YARN: Animating Software Evolution

Abram Hindle, Jack ZhenMing Jiang, Walid Koleilat, Michael Godfrey, Ric Holt
University of Waterloo
{ahindle,zmjiang,wkoleila,migod,holt}@cs.uwaterloo.ca

Abstract

A problem that faces the study of software is to be able to see the aggregated and cumulative effects of fine grained change (source control changes).

In this paper we describe an approach to animating the evolution of a software project. We build our tool YARN (Yet Another Reverse-engineering Narrative) which implements this approach. YARN generates YARN balls (animations) which a viewer can unravel (watch). YARN animates changes in the dependencies between modules in a software system. These dependencies are extracted from a software repository (CVS) and then used to generate a animation. The animation features a static layout out of modules connected by animated edges. Edges are weighted by the number of dependencies or importance of the changes.

1. Introduction

One problem with studying software evolution is how to visualize the change over time. On paper we have static images of architectures and their dependencies but in real software systems those dependencies change over time. The finer the granularity of change (releases, versions, commits), the more changes to evaluate. Important change often occurs at the fine-grained level of CVS commits, thus we wanted to show for a large software system such as *PostgreSQL* what changes were occuring.

One way to represent this forth dimension is to use animation. We can show the state of the dependencies in the system while showing how they change over time with animation. By using time we can take advantage of cumulative views and see differences in the context of the whole rather than just in the context of the exact. This is often a problem with extractors because you are given a snapshot of a change and not necessarily the state of the system.

Software Evolution deals with such large data set that it is often impossible to view all the data at once. Animation enables us to travse this rich and large information space and interpret the data in a visual way rather than a textual or statistical way.

Other tools which animate graphs move the nodes around, instead we assume that all the modules persist throughout the entire time of the animation. Thus we avoid moving the nodes around.

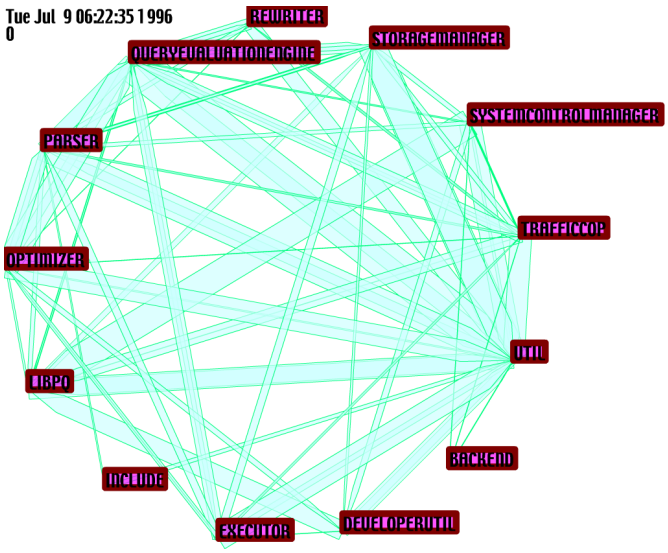
Animation gives a more temporal feeling to the data in the repository and better illustrates the dynamic changing coupling over time.

There are many ways one can draw the coupling, most notably both culmulative addition of dependency or just the difference of edge weights between changes.

1.1 Problem

Thus our problem is how do we animate aspects of this information in a useful and informative way. How do we show the architectural evolution of a software system? What can we gain from such a visualization? Given the large amount of data available what is a compact way to explore it? How can we represent the cumulative effects of change? We can show change through various parameters:

Color: In section **??** we'll discuss the various ways we can



use edge color to describe change and highlight various aspects of change.

Edge Width: Edge Width can indicate the size of changes, unfortunately it is hard to represent negative values with edge width. Edge Widths can also be used to indicate age of changes. Edges can shrink as they age.

Edge Existence: Edges can be shown cumulatively or non-cumulatively, we can show the change itself or we can show the cumulative effect of the change.

The values these attributes represent can be much more complex than a single attribute. We can even attempt to encode an emphasis or a focus on a certain kind of behaviour.

2. Background

There is much related work to this problem. Much of the research comes from the work done in the Mining Software Repositories [3], Software Evolution and Software Maintenance [6] research communities. Lehman's was influential in relation to these research communities. Much of our analysis is related to work by Lehman such as "Programs, Life Cycles and Laws of Software Evolution" [8].

Our *PostgreSQL* architecture is inspired by the architecture of another class project during a previous year, "Conceptual/Concrete Architecture of PostgreSQL" by Xinyi Dong, Lijie Zou and Yuan Lin [4].

3. Tools

Figure 3 illustrates the extraction flow we have used in our analysis. We first use *CVSup* to obtain a local copy of *PostgreSQL* source code repository. Then *C-REX* is used to extract the architectural information for the repository. Once extraction is done, we run *HistODiff*, which makes use of *C-REX*'s functional dependencies to compute the number of dependencies between subsystems, output the dependency graph and select architecturally important changes for us to investigate manually. We use *LogGen* to aid our manual inspection for architectural important changes. Finally, all the dependency graphs are read in and visualized by *YARN*.

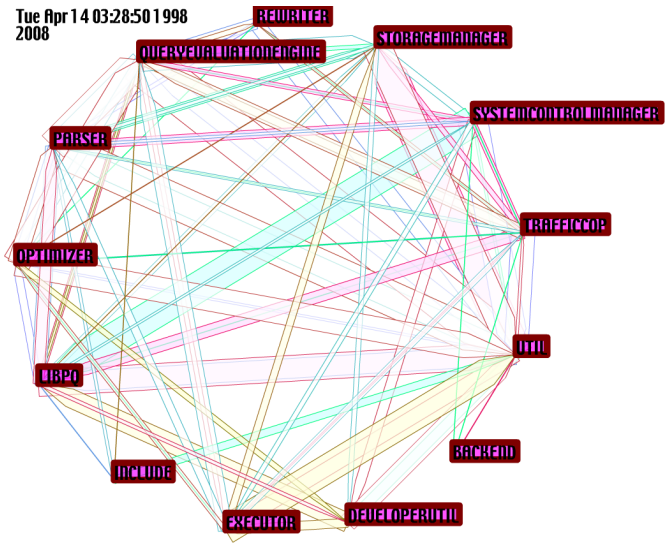
3.1 C-Rex

We choose to use *C-REX* [7] as our fact extractor. *C-REX* is an ideal tool for conducting our historical analysis. Unlike traditional snapshot extractors like *LDX* [10], *RIGI* [9], or *CPPX* [1], which can only retrieve architectural information from one version of the system; *C-REX* is an evolutionary source code extractor which extracts information from version control systems and recovers architectural information over a period of time. Furthermore, source code might not compile properly due to various errors such as dialects of programming language, syntax errors, etc. In this situation, parser-based extractors, like *CPPX* or *LDX*, will fail. On the other hand, *C-REX* avoids parsing the source code by adopting a source code tagging tool called *ctags* [2]. This makes *C-REX* more robust than most of the extractors. Last but not least, most of the extractors operate on the preprocessed code or the object code. Because of compilation flags, the parser-based extraction result contains information only specific to certain configuration. On the other hand, *C-REX* operates on the source code, therefore it extracts more information relevant to software evolution than parser-based extractors.

C-REX analyzes the main branch of *PostgreSQL* source code repositories. CVS stores changes in terms of file revisions. *C-REX* extracts all the changes from each revision and groups revisions into transactions. It outputs two types of information: *Global Symbol Table* and *Transaction Changes*.

Global Symbol Table: maps all the entities ever defined during the history of software development and to the file locations where these entities are actually defined. Entity can be of any C language types, such as macro, variable, function, struct, enum, etc.

Transaction Changes: each transaction change consists of a list of entity changes committed by the same author, at approximately the same commit time, with the same log message. It contains the author's name, a unique hash value to identify this transaction, the commit time, and the log message as well as detailed entity changes.



Entity change can be one of three types depending on the scope of the entity:

Modified Entity: refers to an entity which exists in both previous and current revisions with its contents get modified.

Added Entity: refers to an entity which is added to the newer version but not exists in previous version.

Removed Entity: refers to an entity exists in previous version but got removed from the newer version.

Within each changed entity, *C-REX* keeps tracks of changes in entity types, dependencies, comment changes. If the entity is a function, *C-REX* also keeps track of changes in parameters and return types.

3.2 HistoDiff

HistODiff is an all encompassing program which associates symbols to files, and can handle incrementally changing architectures. It outputs graphs and produces reports of changes which are deemed interesting.

HistODiff relies on the previous assumptions that files are always associated to one module in a hierarchy.

Symbol Mapping: *HistODiff* has to handle the context of multiple symbols meaning different things like functions and macros. The symbols are supplied by *C-REX*'s output. The symbol mappings are important because the changes provided indicate the symbols the changes depend upon.

Architectural Mapping: *C-REX* produces a list of transactions where entities such as functions, macros and variables are added, deleted and modified. *HistODiff* uses this list of transactions and updates the architectural dependency graph with the change in dependencies.

A less cluttered graph is produced, it is the high level module graph, where only user defined subsystems are shown. Essentially this is an lift operation of all the dependencies up to the user defined module level. It outputs two kinds of graphs: dependency graphs between subsystems overtime as well as the difference graph indicating the dependencies changes among subsystems before and after the transaction.

The graphs produced have directed weighted edges which indicate the number of calls between modules.

Filtering: In large projects such as *PostgreSQL* there are many transactions. The significance of each is different. We want to look for changes to the architecture which might not be found. Large changes are often noticeable because they affect many files or had a large line count, but small changes of 1 or 2 lines can add dependencies between modules. These are important changes because they could indicate some kind of architectural violation.

Our filtering heuristics highlight transactions which affect the number of dependencies between the top level subsystems and it satisfies one of the following criteria:

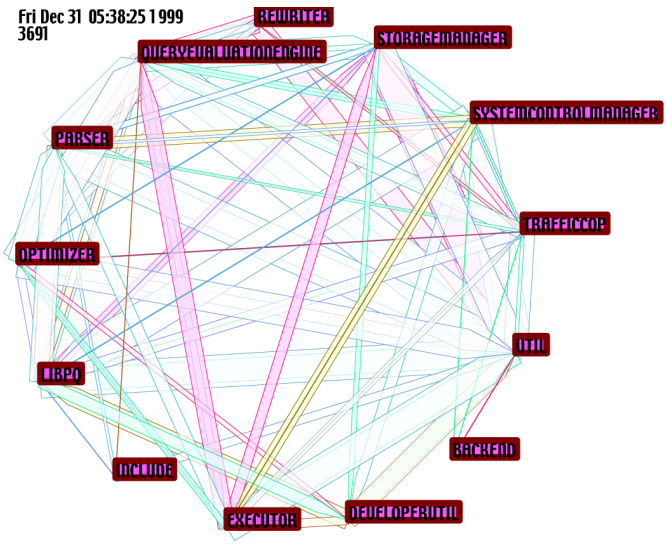
- if there appears to have dependencies between the two subsystems which previous are not connected; or
- the opposite: dependencies between the two subsystems are all removed and two subsystems become disconnected; or
- it doubles the number of dependencies between two subsystems; or
- it reduces the number of dependencies by half.

3.3 YARN

YARN (Yet Another Reverse-engineering Narrative) tries to provide a narrative, a story of the evolution of a software project over time. *YARN* takes in a set of parameters and data then generates, *YARN* Balls. *YARN* Balls (the animations) can be unravelled (watched) to reveal the narrative of how the system changed over time.

YARN uses *HistODiff*'s graph output to produce a graphical animation of graph of modules represented as nodes and their edges as dependencies. These animations are SWF animations which ca be embedded into webpages. The thickness of the edges suggests how many dependencies exist between two modules, we use the measurement $log^2(weight(u, v))$ to determine the edge's thickness based on its weight. The nodes are statically laid out so they don't change position over time. This allows for some sense of coherency between changes.

YARN animates the changes to the dependencies of a graph over time. The modules of the software are layed out statically while the edges are animated.



Edges are directional, when displayed, the edge of lesser weight is displayed inside of the edge going in the reverse direction. Edges are also rendered transparently, thus intersections of edges are both visible and visually resolvable.

Edges can be animated into two different ways:

- Edges exist over the entire time that there is a dependency between two modules. This view emphasizes the current state of the system and what edges are being relatively changed. This view is the cumulative view.

- Edges only exist per each change. This view emphasizes what the actual changes are by removing the extra information. This kind generally lacks the cumulative aspects of the previous, but the edge weight

There are multiple coloring algorithms. Each uses color in a different way in the animation to emphasize certain aspect of the changes over time:

Color Changes on Change: Each edge that changes, changes color, this serves to highlight edges that change. Edges that are changed are highlighted in one color. Thus the more times an edge changes color, the higher the frequency of change that occurs between those two modules. The colors cycle over time thus new colors indicate new changes and older colors which have lasted show edges which haven't recently changed. Unfortunately this color function leads to ambigiuity. The brightening of edges often gives a sense of decay which might not be accurate.

Highlight and Decay: Each edge is highlighted when a change occurs. It is highlighted by changing the color to a bright bold color, then over the period of a few changes the color of that edge decays back down to a neutral color. This color function emphasizes recent changes.

Highlight the Important changes: This coloring algorithm is much like highlight and decay algorithm except only the important changes are highlighted instead of just new changes. This view emphasizes changes which have been tagged as important.

All of the these algorithms assume that the edges are growing and shrinking in width and that only the edges are being animated.

Edge widths can be used in many different ways:

Cumulative Width: Using a scaling function such as $log^2(w)$ we scale the cumulative weight of an edge over.

Decaying Edge Width: The older an edge gets the more it decays in width. Over time an edge is decayed until it reaches a minimum width. When a change occurs it modifies the width of the edge back to it's width according to the scaling function.

Edge Width as Age: Instead of the number of dependencies, edge width alone could indicate when the last change to the dependencies between two modules.

The changes animated are transactions, one frame represents one transaction. For instance, *PostgreSQL* had over 10,800 frames of animation.

In the top corner, the current date of the transaction is shown. Underneath it is the order of the revision. At the bottom a time-line relationally shows where the transaction is with respect to the other revisions. The time-line is click-able to allow jumping throughout the evolution of the project.

The animation produced is a Macromedia Flash animation. It uses vector graphics and is easily embeddable into web pages. This could be used in the such systems like the Software Bookshelf [5] or SoftChange.

Modules can be layed out manually or automatically. Automatic layout algorithms currently include radial or matrix layout.

Figure **??** depicts 6 frames from *YARN* (cropped) over time. Figure 2 depicts a screen-shot of *YARN* in action.

4. Case Study of Postgresql

4.1 Architecture

PostgreSQL has a very well defined layered architecture. The layers pattern gives the software many advantages including the separation of concerns and abstraction.

The three layers are :

Client Interface Layer: This layer accepts inputs from the users through a variety of user interfaces. It submits the queries to the Backend layer below and returns the answers.

Backend: This layer parses the user's query, expands it and presents it to the optimizer which uses information to produce the most efficient execution plan for the evaluation. In order to execute the plan tree, this layer uses the file manipulating functions in the lowest layer.

Data Store Layer: This layer deals with managing space on disk, where the data is stored. Upper layers requires this layer to write or read pages.

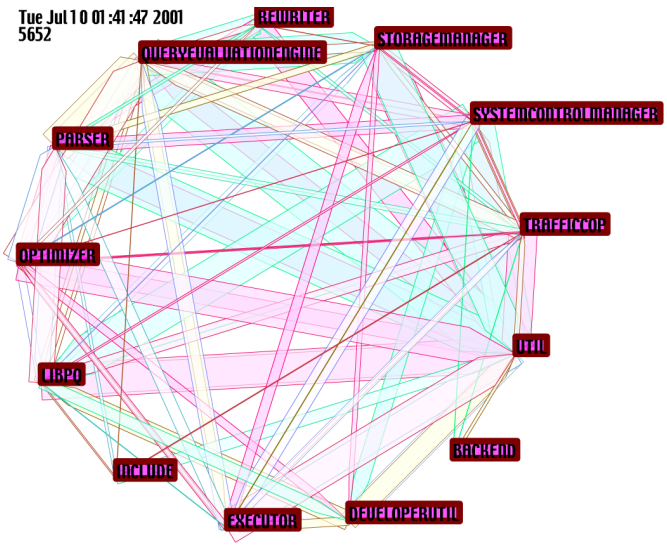
LibPQ: The module contains all the procedures that are involved in providing an interface to the client side application programs. These procedures will be used by the clients to send queries to the lower layer and return results back.

System Control Manager: When a client first tries to access the server, it will spring a `postgres` process and authenticates the client. Once it creates the process for the client, it establish a dedicated communication between the client and the backend. Next, it monitors the connection and records statistics. When the client is not interested in querying the server, the System Control Manager is responsible for disconnecting the communication line cleanly.

Traffic Cop: This module acts as an entry point to the Query Evaluation Engine. After the connection between the client and the backend is established, the client starts sending the queries straight to the Traffic Cop. The Traffic Cop then decides whether the query is simple and doesn't need optimization. If that is true, it sends the query to the Command module to take care of it. Otherwise, the query is complicated, then the query is evaluated, optimized and the result is returned. The Traffic Cop controls the execution of all the modules in the Query Evaluation Engine subsystem. It also directs the data structure through out the different modules.

Parser: This modules tokenizes and parses SQL queries. It also creates Query structures for the various complex queries that will be passed later to the optimizer and then to the executor.

Rewriter: It is the primary module for query rewriting. Although the rewriter is not always called but it is an option that is used when the query uses special constructs. It is also responsible for handling rules which are fired for certain queries. In addition, the rewriter expands the aggregates in the query and deals with recursive statements that have sub links.



Optimizer: This module take the Query structure returned by the Parser, and generate a plan that is sent to the Executor.The Optimizer actually generates all possible ways to join tables and inserts all the necessary preprocessing steps for special queries. In summary, it takes the parse tree and generates all possible paths in which the query can executed. Then takes these paths and calculates which path is the cheapest and then gives the Executor the tree to execute.

Executor: This module takes the plan (execution tree) with the cheapest path and the query, then accesses all the support modules in the lowest layer to execute the query. The execution is monitored and results in a retrieval of information from the database or persistence of data.

Command: This module is called by the Traffic Cop when the query sent by the client is simple enough to skip the Optimizer.

Catalog: This where the RDBMS stores the meta data, e.g information about tables, columns, values etc.

Nodes: This module is responsible for storing the queries in a specified common data structure called Nodes Structure. It is a data structure that allows the storing of any data type and contains procedures for its manipulation.

Util: This consists of procedures and routines that different modules use to do their jobs. It acts more as a support module for the entire Backend.

Data Store Layer:

Storage Manager: This module contains procedures that are related to managing the files and pages of the database on the hard disk.

4.2 A Visual Walk Through of Postgresql

We evaluate the history of architectural changes of *PostgreSQL* by manually inspecting a few "architectural important" transactions. *HistODiff* has flagged out about 100 transactions which are considered as "architectural important" changes. They have either added or removed dependencies which makes two subsystems connected or disconnected, or significantly increased or decreased the degree

of coupling between subsystems. We divided these transactions into 3 parts of equal number of flagged changes. Each part ranged over a different amount of time because less and less of the changes we were looking for occurred:

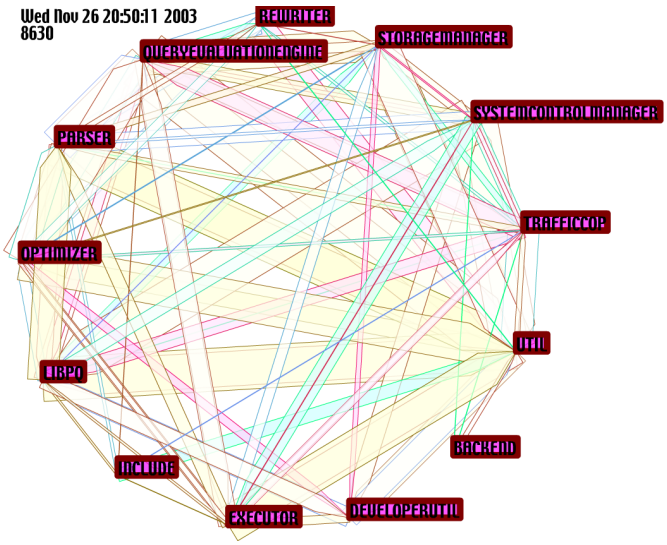
- 1996 to 1998: *PostgreSQL* is released as open source software. Portability and reimplementation of features such as ODBC are included.
- 1998 to 2000: *PostgreSQL* is still in flux, ODBC updates occurred and *PostgreSQL* was extended by the PL/pgSQL language.
- 2000 to 2005: *PostgreSQL* was being maintained, less features are added, more auditing and bug fixing occurs.

4.3 *PostgreSQL*: 1996 to 1998

The theme of *PostgreSQL* from 1996 to 1998 was the transition from a closed project to an open source project used by many users on many different platforms ranging from UNIX to Win32. It covered releases 1.06 to 6.3.

Security: Many changes were related to authentication. These included host based authentication, plain-text password authentication, and checking if *PostgreSQL* was running as the root user.

Portability: There were two kinds of portability issues dealt with: wrapping systems calls and distributing the output of tools such as bison and flex. System call wrapping included wrapping signal with portable signal call code, as well as providing portable file open and close calls (for SunOS). Flex and Bison output was added for systems which lacked compatible versions of the utilities (FreeBSD and AT&T UNIX).



Bugs: There were issues regarding the precedence of attributes from certain tables not being handled appropriately.

Extensibility: The server programming interface was added, it allows *PostgreSQL* to be used from languages such as C and TCL. As well PL/TCL was added which allows stored procedures to use TCL.

Optimization: The heap tuples code was optimized by making it inline.

Updated Code: Old transactions code *Time Travel* was removed. The old ODBC driver was updated with a new ODBC driver.

4.4 *PostgreSQL*: 1998 to 2000

The next period is from May, 1998 to September 2000, which approximately corresponds to Releases 6.4 to Release 7.0.3. There are total 11 releases during this time period.

ODBC they have updated ODBC driver to version 0.0244.

Extension They added a second procedural language PL/pgSQL and then later moved into the offical distribution. Another large change was when PL/pgSQL was migrated from the contribution into the core distribution of postgresql.

Enhancements: Developers improved *PostgreSQL*:

- improved the client/server asynchronous communication;
- added support for outer joins, and read/write locks; and
- rewrote the *Memory Management Process(mmgr)* utilities for better handling of out of memory error;
- and changed the header comments of functions.

4.5 *PostgreSQL*: 2000 to 2005

The period 2000 to 2005 covered Postgresql releases 7.0.3 to 7.4.8. This period seemed contained 1/3 of the revisions in the repository. Large changes to the source included new implementations of SQL statements, improving triggers, JOINS and dropped columns. Most of the changes were maintenance and improvement of properties such as robustness, security and performance. Security improvements were spurred by problems found by “white hat hackers”, who found flaws related to interupt handling and critical sections. Figure 1 depicts the changes made to the system starting from 2001 to 2005.

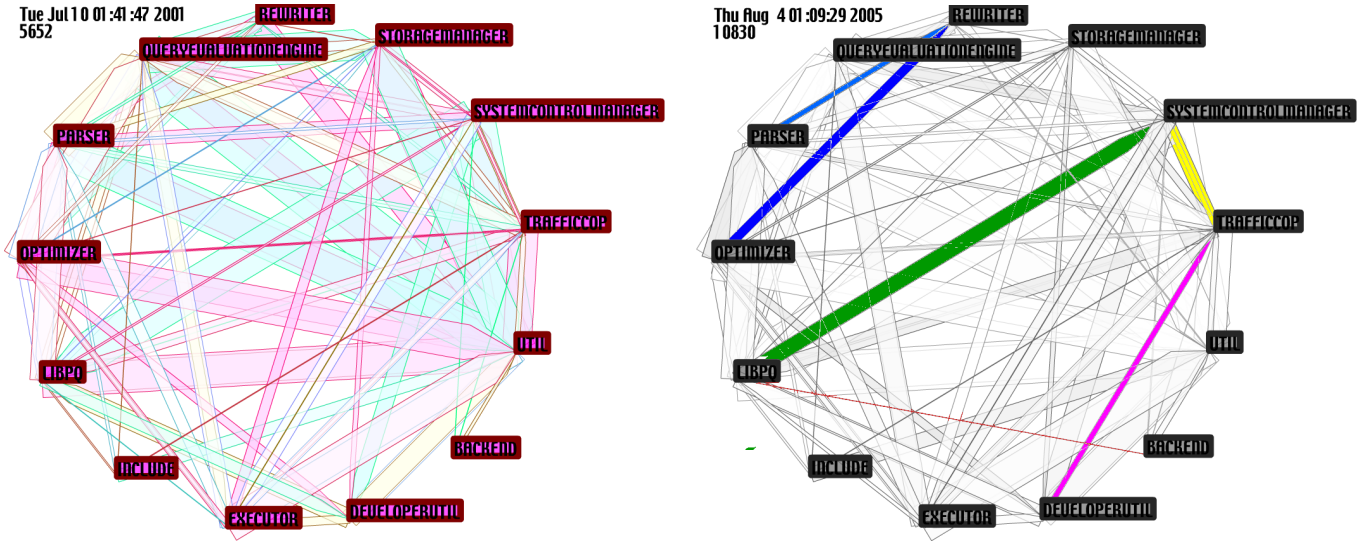
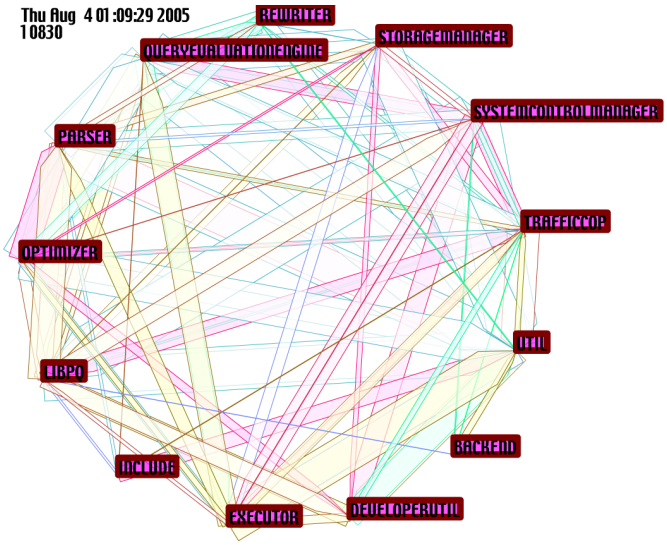


Figure 1. Important architectural changes done during the last 5 years

5. Future Work

Essentialy most extension to YARN is to it's user interface or how it presents information. More layout algorithms will be added to YARN. We plan to add more interactivity to the animation. This would include dragging and dropping modules as well as opening up sub modules. This would allow for hierarchical navigation of sub modules. Different views of the architecture are planned such as source control view which shows which modules are coupled together per commit.

Other work includes evaluating the use of animation for maintenance work. We have yet to answer the question if this animation is useful to developers or just researchers.



6. Conclusions

In conclusion we proposed and implemented a system which can extract the dependency information from a repository and aggregate it into an animation. These animations, or yarn balls, then can be embedded into webpages or shared in order to communicate change based dependency information about software projects. Many different kinds of animations can be produced.

The contributions of this paper thus included: an approach for animated the evolution of a project's dependencies in a coherent static manner; a system to view changes and the cumulative effects of the changes; provide a fine grained view of the evolution of the project.

References

[1] CPPX: Open Source C++ Fact Extractor. Technical report, University of Waterloo. http://swag.uwaterloo.ca/ cppx.

[2] Exuberant Ctags. Technical report. http://ctags.sourceforge.net.

[3] D. Church. A survey of techniques for the recovery and observation of software evolution. Unpublished, 2004.

[4] X. Dong, L. Zou, and Y. Lin. Conceptual/concerte architecture of postgresq. Technical report, University of Waterloo, 2004.

[5] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Muller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. pages 564–593, November 1997.

[6] A. Hassan. Mining software repositories to guide software development. http://plg.uwaterloo.ca/ aeehassa/home/pubs.html, 2005. Accessed July.

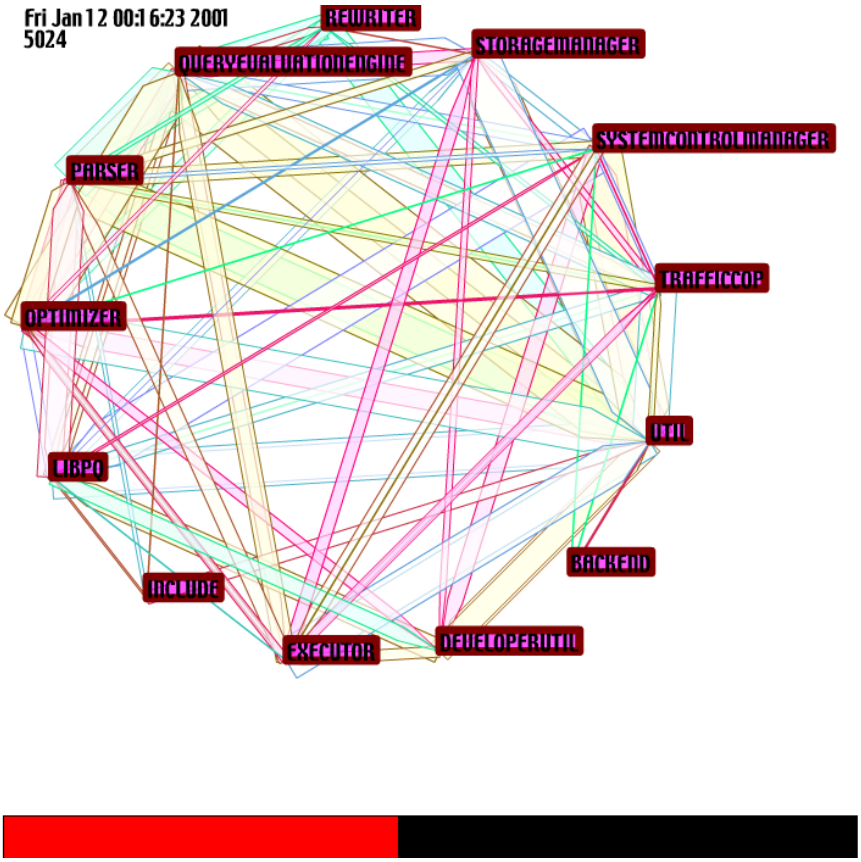


Figure 2. Screen-shot of *YARN* with *PostgreSQL*

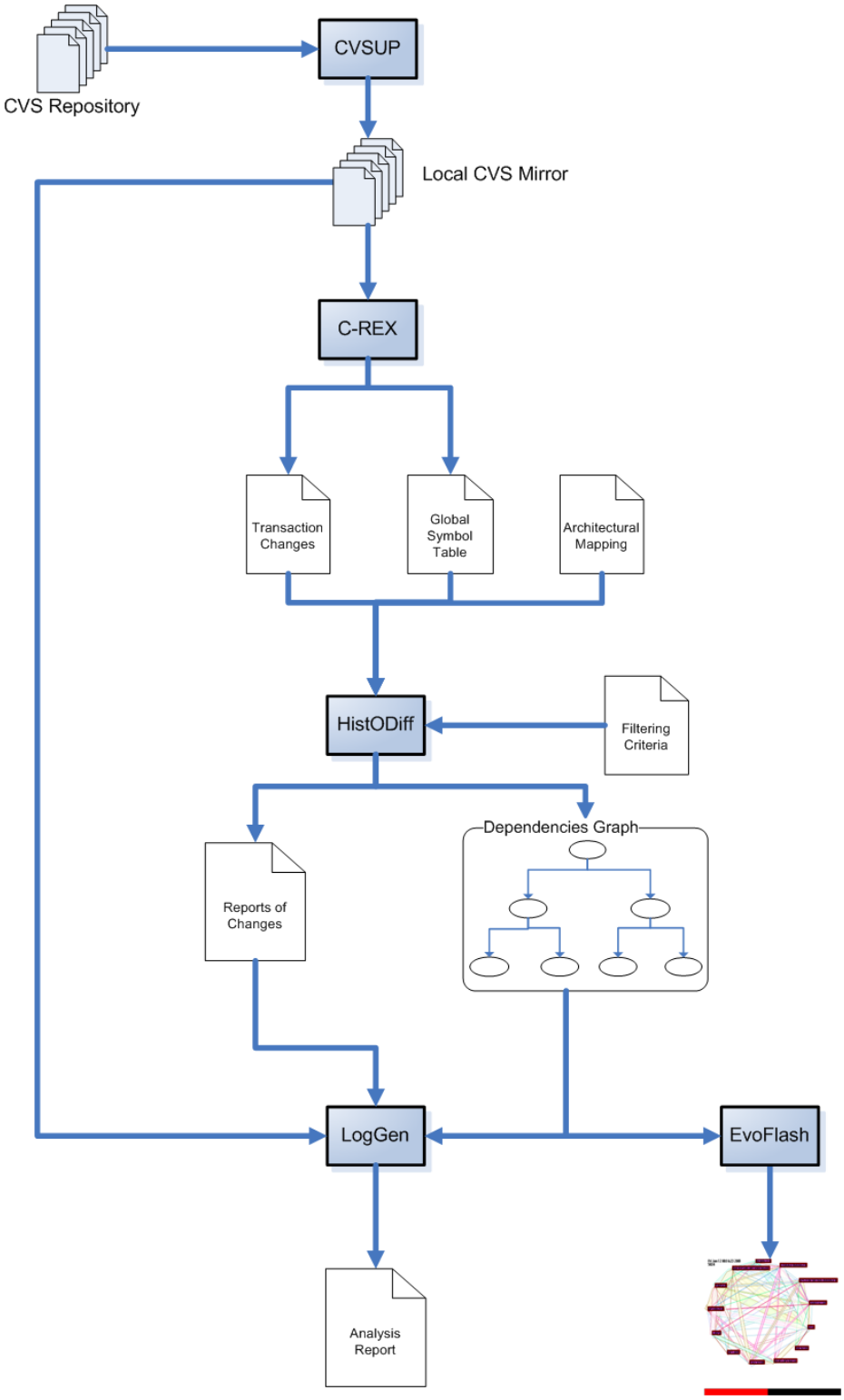


Figure 3. Flow Control of Extraction with our tools

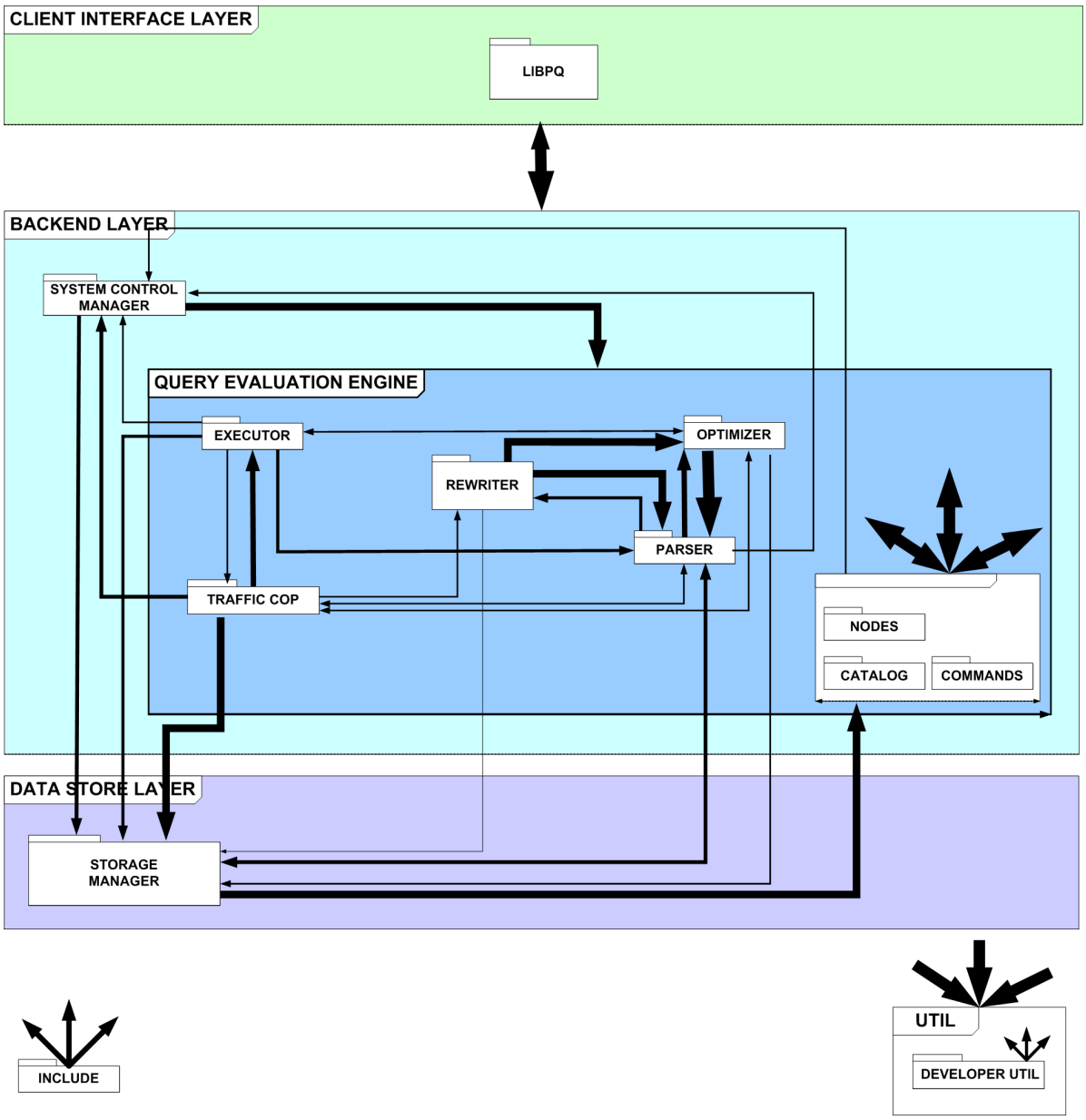


Figure 4. Top Level Architecture of *PostgreSQL*

[7] A. E. Hassan and R. C. Holt. C-REX: An Evolutionary Code Extractor for C - (PDF). Technical report, University of Waterloo. http://plg.uwaterloo.ca/ aeehassa/home/pubs/crex.pdf.

[8] M. M. Lehman. Programs, life cycles and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, Sept. 1980.

[9] H. A. Muller and K. Klashinsky. Rigi - A System for Programming-in-the-large. Technical report, In IEEE 10th International Conference on Software Engineering(ICSE-1998), April 1998.

[10] J. Wu and R. C. Holt. Linker-Based Program Extraction and Its Uses in Studying Software Evolution. Technical report, In Proceedings of the International Workshop on Unanticipated Software Evolution (FUSE 2004), March 2004.