


# Release Pattern Discovery: A Case Study of Database Systems

Abram Hindle, Michael W. Godfrey, Richard C. Holt  
University of Waterloo  
{ahindle,migod,holt}@cs.uwaterloo.ca

## Abstract

*Studying the release-time activities of a software project — that is, activities that occur around the time of a major or minor release — can provide insights into both the development processes used and the nature of the system itself. Although tools rarely record detailed logs of developer behavior, we can infer release-time activities from available data, such as logs from revision control systems, bug tracking systems, etc. In this paper, we discuss the results of a case study in mining patterns of release-time behavior from the revision control systems of four open source database systems. We partitioned the development artifacts into four classes — source code, tests, build files, and documentation — to be able to characterize the behavioral patterns more precisely. We found, for example, that there were consistent activity patterns around release time within each of the individual projects; we also found that these patterns did not persist across systems, leading us to hypothesize that the four projects follow different but consistent development patterns of activity around releases.*

## 1. Introduction

In this research we attempt to  cover release patterns, behavioral patterns of software projects, observed during release-time. ~~We theorize that the release patterns are part of larger patterns which are actually stages of software development processes which the project is following. We hope release patterns will provide users with useful process information.~~ We observe and extract these patterns from the software artifacts available for use. Since many of the development decisions and behaviors are not regularly logged we rely on systems which automatically log activity, such as a project's Version Control System (VCS).

Ever since the Cathedral and Bazaar [10], there has been interest from businesses, developers, managers, and researchers about how Open Source Software (OSS) is created. Previous attempts at investigating the development

processes of Open Source Software (OSS) have analyzed Bugzilla repositories and mailing-lists [4]; these artifacts do not offer the fine granularity of activity records that revisions, from a VCS, provide. We hope to extract this behaviour so that stakeholders in the project have a way to extract and analyze the behaviour that is reflected within the source control repository.

Given that many software processes are composed of stages we suspect we can better observe a system's behavior by looking for changes or ~~revision which~~ are related to stages of software development. For ~~instance~~ maintenance and implementation stages are related to changes in source code files; integration and test stages are related to changes of test files, benchmarks, test framework code, build files and configuration management files; requirements, specification and design stages are related to changes in documentation files.

In this research we will study Open Source Software (OSS) Database Systems (RDBMSs). Many OSS RDBMSs started as proprietary software either commercially or academically. Each project we study has a long history of use and change. We assume that database systems are relatively stable and have similar architectures or provide a similar functionality which might help us compare them.

This work, when incorporated into a more general framework, can help people such as managers and programmers. ~~This work and approach would allow managers to analyze project behaviour around events such as release. It would allow managers to extract the software development process from a project and verify what process the programmers were following. This would reduce the need for managers to trust programmers when they ask them what practices they are following.~~ For example our work can be used to determine when documentation or testing occurs around an event or if activities such as coding and testing occur at the same time.

Programmers could investigate how their project is being maintained. Newcomers to the project could determine the process followed by the programmers and figure out the workflow of the project. Programmers could also ask

the repository, relative to events, when documentation took place.

Researchers could benefit by correlating the behaviour of successful projects, thereby deriving successful software development processes. Researchers could also validate the behaviours of developer against the software development process the developers claimed they were following.

This work does not provide the full analysis of project behaviour, it is the analysis of project behaviour around release time.

In this paper we look explicitly for a common behaviour among all of the projects, since they are all databases perhaps their release-time behaviour is similar.

### 1.1. Background

The *stages* of software development we want to identify, are derived from various software development models such as the Waterfall model [11], the Spiral model [1] and OMG's Unified Process [7]. Example stages include: requirements, design, implementation, documentation, testing, release, etc. Depending on the software development model, stages are followed linearly (Waterfall), or recursively (OMG and Spiral Model). The recursive models are split into smaller iterations. Each iteration mixes together relevant stages much like mini-waterfall models. These iterative models are often called "evolutionary models", as the process elicits feedback from previous iterations to enable the software to react to changes in requirements or design.

*Software evolution* is the study of how software changes over time [9] based on the artifacts left behind by developers. These artifacts include mailing lists, change-logs, program releases, source code, version control systems, revisions, etc. These artifacts often need to be measured or aggregated to study. Common software evolution metrics include lines of code, clean lines of code (no comments or extra whitespace), number of lines changed, lines added, lines removed, etc. Some software evolution metrics measure systems before and after a change, as well as measuring change itself [8, 5].

Process Discovery and extraction is the study of how to determine the process or behavior a project is following either by tooling the development process [3] or by analyzing the software artifacts, such as revisions [6].

Our work is a case study of multiple Open Source systems. There have been many cross project studies of open source systems such as those by Capiluppi et al [2].

### 1.2. Terminology

This section introduces terminology which we use throughout the rest of this paper.

*Version control systems* (VCSs) track and maintain the development and revision history of a project. VCSs are repositories which store revisions to files such as source code and documentation. In this study we used CVS and BitKeeper.

*Revisions* are changes to files stored in a VCS. They are not the actual files themselves but the changes that occur between versions of a file.

*Commits* are the actions taken that add, submit or record revisions to the VCS.

*Releases* are the events where software is bundled for distribution. Usually when a version of the software in VCS is decided to be the release, it is checked out and packaged. There are two kinds of releases: *major releases* and *minor releases*.

*Major Releases* are releases which are large changes to the software that often affect the software's architecture. Generally developers will indicate a major release by using a large change in their version numbering of a release (e.g. MS Windows 95 to MS Windows 98 or Linux Kernel 2.2 to 2.4).

*Minor Releases* are generally smaller than major releases and are usually indicated by a smaller change in the version number of a release (e.g. MS Windows XP SP1 to MS Windows XP SP2 or Linux Kernel 2.4.19 to 2.4.20).

*All releases* include both major and minor releases.

*Release revisions* are revisions that are near a release. Given an interval around a release, a release revision is a revision that occurs within that interval.

*A partition* is one of the four sets of files stored in a VCS that we analyze: *source*, *test*, *build*, and *documentation*.

*Revision classes* are the sets of revisions in a VCS which belong to a corresponding file partition (source, tests, etc.).

*Source revisions* are revisions to source code files.

*Test revisions* are revisions to files that are used for testing the project. Test files include regression tests, unit tests, and other tests that may be added to the repository.

*Build revisions* are revisions to build files such as those associated with GNU Auto-tools (make, configure, automake, etc) and other build and configuration utilities.

*Documentation revisions* are revisions to documentation files, which include files such as README, INSTALL, doxygen files, API documents, and manuals.

*Project behavior* is the behavior of the revisions stored within the project's VCS.

*Release patterns* are the patterns of project behavior that are observed around a release.

## 2. Methodology

This section presents our methodology for analyzing release patterns of a project; we will present the steps involved and then we will followup with an application

Project	Major	Minor	All
Firebird	5	5	10
MaxDB 7.500	1	12	13
MaxDB 7.600	2	11	12
MySQL 3.23	2	68	70
MySQL 4.0	4	110	114
MySQL 4.1	4	110	114
MySQL 5.0	4	110	114
MySQL 5.1	4	110	114
PostgreSQL	7	27	34
Total	33	563	595

**Table 1. Total number of releases per project (Note MaxDB and MySQL fork their repositories so each repository contains most of the releases)**

of our methodology in a case study (section 3). Our project analysis methodology can be summarized as: extracting data for revisions and releases; partitioning the revisions into their revision classes; grouping revisions by aggregation and windowing; producing plots and tables; analyzing summaries of the results with our STBD notation (see section 2.1). For a more detailed description of our methodology please see [6].

We extract revision data from the various repositories using `softChange`, `CVSSuck` and `bt2csv` as described in section 3.2. We then mine mailing-lists, change-logs, manuals, and repository tags for release information. The release information we extract includes the version number, the date, and whether the release is a major release or not.

We partition the revisions into their four revision classes based on the files they are associated with. This is primarily done by checking extensions and substrings of their pathnames. If necessary, one can create a list of files in each partition.

We aggregate and group our revision class partitions for plotting and analysis. Aggregates include averages, summations and windowed functions. In this study we aggregated the revisions by day and filtered the revisions into per-day bins before and after release for a given interval.

Using the aggregated and partitioned data we then plot various graphs such as revision frequency before and after a release, and the linear regression of changes before and after a release. From these plots we generate summaries of the slopes of the linear regression or other aspects of the graphs themselves. These summaries often use the STBD notation which we describe in detail in section 2.1. We then analyze the summaries and try to discover release patterns from the data and our summaries.

Once these steps are done, for each project analyzed

we compare and contrast the summaries of the projects to determine if they have a similar relative behavior.

## 2.1. STBD Notation

We introduce STBD notation as a notation to summarize the results of metrics on our four classes of revisions. We assign a letter to each revision class (S for source, T for test, B for build, D for documentation). We order the class letters from most frequent class to least frequent class: S, T, B, D.

The format of the summary is  $S^*T^*B^*D^*$  where  $*$  could be characters such as  $\nabla$  (down/before),  $\square$  (equals),  $\blacktriangle$  (up/after) (but other characters could be used as well). Generally we try to choose intuitive mappings for these characters, for instance  $\nabla$ ,  $\blacktriangle$ , and  $\square$  map well to the slope of a line (such as the linear regression of a set of points).  $\blacktriangle$  and  $\nabla$  work well for comparing two values, much like less than and greater than. In this paper when we compare two metrics such as revision frequency before and after a release,  $\nabla$  means the revision frequency was greater before a release, and  $\blacktriangle$  means the revision frequency was greater after a release.

An example instance of STBD notation would be,  $S\nabla T\nabla B\blacktriangle D\square$  for revision frequency before and after a release; this would indicate source and test revisions where more frequent before the release,  $S\nabla T\nabla$ , while build revisions where less frequent before,  $B\blacktriangle$ , and documentation revisions were equally frequent,  $D\square$ . We can also omit classes which we are not interested in or did not change. For instance if only source revisions changed or we only want to focus on source revisions we could show  $S\blacktriangle$ .

This notation is very flexible and allows us to identify patterns and make quick judgments based on actual data, the repetition of the class letters helps readers avoid memorizing the ordering. Metrics we can use with STBD notation include: linear regression slopes, average LOC per revision, average frequency of revision, relative comparison of frequencies, the sign of a metric, concavity of quadratic regression, etc.

## 3. Case Study

In this case study we study 4 OSS databases: PostgreSQL, MySQL, Firebird and MaxDB. We start with initial questions and predictions. We then apply our methodology, as discussed in section 2, and then analyze and compare these projects.

### 3.1. Questions and Predictions

For each revision class, we ask these questions:

Project	Major	Minor	All
Firebird	S▼T▲B▼D▲	S▼T▼B▲D▼	S▼T▲B▼D▲
MaxDB	S□T▼B▼D□	S□T▲B□D□	S□T□B□D▼
MySQL	S▼T▼B▲D▼	S▼T▼B□D▼	S▼T▼B□D▼
PostgreSQL	S▲T▲B▲D▲	S▲T▼B▼D▲	S▲T▼B▲D▲

**Table 3. Summary of the four major project's revision frequencies before and after a release with majority voting across the branches, where □ is when no majority is found, ▼ means revisions are more frequent before, ▲ means revisions are more frequent after. S - source, T - test, B - build, D - documentation**

Project	Major	7 days	14 days	31 days	42 days
Firebird	Major	S▲T▼B▼D▲	S▼T▲B▼D▼	S▼T▲B▼D▲	S▼T▲B▼D▲
MaxDB 7.500	Major	S▼T▼B▼D▼	S▼T▼B▼D▼	S▼T▼B▼D▼	S▼T▼B▼D▼
MaxDB 7.600	Major	S▲T▲B▼D□	S▲T▼B▼D▼	S▲T▼B▲D▲	S▼T▼B▼D□
MySQL 3.23	Major	S▼T▲B▲D□	S▲T▼B▲D▼	S▲T▼B▲D▼	S▲T▼B▲D▼
MySQL 4.0	Major	S▼T▼B▲D□	S▼T▼B▲D▼	S▼T▼B▲D▼	S▼T▼B▲D▼
MySQL 4.1	Major	S▼T▲B▲D□	S▼T▼B▲D▼	S▼T▼B▲D□	S▼T▼B▲D▲
MySQL 5.0	Major	S▼T▼B▲D▼	S▼T▼B▲D▼	S▲T▼B▲D▼	S▼T▼B▲D▲
MySQL 5.1	Major	S▼T▼B▲D▼	S▼T▼B▲D▼	S▲T▼B▲D▼	S▼T▼B▲D▲
PostgreSQL	Major	S▼T▲B▲D▼	S▲T▲B▲D▲	S▲T▲B▲D▲	S▲T▲B▲D▲
Project	Major	7 days	14 days	31 days	42 days
Firebird	Minor	S▼T▼B▲D▼	S▼T▼B▲D▼	S▼T▼B▲D▼	S▼T▼B▲D▼
MaxDB 7.500	Minor	S▼T▲B▼D▲	S▼T▲B▼D▲	S▲T▲B▼D▲	S▲T▲B▲D▲
MaxDB 7.600	Minor	S▲T▲B▲D▼	S▲T▲B▲D▼	S▲T▲B▲D▲	S▲T▲B▲D▼
MySQL 3.23	Minor	S▼T▼B▼D▼	S▼T▼B▼D▼	S▼T▼B▼D▼	S▼T▼B▼D▼
MySQL 4.0	Minor	S▼T▼B▲D▼	S▼T▲B▲D▼	S▼T▼B▼D▼	S▼T▲B▼D▼
MySQL 4.1	Minor	S▼T▼B▼D▼	S▼T▼B▲D▼	S▼T▼B▲D▼	S▼T▼B▼D▼
MySQL 5.0	Minor	S▼T▼B▼D▼	S▼T▼B▲D▼	S▼T▼B▲D▼	S▼T▼B▼D▼
MySQL 5.1	Minor	S▼T▼B▼D▼	S▼T▲B▲D▼	S▼T▲B▼D▼	S▼T▲B▼D▼
PostgreSQL	Minor	S▼T▼B▼D▲	S▲T▲B▲D▲	S▲T▼B▼D▲	S▲T▼B▼D▲
Project	Major	7 days	14 days	31 days	42 days
Firebird	All	S▼T▼B▼D▲	S▼T▲B▼D▼	S▼T▲B▼D▲	S▼T▲B▼D▲
MaxDB 7.500	All	S▼T▼B▼D▼	S▼T▼B▼D▼	S▼T▼B▼D▼	S▲T▲B▼D▼
MaxDB 7.600	All	S▲T▲B▼D▼	S▲T▲B▲D▼	S▲T▲B▲D▲	S▼T▼B▼D▼
MySQL 3.23	All	S▼T▼B▼D▼	S▼T▼B▼D▼	S▼T▼B▼D▼	S▼T▼B▼D▼
MySQL 4.0	All	S▼T▼B▲D▼	S▼T▲B▲D▼	S▼T▼B▼D▼	S▼T▲B▼D▼
MySQL 4.1	All	S▼T▼B▼D▼	S▼T▼B▲D▼	S▼T▼B▲D▼	S▼T▼B▼D▼
MySQL 5.0	All	S▼T▼B▼D▼	S▼T▼B▲D▼	S▼T▼B▲D▼	S▼T▼B▼D▼
MySQL 5.1	All	S▼T▼B▼D▼	S▼T▲B▲D▼	S▼T▲B▲D▼	S▼T▲B▼D▼
PostgreSQL	All	S▼T▼B▼D▲	S▲T▲B▲D▲	S▲T▼B▲D▲	S▲T▼B▲D▲

**Table 4. A STBD notation summary table of project revision frequencies across release types, and interval lengths.**

Project	Major	Minor	All
Firebird	S▼T▲B▼D▲	S▼T▼B▲D▼	S▼T▲B▼D▲
MaxDB 7.500	S▼T▼B▼D▼	S□T▲B▼D▲	S▼T▼B▼D▼
MaxDB 7.600	S▲T▼B▼D□	S▲T▲B▲D▼	S▲T▲B□D▼
MySQL 3.23	S▲T▼B▲D▼	S▼T▼B▼D▼	S▼T▼B▼D▼
MySQL 4.0	S▼T▼B▲D▼	S▼T□B□D▼	S▼T□B□D▼
MySQL 4.1	S▼T▼B▲D□	S▼T▼B□D▼	S▼T▼B□D▼
MySQL 5.0	S▼T▼B▲D▼	S▼T▼B□D▼	S▼T▼B□D▼
MySQL 5.1	S▼T▼B▲D▼	S▼T▲B▼D▼	S▼T▲B□D▼
PostgreSQL	S▲T▲B▲D▲	S▲T▼B▼D▲	S▲T▼B▲D▲

**Table 5. Summary of table 4 using majority voting where □ means no majority**

Project	Source	Test	Build	Doc
Firebird	40737	7727	3183	534
MaxDB 7.500	10369	4270	298	52
MaxDB 7.600	23456	7087	318	97
MySQL 3.23	4220	1410	421	21
MySQL 4.0	11593	4936	1033	34
MySQL 4.1	31451	16430	2990	88
MySQL 5.0	45946	26373	3908	105
MySQL 5.1	52897	31389	4772	122
PostgreSQL	39153	4906	7172	3084
Total	259822	104528	24095	4137

**Table 2. Total Number of Revisions per class per project**

- Are revisions of this class more frequent before or after a release?
- Is the frequency of change increasing or decreasing before or after a release?

~~Before a release, we expect to see S▼T▼, because we suspect that developers prepare for a release by fixing bugs and by adding more tests to the project. We expect that newer, more risky features are not being added till after the release, which implies that build files will not change much near a release.~~

### 3.2. Tools and Datasets

We used a set of extractors on various datasets and then analyzed the extracted data with our analysis tools.

Extractors we used include: *CVSSuck* is a tool which mirrors RCS files from a CVS repository; *softChange* extracts CVS facts to a PostgreSQL database; *bt2csv* converts BitKeeper repositories to facts in CSV databases.

Datasets we used include: *Postgresql* (CVS), *Firebird* (CVS), *MaxDB* 7.500, 7.600 (CVS), *MySQL* 3.23, 4.0, 4.1, 5.0, 5.1 (BitKeeper).

Analysis tools we used include: *Hiraldo-Grok* (an OCaml based ~~spin-off~~ of Grok used for answering queries), *R* (a plotting and Statistics Package), *GNUplot* (a graph plotting package), and *Octave* (an OSS Matlab clone).

We selected four successful OSS RDBMSs. Two of the RDBMSs originally started in academia (Postgresql and MySQL) while two were gifted from commercial companies to the Open Source Community (Firebird and MaxDB). Of those RDBMSs, we used multiple forks of the databases (MySQL and MaxDB) primarily because their major release branches did not share the same VCS repositories.

~~Databases were chosen because they implement similar functionality and rely on a large body of experience, thus we expect they share some architectural traits in common. These projects are relatively mature and thus have automated tests, benchmarks and developer documentation.~~

## 4. Results

We have plotted much of the data and provided it in a summarized form. We can evaluate this data from various perspectives or viewpoints such as release types, interval length, project, revision classes, and the linear regression of the frequencies of revision classes.

### 4.1. Indicators of Process

~~Our current results indicate that there was no global consistency across revision classes across all projects. However our results suggest there was some consistency within a project. This suggests that this consistency, this behavior, might be part of the process that the project followed, and that different projects follow different processes.~~

One indicator of process that seemed extractable was how test revisions could relate to the test methodology used by the project. Looking at MySQL 3.23 to 4.1 we can see that source revision activity increased across the release (table 6) while test revision activity decreased (it



Project	Major	Before	After	Both
Firebird	Major	S T B D	S T B D	S T B D
MaxDB 7.500	Major	S T B D	S T B D	S T B D
MaxDB 7.600	Major	S T B D	S T B D	S T B D
MySQL 3.23	Major	S T B D	S T B D	S T B D
MySQL 4.0	Major	S T B D	S T B D	S T B D
MySQL 4.1	Major	S T B D	S T B D	S T B D
MySQL 5.0	Major	S T B D	S T B D	S T B D
MySQL 5.1	Major	S T B D	S T B D	S T B D
PostgreSQL	Major	S T B D	S T B D	S T B D
Project	Major	Before	After	Both
Firebird	Minor	S T B D	S T B D	S T B D
MaxDB 7.500	Minor	S T B D	S T B D	S T B D
MaxDB 7.600	Minor	S T B D	S T B D	S T B D
MySQL 3.23	Minor	S T B D	S T B D	S T B D
MySQL 4.0	Minor	S T B D	S T B D	S T B D
MySQL 4.1	Minor	S T B D	S T B D	S T B D
MySQL 5.0	Minor	S T B D	S T B D	S T B D
MySQL 5.1	Minor	S T B D	S T B D	S T B D
PostgreSQL	Minor	S T B D	S T B D	S T B D
Project	Major	Before	After	Both
Firebird	All	S T B D	S T B D	S T B D
MaxDB 7.500	All	S T B D	S T B D	S T B D
MaxDB 7.600	All	S T B D	S T B D	S T B D
MySQL 3.23	All	S T B D	S T B D	S T B D
MySQL 4.0	All	S T B D	S T B D	S T B D
MySQL 4.1	All	S T B D	S T B D	S T B D
MySQL 5.0	All	S T B D	S T B D	S T B D
MySQL 5.1	All	S T B D	S T B D	S T B D
PostgreSQL	All	S T B D	S T B D	S T B D

Table 6. Linear Regressions of daily revisions class totals (42 day interval): ▲ indicates a positive slope, ▼ indicates a negative slope, □ indicates a slope of 0

Project	Major	7 days	14 days	31 days	42 days
Firebird	Major	▲(0.56)	▲(0.65)	▼(0.50)	▼(0.50)
MaxDB 7.500	Major	▼(0.01)	▼(0.01)	▼(0.01)	▼(0.03)
MaxDB 7.600	Major	▲(0.60)	▲(0.58)	▲(0.53)	▼(0.28)
MySQL 3.23	Major	▼(0.42)	▼(0.32)	▼(0.36)	▼(0.34)
MySQL 4.0	Major	▼(0.40)	▼(0.35)	▼(0.36)	▼(0.36)
MySQL 4.1	Major	▼(0.48)	▼(0.46)	▼(0.45)	▼(0.41)
MySQL 5.0	Major	▼(0.45)	▼(0.47)	▼(0.48)	▼(0.45)
MySQL 5.1	Major	▼(0.45)	▼(0.44)	▼(0.49)	▼(0.46)
PostgreSQL	Major	▼(0.46)	▲(0.64)	▲(0.61)	▲(0.62)
Project	Major	7 days	14 days	31 days	42 days
Firebird	Minor	▼(0.30)	▼(0.37)	▼(0.38)	▼(0.44)
MaxDB 7.500	Minor	▼(0.49)	▲(0.50)	▲(0.72)	▲(0.83)
MaxDB 7.600	Minor	▲(0.66)	▲(0.66)	▲(0.57)	▲(0.51)
MySQL 3.23	Minor	▼(0.31)	▼(0.37)	▼(0.41)	▼(0.43)
MySQL 4.0	Minor	▼(0.45)	▼(0.47)	▼(0.46)	▼(0.48)
MySQL 4.1	Minor	▼(0.45)	▼(0.48)	▼(0.48)	▼(0.48)
MySQL 5.0	Minor	▼(0.47)	▼(0.48)	▼(0.48)	▼(0.48)
MySQL 5.1	Minor	▼(0.48)	▼(0.49)	▼(0.50)	▼(0.50)
PostgreSQL	Minor	▼(0.40)	▲(0.59)	▲(0.52)	▲(0.50)
Project	Major	7 days	14 days	31 days	42 days
Firebird	All	▼(0.37)	▲(0.51)	▼(0.45)	▼(0.48)
MaxDB 7.500	All	▼(0.06)	▼(0.10)	▼(0.32)	▲(0.55)
MaxDB 7.600	All	▲(0.65)	▲(0.65)	▲(0.57)	▼(0.47)
MySQL 3.23	All	▼(0.32)	▼(0.36)	▼(0.40)	▼(0.43)
MySQL 4.0	All	▼(0.45)	▼(0.47)	▼(0.46)	▼(0.48)
MySQL 4.1	All	▼(0.45)	▼(0.48)	▼(0.48)	▼(0.48)
MySQL 5.0	All	▼(0.47)	▼(0.48)	▼(0.48)	▼(0.48)
MySQL 5.1	All	▼(0.48)	▼(0.49)	▼(0.49)	▼(0.49)
PostgreSQL	All	▼(0.41)	▲(0.60)	▲(0.54)	▲(0.53)

Table 7. Comparison of average of total number of revisions before and after a release. ▼ indicates more revisions before a release, ▲ indicates more revisions after a release.

had a negative slope after release as well). This ~~seems to indicate~~ that source revisions were not heavily correlated with test revisions. PostgreSQL and Firebird's test revisions seemed to change in the opposite way of their source and build revisions. MaxDB, on the other hand, had correlated release patterns between source and test revisions. This ~~would imply~~ that MySQL, Firebird, and PostgreSQL did not follow a test-driven development model, or that testing was done at a different time in the process, yet MaxDB followed a release time process or pattern in which test and source changes are correlated.

## 4.2. Linear Regression Perspective

We used the STBD notation to describe the results of multiple linear regressions of revisions per day before, after and during a release (table 6) with before and after intervals of 42 days.



Notable release patterns found from the linear regressions were: S▼T▼ for the before slopes were common for *All* releases for every project except for MaxDB 7.6; a slope of B▼ was common among *before* intervals for *Minor* releases and *All* releases.

*After* release summaries for *All* and *Minor* releases were very similar with the most notable change in build files for MySQL. *Before* release patterns of S▲T▲ were consistent with after patterns of S▼T▼ for MySQL 4.1 to 5.1 and PostgreSQL. This indicates there was a concave up peak around release time. Peaks and dips around release time are interesting because they seem to indicate that an event that changes behavior occurs.

In the *Both* interval the most consistent pattern was B▼ for *Minor* and *All* releases and the opposite for *Major* releases (B▲). The B▲ pattern matches with the summaries in table 5. For all releases and all intervals, except *Major After* and *Major Both*, we observed B▼ for build revisions. This suggests that build revisions are probably less common around release time than during the non-release time. We observed there was almost a freeze in build changes around a *Major* release as if features which include new files or compilation changes were being held back.

## 4.3. Release Perspective

We can see from table 5 that there was a definite difference in behavior between *Major* and *Minor* releases per each project and across projects. These tables summarize the average frequency before and after release. Source revision behavior seemed relatively consistent between *Major* and *Minor* releases, except for MySQL 3.23 and MaxDB 7.500.

Tests were quite different  for all projects other than MySQL 3.23, 4.1 and 5.0,  release patterns were different between *Major* and *Minor* releases. For Firebird,


MaxDB and PostgreSQL, test patterns were completely opposite. Firebird and PostgreSQL modified tests more frequently before a *Minor* release than after, where as for MaxDB and for all the MySQLs tested more often before a *Major* release than after; yet for *Minor* releases test release patterns were the opposite of MaxDB and were inconsistent for MySQL.

Projects with inconsistent build release patterns were Firebird, MaxDB 7.600, MySQL and PostgreSQL. Only Firebird and MaxDB had B▼ for *Major* releases yet mostly B▲ for *Minor* releases. MySQL and PostgreSQL seemed to change build files more before *Minor* releases rather than *Major* releases.

A general release pattern of PostgreSQL, that was consistent across all releases, was that there were more revisions to PostgreSQL after a release than before, except for the one week interval. Perhaps this indicates that PostgreSQL follows a process that relies on code freezes or a delay of patches till after a release.

The differences between *Major* and *Minor* releases seemed similar between projects but were mostly noticeable in MaxDB 7.5. The differences between *Major* and *Minor* releases seemed more prevalent than differences between intervals.

## 4.4. Interval Length Perspective

Tables 4  table 7 all provide summaries of data organized by interval. The one week interval in table 4 showed that *Major* and *Minor* releases seem to act quite differently, but when combined, *All* releases had more changes before release than after. The week *before* release generally had more activity than the week *after*. We can confirm this in table 7, where we see only for one release of MaxDB and the *Major* releases of Firebird, were there more revisions during the week after release than before. Yet for *Major* releases of projects such as MySQL and PostgreSQL there were more build revisions after a release than before.

Another interesting release pattern related to one week intervals was that documentation had the most equally frequent results (MaxDB 7.6 and MySQL 3.23 to MySQL 4.1). These equally frequent results suggest there were not a lot of revisions to documentation during the *Major* releases (alternatively, the behavior was stable). The longer the interval was, the more noticeable and track-able the documentation revisions became. This was most likely due to the infrequency of documentation revisions.

Some projects have shown some stability across intervals, such as the pattern of PostgreSQL for intervals of 14 days or greater for *Major* releases, and the pattern of MySQL 3.23 for *Minor* and *All* releases, for all intervals plotted in table 4. MySQL 3.23 was also somewhat consistent for *Major* releases beyond the one week interval.

Some projects went through a slow transition of behavior from one interval to the next.

This interval based analysis lends itself to per-project analysis since there was inconsistency between projects.

#### 4.5. Project Perspective

Probably the most notable perspective is the per-project perspective. Results are generated internally to a project, and show consistency within a project.

Firebird's shape for source revisions around *Major* releases was a concave up dip in frequency around release time followed by a rise; for *Minor* releases its source revision frequencies had a concave down shape where the frequency peaks around the release. The opposite behavior was observed for test revisions. Source revisions were consistently more frequent before a release across all release intervals. It seemed that Firebird was generally documented around the time of *Major* releases rather than *Minor* releases.

The MaxDB branches are interesting because their behavior between branches is often inconsistent or directly opposite. For *Major* releases MaxDB had consistent revision frequencies of  $T \downarrow B \downarrow$ . The linear regression of *Major* revisions, for the *both* interval and the *before* interval, were consistent and downward sloping; *Minor* releases inconsistently displayed the opposite behavior for everything except documentation revisions. For *All* releases, all intervals and every revision class, except documentation, of MaxDB 7.5 had the opposite slope of MaxDB 7.6. MaxDB 7.5 did not have many revisions immediately after release. Perhaps there were not enough MaxDB revisions but also perhaps the nature of development changed from adding new functionality to maintaining the code.

The MySQL branches are not totally consistent but they show a transitioning consistency through their branches. In general MySQL followed  $S \downarrow T \downarrow D \downarrow$  for revision frequency. Build frequency release patterns were often inconsistent. Although for *Major* releases build revisions were definitely  $B \uparrow$ . According to table 2 we can see that the proportion of test revisions to source revisions grew to over 5 : 3, therefore testing within MySQL seems very important. Slopes of  $S \uparrow T \uparrow$  before and  $S \uparrow T \uparrow$  after, and  $S \uparrow T \uparrow$  before and  $S \downarrow T \downarrow$  after were observed for MySQL's *Minor* and *All* releases. The shape of the curve for source and test revisions around release time for MySQL was a general upward slope or a concave down peak.

PostgreSQL was the only project to have more build revisions than test revisions. For *Minor* and *All* releases, for all revision classes except documentation, PostgreSQL had a concave down peak at release (e.g. slopes of  $S \uparrow T \uparrow B \uparrow$  to  $S \downarrow T \downarrow B \downarrow$ ). For *Major* releases, PostgreSQL had a concave up dip for source, build and documentation revisions and

had a concave down shape for test revisions. For the *both* interval, there was a positive slope for all revision classes except for tests. PostgreSQL seemed to have the most emphasis on frequent change after release; PostgreSQL's behavior seemed most indicative of the kind of project that uses code freezes before release.

#### 4.6. Revision Class Perspective

Source revisions were the most common revisions of all the revision classes. For most intervals and all releases (*Major*, *Minor*, *All*) source revisions were usually more frequent before a release than after. Only for MaxDB 7.6 and PostgreSQL were source changes more frequent after a release than before. It seems that source revision frequency almost returned to an equilibrium during the 42 day window. The slope of the linear regression of source revisions was inconsistent across *major* and *Minor* releases. For *All* releases, source revisions have a consistent positive slope before release, the slope usually changed after release.

Test revisions were the second most numerous kind of revision across all of the projects. For *Minor* releases and *All* releases tests were usually more frequent before a release than after. For *Minor* releases the slope of the frequency change across the release was positive. *Major* releases had a negatively sloped test revision frequency across all intervals. Whereas across *Minor* and *All* releases, tests had a concave down peak around release time. For Firebird and PostgreSQL, tests were more frequent after a *Major* release, where as for MaxDB and MySQL they were often more frequent before release. For *Minor* releases we observed the opposite behavior for all projects except MySQL. Tests seemed to be correlated with the frequency of revisions before and after release.

Build revisions were the third most frequent revisions for all projects except PostgreSQL. For *Minor* releases build revisions were negatively sloped across the release where as for *Major* releases they were positively sloped across the release, and dipped down around release time. Perhaps build revisions were more likely after a *Major* release because new functionality was added that required more configuration changes. Build revisions seemed to be inconsistently frequent for all releases, but had a more consistent negative slope across *Minor* releases.

Documentation revisions were the least frequent revisions for all projects except PostgreSQL. The frequency of documentation revisions indicated that they were changed more often before a release rather than after. Perhaps documentation was left until the functionality was frozen so that it could be described. *Major* releases of Firebird, and all releases of PostgreSQL had documentation changes that were more frequent after release than before. Documentation revision frequencies were the most likely to be equal across a release mainly due to the lack of




documentation revisions. The shape of documentation revisions around *Minor* releases was usually a dip, but inconsistent for *All* and *Major* releases.


Source, test and documentation revisions seemed to be related and often mimicked each other in frequency, but usually not in ~~slope, near release~~. Build revisions seemed to be the most at odds with the behavior of other revision classes, and were the most likely to be more frequent after release. Documentation revisions were the most likely to be missing after release because they were the least frequent.


#### 4.7. Answers to Our Questions

For each class of revision, are revisions in that class more frequent before or after release? We answered this question from section 3.1 based on our frequency tables (tables 5 and 4).

**Source revisions**, in general, were changed more frequently before a release than after. There were exceptions such as the early MySQLs, MaxDB 7.600 and PostgreSQL. 

**Test revisions**, were usually more frequent before either a *Major* or *Minor* release than after. ~~Perhaps testing was done just to verify that the project is ready for release.~~

**Build revisions**, for *Major* releases were usually changed more after a release than before; the opposite was true for the *Minor* releases for all projects, except for Firebird and MaxDB 7.600. 

**Documentation revisions**, were changed more before release than after for MySQL and MaxDB; Firebird and PostgreSQL displayed the opposite behavior for *Major* releases. 

Is the frequency of change increasing or decreasing as we approach a release? To answer this question we rely on the linear regression results from table 6.

**Source revision** frequency had a consistent before release behavior (S▼) for *Minor* and *All* releases, but was inconsistent for *Major* releases. For *Major* releases only PostgreSQL and some MySQL branches had S▲ across the release. Consistent patterns found were both intervals had S▼ (downward slopes), and both intervals had S▲ slopes and concave down dips. Source releases generally increased before a release except with *Major* releases.

**Test revision** slopes were T▼ across *Major* releases, and T▲ across *Minor* releases. The slope across *All* releases was inconsistent across the projects.

**Build revision** slopes for some MySQL and PostgreSQL *Major* releases, increased across the release (B▲). PostgreSQL's *Major* release build revisions formed a concave up shape. For Firebird and MaxDB the general trend was B▼. For *Minor* and *All* revisions PostgreSQL had a concave down peak around release.

**Documentation revision** activity increased across the release (D▲) for both Firebird and PostgreSQL. After a

release, documentation revision frequency often had a flat slope, which indicated that there was no documentation activity after a release for most projects except Firebird, PostgreSQL and MaxDB 7.600.

Thus we can see that due to the differences between projects there was usually no clear consensus on what each revision class does per project, but we saw with MaxDB and MySQL there was some internal consistency between branches.

~~Our predictions were inaccurate, we did not expect such inconsistency between projects.~~ Our prediction of S▼ was consistent for all projects except MaxDB 7.600, MySQL 3.23 and PostgreSQL (table 5). Our prediction of T▼ revision frequency was common for both MaxDB databases and all *Major* releases of MySQL. MaxDB's *Major* releases were inconsistent with their *Minor* and *All* releases. Build revisions before *Major* releases occurred more often after release than before for MySQL and PostgreSQL, ~~where as~~ for *Minor* and *All* releases the general trend was that build revisions were either inconsistent or were more frequent before release.

#### 4.8. Validity Threats

~~Our analysis is really only relevant to 4 OSS RDBMSs discussed here and that any generalizations are meant in reference to these projects and not necessarily the universe of OSS.~~

Our five main threats to validity were: deciding if a release was a *major* release or a *minor* release; determining if branching seen in MySQL and MaxDB affected our results; determining if we had enough revisions per aggregate to be statistically significant; deciding on an appropriate interval length; comparing the projects against each other based on their internal measurements. None of the projects were within a single stage of development, it was a mixture of stages.

#### 5. Future Work

Future work will entail studying more projects, applying more analysis techniques, evaluating the data from different perspectives, and investigating non-release time patterns.

More projects should be analyzed in order to achieve some kind of statistical significance so we can generalize our results about OSS release patterns. We will probably need at least 40 projects before we can generalize about OSS processes and release patterns. Other perspectives from which we could evaluate include: the distributions, the authors, the files, co-changes, forks, branches and other events.

## 6. Conclusions

Through the application of our methodology on four OSS RDBMSs, and the partitioning of revisions into four classes (source, test, build, and documentation) we observed that release patterns exist within projects, although these patterns are not necessarily observed across projects.

One of the observed release patterns is that the frequency of source revisions generally decreases across a release. This might indicate that at release time, developers divert their file updating efforts to activities such as packaging and distribution, which are not recorded in the VCS repository.

The fact that release behavior differs from project to project suggests that the projects we studied follow different processes and/or have different properties.

**Acknowledgments:** This research was partially funded by a NSERC PGS Scholarship.

## References

- [1] B. Boehm. A spiral model of software development and enhancement. *SIGSOFT Softw. Eng. Notes*, 11(4):14–24, 1986.
- [2] A. Capiluppi, P. Lago, and M. Morisio. Characteristics of open source projects. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, page 317, Washington, DC, USA, 2003. IEEE Computer Society.
- [3] J. E. Cook and A. L. Wolf. Automating process discovery through event-data analysis. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pages 73–82, New York, NY, USA, 1995. ACM Press.
- [4] D. M. German. Decentralized open source global software development, the GNOME experience. *Journal of Software Process: Improvement and Practice*, 8(4):201–215, 2004.
- [5] D. M. German and A. Hindle. Measuring fine-grained change in software: towards modification-aware change metrics. In *Proceedings of 11th International Software Metrics Symposium (Metrics 2005)*, 2005.
- [6] A. Hindle, M. Godfrey, and R. Holt. Release pattern discovery via partitioning: Methodology and case study. 2007.
- [7] I. Jacobson, G. Booch, and J. Rumbaugh. *The unified software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [8] T. Mens and S. Demeyer. Evolution metrics. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, New York, NY, USA, 2001. ACM Press.
- [9] A. Mockus, R. T. Fielding, and J. Herbsleb. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):1–38, July 2002.
- [10] E. S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary (O'Reilly Linux)*. O'Reilly, October 1999.
- [11] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering*, pages 328–339. IEEE Computer Society Press, Mar. 1987.