Overview Over My Bachelor's Thesis

Jannis Eichborn

June 5, 2014

*Things related to Usability later in the explanations

Contents

1 Introduction & Motivation

1.1 Coding in an Open-Source Environment

Brief Description and History
Here I want to talk about how software is developed, give a brief history on how open-source developed over time and present different approaches of people working together on large problems.
What are the main goals of this community and how are these goals expressed?

The Open-Source Community is Changing
What are recent developments, how is code shared, where do we want to go? Which ideals are fulfilled and which things are still in a messy state?

What is the Problem with Current Integrated Development Environments?
Introduction of usability in the context of working with IDEs
Description of problems in the workflow regarding spreading of code, packages in different forms and finding suitable discussion and documentation. Reuse of code is bad and tedious.

1.2 Our Idea for Working on Large Projects in the Future

Description of the idea of having more code in one place with interchangeable and competing solutions. Using crowd sourcing for evaluation...
How is the functionality of an online database for code integral part of the IDE's usability?

1.3 My Work & Goals - A Self-organizing Database for Code

What exactly will my work do in this context and what are my goals. Description of the structure of my thesis. What usability criteria do I aim for?

General notes - Abbreviations.

- I will use the term *function* for anything that is a function, method, routine or subroutine in a given language. Maybe just talking about Julia here?

2 Analysis & Formalisation

2.1 Description of Use-Cases - Client Requirements

This part will include extensive formalisations with respect to usability. How can this be formalised, what metrics can be derived? How do I define the client in this scenario and

what is important to this client?
I describe how people are going to use the client IDE and what requirements arise from this. I will talk about portability between different OSs, what performance clients want and other aspects which are important to the implementation. What are possible interfaces and how can I meet them?
First of all I will have to describe potential clients to my system. What kind of applications will make use of the database, how do they want to communicate and what does this interaction imply for the architecture of the database?
Well to begin with is should be obvious, that I will not be concerned with any kind of graphical interface to the database. It would store millions of entries with several different versions of each entry. Of course one could think of means to make an interactive query interface which can display the database in it entirety but in my mind this is no use case. Users should always query the database in the context of a more elaborate system on top. If you wanted to search for documentation or code only there are already thousands of possibilities to do so on the internet (TODO quote). From my point of view the system is designed to be used by an IDE or something equivalent. It does not mean that other more 'traditional' use-cases are not possible, but I will not be concerned with those in this chapter. Also the aimed database-representation would probably perform suboptimal for these kind of uses (multiple rapid queries at a time over simple indices). For these tasks there are solutions (TODO quote) out there and they cannot be the target use-cases for my system.
In my mind the focus of the system must be to aggregate over the contained data for more elaborate queries. For example a query in plain English would be something like this: "Give me all the function signatures which match this given set of parameters" or "Which functions have the same parameters and return the same type?".
These queries result from using the a possible IDE or smart query system on top of the database. Let me stick with using an IDE for a more extensive example: A User writes a function from the top of his head. He does not refer to anything in the database (yet), but instead simply writes ahead.
Right after he has entered the signature the system can check for the following:

1. Is there already a function with the exact *same signature*?
   Should definitely be shown to the user. Maybe somebody has already done all the work he is going to do now. This might even be an expert in this area who has spend approximately 42 hours upon perfecting the runtime-behavior of this single function. In this case users might be happy to just plug-in the given function as is, or at least make use of the code to get ideas for his own improvements. I think this case might sound pretty uncommon but with thousands of users which roughly confirm to naming-standards of functions, this feature might work pretty fine right out of the box. Also users could then go ahead and write their own variants of this function an re-upload it. This way entering the desired signature alone can be seen as a means to search for functionalities using all the knowledge that you put into the signature.

2. Is there already a function with the *same name*?

Even if the signature is no a match, there might still be quite a lot of use in showing functions which have the same name. They might be different approaches to the same functionality - variations in the parameters might just be due to implementation details - or functions which are similar in their behavior and might need only slight changes to fulfil a new role. As it is the intention to make code reusable and having the database managing all the (sometimes rivalling) code-fragments.

3. Is there already a function with a very *similar name*?
   Using some sort of similarity-measure, one could determine a class of function names which are similar enough to the given function name. Best case this is done using the general naming conventions of the language in question. There are two cases that are worthwhile to consider:
   a) Are the parameters the same?
      Then it is most propable that the user will want to see these results. There is quite a lot of information already in the typed parameter to a function. If those correspond it is worth a look to see whether this function might actually what the user wants in this context.
   b) Are the parameters - while not the same - at least similar
      If the parameters are completely different it will be highly unlikely that the function will do what the user wants to achieve. If however the parameters differ only marginally it might be interesting to see also these function in the database.

In general this all means that the database must be able to process these kinds of queries with reasonable results for all the given cases. While case 1) seems pretty straightforward, there might be problems when it comes to versioning of code pieces and finding the "best" functions which have the same signature. It is of no use for the user to find himself confronted with more than a dozen nightly-hacked matrix multiplications before the "standard" built-in matrix-multiplication is shown.
For all cases in 2) and 3) there is some sort of ranking necessary to present the client with usable results. Also one must consider, that you don't ever want to return *all* results as bandwidth would quickly become a limiting factor to the system, which makes ranking even more interesting. The system should make use of as much context-information as necessary, even drop down to comments in order to rank functions for a given query.

Possible queries

- Retrieval of functions:
  - Finding matching code to a signature.(might also include the return type of the function)
  - Find code to similar signatures or even without a name at all, just so you can see what would be possible with a set of arguments.

  - Finding matching code to a block of code. (Potentially hazardous)
  - Retrieving a newer version of code (maybe just the difference)
  - Retrieving a whole package/module with all functions
  - Retrieving meta-information to functions. Documentation, connections to other code, dependencies/using/requirements, variations, older version etc...
- Uploading functions:
  - A completely new function, without any links.
  - Upload an entire module/package
  - An update/extension to a function which the user wrote
  - A variant to a function.
  - Comments to a function
  - A rating for a function
  - A connection between code

Defining the desired experience from a client's point of view

- Most obviously: Having a consistent, simple and well documented interface to the database. It is well defined what queries can be made (and how, maybe using DTD) as well as what one can expect as a response to those queries.
- The most important criteria: Having fast response times for simple queries (meaning super-fast retrieval of single functions). Searches for larger sets should yield responses quickly as well. Initial guess less than 500ms for first results to arrive.
- Finding the most current version of a function
- Finding the most relevant function if there are multiple implementations
- If one searches for a set of functions or if a query has mutliple matching results, the client will expect some kind of order on the results. This order should feel natural and consistent as people have become used to by modern search engines. If the database fails to deliever at this level people are far less likely to interact with the code and make updates to the system.
- Up to this point I do not want to aim for auto-completion. The client should provide one query and expect a timely answer to it (no streaming yet). Later this feature might get interesting, especially once the database gets partitioned over several clusters and results are aggregated from different sub-graphs. For now my goal will be batched processing of one query after the other.
- Having consistent results. The same query deliveres the same answers, similar queries lead to similar results. Priorization should be transparent to a client, or even configurable (using some sort of arguments for a search).

- If a client updates information in the database or uploads new content, he will want to get feedback whether this was successful or not. Some auxiliary information might be interesting (what version, where, what size, how long etc...). If the transaction failed for whatever reason the client wants to know why.

Required funtionalities derived from the desired experience and possible queries

- Fast (indexed) access to the data in various forms:
  - function-names (some intelligent search like elastic search to be able to find similar names)
  - parameters (in combination with the name or without it)
  - return types in combination with the above.
  - metainformation (not a priority)
- Using these finding mechanism the database must be able to return:
  - A whole function, if there is exactly one hit, in a protocolled and serialized manner.
  - Requested parts of functions (comments, ratings, documentation)
  - Being able to traverse the graph in order to find connections (versions, similar functions, parameters, usage links etc...). **This functionalality is very interesting but is also vast and sometimes difficult. Thus it is not a priority.**
- Also the database must return sets of functions:
  - If a client wants to see/retrieve a package.
  - If a client searches functions similar to a given signature.
  - If a client only provides parameters and wants to find matching functions.

2.2 Non-functional (technical) Requirements to the System

This section focuses on things like scalability, robustness, the iteration speed of changes, whether and where changes to the design are possible and how security and safety standards can or have to be met. Which things are necessary and which ones are nice to have?

Building an interface
The first task is to provide the functionality of the database-system to a remote client. There must be a suitable protocol in place which can handle the transfer or queries and the responses. Criteria: Quick, secure, fail-safe, well-known and understood. The datastream needed should be minimal to avoid loss of data as well as reducing the time/bandwidth needed for the transactions. A client should be able to use the system without a top-notch internet connection.

Trying to minimize the time needed for the queries
This is a central requirement.. Making data accessible is what the database is all about. The client-acceptance of the system is dependent on quick response of the system. Having a neat interface (above) is the first step. Furthermore the system must exhibitthe following properties, which I will explain below:

1. Consistency of the data in order to keep the amount of space needed for the data minimal.
2. Monitoring as a means to detect failures and bottlenecks in the system.
3. Scalability.
4. Ranking of results.
5. Feedback when things timeout or fail.

1) Consistency of the data

- Being able to produce a *unique* ID for every entry which encodes name, signature and timestamp for the given function which was uploaded. This ID must be unique for every entry but still encode in an understandable manner. A function can be translated to an ID and it must be possible to retrieve a function from an ID. (Indexing or traversals)
- The database needs a valid scheme which represents structures in the data. Since I will use a graph-based system there has to be a graph-representation of the data, which is extendable and concise at the same time in order to prevent overhead in the data-representation while still having the means to extends functionalities in the system.
- Items which are put into the database have to be addressable, properly connected and indexed, so that there is no zombie-data in the database. Transactions need to be atomic for retries, dispatchement over different threads and support of thousands of clients.
- One very important aspect is proper versioning of functions. Initially the database should be able to store as many version as the user likes. Over time it could be interesting to implement some garbage collection behavior to sort out unused nodes in the database to prevent the system from becoming overloaded in slow. There must be some way to monitor this behavior.
- Having a proper indexing system.

2) Monitoring
In order to assess the functionality of the database there have to be mechanism to generate and use simple metrics over the system. These metrics can then be used to optimize the behavior of the graph and to evaluate the success of different functionalities. Possible metrics are:

- Number of transactions per second.
- Overall size of the graph; number of nodes/edges.
- Average number of connections between nodes. Distribution of types of connections.
- Average number of versions for one function (or other means of central tendency).
- How the client-access is distributed, how many calls of what category?
- Average times for operations.
- Error counts, failure statistics.

3) Scalability

- The performance of the database must be consistent with growing input. One cannot assume constant behavior but linear would already be pretty bad. Mechanisms must be in place which allow efficient indexing and sorting of data.
- Vertical scalability: If a machine has more resources it should be able to make use of those. More cores mean more threads. Atomic transactions have to be distributed over all available threads using the provided memory in the heap. That means the database will have to be designed for heavy concurrency in all possible places (reading, writing, garbage collection, metrics).
- Horizontal scalability: With more and more data it is vital to be able to partition the data over multiple machines/drives. This should be considered from the start. The database should never be a monolithic process in a single JVM, or at least it must be possible to change that without rewriting the whole codebase. This includes considerations about the consistency of the data over multiple instances or partitions.

4) Ranking of results

- Using some metrics to cap the search-space and the amount of data which has to be retrieved.
- This also means not to squander with the number of individual transactions (possibly concurrent) to the database, but to make processing batched. This will help serving a lot of clients while still providing fast answering to every one of them.
- There has to be a good understanding of the use-cases and distribution of usage to tune this system. Therefore the initial capabilities will be limited to rudimentary features.

2.3 Goals of my Work

Which derive from the requirements above. Explicit goals with respect to usability.

2.4 Structure of My Work

What comes first. How do I want to accomplish the goals and what is the prioritization.

3 Relevant Basics

3.1 JVM - Choice of Language and Context

Why it is still relevant in the context of large-scale distributed computation

3.2 SQL to NoSQL - Why Graph Databases?

What are recent developments in the requirements on databases and how are those met. New types of databases are emerging.

3.3 Distributed Databases

Distributed computations require new forms of data-management. Distributed systems lead to more scalability and robustness in the case of hardware-failures.

3.4 General Thoughts on Performance & Scalability in Recent Software Design

What is the bottleneck in performance nowadays, how is that important to me.

3.5 Usability in the Context of Technical Systems

Usability without interfaces. Including thoughts about usability in the structure of a program/package.

4 Evaluation & Verification

4.1 Why Thinking About Testing Right from the Start Is a Good Idea

Test-driven design and other thoughts..
How do I test for the usability I introduced earlier?

4.2 What I Need to Test

Description of formal aspects which have to be tested. Derivation of suitable measures for the problems.

4.3 How I Test

What are the principles in the implementation of my tests.
Implementation of usability measures

5 Design & Implementation

5.1 Used Software Packages and Tools

Java has been around for nearly 20 years [1]

5.1.1 Scala

5.1.2 Akka

5.1.3 Play

5.1.4 Blueprints-Stack

5.1.5 Titan Database

5.2 First Sketch - Design of my Implementation

how do I tackle the problems and requirements?

5.3 More Detailed Description of the Implementation

To a necessary degree of precision that is.

5.4 Evaluation of the First Sketch

What is good, what needs to be done...

5.5 Description of the Iteration

5.6 Further Evaluation

6 Conclusions

6.1 State of the Implementation at the End of My Work

6.2 Comparison to Requirements and Goals

What was fulfilled, what not and what might be critical. Can the application be extended? What have I achieved at what can people to with it at the time?
Is the code I wrote usable from the clients perspective?

6.3 What to Come - Future from here on

What are the next steps and which people can get involved. How do I see the chances in the future. Concluding thoughts on performance and scalability.

6.4 Summary

Testing features

This is a citation [2]
This is an image as a floating object with a ref to it: See Figure 1 on page 12

Figure 1: Ne fiese text is dat

Appendix A - References

References

[1] JAVASOFT SHIPS JAVA 1.0. `http://web.archive.org/web/20080205101616/http://www.sun.com/smi/Press/sunflash/1996-01/sunflash.960123.10561.xml`, 1996. [Online; accessed 1-June-2014].

[2] Fay Chang, Jeffrey Dean, and Sanjay Ghemawat. Bigtable: A distributed storage system for structured data. *... on Computer Systems (...*, 2008.