Thesis Title

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

May, 2015

CERTIFICATE

It is certified that the work contained in the thesis titled **Thesis Title**, by **Anshu Avinash**, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Prof Piyush Kurur

Department of Computer Science & Engineering

IIT Kanpur

May, 2015

ABSTRACT

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

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

Department: **Computer Science & Engineering**

Thesis title: **Thesis Title**

Name of Thesis Supervisor: **Prof Piyush Kurur**

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

Placeholder

Placeholder

Acknowledgements

Placeholder

Contents

List of Tables

List of Figures

List of Programs

Chapter 1

Introduction

Database refers to a collection of information that exists overs a long period. 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.

Database systems changed significantly following a famous paper written by Ted Codd in 1970 [1]. Codd proposed that database systems should present the user with a view of data organized as tables called relations. This paper was foundation for popular relational databases like MySQL and PostgreSQL.

However, 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 NoSQL (Not only SQL) databases. These databases can be classified based on the data models used by them. Amazon's Dynamo [2] is a key-value store. In key-value stores records are stored and retrieved using a key that uniquely identifies the record. MongoDB [3] on the other hand is a document-oriented database. Document-oriented databases are designed for managing document-oriented information, also known as semi-structured data.

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.

In this thesis, we explore the second option and provide a simple interface written in *Haskell* to store large files. We also provide an interface for garbage collection of deleted blobs. We try to provide concurrency without using locks as much as possible.

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 present our design. Chapter 4 describes our implementation. We conclude and present the future work in Chapter 5.

Chapter 2

Related Work

Large object files can either be stored directly in a database or we can store the path to the binary file and other metadata. In this section we will discuss few examples of both. We will also discuss merits and demerits of both the approaches.

2.1 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. In MySQL the data type is called BLOB, and has operations similar to that on a string. Corresponding data type in PostgreSQL is bytea.

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]. GridFS 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.

2.2 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 security settings.

2.3 Comparison of both approaches

Both the approaches have their own benefits and disadvantages.

2.3.1 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).

2.3.2 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.

Chapter 3

Design

Our design is inspired from the maildir format [7]. maildir format stores each message in a separate file with a unique name. All the partially delivered mails are stored 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 *cur*.

Similar to maildir, we also store all large objects in separate files. All the large objects of a database are stored under a single directory which we also call a "BlobStore". The BlobStore contains three subdirectories: *tmp*, *curr* and *old*. We will discuss purpose of these directories later in this chapter.

3.1 Initializing the BlobStore

Before starting to create blobs inside a directory, we ensure that the *tmp* and *curr* subdirectories have already been created. We provide a method `initBlobStore` which takes the path of a directory which is to be used as BlobStore as argument and does the initialization for us.

3.2 Creating a Blob

We provide a method called `createBlob` for creating a new blob. It takes a BlobStore as a parameter and returns a WriteContext. WriteContext contains the file handle of just created blob among other things. All the new blobs are created in the *tmp* folder. We use Version 4 UUID [8] to give unique names to the newly created blobs.

3.3 Writing to a Blob

We only allow to add new data at the end of a given blob. We provide `writePartial` method for this. `writePartial` takes a blob and a WriteContext as arguments and appends the given blob to the WriteContext's blob. Once all the data has been written to the blob, `finalizeWrite` is called. `finalizeWrite` takes a WriteContext as argument and moves the blob from *tmp* folder to *curr* folder. We also rename the file to SHA-512 hash of its contents. `finalizeWrite` returns a BlobId. This BlobId contains the location of the blob. No more updates to the blob are possible after calling `finalizeWrite`.

3.4 Reading from Blob

Reading is also sequential. First the `initRead` method is called which returns a ReadContext, similar to the WriteContext. ReadContext also contains the file handle of the blob which is opened in read mode. `readPartial` takes a ReadContext and number of bytes as input and returns those number of bytes from the blob.

Table 3.1: Interface for operations on blob

Methods	Purpose
`initBlobStore`	Initializes given directory to be used as a BlobStore
`createBlob`	Creates a blob in the given BlobStore
`writePartial`	Takes a blob and appends it to the end of the blob given in the argument
`finalizeWrite`	Takes a WriteContext as input and returns a BlobId
`initRead`	Takes a BlobId as input and returns a ReadContext
`readPartial`	Reads a given number of bytes from a Blob
`finalizeRead`	Completes the read

3.5 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 documents.

3.5.1 Starting the Garbage Collection

The `startGC` method takes a BlobStore as argument and starts garbage collection (GC) for that BlobStore. `startGC` does two things: It first renames the *curr* folder to *old* and then creates an empty *curr* folder. Once a GC has started you can not start another GC on the same BlobStore until the first one finishes - doing so will throw an error. Also, note that creation of new blobs and reading the old blobs can happen concurrently with the GC.

3.5.2 Marking a blob as accessible

Once a blob is marked as not deleted using the method `markBlobAsAccessible`, we move it from the *old* folder to the *curr* folder. This ensures that the blob does not get deleted at the end of the GC.

3.5.3 End Garbage collection

This step involves removal of all the blobs which are not accessible. The `endGC` method takes a BlobStore as argument and delete the *old* subdirectory along with its contents.

Table 3.2: Interface for garbage collection

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

Chapter 4

Implementation

In this section we will describe our implementation.

4.1 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 [9]. In this section we will discuss special characteristics and advantages of functional programming (Haskell in particular).

4.1.1 Referential Transparency

Functional programs contain no assignment statements, so variables, once given a value, never change. More generally, functional programs contain no side-effects at all. A functional 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".

4.1.2 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.

4.1.3 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 4.1 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
```

4.1.4 Lazy Evaluation

In lazy evaluation, an expression is not evaluated until its value is needed. This implies that programs can compose very well. 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 4.2 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
```

Note that we did not specify the type of `primes` and `main`. In haskell, we don't have to explicitly write out every type. Types are inferred by unifying every type bidirectionally.

Chapter 5

Conclusion

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

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

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

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

[11] Mendel Rosenblum and John K Ousterhout. “The design and implementation of a log-structured file system”. In: *ACM Transactions on Computer Systems (TOCS)* 10.1 (1992), pp. 26–52.