adjust the salary of employees based on their performance. The merit's percentages are stored in the `merits` table therefore you have to use `UPDATE INNER JOIN` statement to adjust the salary of employees in the `employees` table based on the `percentage` stored in the `merits` table. The link between the `employees` and `merit` tables is `performance` field. See the following query:

```
UPDATE employees
INNER JOIN merits ON employees.performance = merits.performance
SET salary = salary + salary * percentage
```

emp_id	emp_name	performance	salary
1	Mary Doe	1	50000
2	Cindy Smith	3	66950
3	Sue Greenspan	4	78750
4	Grace Dell	5	135000
5	Nancy Johnson	3	87550
6	John Doe	2	45450
7	Lily Bush	3	56650

How the query works.

- We specify only the `employees` table after `UPDATE` clause because we want to update data in the `employees` table only.
- For each employee record in the `employees` table, the query checks the its performance value against the performance value in the `merits` table. If it finds a match, it gets the `percentage` in the `merits` table and update the `salary` column in the `employees` table.
- Because we omit the `WHERE` clause in the `UPDATE` statement, all the records in the `employees` table get updated.

MySQL UPDATE JOIN example with LEFT JOIN

Suppose the company hires two more employees:

```
INSERT INTO employees(emp_name,performance,salary)
VALUES('Jack William',NULL,43000),
      ('Ricky Bond',NULL,52000);
```

Because these employees are new hires so their performance data is not available or `NULL`.

To increase the salary for new hires, you cannot use the `UPDATE INNER JOIN` statement because their performance data is not available in the `merit` table. This is why the `UPDATE LEFT JOIN` comes to the rescue.

The `UPDATE LEFT JOIN` statement basically updates a record in a table when it does not have a corresponding record in another table. For example, you can increase the salary for a new hire by 1.5% using the following statement:

```
UPDATE employees
LEFT JOIN merits ON employees.performance = merits.performance
SET salary = salary + salary * 0.015;
WHERE merits.percentage IS NULL
```

emp_id	emp_name	performance	salary
1	Mary Doe	1	50000
2	Cindy Smith	3	66950
3	Sue Greenspan	4	78750
4	Grace Dell	5	135000
5	Nancy Johnson	3	87550
6	John Doe	2	45450
7	Lily Bush	3	56650
15	Jack William	NULL	43645
16	Ricky Bond	NULL	52780

In this tutorial, we have shown you how to use MySQL `UPDATE JOIN` with `INNER JOIN` and `LEFT JOIN` to perform cross-table update.

Entities Form

Database Name: ______________________ Revision: ________________

Date: ________________

Author: ________________

Entity Name	Entity Type	Relates to

Notes:

____

____

____

____

Fields Form

Database Name: ____________________ Revision: ______________

Entity Name: ____________________ Date: ______________

Author: ______________

Field Name	Data type	Field Type	Relates to

Notes:

____

____

____

____