Title

Abstract

Introduction

Chapter 2- SYSTEM STUDY & ANALYSIS

• 2.1. System Overview

• 2.2. Problem Description

• 2.3. Existing System, Proposed System

**HARDWARE INTERFACES:**

Processor : Dual core and above

RAM : 1 GB and above

Hard Disk Utilization : 40 GB and above

Input Devices : Mouse, Keyboard, etc

Output Devices : Monitor, Printer, etc

**SOFTWARE INTERFACES:**

Software requirements are:

Operating System: Any

Server: Webrick

Front-end: Rails 3

Back-end: MySQL Server 5.3.1

**Feasibility Study**

**Economic Feasibility**

Economic feasibility attempts 2 weigh the costs of developing and implementing a new system, against the benefits that would accrue from having the new system in place. This feasibility study gives the top management the economic justification for the new system.

A simple economic analysis which gives the actual comparison of costs and benefits are much more meaningful in this case. In addition, this proves to be a useful point of reference to compare actual costs as the project progresses. There could be various types of intangible benefits on account of automation. These could include increased customer satisfaction, improvement in product quality better decision making timeliness of information, expediting activities, improved accuracy of operations, better documentation and record keeping, faster retrieval of information, better employee morale.

**Operational Feasibility**

Proposed project is beneficial only if it can be turned into information systems that will meet the organizations operating requirements. Simply stated, this test of feasibility asks if the system will work when it is developed and installed. Are there major barriers to Implementation? Here are questions that will help test the operational feasibility of a project:

Is there sufficient support for the project from management from users? If the current system is well liked and used to the extent that persons will not be able to see reasons for change, there may be resistance.

Are the current business methods acceptable to the user? If they are not, Users may welcome a change that will bring about a more operational and useful systems.

Have the user been involved in the planning and development of the project?

Early involvement reduces the chances of resistance to the system and in general and increases the likelihood of successful project.

Since the proposed system was to help reduce the hardships encountered. In the existing manual system, the new system was considered to be operational feasible.

**Technical Feasibility**

Evaluating the technical feasibility is the trickiest part of a feasibility study. This is because, .at this point in time, not too many detailed design of the system, making it difficult to access issues like performance, costs on (on account of the kind of technology to be deployed) etc. A number of issues have to be considered while doing a technical analysis.

Understand the different technologies involved in the proposed system before commencing the project we have to be very clear about what are the technologies that are to be required for the development of the new system. Find out whether the organization currently possesses the required technologies. Is the required technology available with the organization?

Chapter 3- DESIGN

• 3.1. INTRODUCTION

• 3.2. BUSINESS RULE

• 3.3. FINAL ENTITIES AND ATTRIBUTES

• 3.4. DATA FLOW DIAGRAMS

• 3.5.ENTITIY-RELATIONSHIP MODEL

• 3.6. PHYSICAL DATABASE DESIGN

**Chapter 4 - SOFTWARE DEVELOPMENT METHODOLOGY**

**Ruby on Rails:**

**Ruby:**

Ruby is a pure object oriented programming language. It was created in 1993 by Yukihiro Matsumoto of Japan. You can find the name Yukihiro Matsumoto on the Ruby mailing list at www.ruby-lang.org. Matsumoto is also known as Matz in the Ruby community.

Ruby is "A Programmer's Best Friend".

Ruby has features that are similar to those of Smalltalk, Perl, and Python. Perl, Python, and Smalltalk are scripting languages. Smalltalk is a true object-oriented language. Ruby, like Smalltalk, is a perfect object-oriented language. Using Ruby syntax is much easier than using Smalltalk syntax.

Features of Ruby

* Ruby is an open-source and is freely available on the Web, but it is subject to a license.
* Ruby is a general-purpose, interpreted programming language.
* Ruby is a true object-oriented programming language.
* Ruby is a server-side scripting language similar to Python and PERL.
* Ruby can be used to write Common Gateway Interface (CGI) scripts.
* Ruby can be embeded into Hypertext Markup Language (HTML).
* Ruby has a clean and easy syntax that allows a new developer to learn Ruby very quickly and easily.
* Ruby has similar syntax to that of many programming languages such as C++ and Perl.
* Ruby is very much scalable and big programs written in Ruby are easily maintainable.
* Ruby can be used for developing Internet and intranet applications.
* Ruby can be installed in Windows and POSIX environments.
* Ruby support many GUI tools such as Tcl/Tk, GTK, and OpenGL.
* Ruby can easily be connected to DB2, MySQL, Oracle, and Sybase.
* Ruby has a rich set of built-in functions which can be used directly into Ruby scripts.

**Interactive Ruby (IRb):**

Interactive Ruby (IRb) provides a shell for experimentation. Within the IRb shell, you can immediately view expression results, line by line. This tool comes along with Ruby installation so you have nothing to do extra to have IRb working.

Ruby is a perfect Object Oriented Programming Language. The features of the object-oriented programming language include:

• Data Encapsulation:

• Data Abstraction:

• Polymorphism:

• Inheritance:

An object-oriented program involves classes and objects. A class is the blueprint from which individual objects are created. In object-oriented terms, we say that your bicycle is an instance of the class of objects known as bicycles.

**Defining a Class in Ruby:**

To implement object-oriented programming by using Ruby, you need to first learn how to create objects and classes in Ruby. A class in Ruby always starts with the keyword class followed by the name of the class. The name should always be in initial capitals. The class Customer can be displayed as:

**class Customer**

**end**

You terminate a class by using the keyword end. All the data members in the class are between the class definition and the end keyword.

Rails:

Rails is a web development framework written in the Ruby language. It is designed to make programming web applications easier by making several assumptions about what every developer needs to get started. It allows you to write less code while accomplishing more than many other languages and frameworks. Longtime Rails developers also report that it makes web application development more fun.

Rails is opinionated software. That is, it assumes that there is a best way to do things, and it’ s designed to encourage that best way - and in some cases to discourage alternatives. If you learn”The Rails Way” you’ ll probably discover a tremendous increase in productivity. If you persist in bringing old habits from other languages to your Rails development, and trying to use patterns you learned elsewhere, you may have a less happy experience.

The Rails philosophy includes several guiding principles:

* DRY - ”Don’ t Repeat Yourself” - suggests that writing the same code over and over again is a bad thing.
* Convention Over Configuration - means that Rails makes assumptions about what you want to do and how you’ re going to do it, rather than letting you tweak every little thing through endless configuration files.
* REST is the best pattern for web applications - organizing your application around resources and standard HTTP verbs is the fastest way to go.

**The MVC Architecture**

Rails is organized around the Model, View, Controller architecture, usually just called MVC. MVC benefits include:

* Isolation of business logic from the user interface
* Ease of keeping code DRY
* Making it clear where different types of code belong for easier maintenance

**Models**

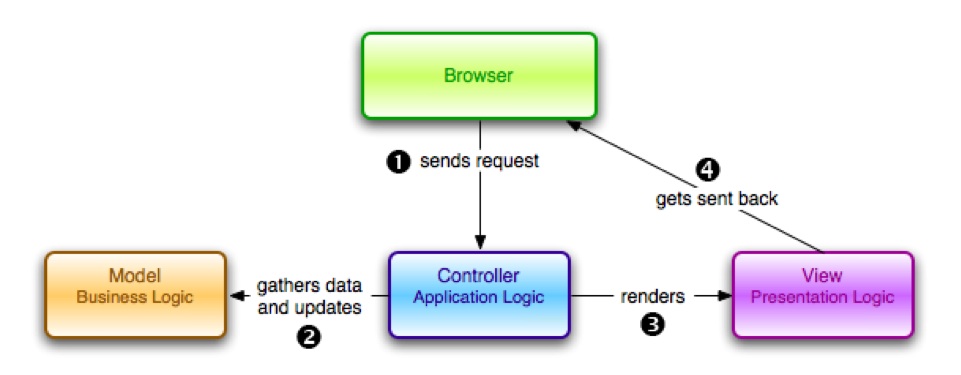
A model represents the information (data) of the application and the rules to manipulate that data. In the case of Rails, models are primarily used for managing the rules of interaction with a corresponding database table. In most cases, one table in your database will correspond to one model in your application. The bulk of your application’ s business logic will be concentrated in the models.

**Views**

Views represent the user interface of your application. In Rails, views are often HTML files with embedded Ruby code that performs tasks related solely to the presentation of the data. Views handle the job of providing data to the web browser or other tool that is used to make requests from your application.

**Controllers**

Controllers provide the ”glue” between models and views. In Rails, controllers are responsible for processing the incoming requests from the web browser, interrogating the models for data, and passing that data on to the views for presentation.



**The Components of Rails**

Rails provides a full stack of components for creating web applications, including:

* Action Controller
* Action View
* Active Record
* Action Mailer
* Active Resource
* Railties
* Active Support

**Action Controller**

Action Controller is the component that manages the controllers in a Rails application. The Action Controller framework processes incoming requests to a Rails application, extracts parameters, and dispatches them to the intended action. Services provided by Action Controller include session management, template rendering, and redirect management.

**Action View**

Action View manages the views of your Rails application. It can create both HTML and XML output by default. Action View manages rendering templates, including nested and partial templates, and includes built-in AJAX support.

**Active Record**

Active Record is the base for the models in a Rails application. It provides database independence, basic CRUD functionality, advanced finding capabilities, and the ability to relate models to one another, among other services.

**Action Mailer**

Action Mailer is a framework for building e-mail services. You can use Action Mailer to send emails based on flexible templates, or to receive and process incoming email.

**Active Resource**

Active Resource provides a framework for managing the connection between business objects an RESTful web services. It implements a way to map web-based resources to local objects with CRUD semantics.

**Railties**

Railties is the core Rails code that builds new Rails applications and glues the various frameworks together in any Rails application.

**Active Support**

Active Support is an extensive collection of utility classes and standard Ruby library extensions that are used in the Rails, both by the core code and by your applications.

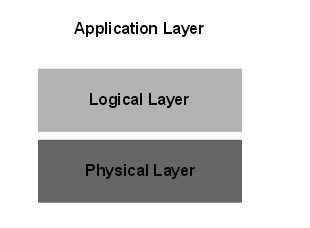
**GEMS:**

Ruby Gems is a package manager for the Ruby programming language that provides a standard format for distributing Ruby programs and libraries (in a self-contained format called a "gem"), a tool designed to easily manage the installation of gems, and a server for distributing them.

**MySql:**

**General RDBMS Architecture**

It was found in all of the consulted resources that all database systems, can be viewed as a Garlan & Shaw layered architecture at the highest level of abstraction. It has three main components as outlined in the following diagram:



Database Layered Architecture

**Application Layer**

The application layer represents the interface for all the users of the system; it essentially provides the means by which the outside world can interact with the database server. In general, it has been found that users can be categorized into four main groups:

Sophisticated users interact with the system without using any form of application; they form their requests directly with the use of a database query language.

Specialized users are application programmers who write specialized database applications that do not fit into the traditional data processing framework.

Naïve users are unsophisticated users who interact with the system by invoking one of the permanent application programs that have been previously written.

Database Administrators have complete control over the entire database system. They have a wide variety of responsibilities, including schema definition, the granting of access authorization, as well as the specification of integrity constraints.

All database systems provide extensive services for each of these groups; it is then left to the logical layer to appropriately process the varying requests.

**Logical Layer**

The core functionality of the RDBMS is represented in the logical layer of the architecture; it is in this portion of the system that there are a wide variety of vendor specific implementations However, upon reading a number of general database management texts, it was found that general core logical functionality can indeed be abstracted. The following diagram is a high level abstraction of the modules found in any robust RDBMS.

**Physical Layer**

The RDBMS is responsible for the storage of a variety of information, which is kept in secondary storage and accessed via the storage manager. The main types of data kept in the system are:

1. Data files, which store the user data in the database

2. Data dictionary, which stores metadata about the structure of the database

3. Indices, which provide fast access to data items that hold particular values

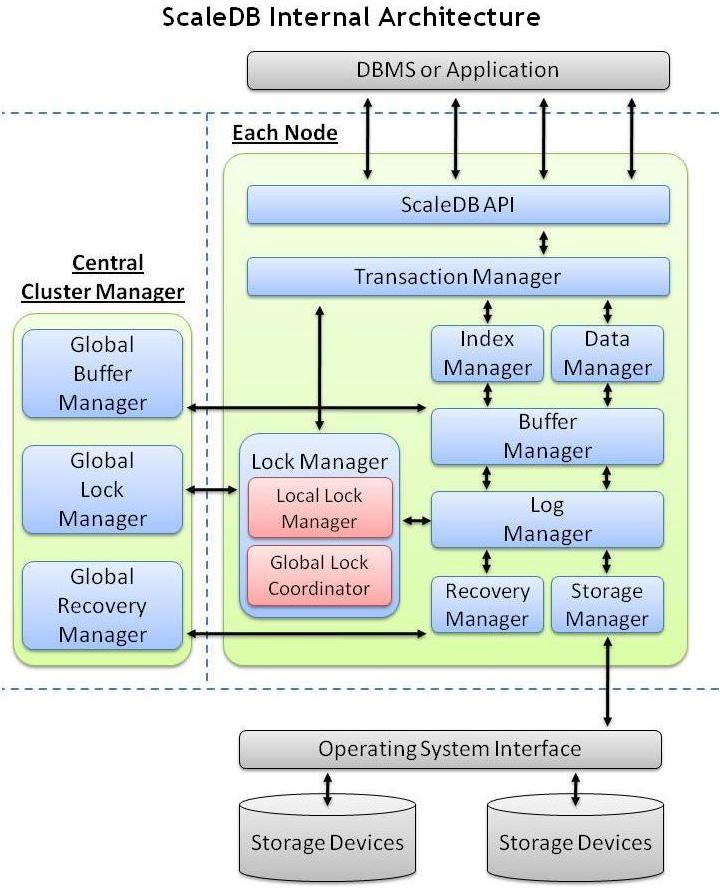
4. Statistical Data, which store statistical information about the data in the database; it is used by the query processor to select efficient ways to execute a query.

5. Log Information, used to keep track of executed queries such that the recovery manager can use the information to successfully recover the database in the case of a system crash.

**MySQL Architecture**

Once the research pertaining to general RDBMS architectures was complete, the specific MySQL documentation could then be examined in order to extract the vendor specific conceptual architecture. The ensuing section begins by describing the detail of the layered architecture from Figure 2 and how the specific MySQL components map onto Figure 1. As mentioned above, however, the core of the functionality is found in the logical layer; the remainder of the section details the key subsystems in this layer. The conceptual architecture of MySQL is illustrated in Figure 3 below.

The architecture depicted in Figure 3 is a view of the control flow of the My SQL system. It is an expansion of the simple architecture described in Figure 2. It should be noted that the architecture described is a layered architecture as described by Garlan and Shaw. There exists a pipeline architecture, also described by Garlan and Shaw, represented in the Query Processing layer between the Embedded DML Precompiled and the Execution Engine. For sake of simplicity, flow back up the architecture has been left out and should be deemed as implicit. For example, calling a simple SQL command, such as SELECT \*, would require information to be brought back up the system. This flow is implied and is not mentioned in the diagram. Each layer in Figure 3 is described below with the core of the functionality found in the Logical Layer. As a result, this is the layer that will be directly explained in further detail.



**Application Layer**

The MySQL application layer is where the clients and users interact with the MySQL RDBMS. There are three components in this layer as can be seen in the layered MySQL architecture diagram in Figure 3. These components illustrate the different kinds of users that can interact with the MySQL RDBMS, which are the administrators, clients and query users. The administrators use the administrative interface and utilities. In MySQL, some of these utilities are mysql admin which performs tasks like shutting down the server and creating or dropping databases, which help to perform table analysis and optimization as well as crash recovery if the tables become damaged, and mysql dump for backing up the database or copying databases to another server. Clients communicate to the MySQL RDBMS through the interface or utilities. The client interface uses MySQL APIs for various different programming languages such as the C API, DBI API for Perl, PHP API, Java API, Python API, MySQL C++ API and Tcl. Query users interact with the MySQL RDBMS through a query interface that is mysql. Mysql is a monitor (interactive program) that allows the query users to issue SQL statements to the server and view the results.

**Logical Layer**

It was found that MySQL does indeed have a logical architecture that is virtually identical to the one depicted. The MySQL documentation gave an indication as to precisely how these modules could be further broken down into subsystems arranged in a layered hierarchy corresponding to the layered architecture in Garlan and Shaw. The following section details these subsystems and the interactions within them

**Chapter 5 - SYSTEM TESTING**

Software Testing is the process used to help identify the correctness, completeness, security, and quality of developed computer software. Testing is a process of technical investigation, performed on behalf of stakeholders, that is intended to reveal quality-related information about the product with respect to the context in which it is intended to operate. This includes, but is not limited to, the process of executing a program or application with the intent of finding errors. Quality is not an absolute; it is value to some person. With that in mind, testing can never completely establish the correctness of arbitrary computer software; testing furnishes a criticism or comparison that compares the state and behaviour of the product against a specification. An important point is that software testing should be distinguished from the separate discipline of Software Quality Assurance (SQA), which encompasses all business process areas, not just testing.

There are many approaches to software testing, but effective testing of complex products is essentially a process of investigation, not merely a matter of creating and following routine procedure. One definition of testing is "the process of questioning a product in order to evaluate it", where the "questions" are operations the tester attempts to execute with the product, and the product answers with its behavior in reaction to the probing of the tester[citation needed]. Although most of the intellectual processes of testing are nearly identical to that of review or inspection, the word testing is connoted to mean the dynamic analysis of the product—putting the product through its paces. Some of the common quality attributes include capability, reliability, efficiency, portability, maintainability, compatibility and usability. A good test is sometimes described as one which reveals an error; however, more recent thinking suggests that a good test is one which reveals information of interest to someone who matters within the project community.

In general, software engineers distinguish software faults from software failures. In case of a failure, the software does not do what the user expects. A fault is a programming error that may or may not actually manifest as a failure. A fault can also be described as an error in the correctness of the semantic of a computer program. A fault will become a failure if the exact computation conditions are met, one of them being that the faulty portion of computer software executes on the CPU. A fault can also turn into a failure when the software is ported to a different hardware platform or a different compiler, or when the software gets extended. Software testing is the technical investigation of the product under test to provide stakeholders with quality related information.

Software testing may be viewed as a sub-field of Software Quality Assurance but typically exists independently (and there may be no SQA areas in some companies). In SQA, software process specialists and auditors take a broader view on software and its development. They examine and change the software engineering process itself to reduce the amount of faults that end up in the code or deliver faster.

Regardless of the methods used or level of formality involved the desired result of testing is a level of confidence in the software so that the organization is confident that the software has an acceptable defect rate. What constitutes an acceptable defect rate depends on the nature of the software. An arcade video game designed to simulate flying an airplane would presumably have a much higher tolerance for defects than software used to control an actual airliner.

A problem with software testing is that the number of defects in a software product can be very large, and the number of configurations of the product larger still. Bugs that occur infrequently are difficult to find in testing. A rule of thumb is that a system that is expected to function without faults for a certain length of time must have already been tested for at least that length of time. This has severe consequences for projects to write long-lived reliable software.

A common practice of software testing is that it is performed by an independent group of testers after the functionality is developed but before it is shipped to the customer. This practice often results in the testing phase being used as project buffer to compensate for project delays. Another practice is to start software testing at the same moment the project starts and it is a continuous process until the project finishes.

Another common practice is for test suites to be developed during technical support escalation procedures. Such tests are then maintained in regression testing suites to ensure that future updates to the software don't repeat any of the known mistakes.

It is commonly believed that the earlier a defect is found the cheaper it is to fix it.

White-box and black-box testing

To meet Wikipedia's quality standards, this section may require cleanup.

Please discuss this issue on the talk page, and/or replace this tag with a more specific message.

White box and black box testing are terms used to describe the point of view a test engineer takes when designing test cases. Black box being an external view of the test object and white box being an internal view. Software testing is partly intuitive, but largely systematic. Good testing involves much more than just running the program a few times to see whether it works. Thorough analysis of the program under test, backed by a broad knowledge of testing techniques and tools are prerequisites to systematic testing. Software Testing is the process of executing software in a controlled manner; in order to answer the question “Does this software behave as specified?” Software testing is used in association with Verification and Validation. Verification is the checking of or testing of items, including software, for conformance and consistency with an associated specification. Software testing is just one kind of verification, which also uses techniques as reviews, inspections, walk-through. Validation is the process of checking what has been specified is what the user actually wanted.

• Validation: Are we doing the right job?

• Verification: Are we doing the job right?

In order to achieve consistency in the Testing style, it is imperative to have and follow a set of testing principles. This enhances the efficiency of testing within SQA team members and thus contributes to increased productivity. The purpose of this document is to provide overview of the testing, plus the techniques.

At SDEI, 3 levels of software testing is done at various SDLC phases

• Unit Testing: in which each unit (basic component) of the software is tested to verify that the detailed design for the unit has been correctly implemented

• Integration testing: in which progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software works as a whole.

• System testing: in which the software is integrated to the overall product and tested to show that all requirements are met

A further level of testing is also done, in accordance with requirements:

• Acceptance testing: upon which the acceptance of the complete software is based. The clients often do this.

• Regression testing: is used to refer the repetition of the earlier successful tests to ensure that changes made in the software have not introduced new bugs/side effects.

In recent years the term grey box testing has come into common usage. The typical grey box tester is permitted to set up or manipulate the testing environment, like seeding a database, and can view the state of the product after his actions, like performing a SQL query on the database to be certain of the values of columns.

Test cases, suites, scripts and scenarios

A test case is a software testing document, which consists of event, action, input, output, expected result and actual result. Clinically defined (IEEE 829-1998) a test case is an input and an expected result. This can be as pragmatic as 'for condition x your derived result is y', whereas other test cases described in more detail the input scenario and what results might be expected. It can occasionally be a series of steps (but often steps are contained in a separate test procedure that can be exercised against multiple test cases, as a matter of economy) but with one expected result or expected outcome. The optional fields are a test case ID, test step or order of execution number, related requirement(s), depth, test category, author, and check boxes for whether the test is automatable and has been automated. Larger test cases may also contain prerequisite states or steps, and descriptions. A test case should also contain a place for the actual result. These steps can be stored in a word processor document, spreadsheet, database or other common repository. In a database system, you may also be able to see past test results and who generated the results and the system configuration used to generate those results. These past results would usually be stored in a separate table.

The term test script is the combination of a test case, test procedure and test data. Initially the term was derived from the byproduct of work created by automated regression test tools. Today, test scripts can be manual, automated or a combination of both.

The most common term for a collection of test cases is a test suite. The test suite often also contains more detailed instructions or goals for each collection of test cases. It definitely contains a section where the tester identifies the system configuration used during testing. A group of test cases may also contain prerequisite states or steps, and descriptions of the following tests.

Collections of test cases are sometimes incorrectly termed a test plan. They might correctly be called a test specification. If sequence is specified, it can be called a test script, scenario or procedure.

A sample testing cycle

Although testing varies between organizations, there is a cycle to testing:

1. Requirements Analysis: Testing should begin in the requirements phase of the software development life cycle.

During the design phase, testers work with developers in determining what aspects of a design are testable and under what parameter those tests work.

2. Test Planning: Test Strategy, Test Plan(s), Test Bed creation.

3. Test Development: Test Procedures, Test Scenarios, Test Cases, Test Scripts to use in testing software.

4. Test Execution: Testers execute the software based on the plans and tests and report any errors found to the development team.

5. Test Reporting: Once testing is completed, testers generate metrics and make final reports on their test effort and whether or not the software tested is ready for release.

6. Retesting the Defects

Not all errors or defects reported must be fixed by a software development team. Some may be caused by errors in configuring the test software to match the development or production environment. Some defects can be handled by a workaround in the production environment. Others might be deferred to future releases of the software, or the deficiency might be accepted by the business user. There are yet other defects that may be rejected by the development team (of course, with due reason) if they deem it inappropriate to be called a defect.

**Chapter 6- OUTPUT FORMS & REPORTS**

Chapter 7 - CONCLUSION

Chapter 8- FUTURE ENHANCEMENT(S)

**BIBLIOGRAPHY**

**Ruby on Rails:**

* <http://ruby.railstutorial.org/>
* [www.slideshare.net](http://www.slideshare.net/)
* [www.youtube.com](http://www.youtube.com/)
* [www.google.com](http://www.google.com/)
* <http://www.netbeans.org/kb/docs/ruby/rapid-ruby-weblog.html>
* <http://guides.rails.info/getting_started.html>
* [www.rubyonrails.org](http://www.rubyonrails.org/)
* [http://www.tutorialspoint.com/ruby-on-rails-2.1](http://www.rubyonrails.org/)
* <http://guides.rubyonrails.org/>

**MySql :**

* <http://www.troubleshooters.com/codecorn/mysql/basictutorial.htm>
* <http://www.tutorialspoint.com/mysql/index.htm>
* <http://www.w3schools.com/sql/sql_quickref.asp>

**Books:**

* <http://ruby.railstutorial.org/ruby-on-rails-tutorial-book>
* <http://it-ebooks.info/book/1276/>
* <http://www.rubyinside.com/michael-hartls-rails-3-tutorial-book-3583.html>
* <http://pragprog.com/book/rails4/agile-web-development-with-rails-4>

APPENDICES