Building a Bloodbank Database

Nicholas Gordon

University of Memphis

Abstract A report on the appropriateness, development, and construction of a database suitable for use by a blood bank. Includes short introduction to databases and competing vendors and web technologies as well. Discussed are the design decisions made, including a discussion of the schema used for the database, as well as the user interface design. Finally, a discussion of the implementation itself is given, including problems faced, analysis of the solution, and future expansions of the solution.

1 Introduction

Databases are ubiquitous in computers. They are an extremely powerful tool for organizing data, which is an imperative in computer science; randomness is tantamount to uselessness. Once data is organized, it can be manipulated in powerful ways to derive meaning from data. One useful example of this is a blood bank. In a blood bank, the data are a pool of patients, each with varying blood types, blood-borne diseases, names, addresses, etc. Without organization, the best you could hope to do is keep track of everyone. With organization, you can generate lists of patients to contact for donation, who can donate to some arbitrary patient, or even location-based demographics on frequency of donation, disease distribution, etc.

Now we come to the database itself. We could simply organize these lists by hand, periodically generating all of these metrics, but we can use computers to muscle this data into formats we perfer for us. This is the impetus for this project: generate a database that can be populated with data from a blood bank which will provide to the bank useful meaning from their data. The rest of this report will deal with some knowledge upon which this report is incumbent, such as differing database vendors and web technologies available for interfacing with these databases, the overall design of the database that has been produced, including the structure of the tables and the database itself, and a discussion of the problems faced, the solution presented, and its appropriateness to the problem at hand.

2 Related Work

In order to fully understand the decisions made during this project and the design of the database given, some background knowledge needs to be established. Given that a not-narrow audience is likely to be reading a paper such as this, this explanation is warranted. Databases often are just referred to as "the database" because of their ubiquity, but the truth of the matter is that there are a great many databases that any sysadmin must choose from, each having their own quirks, costs, and intended uses. According to db-engines.com, which uses a variety of metrics to determine database popularity, the five most popular databases in February 2016[2] are, in order:

1. Oracle
2. MySQL
3. Microsoft SQL Server
4. MongoDB
5. PostgreSQL

This list is not linear, however. The top three entries have reasonably close numbers in terms of popularity, but MongoDB and PostgreSQL have only about 30% each of the share of Microsoft SQL Server. Most databases are relational, which means their records are stored in tables or similar, and relate multiple fields together in items called records. These databases are the dominant paradigm, and only MongoDB on this list is not a relational database. MongoDB fits into another paradigm known as document-store database, which effectively are a dictionary that stores complex objects in a key-value system. The differences between these databases largely come down to individual feature support, cost of use, and enterprise-support. Of note is that MySQL, MongoDB, and PostgreSQL are all open-source software and as such are free to use or have a free version available.

Racket is a Scheme derivative, which emphasizes being a platform for extending the language and being a full-spectrum programming language suitable for a broad variety of uses.[4] It includes as built-in packages tools to develop a dynamically-generated website, as well as for interfacing with a large variety of databases from different vendors.

With regards to the bloodbank itself, when determining something as straight-forwarded as who can donate blood to who, we have an important classifier to understand: the ABO blood group system and Rh factor. These two qualifiers decide what kind of blood any given person can receive.[5] This is critically important to the process; if you give a patient blood they cannot receive, their body could incorrectly identify the transfused blood cells as foreign invaders, and as a result their immune system could have a major reaction, causing severe bodily danger, such as shock or kidney failure.[3]. As a result, there is a strong motivation to get the relevant information correct.

Additionally, there exist infections that reside in the blood of the infected. Many of these infections can be transmitted through the blood, including dengue fever, the hepatitus viruses, HIV, and malaria among others.[1] As such, donations must be screened before they can be given to a needful recipient to avoid causing transmission of these infections. Further, a record of which donors are known to have certain diseases can help a bloodbank prioritize which people they contact to request donations, or provide useful information to viral epidemiology studies.

3 Design

The database itself is described by a schema, which is essentially some logical design of the database stipulating things such as how the tables are interrelated, what tables have which attributes, and how those attributes are constrained. The schema for the table in this database is as follows:

Donor

Attribute	Constraints
id	serial primary key
lastname	varchar(255) not null
firstname	varchar(255) not null
bloodtype	varchar(3) not null
address	varchar(255)
testsdone	text[]
knowndiseases	text[]
lastdonationdate	date
phone	char(14)

As a demonstration of the database's capability and real-world usefulness, a webpage was built. This webpage permits a user, here speculated to be bloodbank nurses or doctors, to view donors, and to manipulate this data accordingly. They are be able to view details about a patient they select from a brief view, and then find a list of patients who could donate blood to this user. Naturally, such a webpage should also enable the user to add, remove, and modify the patients in the database. PostgreSQL was chosen as the backend database system to implement the solution on.

The web-based application design that was chosen is structured as follows: the builk of the application is hosted on the initial site: a search form to narrow results down, and the related list of donors beneath that. For editing donors, viewing more donor details, and finding a list of compatible donors, you are taken to new pages for each of these things. Hosted on these pages are the relevant forms listed in HTML tables, and the content-modifying actions (edit, delete, add) all have additional confirmation pages before any true action is committed.

4 Analysis

My experiences and encounters with this project have been varied and interesting. The biggest stumbling block has by far been the integration of the database with the developed frontend website. Both design choices and implementation issues will be discussed, including schema design, variable typing, data validation, and the peculiarities of Racket in general.

Racket was chosen due to the language having built-in packages for both web-page generation and serving and database interaction. As a result, there was no required code to transform the data between systems or to act as 'duct tape'. This makes the system simpler to maintain, and means that the required skill level to maintain the code is lower than it would otherwise be, using say using some typical combination as LAMP, which requires four separate skill-sets. The Racket/PostgreSQL stac used here requires only two. Additionally, Racket was a good choice because of my prior familiarity with it, avoiding the need to inves time in learning a new language.

I correctly expected addition and updating to be more challenging than the other features, owing to considerations like data validation and integrity requirements. Because of the breadth of information regarding a blood bank, this was left mostly-undeveloped; a future iteration of this could include this.

I expected addition and updating to be more challenging, owing to data validation requirements.

I've chosen Racket because of the aforementioned tight integration of website devlopment capability and database interaction. These qualities make Racket an ideal platform for implementing this design.

First, decisions about what fields to include in the 'donor/patient' model had to be made. Ultimately, the choice was to make the model as broad as possible, hoping to cover as many use-cases as possible; this is the justification for making only first name, last name and bloodtype requirements. This selection of constraints covers the greatest cross-section of possible donors. Further, the decision to limit retained attributes to simple demographics and medical information is practical. A division of information exists between clerical information, such as billing, weight requirements, blood quality, reimbursement, etc., and less ephemeral, qualitative data like name, blood type, address. While a database could be used similarly to retain and manage the clerical information, the solution presented here was intended to be a broadly-applicable one, and so an omission of complexity was valid.

Second, a decision had to be made about how to store the list of tests that have been done on the donor's blood, and what diseases a given donor is known to have. Due to the nature of arrays, the traditional SQL approach to arrays is to instead use additional tables and join those tables as a foreign key. The motivations for eschewing this and using Postgres's built-in, declarative-like array structures are twofold: it allows for more intuitive abstraction about the data, and it simplifies the work needed to convert the data gathered in the frontend to the backend database. Additionally, because the number of entries in these arrays will likely be small (<10), any performance advantage that would be gained from the traditional approach is negligibly small.

After the implementation of this system, it has become clear that none of the features showcased are vendor-dependent; the database backend for this is quite flexible, pending only some code tweaks to the database interfacing code. This is not a huge revelation considering the scope and complexity of the web app.

5 Future Works

While this web application is a working system, it clearly lacks many features and lots of 'polish'. Due to time constraints I have not been able to add some features in, and future development could yield improvements. Several of the most important features are listed here:

– The aesthetics of the page could be improved by use of CSS with appropriate re-design of the underlying HTML.
– An interface redesign could also provide cleaner results, using bigger, modern buttons in place of the standard HTML form buttons.
– The current system for updating and adding patients provides no systematic validation for what blood diseases and tests are permitted; an improved system would be to maintain a whitelist for each set and check inputs against them.
– Calendar functionality would be useful, with regards to scheduling people for donations. Such a system might permit, for example, bringing up a list of donors scheduled for a day, so a worker could call and remind them of an appointment.

It is also not difficult to see how a system like this could be monetized. In a world where HIPAA places draconian requirements on the currency of software used to process and store patient data, software-as-a-service (SAAS) is becoming more popular, as companies seek to reduce their number of technology headaches. As this is built on a web platform, it is on a clear track to SAAS.

6 Conclusion

In conclusion, it is easy to see why a blood bank would need such a database. The test database was seeded with 1000 records, and response time was nearly-instant on a consumer-grade laptop. Compare that to a paper storage system, where information fetch times become the biggest bottleneck to effective operation, and not logistics. Regarding the technical implementation, this project was a lesson in how quickly a piece of software can grow in complexity; from simple create-read-update-delete the project grew to specific reads, data-type validation, constrained views, and an interesting exercise in user experience: do you force two clicks to do a content-modifying action, or not? Do you present all the listings at once, or have a 'more' button? Regardless, a system such as the one presented here is not technically demanding, and in fact very few good 'how-to's exist for this sort of Racket project; this was mostly free-handed. However, a lack of technical complexity is not to discount interface complexity or use-case complexity. This project illuminated many of the issues associated with end-user software, like sandboxing and security, too.

References

1. Cdc - diseases and organisms - blood safety. Online (April 2016), http://www.cdc.gov/bloodsafety/bbp/diseases_organisms.html
2. Db-engines ranking. Online (February 2016), http://db-engines.com/en/ranking

3. L., D.: Blood groups and red cell antigens. Online (2005), http://www.ncbi.nlm.nih.gov/books/NBK2265/
4. The racket language. Online (March 2016), http://www.racket-lang.org/
5. Blood types chart | blood group info. Online (April 2016), http://www.redcrossblood.org/learn-about-blood/blood-types