Refactoring the Candidate Cancer Gene Database

A Thesis

SUBMITTED TO THE FACULTY OF THE

UNIVERSITY OF MINNESOTA

BY

Christopher T. Tastad

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

Adviser: Timothy K. Starr, PhD

[full month and year of degree conferral]

Contents

List of Tables

List of Figures

Chapter 1: Introduction

Refactoring is intended to improve the design, structure, and/or implementation of the software (its non-functional attributes), while preserving its functionality. Potential advantages of refactoring may include improved code readability and reduced complexity; these can improve the source code's maintainability and create a simpler, cleaner, or more expressive internal architecture or object model to improve extensibility. Another potential goal for refactoring is improved performance; software engineers face an ongoing challenge to write programs that perform faster or use less memory[4]

1.0.1 the challenge

field of bio has ever increasing issues in data management and doesn't apply tech solutions very effectively. Ability to move past code to data manipulation and presentation. Deployment and the integration of common programming practices

In many ways, the field of Bioinformatics exists as the result of a problem rather than a discovery, as with other scientific fields. The nature of the challenges have evolved from a growth in the composition of primary data and a lacking compensatory set of technical skills to evaluate it. With the bottle neck that exists at the center of this problem, previously simple notions of collating, analyzing, and visualizing results no longer scale. As a result, a measurable area of focus is not just in the novel discovery of Biological relevance but in advancement of those engineer tools that offer new or dynamic means to solve old problems with new solutions.

1.0.2 the tech has evolved

Transitioning (implementing) existing solutions with new paradigms. Improved data management (manipulation, collation, visualization). Web front-end tech (web dev). Technologies that enable reformed development (javascript, markdown, rstudio engine). Increased accessibility to non-tech users.

1.0.3 applications of the tech in bio, specifically the CCGD

gene screens in cancer bio [2]

Cancer gene screening is a hallmark case for this exploration as it lives at the nexus of being an area of high impact research and a process that can produce a deluge of raw data. In tandem, the state of web front ends has advanced in a manner that more completely allows [6] In this work, we present a small but emblematic case that illustrates these shifting paradigms. With these incremental changeovers, no mater the impact, the field takes on a new character in the management of its growing data challenges. The CCGD as a concept set out to tackle some of these original problems of filtering handling large quantities of data. We expand on this by re-implementing the same achievement through a modern implementation.

Chapter 2: Body

2.1 Methods

2.1.1 Project Requirements

This project's goal was ultimately to upgrade the existing CCGD. In setting out to do this, we established several requirements that took into consideration the opportunity to seize a routine set of server transitions to implement improved functionality. Specifically, the central output of the service should be retained in the eyes of the end user. Beyond this, attempting to fully reconstruct the backend provided for significant improvements (see Table 2.1). These goals more broadly were: upgrade the existing server build to RHEL 7; rewrite web interface in a more modern framework; streamline the central table construction process using Rshiny; implement improved ability to manipulate table by product owner; employ improved software development best practices; generally seek opportunities for codebase and resource utilization improvements. Ultimately, all of these taken together were intended to serve some form of modernization and simplification.

Table 2.1: Project Requirements

	Feature	Goal	Framework	Description
1	server OS	upgrade server OS	RHEL7	Transition of architecture for public server host. This is required by University OIT due to end of life schedule for RHEL6.
2	web front-end	rewrite web interface	Rmarkdown	Improvements to the web interface written in a modern, simplified language. This will improved access to content creation and allow for automation in site rendering.
3	table build	rewrite table build	Rshiny	Rshiny offers a dramatic improvement to replace the existing process by merging the table build back-end with a modern web display of the app interface.
4	table update	improved admin controls	BASH/R	Existing app version confined some content controls to app author, preventing contributions by app owner.
5	version control	implement best practices	git/docs	No version control was used in the original development of the app. This and other documentation practices were expanded in the rewrite
6	n/a	resource improvements	codebase	Due to the lack of some best practices, there were many opportunities to make impactful resource improvements.

2.1.2 Server Setup

At it's core, the OS transition from RHEL 6 to 7 was the original project requirement that spurred this work. Existing hosting for the CCGD was owned by university OIT who established an end of life date for the current server OS. As a result, the bulk of core architecture implementation followed a narrow prescription for what was termed a "Fully Managed" RHEL7 install as detailed by the OIT-LPT Public-Docs [5]. Contrary to the implications of this installation name, a Fully Managed administration carried a mixture of privileges and management roles that were shared between OIT and our group. This was accomplished by leveraging a chef admin paradigm that was allowed limited sudo privileges within the hosted ecosystem. Importantly, items that extend from the core of server administration such as virtual machine creation, backups, and core security were handled by OIT. This left our group with the more focused task of administering exclusively our application.

[to what level of detail should i describe the server setup]

Apache

2.1.3 Web Framework

The upgraded build of the CCGD employed a completely different front-end architecture centralized around R web services environment. Built with dynamic modularity and data transportability in mind, the Rmarkdown framework offers the ability to render several independent markdown files into a unified website. We applied this feature set to produce our general site structure and content as it allowed for a massive simplification in web design. While following the existing site map, we recreated the raw content of the existing application as independent markdown files. Within the Rmarkdown format guide. The keystone feature of the Rmarkdown framework that allows for this unified rendering is the use of a YAML configuration header (see Figure 2.2b). The `rmarkdown::render_site` function leverages this configuration file to align and sequentially render content and styling in tandem for a complete site map [citeme rmarkdown docs]. Additionally, individual markdown pages employ this same configuration paradigm for page-specific parameters and formatting guidelines (see Figure 2.2a).

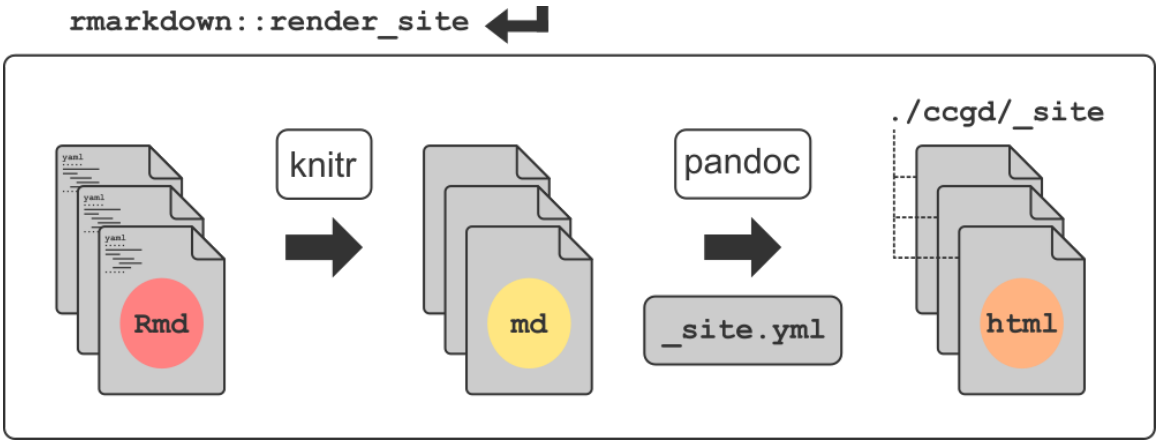

Figure 2.1: **Rmarkdown Rendering Flow** The core web framework was created using a stitch process composed of two document conversion steps, which are both contained in the single `rmarkdow::render_site` command. These two transformations are done by the R-specific knitr package and then the open-source pandoc tool. The first phase holds less relevance in particular as its role is to execute R code chunks, of which there are few in our page content. The markdown output from this step is seamlessly injected into a pandoc conversion which offers a large variety in document output. Specific to our case, the pandoc conversion paired with a companion yaml configuration file allows for multidocument markdown-to-html rendering which is assembled as a complete site stack. As a single process, this function generates an integrated, modern web front-end from content written in a light-weight syntax with almost none of the typical complexity of full-scale web page creation.

responsive web design

An important distinction in summarizing the implementation of web framework is to point out

```yaml
title: "Candidate Cancer Gene Database"
bibliography: refs/ccgd_paper.bib
nocite: '@*'
output:
    html_document:
        includes:
            in_header: "styles/favicon.html"
            after_body: "styles/footer.html"
        css: styles/styles.css
        theme: readable
---
```

(a) index.Rmd individual page header

```yaml
name: "Candidate Cancer Gene Database"
navbar:
    title: "CCGD"
    left:
        - text: "Home"
          href: index.html
        - text: "Search"
          href: search.html
        - text: "Help"
          href: help.html
        - text: "References"
          href: references.html
        - text: "Contact"
          href: contact.html
```

(b) _site.yml: file for site assembly

Figure 2.2: **YAML Configuration Segments** Pictured above are the two forms of complete yaml configuration headers utilized in Rmarkdown rendering. The central takeaway from this illustration is to recognize the heavy lifting performed by these relatively small code segments. Figure 2.2a provides an example of the header found in every .Rmd file that make-up the respective pages of the site. These headers provide primarily formatting configuration for the rendering process but include broader functions as well. The header can be as simple as listing a title and pandoc output type or include LaTex bibliography functionality and read-in style sheets like the one shown. Either way, a few lines of configuration generate wildly different outputs. Figure 2.2b presents the yaml file which serves to assemble independent Rmarkdown files into a joined website as described in figure 2.1. There is less configuration here as the only real purpose is to define the site tree for the `rmarkdown::render_site` process.

that the Rshiny app, which contains the gene table, is assembled and hosted as a function separate from the general site rendering. More specifically to the web design, the shiny app is hosted at the common public repository `shinyapps.io` and rendered in an inline iframe on the site search page. In the eyes of the user this presents a seamless transition, but the functional contrast permits for differential rendering between the two. This allows for the continuous integration of external sources that feed the table while eliminating the need to regularly render the static site content.

2.1.4 Data Integration Process

Improvements to the data integration process were accomplished using a more simplified framework to eliminate redundancy and provide a more standardized schema. The prior implementation employed some tools that are common place, but they were implemented in a fashion that lacked simplicity at times. Also, certain proprietary widgets presented functionality and compatibility choke points that generated road blocks to maintenance. The most substantial of the changes that addressed these issues was the transition to use of R as the data manipulation toolset. More specifically, the `build_table.R` script established a lean ETL process that took existing table data and external reference data to perform a transformation which collated old and new data. The product of this process could then be delivered to the R-based shiny app deployment generated by the `app.R` script.

Redundancy reduction was acheived within scipt functionality and storage utilization. Dedicated script actions that performed house keeping functions, as seen in the `build_table.sh` and `backup.sh` (see Figure 2.3), made specific effort to clear all transient data content that otherwise did not require persistent storage. This accounted for a large portion of persistent data that was previously left on the server. Additionally, script reduction was improved through a reorganzition of the general subroutines within the application. A reassignment of script actions were assigned in a more cohesive manner which allowed for both improved simplicity and modularity. This is most effectively seen in the manual control `ccge_upload.sh` script as the developer has the flexibility to opt out of executing several functions within the app.

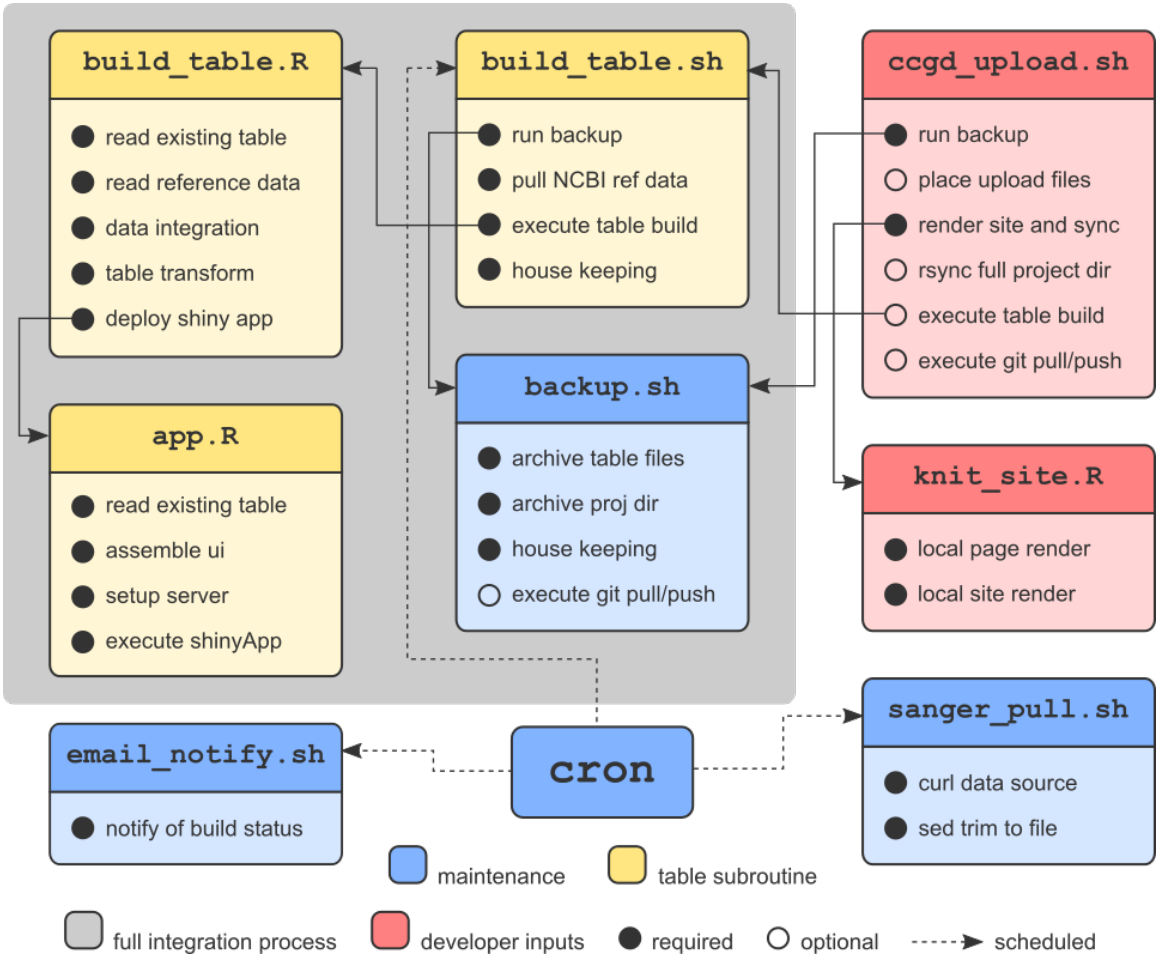

Figure 2.3: **Subroutine flow** This diagram offers an overview of the actions and relationships within the executable scripts of the CCGD. Prior functionality was more effectively delineated into subroutines for better modularity. Also shown is the added functionality brought by the rebuilt data integration process. A hallmark of the difference between this new and old process is the use of R and shiny as the central frameworks to build the table as implemented in the build_table.R and app.R scripts. It can also be seen that the integration process is automated with a cron job among other maintenance tasks. Last, the ccgd_upload.sh and knit_site.R scripts were created to allow for introduction of new table data or the manual execution of any of the app subroutines.

2.1.5 Rshiny App

In order to replace the existing CCGD search functionality our group employed the Rshiny framework. Simply put, the shiny app that now enables this central purpose is an interactive web interface for a single table. Several changes were made to the existing integration pipeline in order to achieve this simplicity.

The structure of a general shiny app follows a highly reduced and consistent paradigm, and ours is no different. The broad format of this work flow is `ui + server` → `deployment`. The server specifically has a discrete composition made up of reactive constructs of inputs, expressions, and outputs that shape the app's reactive graph, which forms, more accurately, the discrete operation of the application. The assembly and movement of this graph is both emblematic and practical in displaying the modularity and improved resource utilization of the reactive elements. Without belaboring the description of reactive programming, this paradigm employs a declarative programming style that allows for lazy code execution [7]. What this amounts to in application is for the developer to define loose controls at a higher level while allowing the code to passively fill these requirements. This translates to greater flexibility to both the developer and the machine as resource use can be reduced through modularity in code execution.

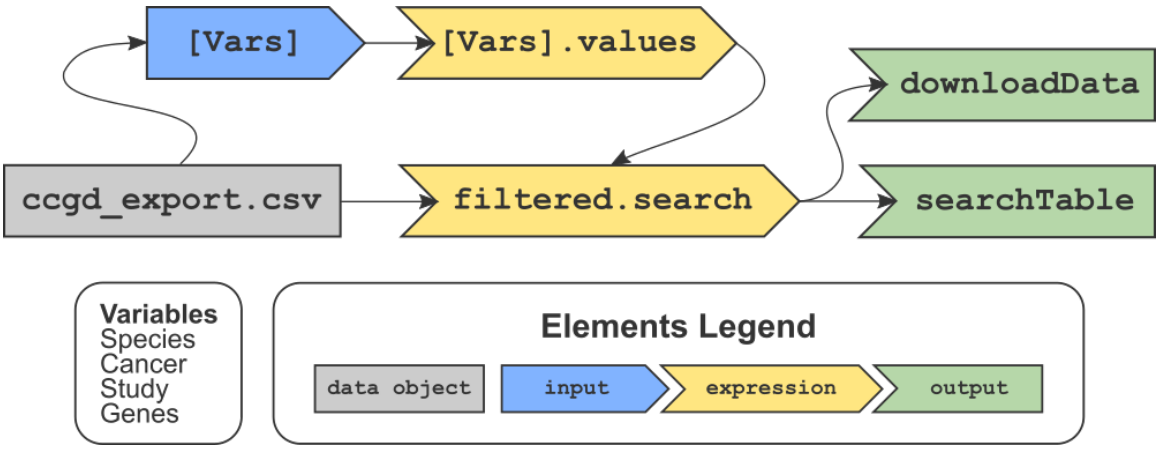

Figure 2.4: `table_app/app.R` **Reactive Graph** An Rshiny app design can be defined by its reactive graph. The graph above displays the process flow of our app by highlighting modularity in the data streams around the table filtering process. The reactive web design is intended to allow for the automatic implementation of changed states applied through the alternate paths of the graph. This allows for the passive display of a static representation of the table while allowing for dynamic adjustment to filtering inputs through the `[Var]` → `[Var].values` path. Additionally, this declarative model provides for an interactive table array while only drawing computation when the appropriate event is called.

Within the scope of the `table_app/app.R` server function we had a relatively simple goal of

defining variable filters across the CCGD data table. Taking advantage of Rshiny's modularity, we created a two state path for the imported data object that allowed for both a variable-specific filter step along with a static, unchanged display of the table (see Figure 2.4). The advantage in doing this is that the table is displayed immediately on page load without the need to initiate a query. Additionally, the full, unfiltered table is available from the `downloadData` output at this state as well. For processing going down the filtered path, independent variable inputs are available to the user which are passed through two expression elements. In selecting a variable filter the `[Vars].values` expression applies a syntactic subsetting of the variable's data list which is then past to the central `filter.search` where the broader subsetting is carried to the rest of the table. The `searchTable` function takes the output of the filter and applies several text mutations to add external linking to several associated resources.

A small but fundamental element of our shiny app design was the use of the `renderDataTable` function in generating the `searchTable` output. The prior iteration of the CCGD architecture employed a relatively disproportionate MySQL database to store and serve numerous component elements that made up the table. Instead, the new implementation of our app now leverages the server-side processing capability of the DataTables JavaScript library. More specifically, we set parameters to create a reactive version of our function which allows for paging, searching, filtering, and sorting in real-time within the shiny server infrastructure [citeme DataTables]. Through this function alone we established new server paradigm that eliminated the MySQL database.

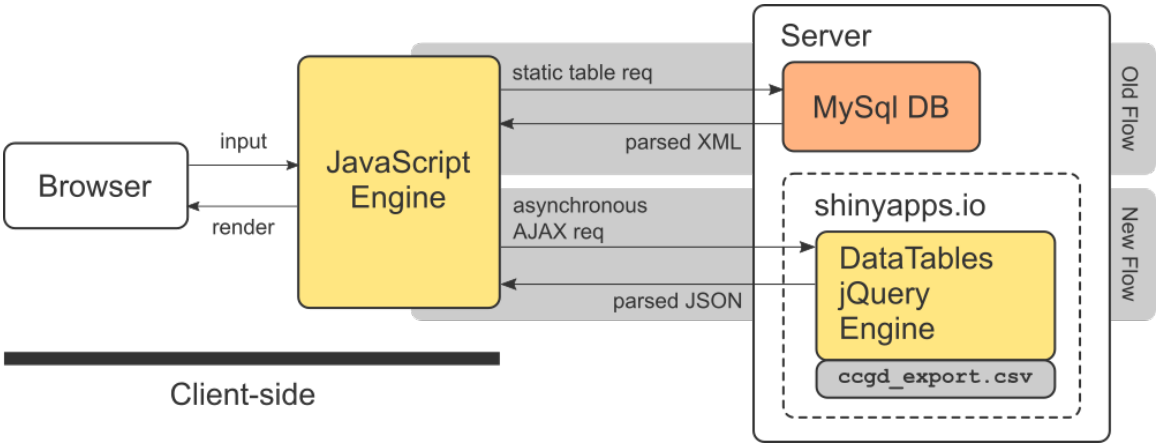

Figure 2.5: **Server-side Processing Flow** This diagram illustrates a small but fundamental change applied to the application table processing through the lens of the server-client relationship. The "old" flow shows the prior table processing procedure, which is highly typical of a classic Linux-Apache-MySql-PHP (LAMP) stack website. Inputs and calls made by the browser occur from and within a client-side rendering engine which delivers a static request to the server-side SQL database. The database returns parsed XML to the browser that is rendered and displayed. Our fundamental shift was to move this data processing entirely to JavaScript. Typically, this would result in client-side processing that would prohibit large tables like ours. Instead, we leveraged a server-side engine to handle asynchronous AJAX requests which are returned with parsed JSON, shown in the "new" path. This processing flow allows for the use of a single .csv file as the data source in table presentation, ordering, and pagination, ultimately eliminating the need for a SQL architecture.

2.1.6 Best Practices

Some features of the software best practices from the previous version of the CCGD were done quite well such as the level of detail in the documentation. At the same time, many other important areas were missing in the development and maintenance of the application. Most prominent among these were the lack of any degree of version control and relatively inconsistent design principles. As a result, our group employed or improved upon several methods in an effort to elevate the quality, usability, and transportability of our codebase and application [3].

Throughout the course of the project we utilized a private github code repository for hosting and change tracking. This work has since been published on a public repository and assigned a DOI number through Zenodo. Furthering the spirit of project integrity, we expanded on the nature of the documentation for the user and, more prominently, the application owner. Specifically, documentation that was previously written in a Microsoft Word file was now recreated in markdown, and a notable tone and detail shift was made in serving the information transportability between administrators.

In approaching the process of development, we applied a philosophy of tidy code creation to

both syntax and design. A supporting method in creating a more elegant code syntax was the use of R universe tools, tidyverse and Rmarkdown. Both offer a coding grammar that is centered around accessibility and simplification of high-level language organization [8]. Furthering this simplification, we sought opportunities to deduplicate and extract subroutines in the table integration process in an effort to streamline resource constraints. Finally, another notable area for improvement we sought was in disk space utilization. The existing integration process relied on the continuous presence of what were several intermediary files that occupied a sizable disk capacity. To combat this we specifically reworked several data transformation steps to allow for the post-process elimination of these temporary data objects.

2.2 Results

2.2.1 Architecture Upgrades

previous installation used traditional lamp stack

Codebase reduction.

Need for sql database eliminated.

Interface improved. Table content contained within single page.

The rebuilt data integration process represents a core accomplishment of the refactoring project. A reconstruction of this process and the frameworks utilized to execute it offered notable improvements

Figure 2.6: **Project Directory File Tree Comparison** The above figure offers a qualitative assessment of an element of technical debt that was improved in refactoring. Shown are the file trees of the site directories of the new and old projects. What is illustrated is a dramatic improvement in the tree size and complexity through the elimination of a large degree of content, log, and style files. This is accentuated even forther when comparing the source code. This was accomplished both in the intentional design of eliminating content redundancy and in the streamlined structure of the Rmarkdown rendering process. This allowed for the elimination of several dimensions of broader web design and table build data elements.

Following the goals set out in our project requirements, we chose Rmarkdown as a replacement framework to develop the application's web front-end. Previously, the web interface for the application had been developed using Adobe Dreamweaver [1]. For the time Dreamweaver served the purpose of simplifying the process of interface construction and web development, yet we hoped to extend the theme of simplified content creation into a modern framework. Further, utilizing a language such as Rmarkdown almost fully carries the development process into a state of pure content creation.

The original version of the CCGD front-end was developed using Dreamweaver [1]. While this may have suitable at the time, several generations of web development have brought the field to be able to move beyond the process of design and focus more readily on content.

In an effort to reduce code complexity and improve access to content creation, an engineering choice was made to redesign the web front-end in Rmarkdown. While an approach such as this can reduce certain feature availability and overall design flexibility, a conscious priority was placed on the content of the page over the web functionality. In the spirit of our project goals,... A framework such as Rmarkdown provides a development paradigm that dramatically shifts the effort of creation into that content rather than the creation broader page architecture [citeme rmarkdown docs]. The work of assembling that architecture is largely done for the user in leveraging

2.2.2 Resource Utilization Improvements

Space constraints eliminated. Server resource reqs reduced overall.

Interface now streamlined to single page instead of a bunch of pages.

First, changes to the backend process allowed for the elimination or significant reduction of several large, persistent data files making them transient through asynchronous update. (The largest of these are highlighted with asterisk.) Second, the major architecture choice to handoff server-side table processing to a javascript engine instead of MySQL allowed for the elimination of the 10GB database. Last,

sql elimination

persistent data files reduced

build table process constructed to make more files dynamic and transient

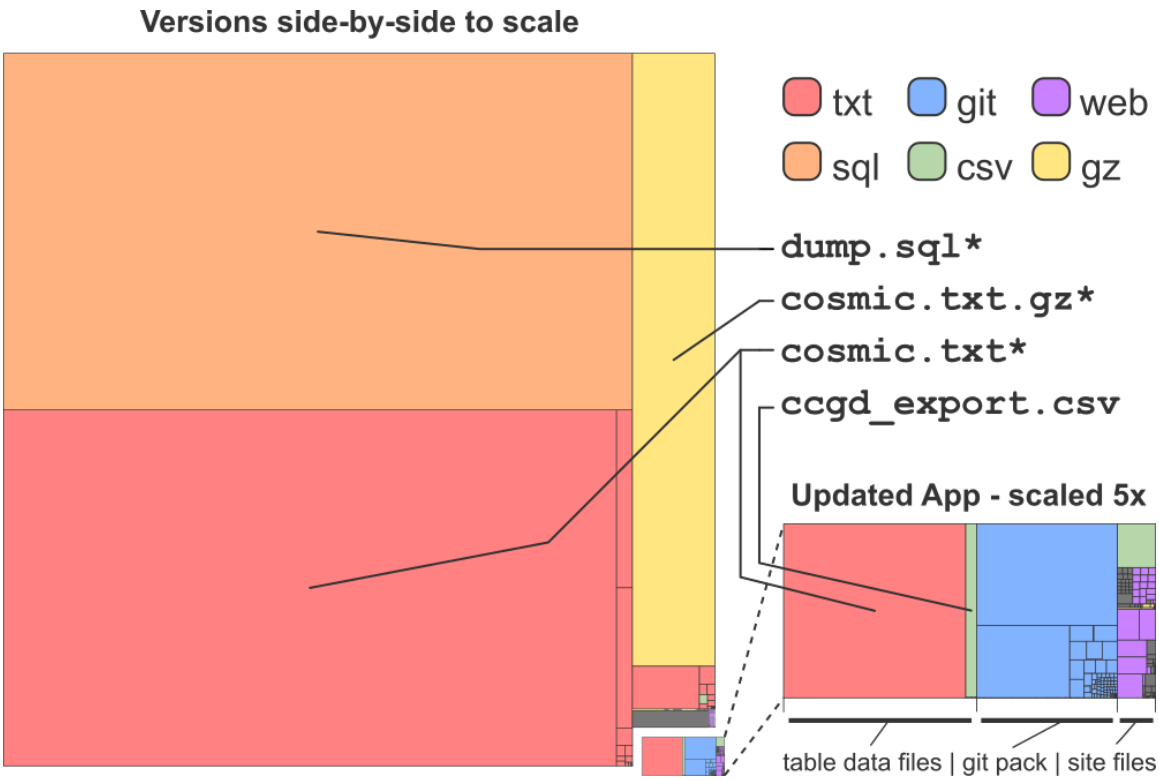

Figure 2.7: **Disk Use Comparison** shows a treemap which illustrates the substantial difference in disk utilization between CCGD versions. The original to-scale map on the left gives an exploded view distinguishing the updated app treemap in the bottom right corner with all remaining elements of the map belonging to the old version. This visual comparison emphasizes the [10x] space decrease made during the upgrade. The fruits of several design choices are present in this space reduction. The most notable impacts of these choices are several massive files eliminated or substantially reduced (noted by asterisk). Notably, the dump.sql file shown represented the disproportionate size of the database and space payoff earned by eliminating it. Also, it can be seen that the persistent cosmic.txt data file is present in both versions. Improvements made in post-process trimming of files like this also produced substantial space returns. Last, the rebuilt table build process transitioned several data objects shown from persistent to transient.

Ref crossdirstat

2.2.3 Administration Streamlined

server admin and app controls streamlined. Site render newly automated.

Chapter 3: Conclusions

Best practices. Discuss open source.

Bibliography

[1] Kenneth L Abbott et al. "The Candidate Cancer Gene Database: a database of cancer driver genes from forward genetic screens in mice." In: *Nucleic acids research* 43.Database issue (Jan. 2015), pp. D844–8. ISSN: 1362-4962. DOI: 10.1093/nar/gku770. URL: http://www.ncbi.nlm.nih.gov/pubmed/25190456%20http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC4384000.

[2] Christopher R. Clark, Wilaiwan DuRose, and Timothy K. Starr. "Cancer Gene Discovery: Past to Present". In: 2019, pp. 1–15. DOI: 10.1007/978-1-4939-8967-6_1. URL: http://link.springer.com/10.1007/978-1-4939-8967-6%7B%5C_%7D1.

[3] Robin Fincham et al. "Software Development Practices". In: *Expertise and Innovation* 15.1 (2011), pp. 168–188. DOI: 10.1093/acprof:oso/9780198289043.003.0008.

[4] Joshua Kerievsky. "Refactoring to patterns". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2004. ISBN: 354022839X. DOI: 10.1007/978-3-540-27777-4_54.

[5] University of Minnesota OIT. *OIT-LPT/Public-Docs: This is where all of our public-facing documentation will live.* URL: https://github.umn.edu/OIT-LPT/Public-Docs (visited on 01/02/2021).

[6] Daniel J Rigden and Xosé M Fernández. "The 27th annual Nucleic Acids Research database issue and molecular biology database collection". In: *Nucleic Acids Research* 48.D1 (Jan. 2020), pp. D1–D8. ISSN: 0305-1048. DOI: 10.1093/nar/gkz1161. URL: https://doi.org/10.1093/nar/gkz1161%20https://academic.oup.com/nar/article/48/D1/D1/5695332.

[7] Hadley Wickham. *Mastering Shiny: a book.* URL: https://github.com/hadley/mastering-shiny (visited on 01/05/2021).

[8] Hadley Wickham et al. “Welcome to the Tidyverse”. In: *Journal of Open Source Software* 4.43 (Nov. 2019), p. 1686. ISSN: 2475-9066. DOI: 10.21105/joss.01686. URL: https://joss.theoj.org/papers/10.21105/joss.01686.