Free University of Bolzano - Bozen

Bachelor Thesis

Designing a Fast Index Format for Git

Author:
Thomas Gummerer

Supervisor:
Dr. Johann Gamper

A thesis submitted in fulfillment of the requirements
for the degree of Bachelor in Computer Science

in

July 2013

Contents

List of Figures

Abstract

Version Control Systems allow users to track the history of their projects, by keeping track of versions of the project, so called revisions. Git is a distributed version control system, which means that every user has his own repository and no central server is needed. To detect changes of the files on the file system, Git uses an index file, which is a list of all files in the repository with associated meta-data. The current index file format (index v2) is slow when working with large repositories, because it is impossible to read or write it partially. Unfortunately, because of the design of the index file, it is not possible to just change the algorithm to avoid those disk operations. The solution is a new index file format, which allows partial reading and writing. The new index file format was sketched with python prototypes and then integrated into Git in C.

Chapter 1

Introduction

1.1 Motivation

Version Control Systems allow users to track the history of their projects by storing the versions of the project so called revisions. This versions can be retrieved at a later time, so if something goes wrong in the development, the code can be reverted to an old (working) state. Using the history, the contributions of each developer can be tracked too.

Git is a distributed version control system, which means that every user has his own repository, and no central server is needed. In a distributed version control system, a central server can be set up, to which the users can push, when they have a network connection, but no network connection is needed when working with the local repository. Co-workers can also share branches between them to test them, before sharing them with the public or a central repository.

To detect changes of the files on the file-system, git uses the index file, which is a list of all files in the repository with associated meta-data. The meta-data consists of the stat data of the file, a sha-1 sum of the file, and some flags that are used by git internally. The stat-data is a heuristic to check if the file has changed on the file-system without reading the whole file. If the stat-data is not enough to determine if a file has changed, the file is read and compared with the sha-1 sum in the index. The index file can also contain additional information in the form of extensions. Two extensions are supported in the current index v2, the cache-tree and the resolve undo extension.

The current index format (index v2) is slow when working with large repositories, because it is impossible to read or write it partially. Even if only the data of a subdirectory is needed, with the current design the whole index file has to be read. The same is true for changes in a file. Even if only a single file has been changed, the whole index file has to be rewritten. Because of the design of the index file, it is impossible to change the algorithm to avoid this excessive disk operations.

1.2 Problem Description and Objectives

The index is a simple list of files, with the meta data being a part of each of them. Each of those index entries has a different length because of the file names, making it impossible to bisect the index, or search for a particular entry without scanning the whole index linearly until the entry is found. Because of this it takes O(n) time to read a part of the index (n is the number of files in the index). Bisectability could be enabled by adding a table with the offsets to each file as extension at the end, but there is no offset to the extension, so that the whole index file would have to be read first, making it useless.

To ensure the integrity of the index file and avoid corrupting the repository because of a hardware failure or manual tinkering with the index file, a sha-1 checksum at the end of the index protects it. Because this sha-1 is calculated over the whole index file, it has to be completely read to check its integrity. In addition, when something changed in the index file, the sha-1 checksum has to be recalculated over the whole data that will be written.

While this is not a problem for small to medium sized git repositories, it is problematic for large repositories, such as WebKit [1]. Repositories will probably grow larger in the future, making the change of the index file format even more sensible.

The in-memory format of the Git index is just a simple array of all files with the attached information. The array is accessed directly, instead of being accessed through an API. New index formats can't simply change the in-memory format, without touching a large portion of the Git source code. To be more future proof and clean up the source code, it would also be good to have an API to access the index file structures.

The objective for the new index file format is to address those problems, and get average partial reading and writing times of O(log(n)), where n is the number of files in the repository. While addressing the problems, data integrity is always the first priority, and cannot be sacrificed for speed. Full reads should also get faster, by compressing the index file.

The long term goal is to provide an API, through which the index file will be accessed and which could take full advantage of the new index file format.

1.3 Organization of the Thesis

The Thesis is organized in the following sections:

- Chapter 2: A short overview over current version control systems and Git.
- Chapter 3: An overview of the old index, the design of the new index format, and solutions for a new index format that were considered, but not deemed feasible for various reasons.
- Chapter 4: An overview of how the new index file format was implemented.

[1] http://webkit.org

- Chapter 5: Some experiments of how the new index file format stacks up against the older formats.
- Chapter 6: The conclusion with an overview of the work.

Chapter 2

Related Work

2.1 Version Control Systems

"Most major software development projects involve many different developers working concurrently on a variety of (overlapping) sets of files, over a long period of time. It is therefore critical that the changes made by these developers be tacked, so that you can always tell who is responsible for any changes, as well as what your source file looked like some time ago. Furthermore, it's just as important to be able to merge the contributions of those many developers into a single whole. This is where version control comes into play.

The basic functionalities of any version control system are to keep track of the changing states of files over time and merge contributions of multiple developers. They support this, for the most part by storing a history of changes made over time by different people. In this way, it is possible to roll back those changes and see what the files looked like before they were applied. Additionally, a version control system will provide facilities for merging the changes, using one or more methods ranging from file locking to automatic integration of conflicted changes." [1]

2.2 Types of Version Control Systems

The most basic type of version control systems are local Version Control systems, that have a simple database that keeps all changes to files under version control.

This kind of version control is nearly useless when multiple developers on different systems need to collaborate. To solve this problem, Centralized Version Control Systems (CVCSs) such as CVS, Subversion and Perforce were developed. They have a single server that contains all versioned files and a number of clients that check out files from that central place. While this setup offers many advantages, such as that everyone knows to a certain degree what everyone else on the project is doing and that administrators have fine-grained control over the repository it also has some serious downsides. There is only a single point of failure. If the server is down, during the downtime nobody can collaborate or save versioned changes. If the central database becomes corrupt and

proper backups have not been kept, everything is lost, the whole history, except the snapshots that people have on their machines.

This problem is solved by Distributed Version Control Systems (DVCSs). In a DVCS such as Git, Mercurial or Bazaar clients don't just check out the latest snapshot of the files, but they mirror the full repository. If the server crashes the whole repository including the history can be restored by copying back one of the client repositories. In addition to that, many of these systems deal well with having different remote repositories, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows workflows that aren't possible in centralized systems. In addition, all repository operation except remote operations are available offline. Users can work offline and when a network connection is available, they can transfer all the information to the server. [2]

2.3 Git

Gits history is strongly linked with the history of the Linux kernel. The Linux kernel was maintained from 1991-2002 by passing around changes to the software as patches and archived files. [2] After that, BitKeeper[1] was used as Version Control System. It was the "best tool for the job", but there was a lot of criticism, because while it was free to use for kernel developers, it was not open source software, which was bugging some developers. Andrew Tridgell then wrote a tool of his own, after reverse-engineering the BitKeeper protocols. After that incident, Larry McVoy, CEO of the company that created BitKeeper, decided to no longer let the kernel developers use BitKeeper for free. [3]

In the search for a new version control system, Linus Torvalds decided to write a new tool based on the lessons learned while using BitKeeper. The goals for the new system were as follows: [2]

- Speed
- Simple design
- Strong support for non-linear development
- Fully distributed
- Ability to handle large projects like the Linux kernel efficiently

After the 2.6.12-rc2 kernel release on April 3th 2005 Linus Torvalds started developing the new version control system. [4] It was announced by Linus Torvalds on April 6th 2005 on the linux-kernel mailing list. [5] The first kernel commit in Git was done on April 16th. [4] Linus Torvalds stopped maintaining Git on July 27th 2005. The maintainership was given to Junio C Hamano, who is still the maintainer to this date. Linus Torvalds is still working on the project as contributor from time to time.

[1] http://bitkeeper.com

The README of the project explains the name “Git” as follows: [6]

”Git” can mean anything, depending on your mood.

- random three-letter combination that is pronounceable, and not actually used by any common UNIX command. The fact that it is a mispronunciation of ”get” may or may not be relevant.
- stupid. contemptible and despicable. simple. Take your pick from the dictionary of slang.
- ”global information tracker”: you’re in a good mood, and it actually works for you. Angels sing, and a light suddenly fills the room.
- ”goddamn idiotic truckload of sh*t”: when it breaks

Chapter 3

A New Index for Git

3.1 The Git Index File

Git uses the index file as virtual working tree state. It records a list of paths and their object names and serves as a staging area. The saved date is then written out as the next object that will be committed. The index is also used to check for changes of files on the file system, for example in '*git status*'. The state is "virtual" in the sense that it does not necessarily have to and often does not, match the files in the working tree. [7]

In order to speed up the comparison between the files in the working tree and the index entries, the index entries record the stat[1] information obtained from the file system via the '*lstat(2)*' command, when they were last updated. When checking if they differ, Git first checks the stat data of the files. If some of these "cached stat information" fields do not match, Git can tell that the files are modified without even looking at their contents. [7]

Example 3.1. *Let x be an empty file checked in to the repository, with the following data (The output comes from '*git ls-files –debug*' and the sha-1 checksum from '*git ls-files -s*').*

```
1 x
2   ctime: 1354663449:395880439
3   mtime: 1354663449:395880439
4   dev: 64769 ino: 1739333
5   uid: 1000 gid: 1000
6   size: 0 flags: 1
7   sha-1: e69de29bb2d1d6434b8b29ae775ad8c2e48c5391
```

After changing the file, the file size, the mtime and the ctime change. Depending on the OS and the editor the inode number may change too. Because the contents of the file changed the sha-1 changed too.

*When executing a command that checks the index if a file has changed, for example '*git diff-files*', first the ctime, mtime, dev, inode, uid, gid and the size are checked for changes. Because the ctime, mtime and size changed (ctime to 1354663630:158023998, mtime to 1354663630:52021582 and size to 6), '*git diff-files*' does not need to check the file contents, speeding up the operation. The command outputs the following:*

[1] ctime, mtime, device, inode, user id, group id and file size

```
:100644 100644 e69de29bb2d1d6434b8b29ae775ad8c2e48c5391
 0000000000000000000000000000000000000000 M x
```

*After adding the changes to the index using '*git add x*', the changed ctime, mtime, size and sha-1 checksum are represented in the index too.*

```
x
  ctime: 1354663630:158023998
  mtime: 1354663630:52021582
  dev: 64769 ino: 1739333
  uid: 1000 gid: 1000
  size: 6 flags: 1
  sha-1: 4cf5aa5f9a644263dbe3d6e78bcbef45487a802c
```

For the future examples in this thesis, an example repository will be used, containing the following files:

- main.c
- revenues.h
- revenues.c
- db.helper.c
- db.helper.h
- db/sqlite.h
- db/sqlite.c
- db/dbstructure.sql
- db/sample/smalldb.sql
- db/sample/largedb.sql

3.2 Index v2

Index v2 is the current index format used by Git. It starts with the header, which contains the signature (DIRC), the index version (2) and the number of entries in the index. The header is followed by a list of index entries, each of them having the ctime, mtime, dev, inode, mode, uid, gid, file size, the sha-1 of the represented object, a flags field and the path relative to the top level directory. The path is padded with null bytes to a multiple of eight bytes while keeping it NULL-terminated. The index entries are lexically sorted.

After the index entries, there may follow a number of extensions, which are identified by a 4 byte signature, which is followed by the extension size, and the extension data. Currently, Git supports cached tree and resolve undo extensions. When a extension is not understood, it can be ignored by git.

The index file is concluded with a sha-1 checksum over the content of the file before the checksum. For exact byte definitions of the file see: [8].

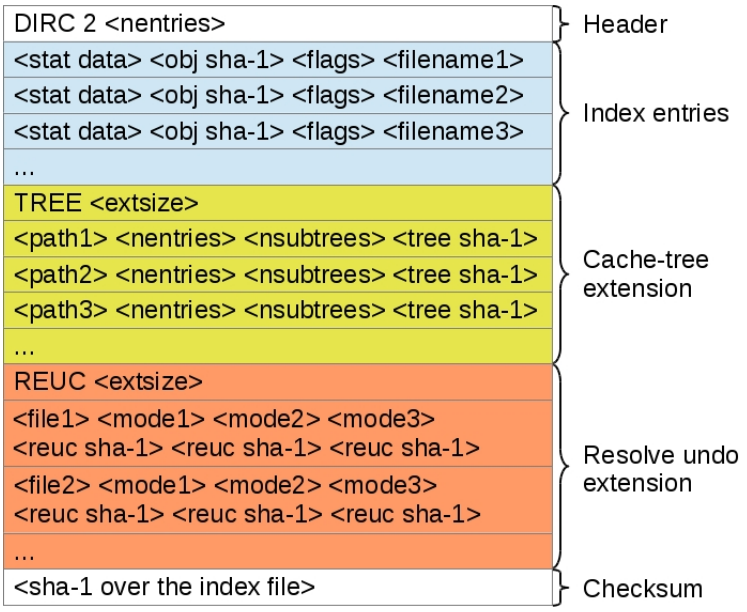


FIGURE 3.1: The structure of the Index v2 file format

Figure 3.1 shows the structure of the index file format v2. Everything between angled brackets, is replaced with real values in the actual index.

The header contains the signature, version and number of index entries. The signature and the index version have to be present in every index file, regardless of the version to do the initial checking. After that part, there may be a number of entries specific to each index format.

The second block are the actual index entries. There may be an arbitrary number of index entries, with the number being defined in the header. For simplicity of the illustration, the stat data also includes the file mode.

The third and forth blocks are the optional cache-tree and resolve undo extensions. There may be more such extensions in the future, but currently only those two are supported. They both start with the extension header, which includes the signature and the extension size. After that follows the extension data, whose format depends on the extension.

The index is concluded by the sha-1 checksum over the whole index, again drawn in white.

Example 3.2. *Below you can find an example of the index file of the example repository in 3.1. The numbers are not converted to network byte order, in order to make them human readable, but they are otherwise real numbers from an actually created repository. It also contains only the resolve undo extension for simplicity.*

```
1 DIRC 2 10
2 1354809222:350491303 1354809222:350491303 64769 1736643 100644 1000
3 1000 100 76aaf436fbdb61e4a839f845af59a638a0984d7b b db.helper.c
4 1354809315:12235636 1354809315:12235636 64769 1736438 100644 1000 1000
5 101 50834ad8d16290f692ef07445cd6b157b3002b5c b db.helper.h
```

```
1354821580:899517114 1354821580:899517114 64769 1736586 100644 1000
1000 99 ocf7cc61f340c59e4dc3906105bc5ad6db27e6f4d 12 db/dbstructure.sql
1354818695:467814725 1354818695:467814725 64769 1736665 100644 1000
1000 182 783b425b53077963dad271738eabee00ea40cf3cd 13 db/sample/large.sql
1354818614:282067768 1354818614:282067768 64769 1738590 100644 1000
1000 56 b18c06ba820dece04dee9f027cca4a56eaeb6992 13 db/sample/small.sql
1354817219:792911291 1352507303:0 64769 1738383 1000644 1000 1000
4850710 22d5dea0e0276cc64c12487b3b092ced2e1e9390 c db/sqlite3.c
1354817219:795911353 1352507302:0 64769 1738543 100644 1000 1000
342230 eb8596292d1599a5a467912824c464262b93e793 c db/sqlite3.h
1354809936:598423356 1354809936:598423356 64769 1736635 100644 1000
1000 153 1041f31643bf94ea23293368c9c2a1e80cc69122 6 main.c
1354810109:741572481 1354810109:741572481 64769 1736658 100644 1000
1000 113 1a99e671bcfc82f9db774baf6be9bec0df3d9b50 a revenues.c
1354810062:794157341 1354810062:794157341 64769 1736657 100644 1000
1000 109 4a69d3b236a5fc54b4bbbf2985be079029d13d17 a revenues.h
REUC 93
db.helper.c 100644 100644 100644 76aaf436fbdb61e4a839f845af59a638a0984d7b
5152b85b8409db4434091e4552bb93c1b20de161 4037f599dcc0e5b3a4c7af29fba57980bbef1105
70a41646f300cadd070ae9779f3d914e34da8a17
```

The header part is in line number 1 in the above listings, starting with the signature "DIRC", the version number 2 and the number of files, 10 in this example repository.

The following lines, 2-21 show the actual index entries, where each index entry uses two lines, with the following fields: The mtime and ctime are written as time seconds:time nanoseconds. Nanoseconds can be enabled or disabled at compilation time of Git, because they are not reliable on every system. If they are disabled, their value is always 0. Following to the times are the device id and the inode number. In this example the device id is always the same because the repository is not spread over different devices. Following after that is the file mode. Git currently supports only 4 file modes: normal files, with or without executable bit, Git links and symlinks. The file mode is in octal format. Following the modes are the user and group id of the file, followed by the sha-1 checksum. After that follow the flags, which are used internally by Git, and which include the length of the file name for more efficient reading.

Lines 22-24 contain the resolve undo extension, which for this index file contains only one file, which was conflicted in a previous merge. There are three file modes, denoting the files in the three stages. If a stage is not present during the merge, the mode for it will be 0. Following to them are three sha-1 checksums for the file in the different stages. If a mode is 0, the checksum will be missing.

The last line (25), contains the sha-1 checksum over the whole index file.

Index v3 has the same format as index v2, but it supports extended flags. If extended flags are present, a flag is set in the flags field, and 2 bytes are added to the specific index entry, for additional flags for the internal usage in Git.

Index v4 has the same structure as index v2 and index v3, but the filenames are prefix compressed, saving disk space. The number of accesses to the disk is lowered and is therefore slightly faster (See Chapter 6 for file sizes and timings). The index still has to be completely read/written with every operation.

3.3 The New Index File Format v5

Index v5 is designed to eliminate the problems of the previous versions of the index file and to allow partial reading and writing. It is a "contiguous tree" format, which reflects the directory structure of the repository. The sha-1 at the end of the index file is replaced by crc checksums, which guard the individual entries.

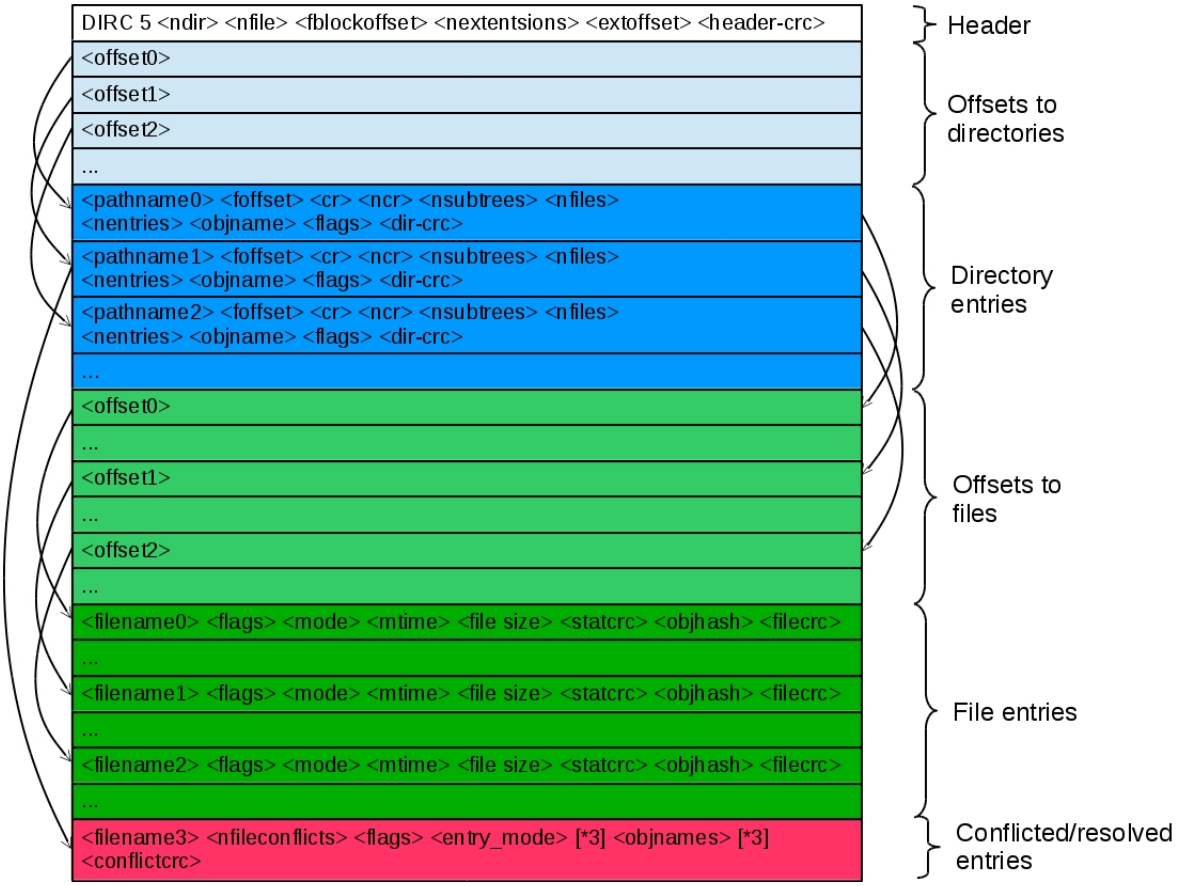


FIGURE 3.2: The structure of the Index v5 file format

Index v5 is split into different sections, as shown in Figure 3.2. Each of them will be described below.

The header contains the general information about the index file. Starting with the signature and version number, and then followed by the number of directories in the index, the number of files in the index, and the offset to the first file. Following that is the number of extensions and the offset to those extension, so that they can be read directly without accessing the rest of the index first. Currently, no extension is supported, so the number of extensions is always 0, and no offset is in the index file. The header is concluded by a crc checksum to verify it.

The second block following the header are the offsets to the directories. They are used for bisectability of the directory data. The offsets are also used to determine the length of the pathname, saving a strlen for each pathname that is read. This is a small optimization, because strlen is a slow operation. To allow this optimization for the last directory too, there are n + 1 offsets in the block, where n is the number of directories in the index.

The last offset is the offset to the end of the block of path entries, and thus to the beginning of the file offsets. The offsets are relative to the beginning of the block of path entries.

The third block is the block of pathentries, which starts with the pathname, followed by the offset to the first file offset of the files in the specific directory. Then, the offset to the conflict data and the number of conflicts are following. The offset to the conflict data is relative to the beginning of the index file. The next field is the number of subtrees of the directory. The number of files, is the number of files in each directory, which is both used when reading the whole directory, and when bisecting for a specific file. The number of entries and the objname are cache-tree data. The flags field is reserved for future use, such as an implementation of Git, which also tracks directories, instead of just files. All directories, except the root directory, are saved with a trailing slash. Each directory entry is concluded with a crc checksum.

The next block is the block of offsets to the file entries. They are similar to the offsets to the directory entries and are used both for bisectability of the files, and to save the call to strlen when reading the file entries. The file offsets are referenced by the pathentries to make sense of the files. The offsets are relative to the beginning of the block of files, and the foffsetblock in the header is used to get to the first file.

The block of files, starts with the filename, followed by a flags field and the file mode. The file mode only uses 16-bit, instead of the 32-bit which were used in index v2. Following that is the mtime and the file size. The statcrc is a crc checksum, which includes the rest of the stat data (ctime, dev, inode, uid and gid). Following that is the objhash, the hash of the file which the entry represents. Every file entry is concluded by the file crc, a crc checksum, over the file entry and the offset to the offset, as save guard against possible corruption.

The block of conflicted/resolved entries is the last block in index v5, as long as there are no extensions. The conflicts are at the end of the index, so that the index doesn't have to be rewritten when the conflicts are being resolved, but instead just a bit will be flipped to make resolve-undo data out of the conflict data. Following the filename is the number of conflicts for the file. There may be up to three, but some stages may be missing, followed by the flags, modes and hashes for each stage. The flags include the stage of the conflict, and the conflicted/resolved state. The flags are the same for each conflict part, but the state of the entry is always only checked for the lowest stage entry. The conflicted entries and the resolve undo data is referenced from the path entries, so that only the filename can be saved instead of the full pathname. Each conflict entry is concluded by a crc checksum over the entry.

All offsets are 32-bit values, which allows for an index size slightly above 4GB. The index of the WebKit repositories, which is one of the largest indices of a open source repository, is currently about 26MB, and 15MB in the new index v5 format. When an index reaches close to that mark, there are a lot other code parts, that would have performance issues, that would make it impractical to work with such a huge repository.

The index entries in index v5 are tightly packed in the index. If an entry is added or removed, the whole index file will be rewritten. This tradeoff is taken to speed up full reads of the index file, which are a common operation. In addition, changes to existing files are far more common than adding files to and removing files from a repository.

The stat crc includes all stat data fields, except mtime and the file size. This compression is possible, because the fields are only used for comparison, but the actual value is never used. This saves 16 bytes per index entry. This may not seem much, but for the 180.000 files in the Webkit index it saves about 3MB on a originally 26MB index. Mtime is kept out of the stat crc, because it is needed to check if an index entry is racily clean, and the file contents need to be checked. The file size could theoretically be included in the stat crc, but is kept out for two reasons. Other Git implementations (e.g. JGit) may not have all the stat fields available, or the ctime may not be reliable on some file systems. In this case the stat crc is set to 0, and ignored when checking the heuristic. There may also be collisions in crc checksums, even if they are not likely to happen. In case they do happen, the file size is an additional security, as it's the field that is most likely to change when a file changes. Compressing the fields into a crc hash was suggested by Michael Haggerty[2].

For each file that has more stages (thus is conflicted), one file entry (the one with the lowest stage) is saved in the block of files, and in the block of conflicts. When a conflict is resolved, the flag in the conflict block can be switched and the file in the file block can be replaced with the data of the merged file.

Example 3.3. *Following is an example of the new index format, for the same repository as in the introduction. As the example for index v2, there is no cache-tree data, and no conflicts, but there is a resolve-undo data for db.helper.c. The example doesn't use network byte order so that it is human readable. The crc sums are from the actual index file, so they won't match with the contents as shown below, but they match with the contents in the index file.*

```
1  DIRC 5 3 10 254 0 9b4395fc
2  0
3  51
4  105
5  166
6   0 803 1 1 5 0 0 0 b527d637
7  db/ 20 0 0 1 3 0 0 0 b076b9fb
8  db/sample/ 32 0 0 0 2 0 0 0 310f8e8e
9  0
10 56
11 112
12 163
13 218
14 273
15 333
16 387
17 441
18 495
19 549
20 db.helper.c 0 100644 1354874688:847404995 129 8147a76d
21 76aaf436fbdb61e4a839f845af59a638a0984d7 f5658077
22 db.helper.h 0 100644 1354809315:12235636 101 cf227c50
23 50834ad8d16290f692ef07445cd6b157b3002b5c 4e5f2d56
24 main.c 0 100644 1354809936:598423356 153 4b81d37c
25 1041f31643bf94ea23293368c9c2a1e80cc69122 93f3904a
26 revenues.c 0 100644 1354810109:741572481 113 f3ca60d
27 1a99e671bcfc82f9db774baf6be9bec0df3d9b50 dc95697
28 revenues.h 0 100644 1354810062:794157341 109 ed1d3125
29 4a69d3b236a5fc54b4bbbf2985be079029d13d17 6870051f
30 dbstructure.sql 0 100644 1354821580:899517114 99 5a05e708
31 cf7cc61f340c59e4dc3906105bc5ad6db27e6f4d e0ce25bd
32 sqlite3.c 0 100644 1352507303:0 4850710 3884bb68
33 22d5dea0e0276cc64c12487b3b092ced2e1e9390 6458cc02
34 sqlite3.h 0 100644 1352507302:0 342230 12ba6371
```

[2]Michael Haggerty, mhagger@alum.mit.edu, http://softwareswirl.blogspot.it/

```
35 eb8596292d1599a5a467912824c464262b93e793 e3b30ecc
36 large.sql 0 100644 1354818695:467814725 182 1afca0ca
37 783b425b53077963dad271738eabee00ea40cf3cd 5de623dc
38 small.sql 0 100644 1354818614:282067768 56 41e7e004
39 b18c06ba820dece04dee9f027cca4a56eaeb6992 989428de
40 db.helper.c 3 2000 100644 76aaf436fbdb61e4a839f845af59a638a0984d7b
41 4000 100644 5152b85b8409db4434091e4552bb93c1b20de161 6000
42 100644 4037f599dcc0e5b3a4c7af29fba57980bbef1105 2a96427f
```

The different sections of the index file format v5 are very well visible in the above example, starting with the header in line 1, showing the signature, that the index file is version 5, there are three directories and 10 files in the repository, the first file entry starts after 254 bytes and there are zero extensions. It ends up with the crc checksum over that whole data.

The lines two to five are the offsets to the directory entries, relative to the beginning of the first directory. The offset to the first directory can be easily calculated by using the header size, and the number of offsets plus one, times 4 bytes, as each offset takes 4 bytes.

The lines six to eight are the actual directories that are in the repository. Starting with the root directory followed by the lexically sorted subdirectories. Taking the root directory as example, the name is an empty string, followed by the offset to the file offset, which is 0 for the root directory. After that follows 803 which is the offset to the single conflicted file in this git repository, followed by the number of conflicted files in the directory, which is 1. After that follow one for the number of subdirectories, five is the number of files in the directory. The following two zeros are cache-tree data, the number of entries and the object sha-1. The last 0 is the flags field, which is currently unused. b527d637 is the crc32 checksum for the entry. This information is included in the index entries and in the cache-tree extension in index v2. If the db/sample subdirectory is taken as an example, we have two files in the index v2 file, on line 8-11 in example 3.2.

Lines 9 to 19 contain the offsets to the relative to the first file entry. The offset to the first file entry is stored in the header.

Lines 20 to 39 contain the file entries. They correspond to the index entries in example 3.2 from line 2 to 21, with a few differences. The sorting is different, as in index v2 the entries are sorted lexically, while in index v5 they are sorted by directory. In addition, in index v5 the stat data is compressed to a single crc32 checksum, instead of using the full fields. In cases where there are conflicts in the index, the entries are not the same, because in index v2 conflicts are stored as index entries, while in index v5 they are stored in the block after. Each file entry takes two lines in the example.

The last three lines, 40 to 42 contain the resolve undo data, which is on line 23 to 25 in the example for index v2 (example 3.2). The format again is similar, except that in index v5 the entry is referenced from the directory block, and does not include the name of the subdirectory. Each stage also contains flags, that show the stage of the entry, and if it is conflicted or a resolved conflict. If there are conflicted entries in the index, they are added in this block.

The example above also shows the in tightly packed index entries, which make it impossible to add or remove an entry from the index file, without rewriting it. Changing a stat field, which is a more common operation on the other hand is easily possible, by just replacing the field.

Chapter 4

Implementation

4.1 Convert Index

The first step was to implement a python script, which converts an index v2 to an index v5, called git-convert-index.py. This script was written in python to get more familiar with both index formats. The choice for python was made, because it is a simple scripting language, which allows for rapid prototyping and sketching of the algorithm. In addition the struct module allows easy access to the binary data. The script also has the possibility to print part of the index, which was used during development, to compare it with the output of Git commands like '*git ls-files*', to check the correctness of the script.

By default the script reads the index from .git/index, which is the default location for the index file, and by default writes it to .git/indexv5, so that if anything goes wrong, the original index isn't overwritten. This locations can be changed by giving different locations as arguments to the script.

The git-convert-index script was only planned as a prototype, until the writing part was implemented in core Git. The index format changed after that, due to some optimizations of the format, and the prototype was not brought up to date anymore, and thus doesn't produce a valid index v5 index file when executed.

4.2 Read Index

Before implementing the reading in the Git code, a prototype was implemented in Python, both to test convert index script, and to sketch the reading algorithm. The reading algorithm is a recursive algorithm, because of the way the entries are saved in the internal memory format, as opposed to the way they are sorted in index v5.

Example 4.1. *Let's assume a repository with the files a, c, and the directory b with a file b in it. In-memory, the files are sorted a - b/b - c. In index v5 on the other hand they are sorted as a - c - b/b on disk, because they are sorted after the directory.*

Below is the pseudo code for the reading algorithm, that is used in the read index script.

```
1  function read_files(dir):
2      for(i = 0; i < dir->nfiles; i++):
3          queue.enqueue(read_file(dir->pathname))
4      while (queue):
5          if (dir->next && queue[0]->name >
6              directories->next->pathname):
7              dir = read_files(dir->next)
8          else:
9              entries->add(queue.dequeue())
10
11     return dir
12
13 read_header()
14 seek_to_first_dir()
15 dir = read_dirs()
16 seek_to_first_file()
17 read_files(dir)
```

Reading the header and reading the dirs is straight forward. First the header is read and with the number of directories, the offset to the first directory is determined and that part is skipped, because we don't need it for a full read. The directories are just read sequentially. Then the offsets for the files are skipped, because the index entries are tightly packed they are not needed for a full read.

The read_files function is a recursive function, which first reads all entries in directory into one queue and then dequeues them, while recursing into the next directory, if that comes before it. All the entries are then sequentially saved in a list. That list is then used to produce the output.

To test the read index script, it is able to read an index v5 file from any location, by default it is read from .git/indexv5, which is the default output location from the convert index script. It can display the contents of the of the file in the same way as 'git ls-files' does, so the index v2 and index v5 files are easily comparable.

4.3 Internal Implementation

The in-memory format is the way Git stores the index entries internally. The index entries are stored sequentially in an array and are sorted lexically. There is currently no api to access the index entries, but instead they are accessed directly throughout the Git source code. This makes it hard to change the in-memory format, as it is necessary to touch a large part of Git.

The first step in the implementation is to change the index file format without changing the in-memory format. Git can however benefit from the new index format even without changing the in-memory format, because of the compression. The performance difference will be shown in Chapter 6. To take full advantage of the new format however, a index api needs to be implemented, which will allow partial loading and writing.

It is necessary that the implementation is backwards compatible in order to allow existing users of Git to continue to use the old repositories and if they wish, to update them to the new index version. Currently the code that reads the index file is in a single file read-cache.c. To achieve clean code and to separate the versions cleanly, the file is split into 4 files. read-cache.h contains all the definitions for reading the index, which previously were in cache.h. cache.h is a general definitions file, which is included in virtually every

Git source file. read-cache.c contains all the general functions, that are necessary for both index format and used by Git to access the index. read-cache-v[25].c meanwhile contains all the code that only affects the respective index versions. V2 always stands for the versions 2, 3 and 4, because they are very similar and thus use mostly the same codepaths.

To read the index entries and save them in the old in-memory format, the recursive algorithm, that was shown in section 4.2 is used. In this stage of the implementation neither partial reading nor partial writing is possible, although partial reading was implemented in a side branch as a quick and dirty proof of concept. This partially read index cannot be written, and because of that is only usable in commands that only read the index, for example '*git ls-files*' or '*git grep*'.

The cache-tree data and the resolve undo data are automatically read, when reading the directories, and the conflicts. If available, they are then internally converted to their respective data structures, which are a tree for the cache-tree and a string list for the resolve undo data. The sorting of the cache-tree is interesting, because it is sorted after the length of the directory entry as first parameter and only after that sorted lexically. E.g. if there are two directories aa and z, z comes before aa in the cache-tree.

Another problem is that the index file may change while it is read. Especially when partial writing is implemented, a index entry may have changed, but its crc checksum still isn't updated. This is a very rare case, but cannot be neglected. To avoid problems, Git doesn't die when a wrong crc checksum is detected, but tries to re-read the index. As an additional check, the stat data is checked before the index is read, and again after the index is read. If the crc is wrong, the algorithm is executed a limited number of times before failing. In that case it is highly likely that the crc is really wrong and the index entry was not just overwritten without updating the crc checksum yet. The pseudo code for the algorithm can be found below.

```
1 i = 0
2 do {
3     old_stat = stat(index_file)
4     wrong_crc = read_index()
5     new_stat = stat(index_file)
6     if (i >= 50 && wrong_crc)
7         die(``index file corrupt'')
8     i++
9 } while (old_stat != new_stat)
```

Writing of the index is done in two steps. First the offsets and the directory data is calculated, and then they are written sequentially. The algorithm filters the directories, files and the conflicts out from the list of index files. Because of the sorting of the in-memory format, the directory entries can not just be found out sequentially. Another problem are directories which have no files in them. They have to be in the index, because of the cache-tree, but are not there when just using the directory for each file.

The function that compiles the directory data uses a hash-table which is implemented by Git internally. The hash table uses crc codes as hashes, and chaining to resolve hash collisions.

Below you can find the pseudo code for the compile_directory_data function.

```
1 found_dir = init_directory_entry("")
2 insert_hash(found_dir)
3
```

```
4  foreach(entry in index_entries) {
5      if (dir_is_not_the_one_used_in_the_last_loop)
6          found_dir = find_dir_in_hash_table(dir(entry))
7      if (!found_dir) {
8          fount_dir = init_directory_entry(dir(entry))
9          insert_hash(found_dir)
10     }
11     // A new entry is either the stage 0 entry, or the
12     // lowest stage entry if there is a conflict
13     if (new_entry(entry)) {
14         add_entry_to_dir(found_dir)
15     }
16     if (stage(entry) > 0) {
17         if (new_entry(entry))
18             conflict_entry = create_new_conflict_entry(entry)
19         add_conflict_part_to_conflict_entry(conflict_entry)
20     }
21     if (new_dir(found_dir)) {
22         add_all_super_directories_to_hash_table(found_dir)
23     }
24     if (cache_tree)
25         convert_cache_tree()
26     convert_resolve_undo
27 }
```

struct index_state is the internal struct that stores the index entries, the cache-tree, the resolve undo data and other index related information.

To split the index-specific functions into their specific files, a index_ops struct is added to struct index_state. The index_ops can be either assigned the v2_ops or the v5_ops. The respective function in read-cache.c then acts as a wrapper function for format specific function. The index_ops are assigned after checking the header, and may be changed after reading the index, to write the index in another format. Using the index_ops was first suggested by Nguyen Thai Ngoc Duy. [1]

```
1  struct index_ops {
2      int (*match_stat_basic)(struct cache_entry *ce, struct stat *st, int changed);
3      int (*verify_hdr)(void *mmap, unsigned long size);
4      int (*read_index)(struct index_state *istate);
5      int (*write_index)(struct index_state *istate, int newfd);
6  };
7
8  int read_index(struct index_state *istate) {
9      return istate->ops->read_index(istate);
10 }
```

4.4 New In-memory Format

To take full advantage of the new index file format, a new, tree-structured in-memory format is needed. As with the reading algorithm, the in-memory format is different for index v2 and index v5. While index v2 is more efficient as a list, index v5 is already structured as a tree on-disk, and is more efficient when implemented as tree-structure in-memory. The data-structure, that saves the index entries in-memory is shown in the listing below.

```
1  struct dir_tree_entry {
2      int level_read;
3      struct dir_tree_entry **down;
```

[1] Nguyen Thai Ngoc Duy, pclouds@gmail.com

```
4     struct cache_entry **ce;
5     struct conflict_entry **conflict;
6     unsigned int conflict_size;
7     unsigned int de_foffset;
8     unsigned int de_cr;
9     unsigned int de_ncr;
10    unsigned int de_nsubtrees;
11    unsigned int de_nfiles;
12    unsigned int de_nentries;
13    unsigned char sha1[20];
14    unsigned short de_flags;
15    unsigned int de_pathlen;
16    char pathname[FLEX_ARRAY];
17 };
```

The level_read variable is used to check if the index entries and conflicts in a directory have been read, or if only the the directory has been read. Only if the level has been read, index entries and conflicts can be added to the directory.

The down array is used to store sub-directories of the current directory, and the ce and conflict arrays to store the index entries and conflicts in the directory. The conflicts include the resolve-undo data, because they logically belong together in index v5. The rest of the dir_tree_struct is the directory data.

For the partial writing, another variable will be added, to indicate if the whole tree has to be rewritten, or if only a specific directory has some changes.

4.5 Index Reading API

The new in-memory format is not directly wired into the Git source code, but instead accessed through a new index api. This facilitates both the implementation and future changes to the api or the in-memory format. With the details being hidden behind an api, the two in-memory formats can coexist and the advantages of index v5 can be enabled without having any disadvantage when still using index v2. In addition the calls to the api produce a cleaner and more meaningful code. The index is currently a functional proof of concept, and will be implemented further in the future.

Similar to the index_ops, used to read the different versions of the index, the index api uses internal_ops, which are set to the version of the in-memory format.

The api consists of the following functions:

`cache_open_from(path)`: Open the index from a specific path, and to the basic checking for index corruption, and the index version. This function only has to be called if the index is at a location other than the default location. For the default location, the index is opened automatically.

`read_cache_full()`: This is a compatibility function for the old in-memory format. It reads the whole index and saves it to the flat in-memory format, regardless of the on-disk version.

`read_cache_filtered(opts)`: Read the index filtered by the opts. This function is used to read the cache-tree and the resolve undo information.

`for_each_cache_entry(fn, cb_data)`: Calls the function given by the fn parameter for each index entry. Arguments can be given to the function via the cb_data parameter, which is a void* pointer.

`for_each_cache_entry_filtered(opts, fn, cb_data)`: Calls the function given by the fn parameter for the index entries filtered by the filter_ops. Arguments can be given to the fn function via the cb_data parameter, as for the for_each_cache_entry function.

`get_cache_entry_by_name(name, namelen, opts)`: Get a specific cache entry, with the name given as parameter. The opts are for filtering what part of the index will be read.

`get_cache_entry_stage(name, namelen, stage, opts)`: Get a specific cache entry with the stage given as parameter. If a merge is in progress, the cache entry with the given state is returned. The opts are again for filtering the part of the index that will be read.

The filter_opts are shown in the listing below:

```
1  struct filter_opts {
2      const char **pathspec;
3      char *seen;
4      char *max_prefix;
5      int max_prefix_len;
6
7      int read_staged;
8      int read_cache_tree;
9      int read_resolve_undo;
10 };
```

The pathspec is the regular expression, by which the index entries should be filtered. seen, max_prefix and max_prefix_len are options for the internal match_pathspec function, which is used to match the path read with the given pathspec.

If the read_staged variable is set, the conflicts are read. Similar to that are the read_cache_tree and the read_resolve_undo variables, which determine if the cache-tree data or the resolve undo data are read.

4.6 Writing the New Index Format

The current writing process is designed to write the index sequentially, without any tools to replace an index entry, or write only a part of the index.

The lock file is used to make sure operations on the index are not intercepted by any other command, and that they don't fail, if the computer crashes while writing the index file. With the current system, the new index is written to a lock file .git/index.lock, which then is moved to .git/index, when the index is fully written.

The lock file structure has to be changed for index v5 to allow partial writing. It will be still the same when the index has to be fully rewritten (whenever a file is moved, added or deleted). When replacing one or more index entries, the changed entries will first be written to the lock file, and then replaced in the actual index file. If the computer crashes during the write, Git can recover by reading the lock-file, and trying to replace the index entries in the index file again.

It is determined by the internal code if the index has to be fully rewritten, or if only a few entries can be replaced. A variable will be added to the struct dir_tree_entry, which indicates either which part has to be rewritten, or that a full rewrite has to be done.

Even though this is not implemented yet in the Git source code, a proof of concept exists as a modified read-index script. This was used to test the re-reading of the index file, in case a crc checksum is wrong. It's done by saving the offset relative to the beginning of the file when reading the index file, and then replacing it an entry with random stat data for testing purposes.

Chapter 5

Discussion

5.1 Alternative Solutions

The “contiguous tree” format was not the only format that was discussed in the process of finding the best index file format for git. The formats will be discussed briefly below, without going into detail. The details are spared, because the formats were only discussed briefly and then discarded.

5.1.1 B-tree Format

The B-tree format would allow changes to the index and reading a index partially in O(log(n)) time, with n being the number of entries in the index.

To avoid the corruption of the index, and thus erroneous data in the repository, each entry will have a crc32 checksum. Only the parts which are really read will have to be checked, and when writing only the checksums of a particular part of the tree will be recalculated.

To compress the index, the path would only be stored once for each directory. This will make the index smaller by removing redundant data.

This solution is very similar to the ”contiguous tree” format and would even be faster in the case of adding or deleting a file. The ”contiguous tree” will however have faster reading time, since the format is smaller. The B-tree format would also be more complex, and thus increase the probability of errors.

5.1.2 Append-only Data Structure

An append-only data structure will allow for changes to the index in O(k) time, with k being the number of files that changed. To reach this goal, the first part of the index will be sorted and changes will always be written to the end, in the order in which the changes occurred. This last part of the index will be merged with the rest using a heuristic, which will be the execution of *git commit* and *git status*.

To make sure the index isn't corrupted, without calculating the sha1 hash for the whole index file every time something is changed, the hash is always calculated for the whole index when merging, but when only a single entry is changed the sha-1 hash is only calculated for the last change. This will increase the cost for reading the index to log(n) + k * log(n) where n is the number of entries in the sorted part of the index and k is the number of entries in the unsorted part of the index, which will have to be merged with the rest of the index.

The index format shall also save on file size, by storing the paths only once, which currently are stored for every file. This can make a big difference especially for repositories with deep directory structures. (e.g. in the webkit repository this change can save about 10MB on the index). In addition to that it can also give the index a structure, which in turn makes it easier to search through. Except for the appended part, but that should never grow big enough to make a linear search through it costly.

This idea was dropped, because as mentioned in the problem description reads are more common then writes and therefore trading write speed for read speed is not a good trade-off. The slowdown depends mostly on the size of the appended (unsorted) part, which in most cases will be much less then the size of the sorted part, but in other cases might grow quite big. It is preferable to have a structure that is usually faster on the read operation and where the read speed will not depend on the way the user works. This is worth the slight trade-off in write speed. In addition the append only structure doesn't allow simple partial reading of the index, which is another problem.

5.1.3 Database Format

A database as index file will allow for changes to the index in O(log(n)) time for a single change, with n always being the number of entries in the index, assuming a b-tree index on the path in the database.

The check for the index corruption could be left to the database, which would have about the same cost as the check in the tree-structure, in the read and write direction. Partial loading with the right indices would also have the same cost as the tree-structure, by executing a simple select query.

The drawback of this solution however would be the introduction of a database library. Another drawback would be that it's harder to read for programs like libgit2 and JGit and the codebase wouldn't get any cleaner.

5.1.4 Padded Structure

Another index format that was taken into consideration was to create a padded structure. Basically that would be an append-only like structure, but split into sections, where every section would leave some space at the end for appending changes. This would leave us with a complexity of O(k) where k is the number of sections.

This structure will bring advantages for faster partial loading, when splitting the sections in the right way.

The structure was however only briefly taken into consideration, since it would have close to no advantages (except for the partial loading) compared to the append-only structure, would make the index file larger due to the spaces for appending and have the same drawbacks as the append-only structure.

5.2 Discussion

The alternative formats described in the previous section were all discussed with the Git community, on both their mailing list[1] and on IRC [2]. The choice for the "contiguous tree" format was taken with the help of the community.

The choice was taken taking into account different factors. The new format should be at least as fast as the old one when doing a full read or rewrite. Reading operations are more common than writing operations in Git. In addition most operations do only change files, but don't add or remove any. With those factors in mind, it was decided, that the "contiguous tree" format was the best fit for being the new index file format.

[1] Archives at http://thread.gmane.org/gmane.comp.version-control.git/
[2] #git-devel on irc.freenode.net

Chapter 6

Experiments

6.1 File Size

The figures below show the size of the index file of the Git[1] repository (fig. 6.1a), the Linux kernel[2] repository (fig. 6.1c) and the Webkit[3] repository (fig. 6.1b) over time, starting from their first commit to one of the most recent commits. Generally more files are added to each repository over time, which is why their index files grow over time, and will grow further in the future.

In terms of file size, index v5 is nearly always superior to index v2, with the exception of very small repositories without any extension data. The comparison with index v4 depends on the structure of the repository. If a repository has a deep directory structure, and thus a small number of files per directory, index v4 is slightly superior to index v5 in terms of file size. If the repository has a larger number of files per directory, as it is the case with the Git and the Linux kernel repository, on the other hand, index v5 is superior to index v4. Each prefix uses more space in index v5, because it contains more information on the directory, the cache-tree data and the crc checksum. The stat-data compression makes up for that, and the more files there are, the more effective it is.

The percentual gain on the file size is better when there are deep directory structures, because the prefix compression is reduces the index size more than the stat-data comparison.

Example 6.1. *The index files sizes for a recent commit for the Webkit index are (27 files/directory)*

```
1     29062kb index v2 (100%)
2     16734kb index v4 (57.6%)
3     17120kb index v5 (58.9%)
4
```

As comparison the index file sizes for a recent commit in the Linux repository (16 files/directory):

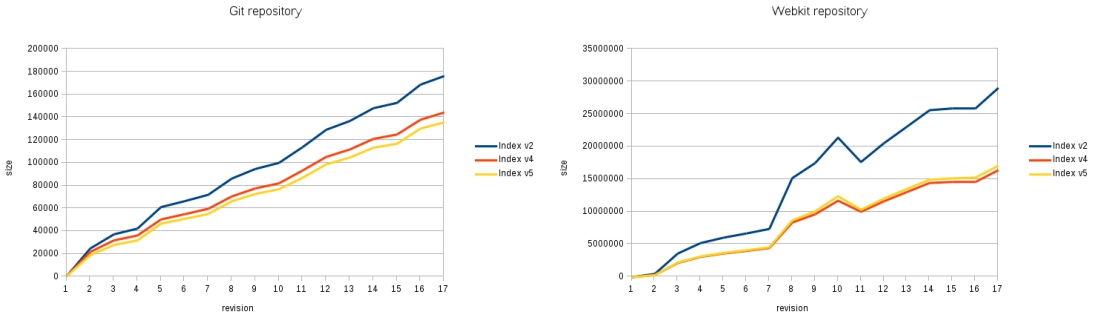
[1] http://git-scm.com
[2] http://kernel.org
[3] http://webkit.org

```
    4063kb index v2 (100%)
    2976kb index v4 (73.2%)
    2682kb index v5 (66.0%)

```

To conclude the comparison the index file sizes for a recent commit in the git repository (8 files/directory):

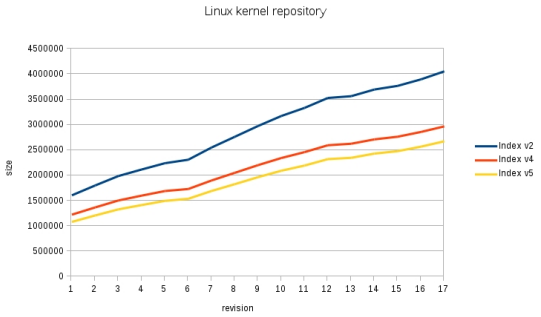
```
    176kb index v2 (100%)
    144kb index v4 (81.8%)
    135kb index v5 (76.7%)

```

The file sizes in the below graphs are all without any extensions. If the cache-tree extension is included in the index file, the results for index v5 are even better, because its file size is the same with or without the cache-tree because the cache-tree data is no longer an optional extension, but always present in the index. In index v4 and index v5 the cache-tree data is optional, and thus their size grows when the data is present.



(A) Git repository

(B) Webkit repository



(C) Linux kernel repositories

FIGURE 6.1: The size of different repositories over time

6.2 Runtime Performance

The reason for index v5 is speeding up reading and writing. The writing part is not yet fully implemented, but partial reading of the index already works. Index v5 also provides faster speed for full reads and writes of the index.

The test repository is the WebKit repository.

```
Test                                                  this tree
-----------------------------------------------------------------
0002.2: v[23]: update-index                           0.21(0.16+0.03)
0002.3: v[23]: grep nonexistent -- subdir             0.10(0.07+0.02)
0002.4: v[23]: ls-files -- subdir                     0.10(0.08+0.01)
0002.6: v4: update-index                              0.18(0.14+0.03)
0002.7: v4: grep nonexistent -- subdir                0.08(0.05+0.02)
0002.8: v4: ls-files -- subdir                        0.08(0.05+0.02)
0002.10: v5: update-index                             0.14(0.11+0.02)
0002.11: v5: grep nonexistent -- subdir               0.01(0.00+0.00)
0002.12: v5: ls-files -- subdir                       0.00(0.00+0.00)
```

Git includes a performance testing framework, which allows to execute different commands and test their performance. The above output comes from this framework, testing various commands with both the old index format and the new one. The tests were done best of 10, to account for variables while executing.

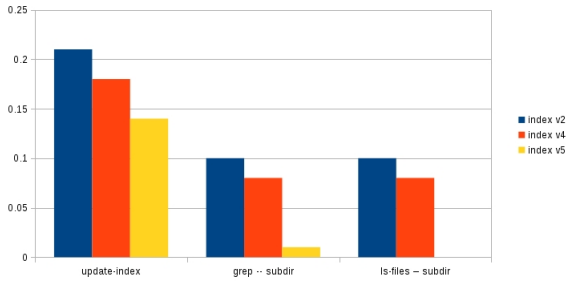


FIGURE 6.2: The above timings shown graphically

The update-index command reads the full index, and rewrites it. It is used to test the read-write performance for the full index. Even though index v5 is slightly bigger in size than index v4 for the WebKit repository it is faster because the offsets do not have to be read.

The grep command searches for for some phrase in a subdirectory, which contains only a few files. Because grepping is very fast for the few files, this mainly tests the index. In both index v2 and index v4, the whole index has to be read, while in index v5 only the part of the index that is really needed is read. This accounts for the improvement from 0.10s to 0.01s.

The ls-files command is basically the same as grep command in terms of reading of the index. It shows a list of files in the specific subdirectory. Because it does not need any time for the grepping, it takes virtually no time execute the command in a subdirectory of the WebKit repository.

The times below are from a sample repository with a 500MB index. I could not find a real world repository this big, but this works well for demonstration purposes.

This special repository allows to show the speed difference in a better way. The test is again best of ten, with the ls-files and grep commands showing improvement from 1.7 seconds to 80 milliseconds with the use of index v5 and partial reading of the index. Note that this test repository is on a external hard disk, which may contribute to the times.

```
Test                                                  this tree
-----------------------------------------------------------------
0002.2: v[23]: update-index                           11.68(2.77+0.95)
0002.3: v[23]: grep nonexistent -- subdir             1.76(1.29+0.37)
0002.4: v[23]: ls-files -- subdir                     1.62(1.18+0.35)
0002.6: v4: update-index                              6.34(2.44+0.58)
0002.7: v4: grep nonexistent -- subdir                1.45(1.03+0.33)
0002.8: v4: ls-files -- subdir                        1.32(0.89+0.33)
0002.10: v5: update-index                             6.45(2.30+0.73)
0002.11: v5: grep nonexistent -- subdir               0.07(0.05+0.01)
0002.12: v5: ls-files -- subdir                       0.08(0.06+0.01)
```

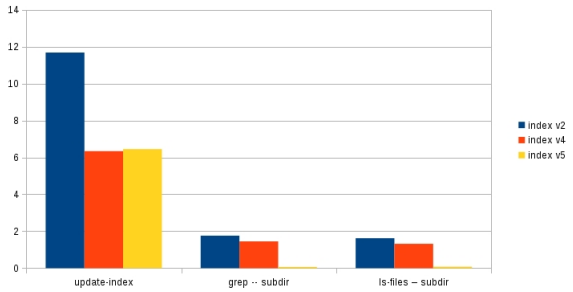


FIGURE 6.3: The timings from the special repository for testing purpose

Chapter 7

Conclusion and Future Work

The new index file format (v5) is superior to the previous index file formats in any aspect, the reading and writing times for the whole index, and especially partial reading of the index, where the main strength of the format is. With the expected growth of repositories in terms of number of files, this advantage will be even bigger.

The full advantage of the new file format will be really noticeable once its the new API is used throughout the Git source code, which will take some time, because the index file is involved in most operations in Git. Even before that, Git benefits from the new index file format, even though to a lesser degree, as shown in Chapter 6.

The compatibility mode for the new index file format is fully implemented and passes the Git test suite without any error and even provides a slight advantage over the old index formats.

The API on the other hand exists as functional proof of concept. It provides a significant advantage for partial loading of the index file, as shown in the Experiments in Chapter 6. The '*git ls-files*' and '*git grep*' commands already use the API and take full advantage of it.

This work will be continued by implementing the use of the index API and thus the new in-memory format throughout the Git source code. This will allow Git to take full advantage of the new index file format. In addition to that a new writing API can be implemented, which will enable partial writing, further enhancing the performance.

Bibliography

[1] William Nagel. *Subversion Version Control*. Prentice Hall, 2005.

[2] Scott Chacon. *Pro Git*. Apress, 2009.

[3] Neil McAllister. Linus torvalds' bitkeeper blunder, May 2005. URL `http://www.infoworld.com/t/platforms/linus-torvalds-bitkeeper-blunder-905`.

[4] Linus Torvalds, February 2007. URL `http://marc.info/?l=git&m=117254154130732`.

[5] Linus Torvalds, April 2005. URL `http://marc.info/?l=linux-kernel&m=111280216717070`.

[6] Linus Torvalds, April .

[7] Junio C Hamano. Use of index and racy git problem, 2006. git-racy.txt in Documentation/technical in the git repository.

[8] Junio C Hamano, Nguyen Thai Ngoc Duy, and Carlos Martin Nieto. Git index format, 2010. index-format.txt in Documentation/technical in the git repository.