Part I

System and Software Design Description (SSDD): Incorporating Architectural Views and Detailed Design Criteria for Hatch

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Project Summary

1.1.1 Background

In the areas of ecology and biology, there is an explosion of data. The amount of bioinformatics data in research institutions doubles every 9 months [**?**]. Gains in processing power, however, are decreasing, as the end of Moore's Law's approaches [**?**]. As processor speed gains decrease, the gap between data and results is growing. More and more data has to be ignored. Data that was expensive to collect.

In spite of the growth of data and platue of processor computing power, solutions exists that allows scientists and mathematicians to conduct their research. High Performance Computing (HPC) is one approach, but is not yet widely available. Most researchers have to instead be more efficient with what data they run computations on, and what data they ignore.

1.1.2 Problem Statement

Currently, researchers' data lie in lots of flat files, in a directory somewhere on a computer. If they are lucky, then their data actually lives in a customized database somewhere. But for those who are lucky, the database is hard to use, and is designed specifically for their data. If a new format for the data is seen, much of the core database code has to be rewritten. This creates a lag and a cost between researchers and their tools.

In both cases, accessing and sharing research data is also expensive. The files structures are too big and hard to filter, and the database interfaces are not very user friendly. The result is a lot of research organizations, with duplicate data. They also have lots of data that they want to share with each other, and would mutually benefit each other. But they have no scalable solutions for sharing the data.

1.1.3 Objectives

The objectives in the Hatch tool are to research existing database technologies and concepts, and find a solution for the data deluge. It needs to have an extremely simple interface, and reuse use cases that researchers already know. Other objectives are to create useful tools that help researchers filter and vizualize their data, and take the pain out of data management. The objective is not to finish the Hatch tool, but to provide a good code and concept foundation for further development.

Chapter 2

Requirements Traceability

2.1 Customer Requirements

2.2 Database

2.2.1 Introduction

One of the biggest challenges with Hatch was how to organize data. Specifically, pretty much every organization with scientific data has their own standard or format on how they store research data. Many of these organizations want to share data between each other, but they cannot decide how to merge the formats. Consider the following examples:

site	datetime	unique fish tag
TUC	02/16/06 19:08:15	3D9.1BF1E7919A
TUC	02/16/06 19:18:36	3D9.1BF1A998FA
TUC	02/17/06 18:21:03	3D9.1BF20E8FE2
...	...	...

Figure 2.1: A database representing for PTAGIS data

unique fish tag	DNA sequence
3D9.1BF1E7919A	ATGCTTAC...
3D9.1BF1A998FA	TTACGATC...
3D9.1BF20E8FE2	GTGGASCT...
...	...

Figure 2.2: A database representation for DNA data

In each of the examples above, the data is represented with **rows** and **columns**, much the same way someone would represent the data in a Microsoft Excel spreadsheet. These structures in a typical **relational database** (SQL, etc). are called **tables**.

The above examples are simplifications of the rows and columns in actual research data. But they highlight one of the biggest issues with data storage using relational databases: they require you to know the column names ahead of time. Not only that, but they require that you know the data types of the values that go in those columns, and once a table is created expecting a certain format, it is hard to change.

The problem with needing to know the structure of research data before designing databases, is that research data **is semi-structured**, at best. Once it does represent some structure, it often changes. For example, once researchers finally decide what columns and data types

should go in the table in Figure 2.1, another researcher suggests more columns that should go in to the table.

This leads to endless edits to the database and program design by some software developer. The standard table format that everyone can agree on isn't useful to many researchers, because it usually leaves out a lot of other needed columns and fields.

A better approach is needed. Researchers, not committees, should decide how to store data. Data should be mergable based on common values in different tables (like the **unique fish tag** column in Figures 2.1 and 2.2. The person who inputs the data should decide how one particular dataset is stored in a database, and should be able to choose to store the same data in a different table format, how they choose. There should be a simple tool that helps them do this.

The following sections describe different approaches to implementing a database design that enables data storage for dynamic or semi-structured data.

2.2.2 Relation Databases with table creations

This approach is the simplest and follows the concept of table creation for data sets pretty closely. Basically, for each input document in the form of a spreadsheet, a new SQL table is created. The columns names and type are determined from the headers and data values in the spreadsheet.

```
CREATE TABLE ptagis_doc1
        (
                id int,
                site char(50),
                read_data_time date,
                tag char(50)
        );
```

Figure 2.3: The SQL syntax for creating the table in Figure 2.1

The biggest problem with this approach with this approach is that each document in the database is a table. When searching for a specific document, the database typically searches for the table name. This search is linear, and with hundreds, thousands, or hundreds of thousands of documents, frequently searching the database to look for values would be infinitely slow and useless.

Another problem with this design, is that building an interface like a web gui would be difficult and complicated.

2.2.3 Relational Databases with tables for each datatype

Another approach is to create column tables for each data type, and let document tables just be collections of columns. Each of the document column values point to respective values in the column tables.

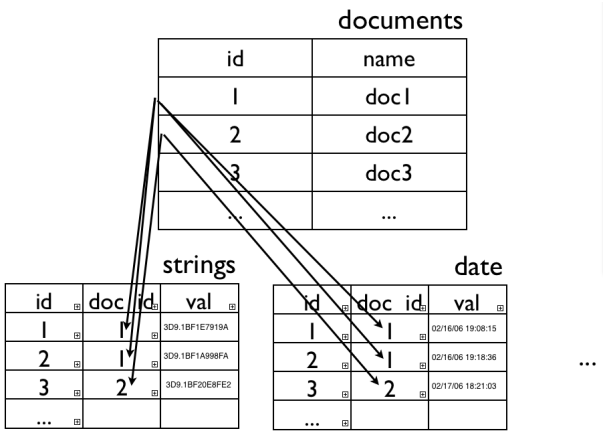

Figure 2.4: Document table as a lookup table

This allows for documents to have a dynamic amount of columns with variable data types. But there are two problems with this approach. First, for every value in every document in the database, they are put into one table (i.e. all values with a 'string' data type go into the 'string' table). With potentially millions of data values, each table becomes an overflowing bucket, and don't utilize the the advantages of storing multiple columns and values in one table. Searches for data would require lots of filtering for just the ones wanted for specific doucments, and would be inefficient.

The other issue is that every retrieval of data from a document would require lot and lots of lookups. Data retrieval over significantly high data sets would quickly become very computationally intensive, and not practical.

2.2.4 CouchDB

When one thinks about the fundemental issues with storing, searching, and merging research data, a core issue is identified: data is semi-structured. This is what makes trying to use relational databases so hard. They were made for datasets where you knew the structure up front, and seldom wanted to change their structures.

The one assumption that Hatch makes about the data is that **there are rows and columns**. This is the only assumption Hatch makes. This leaves the database and interface designs free from whatever changes are needed by the users.

This is done by storing data in JSON format. The data is stored in an array of hashes, modeled exactly like how Ruby on Rails 3 returns records from its Active Record. This allows users to input data however they like, define delimiters for columns using Hatch Input Filters, and Hatch does the rest. It finds the most specific data type the values can be stored in, skips non-matching entries, and populates all the forms on the web pages according to the data. All without actually knowing the structure of the data.

This leads to the technical implementation of the Hatch Database.

2.2.5 Hatch Database

Since Hatch uses Ruby on Rails, there are lots of tools and libraries for using the standard relational database, via Rails' Active Record. In many/most cases, Hatch actually wants to stick to the relational database. With Hatch's internal database structured, order is important. For example, a user will always have a name, email address, etc. A document will always have a name and an owner. But the data in the document is the semi-structured data. Hatch only wants to use CouchDB for that.

The result is a relation - non-relational database hybrid. Hatch uses traditional relational database columns when the columns are ubiquitous, and can add columns to a database entity (called scaffold) on the fly using CouchDB. For example, we create a scaffold called Documents. Document always have a name, id, and collection/folder they belong to. But, documents may have a data section, or they may not. That data section may have infinite columns and rows of data, that would match the flat file they came from (like an Excel file). Or document may have any other data that can be stored in JSON format. It is up to the Hatch interface, not database, to decide. And by allowing the interface to decide the structure and fields of data, Hatch is by extension allowing the user to decide how to format data.

Most Rails applications that use CouchDB completely replace Active Record with some library's version of it, like **couchrest's Active Model**. But if you replace Active Record, you lose tons of support for libraries that lots of Rails developers make, like pagination.

Hatch gets around this by using its database hybrid, using a Ruby library called **stuffing**.

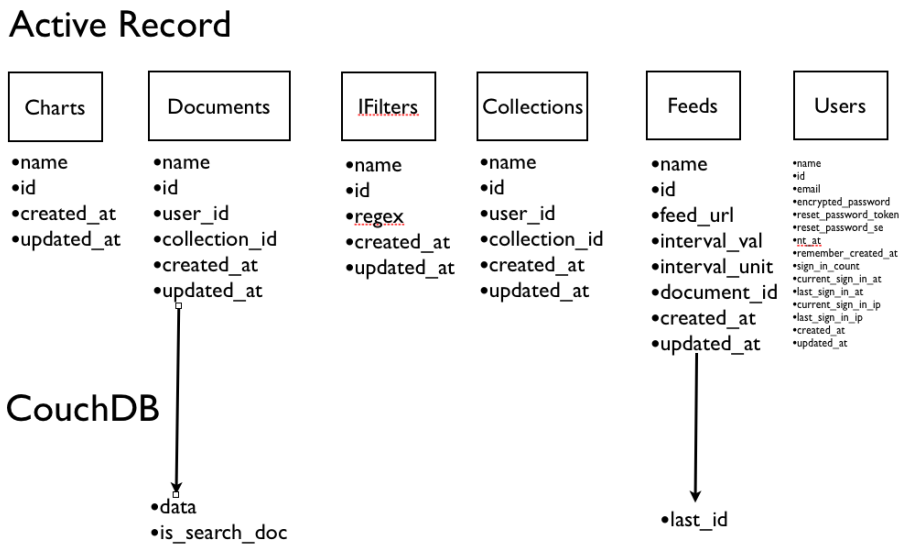

Figure 2.5: The Hatch relational - non-relational database hybrid

Stuffing is a link that ties Rails Active Record records with CouchDB documents. It allows for rapid prototyping in development, and is a nice, modest alternative to ripping out the guts of Active Record. Since the base model for database scaffolds are still Active Record, the huge amount of Rails Active Record based libraries still work.

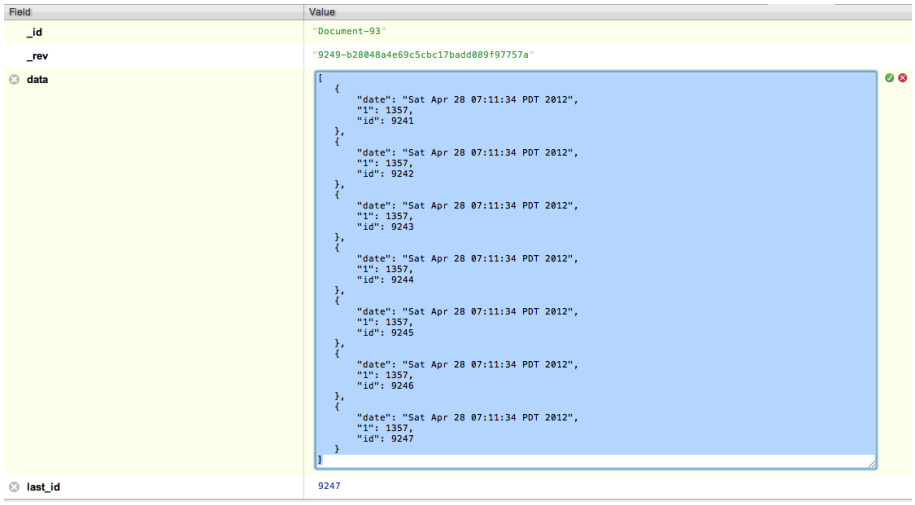

Figure 2.6: Typical representation of document data in CouchDB / JSON

2.2.6 Documents - Requirements

Scaffold

The following are the requirements for Document scaffolds:

1. Active Record
   (a) Documents shall have a **name** field.
   (b) Documents shall have a **collection** field, which is an ID of the parent Collection.
   (c) Documents shall have a belongs_to relationship with Collections, Users, and Feeds.
   (d) Documents shall have a has_many relationship with Charts.
   (e) Documents shall have at least all the scaffold file generated by **rails generate scaffold attributes**.
2. CouchDB
   (a) The CouchDB document will be named 'Document-¡ID¿', where the ID field is the corresponding Document ID in Active Record.
   (b) All documents will have a **data** member. If empty, they will be an empty JSON array ([]).i
   (c) The 'data' member is an array of hashes, and should match the structure that Rails Active Record has for records.

Scaffold - Views

The following are requirements for Document views:

1. edit - view shall be the standard rails generated view.
2. index - view shall start from the standard rails generated view.
   (a) View shall show a Search form at the top.i
   (b) If this document is a result of a search, it shall display a link to 'Show as Document' for search results, and then display the data member search results below.
   (c) If a search has been submitted, then show a list of Documents whose names also match the search.
   (d) For each Document listing displayed, display an additional link that allows for Document CSV download.

3. edit - view shall be the standard rails generated view.
4. show - view shall start from the standard rails generated view. Additionally:
   (a) View shall display the _visualize partial.
   (b) View shall display a form for data manipulation.
      i. Form will have a radio button for each column that can manipulated.
      ii. Form will have a selection list of manipulation functions. The only current function is **categorize**.
   (c) View shall display all the data from the CouchDB document, as a html table.
      i. Each cell in the data table shall be a link to Document Hatch Lucky Search.

Helpers

The following are the requirements for Document Helpers:

1. get_data_colnames(d)
   (a) param d - A CouchDB 'data' member of a Document.
   (b) Return - a list of all 'column names' (unique keys) from the structure.
2. convert_data_to_native_types(d)
   (a) param d - A CouchDB 'data' member of a Document.
   (b) Return - A CouchDB 'data' member of a Document, with data types changed from string to more explicit type, if possible.
   (c) The type conversion priority is listed as follows:
      i. datatime
      ii. decimal/float
      iii. integer
      iv. string
3. get_data_column(d, colname)
   (a) param d - A CouchDB 'data' member of a Document.
   (b) param colname - A string for the colname wanted.

   (c) Return - A CouchDB 'data' member of a Document, but with only hashes with key == colname.

4. get_data_map(d, colname)

   (a) param d - A CouchDB 'data' member of a Document.

   (b) param colname - A string for the colname wanted.

   (c) Return - A CouchDB 'data' member of a Document, with unique values for column == colname in '1' column, and the count of each of the values in '2' column.

5. document_search_data_couch(search, lucky_search = false)

   (a) param search - A string for the search value

   (b) param lucky_search - A bool value, defaults to false

   (c) Return - A CouchDB 'data' member of a Document:

      i. If lucky_search == false, a CouchDB startkey search.

      ii. Else, a CouchDB key search.

2.2.7 Collection - Requirements

Scaffold

The following are the requirements for Collection scaffolds:

1. Collection shall start from the standard **rails generate scaffold**.
2. Collection shall have a has_many relationship with Documents.
3. Collection shall have a belongs_to relationship with Users.
4. Collection shall have a **name** field.

Scaffold - Views

The following are requirements for Collection views:

1. edit - view shall be the standard rails generated view.
2. index - view shall be the standard rails generated view.
3. new - view shall be the standard rails generated view.
4. show - view shall be the standard rails generated view.

Helpers

There are no helper requirements for Collection.

2.2.8 Home - Requirements

Scaffold

Home shall only have one view, and no scaffold.

Scaffold - Views

The following are requirements for Home views:

1. index - a custom view.
    (a) View shall display a quick summary of the project.
    (b) View shall list sponsors and team member names
    (c) View shall link to the senior design website.

Helpers

There are no helper requirements for Home.

2.2.9 IFilters - Requirements

Scaffold

The following are the requirements for IFilters scaffolds:

1. IFilters shall start from the standard **rails generate scaffold**.
2. IFilters shall have a **name** member.
3. IFilters shall have a **regex** member.

Scaffold - Views

The following are requirements for IFilters views:

1. edit - view shall be the standard rails generated view.
2. index - view shall be the standard rails generated view.
3. new - view shall be the standard rails generated view.
4. show - view shall be the standard rails generated view.

Helpers

There are no helper requirements for IFilters.

2.2.10 Data I/O - Requirements

Input

1. The input module will allow users to import CSV native formats into the metadata servers native format.

   (a) *Verify that the CSV inputs correctly into Metadata as a new Data instance.*

2. The input module will allow creation of new input filters as regular expressions, to be applied to every line of a file, for custom file format input/importing. This will be implemented by the Filter scaffold (see Section **??**).

   (a) *Verify that a user can specify filters and then apply them to correctly parse and import files, per the expected behavior of regular expressions.*

3. Once a file is inputed, the wall and system time will be displayed, along with the number of rows created.

   (a) *Verify that the wall and system time, along with the number of rows, are displayed on input.*

4. When a file is inputed, a Metadata instance will be created or updated, with the 'name' field set during the import.

   (a) *Verify that the Metadata instance is created with the 'name' set as the file name.*

5. When a file is inputed, it is represented as a Data instance with the 'name' field set as the name of the imported file.

   (a) *Verify that the inputed file creates a new Data instance, and each row's name column is set to the name of the inputed file.*

6. When a file is inputed, each column will be intepreted as a Rails 3 data type (see Section 2.2.10). The column data type will be applied to all members of all rows for the respective columns.

   (a) *Verify that a column is intepreted to be a support Data Type, and is inputed.*

7. When a column is being considered for which Data Type it is, it will be attempted to be interpreted as a data type in the following order, until success:

   (a) time

(b) date
(c) timestamp
(d) datetime
(e) boolean
(f) binary
(g) decimal
(h) float
(i) integer
(j) string
(k) text

(a) *Verify each kind of data is intepreted correctly.*

Data Types

The supported data types are (in list of precedence):

1. time
2. date
3. timestamp
4. datetime
5. boolean
6. binary
7. decimal
8. float
9. integer
10. string
11. text

2.2.11 Charts - Requirements

Scaffold

The following are the requirements for Charts scaffolds:

1. Charts shall start from the standard **rails generate scaffold**.
2. Charts shall have a belong_to relationship with Documents.

Scaffold - Views

The following are requirements for Charts views:

1. edit - view shall be the standard rails generated view.
2. index - view shall be the standard rails generated view.
3. new - view shall be the standard rails generated view.
4. show - view shall be the standard rails generated view.

Helpers

There are no requirements for Chart helpers.

2.2.12 Visualization - Requirements

The Visualization controller is referred to throughout documentation simply as **Viz**. It will be Rails 3 **views** and **controllers**, without models, since it mainly inteprets data from the Metadata **??** and Data I/O 2.2.10 scaffolds. There will be one view per controller method, unless stated otherwise.

Graphs - Types

Viz graph types will be the following:

1. line (default)
2. bar
3. scatterplot
4. pie
5. more (optional, future; i.e. statistical graphs)

When a view/controller method gets data and no graph type argument, it will use the default type. Otherwise, it will use the specified type.

Data sets - Types

Viz will graph the following data types natively on either x or y axis: precedence:

1. Date / time (may be combined):
   (a) time
   (b) date
   (c) timestamp
   (d) datetime
2. Chunkified data (see Section 2.2.10):
   (a) boolean
   (b) binary
   (c) string

(d) text

3. decimal
4. float
5. integer

Views - Show

1. Viz will contain one show view.
   (a) *Verify that there is a Show view.*
2. Some other controller will pass a Data model id and two column names to graph data by.
   (a) *Verify that this view graphs the correct data for the Data id and two columns listed.*
3. The first passed column will be **x_data** in POST, the second will by **y_data** in POST, for the respective axis.
   (a) *Verify that the respective columns graph on the correct axis.*
4. Controllers may pass a graph type, but if not, the default type will be used.
   (a) *Verify that the data is graphed in the correct graph types.*

Views - Summary

1. Viz will contain one Summary view. This view will take in a Data model data_id POST value, and graph every unique permutation of two columns.
   (a) Verify that the Summary view graphs the correct data for data_id.
2. There will be two Summary graphs per display row.
   (a) Verify that there are two Summary graphs per display row.
3. Summary graphs will graph 10% or 100 points of the data, whichever is bigger.
   (a) Verify that Summary graphs graph 10% or 100 points of the data.

4. Summary graphs will graph any type of data as defined in Data Types (Section 2.2.10).
    (a) Verify Summary graphs graph any Data Type.
5. (a)

2.3 Search

After the EcoData team addressed the issues with storing semi-structured data, the next big problem to address was how to efficiently search through the data. The interface that was desired was one much like Google's search; a simple search box, with two buttons. The interface needs to be extremely simple, but powerful, for it to be affective for non-technical users.

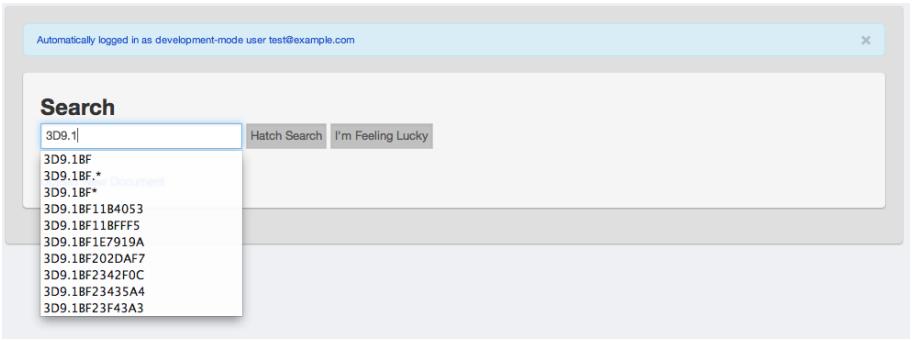

Figure 2.7: The search interface

For example, when a user entered in a search like '3D9.1', it would be assumed that any data starting with '3D9.1' should be returned. Because Hatch assumes that the user would want any data in the same row as the matching data, it returns the entire row.

For search, this leaves the possibility for a huge amount of string comparisons to find all values that match. This would make search impractically slow. Luckily, CouchDB comes to the rescue for us.

2.3.1 Views

CouchDB uses precompiled queries called **views**. Views take developer defined query templates and apply them to every document that is created or updated in the database, when the document is saved. The result are precomipled lookout tables (actually heaps/binary trees), which make searches fast.

Hatch basically creates a view like the following pseudocode:

```
for each document
        for each row
                for each column value in row
                        emit(value, row)
```

emit() is a function that tells CouchDB how to make the search B-Tree. It takes two arguments; the key that the tree node will take, and the value that the node will return if the key matches the search. Hatch says 'every column value in a document is a key, and the return value is the data row it belongs to'. So if a search matches a document value, the search results show the entire data row.

CouchDB makes Hatch's job easier by having internal methods for string matching. For example, if a search for '3D9.1' is used with CouchDB startkey, CouchDB will return any string starting with '3D9.1'. Hatch doesn't have to invent a query language to tell the database lots of parameters for matching values.

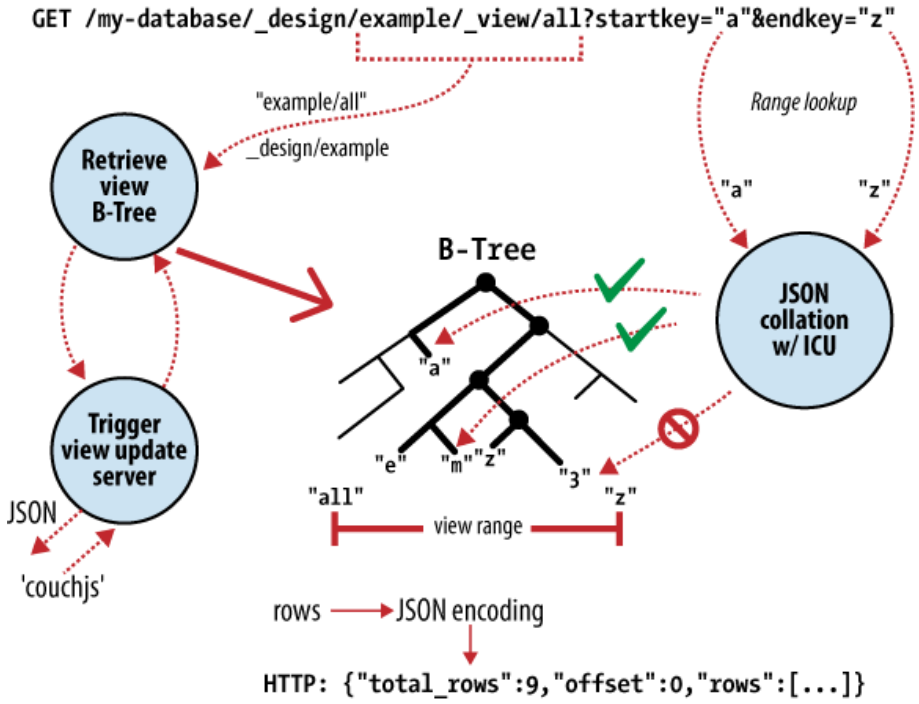

Figure 2.8: CoucbDB search through its internal binary tree.

The other advantage to CouchDB is that it stores values in the B-Tree structure. For string data types, this means that the database only within the '3D9.1' string range in the tree. When the search sees a child node starting with '3D9.0', it instead picks the child node staring with '3D9.1', and the entire '3D9.0' group is never searched through.