Bloc: Library for handling large binary objects in Haskell

A thesis submitted
in Partial Fulfillment of the Requirements
for the Degree of

Master of Technology

by

Anshu Avinash

to the
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
June, 2015

CERTIFICATE

It is certified that the work contained in the thesis titled **Bloc: Library for handling large binary objects in Haskell**, by **Anshu Avinash**, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Piyush Kurur

Department of Computer Science & Engineering

IIT Kanpur

June, 2015

ABSTRACT

Name of student: **Anshu Avinash** Roll no: **10327122**

Degree for which submitted: **Master of Technology**

Department: **Computer Science & Engineering**

Thesis title: **Bloc: Library for handling large binary objects in Haskell**

Name of Thesis Supervisor: **Piyush Kurur**

Month and year of thesis submission: **June, 2015**

In this thesis, we describe a library for handling large binary objects (blob) written in Haskell - a purely-functional programming language. In our design, each blob is stored in a separate file. Lock free concurrent access to blobs is provided by making use of certain atomic operations supported by the underlying file system. We leverage many features offered by Haskell like modularity and strong type system.

Dedicated to the Haskell community

Acknowledgements

This thesis would have been impossible without the guidance from my thesis advisor, Piyush Kurur. He came up with the topic of this thesis, taking into consideration my interest in Functional programming and systems. He was always accessible for discussions, sometimes not necessarily related to the thesis. I would like to express my sincere gratitude towards him.

I would also like to thank the Haskell community (#haskell irc channel, /r/haskell on reddit) for making the journey of understanding Haskell pleasant.

I am thankful to the Department of Computer Science and Engineering, IIT Kanpur, for providing the necessary infrastructure for my research work.

My family and friends always supported me throughout my thesis work - special thanks to them.

Contents

List of Tables

List of Figures

List of Programs

Chapter 1

Introduction

Most of the web applications today require to store some kind of data persistently. For a web application that handles student management - the data can be name, date of birth, photograph and other information about students. These web applications use one of the *databases* to store their data. Database refers to a collection of information that exists overs a long period and a Database Management System (DBMS) is a tool for creating and managing large amount of data efficiently.

Early database management systems evolved from file systems. These database systems used tree-based and the graph-based models for describing the structure of the information in a database. Edgar F. Codd in his seminal paper [1] proposed that database systems should present the user with a view of data organized as tables called relations, setting the foundation for popular relational databases like MySQL and PostgreSQL.

Many of the web applications do not require the complex querying and management functionality offered by a Relational Database Management System (RDBMS). This among other reasons gave rise to several NoSQL (Not only SQL) databases which can be classified based on the data models used by them. Amazon's Dynamo [2] is a key-value store in which records are stored and retrieved using a key that uniquely identifies the record. MongoDB [3] on the other hand is a document-oriented database and is designed for managing semi-structured data. These NoSQL databases offer an important benefit of scalability and availability by sacrificing strong consistency

guarantees offered by RDBMS.

Today's web applications also work with large files like images, music, videos etc. Size of these files can vary from few MBs to tens of GBs. The application developer can decide to store these files directly into the one of the databases mentioned above or store it as a file and save the filename in the database. This large binary data is usually called a blob (**B**inary **L**arge **O**bject).

Concurrency plays an important part in building scalable and fault tolerant web applications. However, building concurrent systems usually requires working with locks and may result into issues like deadlock and starvation.

In this thesis, we provide a library written in *Haskell* for handling blobs. The library provides methods for incremental writing, incremental reading and garbage collection of deleted blobs. By making use of the atomic guarantees provided by the file system on certain file operations, we provide lock free concurrent access to blobs.

The name of our library "Bloc" stands for **B**inary **L**arge **O**bjects with **C**oncurrency, since it deals with blobs and provides support for concurrent operations.

1.1 Organization of the thesis

Chapter 2 discusses the approach of storing large files in databases. It also provides a background for this thesis work. In Chapter 3, we give a brief introduction to functional programming. Chapter 4 describes our design and implementation. We conclude and present the future work in Chapter 5.

Chapter 2

Related Work and Background

2.1 Storing large objects

There are two options for storing large objects of a database - storing the entire large object in the database or storing the path to the binary file corresponding to the large object in the database. In this section we will discuss few examples of both. We will also discuss merits and demerits of both the approaches.

2.1.1 Storing large objects in database

Exodus was one of the first databases to support storage of large object files [4]. It used B+ tree index on byte position within the object plus a collection of leaf (data) blocks. Exodus allowed searching for a range of bytes, inserting a sequence of bytes at a given point in the object, appending a sequence of bytes at the end of the object and to delete a sequence of bytes from a given point in the object.

Popular relational databases like MySQL and PostgreSQL both provide data types to store large object files - BLOB in MySQL and BYTEA in PostgreSQL. PostgreSQL also provides a BLOB data type which is quite different from MySQL's BLOB data type. It's implementation breaks large objects up into "chunks" and stores the chunks in rows in the database. A B-tree index guarantees fast searches for the correct chunk number when doing random access reads and writes.

A similar idea is used by MongoDB, which is a document database. It also divides the large object into “chunks”. It uses GridFS specification for this [5] which works by storing the information about the file (called metadata) in the “files” collection. The data itself is broken down into pieces called chunks that are stored in the “chunks” collection.

2.1.2 Storing metadata and filename in database

Another approach to store large objects is to store only the filename and some metadata in the database. In this case the application has to take care of the all externally attached files as well as the other settings specific to the large objects.

2.1.3 Comparison of both approaches

Both the approaches have their own benefits and disadvantages.

- **Performance**

  When we just store the filename in database, we skip the database layer altogether during file read and write operations. In the paper To BLOB or Not To BLOB [6], performance of SQL Server and NTFS has been compared. The results showed that the database gave higher throughputs for objects for relatively small size ($<$ 1MB).

- **Security**

  Security and access controls are simplified when the data is directly stored in the database. When accessing the files directly, security settings between file system and database are independent from each other.

- **ACID guarantees**

  ACID stands for Atomicity, Consistency, Isolation and Durability. All RDBMS give ACID guarantee on database transactions. If the large object is directly stored in a relational database, all the operations on it also offer ACID guar-

antees. If the large object is stored as a separate file, you don't get any such guarantee.

2.2 Concurrency

A program is said to be concurrent if it allows multiple threads of control. Concurrency is concerned with nondeterministic composition of programs (or their components).

In many of the applications today, concurrency is a necessity. For example, a server-side application needs concurrency in order to manage multiple client interactions simultaneously.

2.2.1 Concurrent Programming Models

There are multiple concurrent programming models like: actors, shared memory and transactions. The traditional way of offering concurrency in a programming language is by using threads which operate on a shared memory. In this model, *locks* are used to achieve mutual exclusion. However, there are several problems with using locks [7]:

- Taking too few locks - One might forget to take a lock, resulting into two variables modifying same variable simultaneously.
- Taking too many locks - Taking too many locks might result into a deadlock or avoid concurrency altogether.
- Taking locks in the wrong order - Acquiring locks in the wrong order can result into a deadlock.
- Error recovery - It is difficult to guarantee that no error can leave the system in an inconsistent state.

The other two approaches try to avoid this problem of using locks. Actor model treats "actors" as the primitives of concurrent computation [8]. It provides

an approach to concurrency that is entirely based on passing messages between processes. In Erlang, which is used to build scalable real-time systems, actors are part of language itself [9].

Software Transactional Memory (STM) is a technique for simplifying concurrent programming by allowing multiple state-changing operations to be grouped together and performed as a single atomic operation.

Haskell supports all the above three concurrent programming models. However, in this thesis we try to achieve concurrency by having a design which uses atomic file operations provided by the underlying file system.

2.2.2 Atomicity of file operations

In this section, we talk about atomicity of operations on file systems which provide a POSIX-file system based view.

We use the `rename` system call [10] for moving a file to a new destination. `rename` is atomic in the sense that if it is used to overwrite a file, it is atomically replaced, so that there is no point at which another process attempting to access the file will find it missing. Also, at any time, a process will not be able to read any "partial" file. This atomic property of `rename` is used by *maildir* [11] to maintain message integrity as messages are added.

Another system call `mkdir`, which is used to create a new directory, is also atomic [12]. So, if two processes try to create the same directory simultaneously, only one of them will succeed and other will receive the "EEXIST: path name already exists" error.

Chapter 3

Functional Programming

In functional programming the fundamental operation is the application of functions to arguments. The main program itself is written as a function that receives the program's input as its arguments and delivers the program's output as its result [13]. In this section we will discuss special characteristics and advantages of functional programming (Haskell in particular).

3.1 Referential Transparency

Functional programs contain no assignment statements, so variables, once given a value, never change. A function call can have no effect other than to compute its result. This eliminates a major source of bugs, and also makes the order of execution irrelevant - since no side effect can change an expression's value, it can be evaluated at any time. Since expressions can be evaluated at any time, one can freely replace variables by their values and vice versa - that is, programs are "referentially transparent".

3.2 Statically typed

Every expression in Haskell has a type which is determined at compile time. All the types composed together by function application should match up. If they don't,

the program will be rejected by the compiler.

In Haskell, we don't have to explicitly write out every type. Types are inferred by unifying every type bidirectionally.

3.3 Algebraic Data Types and Pattern Matching

An algebraic data type has one or more data constructors, and each data constructor can have zero or more arguments. These algebraic data types can be recursive too. We can define functions on algebraic data types using pattern matching. In pattern matching, we attempt to match values against patterns and, if so desired, bind variables to successful matches.

Program 3.1 Pattern matching on algebraic data types

```
data Shape = Rectangle Int Int
           | Square Int

area :: Shape -> Int
area (Rectangle len breadth) = len * breadth
area (Square side)           = side * side

rec = Rectangle 3 4

main = print $ area rec
```

3.4 Lazy Evaluation

In lazy evaluation, an expression is not evaluated until its value is needed. This implies that programs can compose very well [13]. Laziness also allows us to construct infinite data structures. Consider this example of generating primes.

The `primes` method generates an infinite list of primes lazily. In the main method, we take first 10 primes from the list and print them. On running, the above program generates the correct output: `[2,3,5,7,11,13,17,19,23,29]`.

Program 3.2 Program to generate list of primes

```
primes = filterPrime [2..]
  where filterPrime (p:xs) =
          p : filterPrime [x | x <- xs, x `mod` p /= 0]

main = print $ take 10 primes
```

3.5 Foreign Function Interface (FFI)

The Foreign Function Interface allows Haskell programs to call functions written in other languages. Consider an example of calling the power function ("pow") of the libc.

Program 3.3 Calling C's pow function from Haskell

```
foreign import ccall "pow" c_pow :: Double -> Double -> Double

main = print $ c_pow 3.0 4.0
```

When executed, this program calls the pow function of libc and prints the result 81.0. FFI enables us to call system calls like `fsync` from Haskell.

Chapter 4

Design and Implementation

Our design is inspired from the maildir format [11] which stores each message in a separate file with a unique name. The mail user agent (MUA) does not have to worry about partially delivered mail: each message is safely written to disk in the *tmp* subdirectory before it is moved to *new*. When a mail user agent process finds message in the *new* subdirectory, it moves them to *curr*.

Similar to maildir, we store all blobs in separate files. All the blobs of an application are stored under a single directory which is called a "BlobStore". Our library provides three operations on blobs - incremental writes, incremental reads and garbage collection (GC). We need an incremental interface for writing and reading as a blob might not fit entirely into the computer's main memory. Instead of providing a method for deleting the blob directly, we provide an interface for GC of all the deleted blobs, since the same blob might be shared by multiple records of a database.

In this chapter, we discuss implementation of these three operations. We also discuss how our implementation provides a lock free concurrent access on blobs.

4.1 Initializing the BlobStore

All the new blobs of a BlobStore are created inside a subdirectory named *tmp*. After doing a series of incremental writes on a newly created blob, it is moved to another subdirectoy named *curr*. Hence, before starting any operations in a BlobStore, we

ensure that the *tmp* and *curr* subdirectories have already been created. We provide a method `openBlobStore` which takes the path of a directory which is to be used as BlobStore as argument and does the initialization for us. BlobStore is defined as a Haskell newtype. We don't expose the constructor for BlobStore, so the only way to get a BlobStore is by using the `openBlobStore` method.

Program 4.1 Definition of BlobStore

```
newtype BlobStore = BlobStore FilePath
```

The `openBlobStore` method also creates a file named "metadata" in the *curr* directory. Currently this file just contains the version number of our library. Creation on this file ensures that the *curr* directory is never empty.

We use Haskell's `createDirectory` method to create *tmp* and *curr*, which internally uses `mkdir` system call. The POSIX specification of `mkdir` [12] says that if the directory to be created by `mkdir` already exists, `mkdir` should fail with EEXIST error. This guarantee makes concurrent calls to `openBlobStore` safe. If two processes try to call `openBlobStore` at the same time - only one of them will be able to create *tmp* directory and similarly only one of them will be able to create the *curr* directory. This ensures that any data present in *tmp* or *curr* is never lost by a call to `openBlobStore`.

The `openBlobStore` method also does the recovery from a failed garbage collection. This is explained in more detail in the section 4.6.5.

4.2 Creating a new blob

We provide a method called `newBlob` for creating a new blob. It takes a BlobStore as a parameter and returns a value of type `WriteContext`.

WriteContext contains the file handle of just created blob, a value of type `TempLocation` and a hash context. The `TempLocation` type stores the base directory and the filename of the newly created blob. The hash context is used to store

Program 4.2 Definition of WriteContext

```
data TempLocation = TempLocation FilePath FilePath

data WriteContext = WriteContext { writeLoc    :: TempLocation
                                 , writeHandle :: Handle
                                 , hashCtx     :: Ctx
                                 }
```

the SHA-512 hash of the contents that has been written to the blob during the incremental writes.

All the new blobs are created in the *tmp* directory. We use Version 4 UUID [14] to give unique names to the newly created blobs. Creation of new blobs is concurrent in the sense that if two processes call `newBlob` on a same BlobStore, both will be able to create new blobs successfully if the *tmp* directory is always present. We had ensured the presence of *tmp* directory in the `openBlobStore` method. Also, *tmp* directory is never deleted by any of our operations.

4.3 Writing to a blob

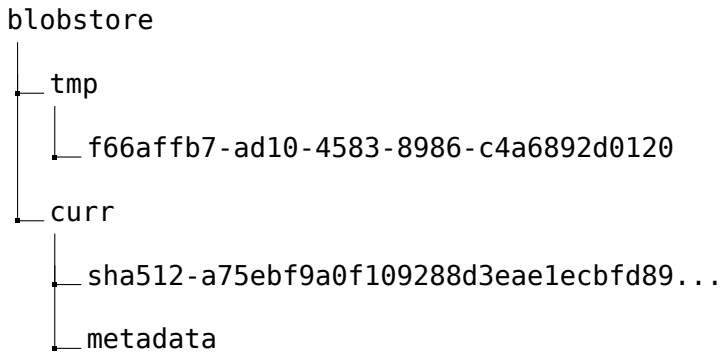
We only allow to add new data at the end of a given blob. The function `writePartial` takes the new data and a value of type `WriteContext` as arguments and appends the new data to the blob.

Once all the data has been written to a blob, we call the `endWrite` method. The `endWrite` method takes a value of type `WriteContext` as argument and moves the blob from *tmp* directory to *curr* directory. Before moving the blob, we ensure that all the data has been written to disk by calling the `fdatasync` system call. We also call `fdatasync` on the *curr* directory to ensure that even in a case of crash, the *curr* directory contains the entry of the moved file.

In the `endWrite` method, we also rename the file to SHA-512 hash of its contents. Using hash of the contents as filename ensures that if multiple blobs have same contents, only one copy is stored. This also provides the benefit of verifying the file

content at a later time - compute the SHA-512 hash of the contents and if does not match with the file name, data has been corrupted. We prefix the filename with the name of the hash function i.e. “sha512”. This is for future compatibility, in case we want to support other hash functions.

Figure 4.1: Directory structure of a BlobStore



The `endWrite` method returns a value of type `BlobId` which contains the location of the blob. No more updates to the blob are possible after calling `endWrite`.

Program 4.3 Definition of BlobId

```
data BlobId = BlobId FilePath FilePath
```

In the `endWrite` method, we move the blob to *curr* directory using the `rename` system call, which is atomic [10]. Before calling `rename`, we have to ensure that the *curr* directory is present. We had created the *curr* directory in the call to `openBlobStore`. However, the `startGC` method described in 4.6.1 renames the *curr* directory to *gc* directory and then creates an empty *curr* directory. During this short interval of renaming *curr* to *gc* and creation of new *curr*, all calls to `rename` in `endWrite` will fail. We solve this problem by taking the optimistic approach - we retry calling `rename` in `endWrite`. The maximum number of such retries can be specified by the user.

4.4 Reading from blob

Similar to writes, reading of a blob is also incremental. First the `startRead` method is called which takes a value of type `BlobId` as argument and returns a value of type `ReadContext`. The `ReadContext` type contains the file handle of the blob which is opened in read mode.

Program 4.4 Definition of ReadContext

```
data ReadContext = ReadContext Handle
```

The `readPartial` method takes a value of type `ReadContext` and number of bytes as input and returns those number of bytes from the blob. While reading from a blob, you can skip ahead using the method `skipBytes`. The method `skipBytes` takes a value of type `ReadContext` and number of bytes, b as input and skips b bytes ahead in the blob. The `endRead` method closes the handle which was opened by `startRead`.

Usually the blob to be read is present in the *curr* directory. However, during a GC, the blob might either be present in *curr* or in *gc* (Refer section 4.6 for details on GC). The `startRead` method has to do three lookups in the worst case. It first tries to read from the `curr` directory. At this time, if the blob is not found in `curr` directory, it might be in the `gc` directory. However the process which is running GC might move the blob back to `curr` before `startRead` reads from GC. The final lookup happens in the `curr` directory.

4.5 Helper methods for small blobs

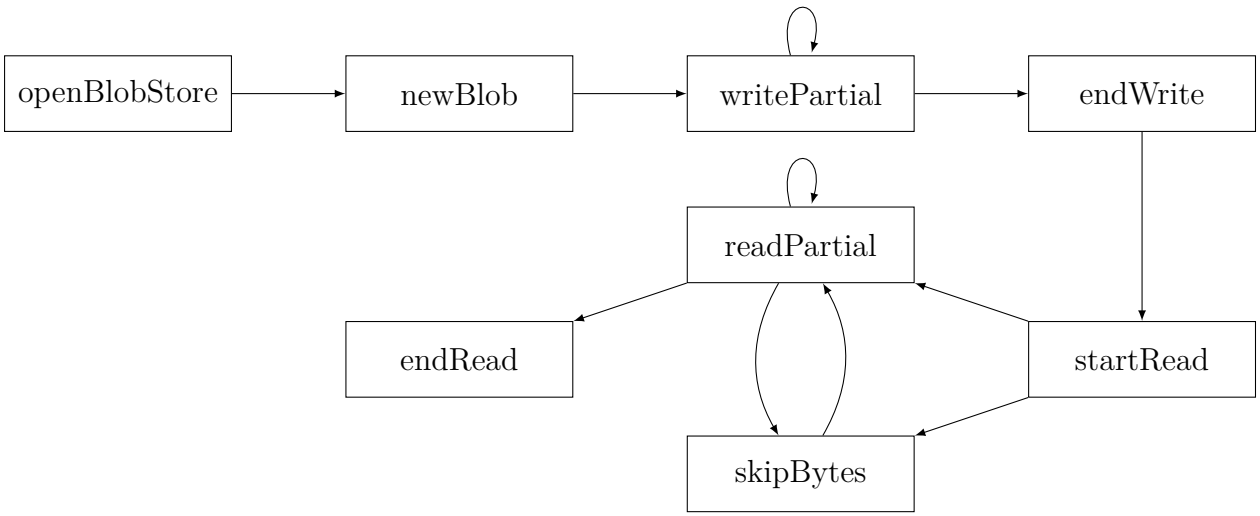
Blobs which can easily fit into the main memory, do not require the incremental write and read methods. For such blobs, we provide helper methods `createBlob` and `readBlob`.

The `createBlob` method takes a BlobStore and the blob contents as arguments

Table 4.1: Interface for operations on blob

Method	Purpose
`openBlobStore`	Initializes given directory to be used as a BlobStore
`newBlob`	Creates a blob in the given BlobStore
`writePartial`	Takes data as input and appends it at the end of the blob given in the argument
`endWrite`	Completes the write on the given blob
`startRead`	Initializes the read of the given blob
`readPartial`	Reads a given number of bytes from the given blob
`skipBytes`	Skips ahead a given number of bytes in the given blob
`endRead`	Completes the read

Figure 4.2: Order of operations on blob



and returns the BlobId of the newly created blob. It basically combines the `newBlob`, `writePartial` and `endWrite` methods.

Program 4.5 Definition of createBlob

```
createBlob :: BlobStore -> Blob -> IO BlobId
createBlob blobstore blob = newBlob blobstore
  >>= \wc -> writePartial wc blob
  >>= endWrite
```

The `readBlob` method takes a value of type `BlobId` as argument and returns its entire contents.

Table 4.2: Helper methods for small blobs

Method	Purpose
`createBlob`	Creates a blob with a given content in a BlobStore
`readBlob`	Reads the entire contents of the given blob

4.6 Garbage Collection

It is quite likely that the same blob would be shared by multiple "values" in the database. For a relational database these values are rows in a table, while for a document-oriented database, these values are documents. Hence, we provide an interface for garbage collecting the deleted blobs.

4.6.1 Starting the Garbage Collection

The `startGC` method takes a BlobStore as argument and starts GC for that BlobStore. The `startGC` method does three things:

1. Renames the *curr* directory to *gc*.

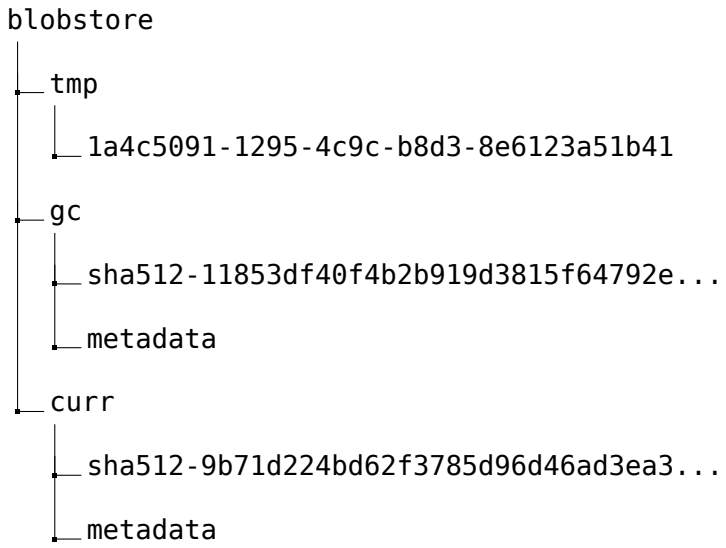
   We use the `rename` system call to atomically move *curr* to *gc*. If a *gc* directory is already present, `rename` will fail with EEXIST or ENOTEMPTY error [10]. This ensures that we have only one GC running at any time.

2. Creates a new *curr* directory.

   We need to ensure that the `endWrite` method is able to move files to the `curr` directory.

3. Copy the metadata file from *gc* to *curr*.

Figure 4.3: Directory structure of a BlobStore during GC



4.6.2 Marking a blob as accessible

Once a blob is marked as not deleted using the method `markAsAccessible`, we move the blob from the *gc* directory to the *curr* directory. We use the `rename` system call, which is atomic. We also call `fdatasync` system call on the file descriptor of the *curr* directory to ensure that the *curr* directory contains an entry for the moved blob, even in case of a system crash.

4.6.3 End Garbage collection

This step involves removal of all the blobs which are not accessible. The `endGC` method takes a BlobStore as argument. It first deletes all the blobs in the *gc* directory as well as the "metadata" file.

The `endGC` method then waits for a user specified interval before deleting the *gc* directory. This is done to restrict the number of lookups during `startRead` to 3 (Refer section 4.4). If we remove this wait, another GC can start and move back the blob which is being looked up `startRead` back to the `gc` directory.

4.6.4 Marking a list of blobs as accessible

Calling `endGC` without marking all the accessible blobs would lead to their deletion. To prevent such accidental calls, we provide a method `markAccessibleBlobs` which takes a list of BlobId as an argument. After calling `startGC`, `markAccessibleBlobs` marks all the blobs in input list as accessible and finally calls `endGC`. This method should be used only when the number of accessible blobs is small.

Table 4.3: Interface for garbage collection

Methods	Purpose
`startGC`	Starts garbage collection for the given BlobStore
`markAsAccessible`	Marks the given blob as accessible
`endGC`	Ends the garbage collection by removing all the unaccessible blobs

4.6.5 Recovering from a crashed GC

A process which had started GC might crash before calling `endGC`. In this case no other process would be able to start a new GC as explained in section 4.6.1. To handle this, we always call `recoverFromGC` in `openBlobStore` method. The method `recoverFromGC` moves all the blobs from the `gc` directory back to the `curr` directory and deletes the `gc` directory. Note that `recoverFromGC` will nullify the effect of a GC which is currently running since it moves all blobs back to the `curr` directory - irrespective of whether they have been deleted or not.

Chapter 5

Conclusion

5.1 Summary

In this thesis we described our design and implementation of Bloc - a library for handling large binary objects written in Haskell. We give a brief introduction to functional programming and describe various features of Haskell which we used in our implementation.

We also describe why programming with locks is difficult and how our design achieves concurrency by making use of certain atomic operations provided by the file systems.

5.2 Future Work

Currently our code uses several functions which are supported only on POSIX compliant file systems. This means that our library will not work on Windows. In the future, we might look into adding support for Windows.

We are also working on creating a key-value store like Bitcask [15]. We plan to add support for storing the blob ids provided by our library as a value in the key-value store.

In this thesis, we tried to give an "informal" explanation of various situations where we handle concurrency. We plan to use π-calculus [16] to give a formal

specification for our design and prove its correctness.

We also plan to test our library by designing an in-memory file system in Haskell which supports the atomic file operations used by us. Using QuickCheck [17] and the in-memory file system, we can formulate and test the properties of our library.

References

[1] Edgar F Codd. "A relational model of data for large shared data banks". In: *Communications of the ACM* 13.6 (1970), pp. 377–387.

[2] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: amazon's highly available key-value store". In: *ACM SIGOPS Operating Systems Review*. Vol. 41. 6. ACM. 2007, pp. 205–220.

[3] Kristina Chodorow. *MongoDB: the definitive guide*. " O'Reilly Media, Inc.", 2013.

[4] Michael J Carey, David J DeWitt, Joel E Richardson, and Eugene J Shekita. *Object and file management in the EXODUS extensible database system*. University of Wisconsin-Madison. Computer Sciences Department, 1986.

[5] David Hows, Peter Membrey, and Eelco Plugge. "GridFS". In: *MongoDB Basics*. Springer, 2014, pp. 101–115.

[6] Russell Sears, Catharine Van Ingen, and Jim Gray. "To blob or not to blob: Large object storage in a database or a filesystem?" In: *arXiv preprint cs/0701168* (2007).

[7] Simon Peyton Jones. "Beautiful concurrency". In: *Beautiful Code: Leading Programmers Explain How They Think* (2007), pp. 385–406.

[8] Carl Hewitt, Peter Bishop, and Richard Steiger. "A universal modular actor formalism for artificial intelligence". In: *Proceedings of the 3rd international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc. 1973, pp. 235–245.

[9] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. "Concurrent programming in ERLANG". In: (1993).

[10] *rename(2) man page*. URL: `http://man7.org/linux/man-pages/man2/rename.2.html` (visited on 05/30/2015).

[11] Daniel J Bernstein. *Using maildir format*. 1995.

[12] *mkdir POSIX specification*. URL: `http://pubs.opengroup.org/onlinepubs/9699919799/functions/mkdir.html` (visited on 05/30/2015).

[13] John Hughes. "Why functional programming matters". In: *The computer journal* 32.2 (1989), pp. 98–107.

[14] Paul J Leach, Michael Mealling, and Rich Salz. "A universally unique identifier (uuid) urn namespace". In: (2005).

[15] DS Justin Sheehy and D Smith. “Bitcask. a log-structured hash table for fast key/value data”. In: *White paper, April* (2010).

[16] Robin Milner. *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[17] Koen Claessen and John Hughes. “QuickCheck: a lightweight tool for random testing of Haskell programs”. In: *Acm sigplan notices* 46.4 (2011), pp. 53–64.