

BLOC: BINARY LARGE OBJECTS WITH CONCURRENCY

Anshu Avinash

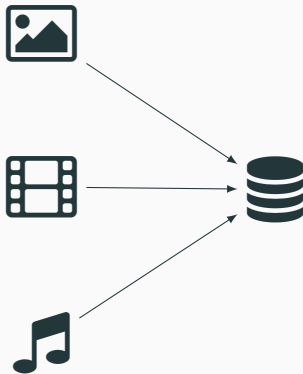
Thesis supervisor: Piyush Kurur

19 June 2015

Department of CSE, IIT Kanpur

WHAT IS A BLOB?

BLOB stands for “**B**inary Large **OB**ject”.



Store the entire content of the blob in the database.

- PostgreSQL breaks large objects into “chunks” and these chunks are stored in rows in the database.
- MongoDB also divides large objects using the GridFS specification and stores them in the “chunks” collection.

Store the blob in a file and store the file name in the database.

- Number and size of blobs are now limited only by the file system.
- Requires to develop an interface to keep track of all the blobs of a database.

We provide a library which will keep track of all the blobs stored in a database.

- Our design is inspired from the maildir format¹.

¹Daniel J Bernstein. *Using maildir format*. 1995.

- Our design is inspired from the maildir format¹.
- All the blobs of a database are stored inside a directory which we also call a **BlobStore**.

¹Daniel J Bernstein. *Using maildir format*. 1995.

- Our design is inspired from the maildir format¹.
- All the blobs of a database are stored inside a directory which we also call a BlobStore.
- Each blob is stored as a different file inside the BlobStore.

¹Daniel J Bernstein. *Using maildir format*. 1995.

- Our design is inspired from the maildir format¹.
- All the blobs of a database are stored inside a directory which we also call a **BlobStore**.
- Each blob is stored as a different file inside the BlobStore.
- We provide an incremental interface for writing to a blob as well as reading from a blob.

¹Daniel J Bernstein. *Using maildir format*. 1995.

```
openBlobStore :: FilePath -> IO BlobStore  
newBlob       :: BlobStore -> IO WriteContext
```

CREATING A NEW BLOB

```
openBlobStore :: FilePath -> IO BlobStore
newBlob       :: BlobStore -> IO WriteContext
```

```
└─ blobstore
   └─ tmp
      ├── 1a4c5091-1295-4c9c-b8d3-8e6123a51b41
      └── 4af0ac03-0b79-43fd-a2b0-2cc7c62947bc
```

```
writePartial :: WriteContext  
             -> Blob  
             -> IO WriteContext  
endWrite     :: WriteContext -> IO BlobId
```

ADDING CONTENTS TO A BLOB

```
writePartial :: WriteContext  
             -> Blob  
             -> IO WriteContext  
endWrite     :: WriteContext -> IO BlobId
```

```
blobstore  
├── tmp  
│   └── 1a4c5091-1295-4c9c-b8d3-8e6123a51b41  
└── curr  
    └── sha512-861844d6704e8573fec34d967e20b..
```

```
startRead      :: BlobId -> IO ReadContext
readPartial    :: ReadContext -> Int -> IO Blob
skipBytes      :: ReadContext -> Integer -> IO ()
endRead        :: ReadContext -> IO ()
```

- A blob can be shared by multiple “records” of the database.

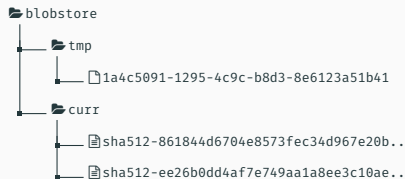
- A blob can be shared by multiple “records” of the database.
- We provide an interface for garbage collection of deleted blobs.


```
startGC :: BlobStore -> IO ()
```

STARTING A GC

```
startGC :: BlobStore -> IO ()
```

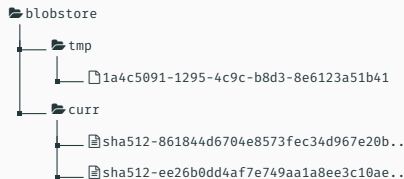
Before startGC



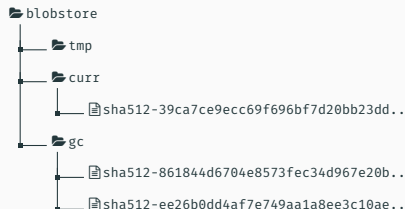
STARTING A GC

```
startGC :: BlobStore -> IO ()
```

Before startGC



After startGC

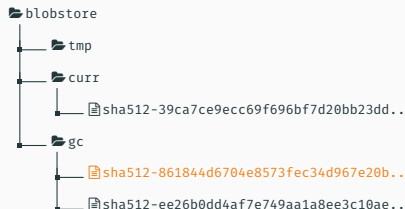


```
markAsAccessible :: BlobId -> IO ()
```

MARKING A BLOB AS ACCESSIBLE

```
markAsAccessible :: BlobId -> IO ()
```

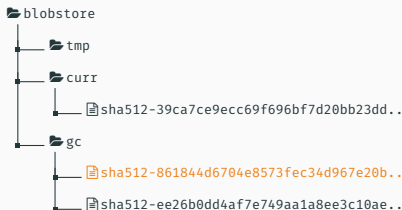
Before



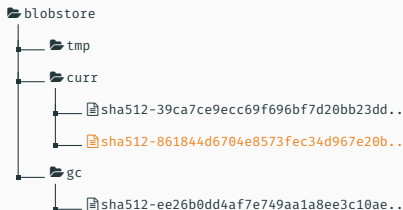
MARKING A BLOB AS ACCESSIBLE

`markAsAccessible :: BlobId -> IO ()`

Before



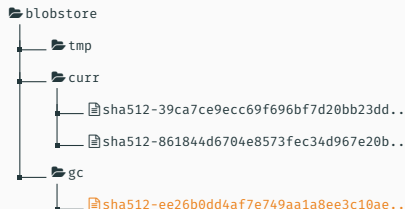
After



```
endGC :: BlobStore -> IO ()
```

```
endGC :: BlobStore -> IO ()
```

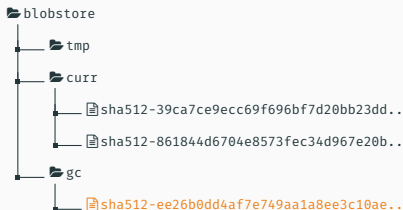
Before



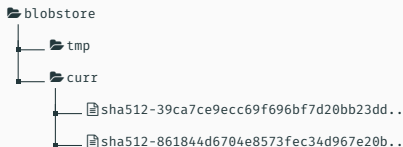
ENDING A GC

`endGC :: BlobStore -> IO ()`

Before



After



HOW DOES THIS DESIGN ACHIEVE CONCURRENCY?

HOW DOES THIS DESIGN ACHIEVE CONCURRENCY?

Using atomic file operations.

HOW DOES THIS DESIGN ACHIEVE CONCURRENCY?

Using atomic file operations.

- rename

HOW DOES THIS DESIGN ACHIEVE CONCURRENCY?

Using atomic file operations.

- rename
- mkdir

HOW DOES THIS DESIGN ACHIEVE CONCURRENCY?

Using atomic file operations.

- rename
- mkdir

Let us revisit our design.

CREATING A NEW BLOB (PART 2)

```
openBlobStore :: FilePath -> IO BlobStore
newBlob       :: BlobStore -> IO WriteContext
```

```
blobstore
├── tmp
│   ├── 1a4c5091-1295-4c9c-b8d3-8e6123a51b41
│   └── 4af0ac03-0b79-43fd-a2b0-2cc7c62947bc
```

ADDING CONTENTS TO A BLOB (PART 2)

```
writePartial :: WriteContext  
             -> Blob  
             -> IO WriteContext  
endWrite     :: WriteContext -> IO BlobId
```

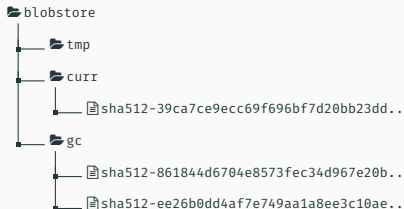
```
blobstore  
├── tmp  
│   └── 1a4c5091-1295-4c9c-b8d3-8e6123a51b41  
└── curr  
    └── sha512-861844d6704e8573fec34d967e20b..
```


READING FROM A BLOB (PART 2)

```
startRead      :: BlobId -> IO ReadContext
readPartial    :: ReadContext -> Int -> IO Blob
skipBytes      :: ReadContext -> Integer -> IO ()
endRead        :: ReadContext -> IO ()
```

GC (PART 2)

```
startGC           :: BlobStore -> IO ()
markAsAccessible :: BlobId  -> IO ()
endGC             :: BlobStore -> IO ()
```



```
createBlob :: BlobStore -> Blob -> IO BlobId
```

```
createBlob :: BlobStore -> Blob -> IO BlobId
```

```
readBlob :: BlobId -> IO Blob
```

SOME HELPER FUNCTIONS

```
createBlob :: BlobStore -> Blob -> IO BlobId
```

```
readBlob :: BlobId -> IO Blob
```

```
markAccessibleBlobs :: BlobStore -> [BlobId] -> IO ()
```

SOME HELPER FUNCTIONS

```
createBlob :: BlobStore -> Blob -> IO BlobId
```

```
readBlob :: BlobId -> IO Blob
```

```
markAccessibleBlobs :: BlobStore -> [BlobId] -> IO ()
```

```
recoverFromGC :: FilePath -> IO ()
```

- fsync a file

- fsync a file
- fsync a directory

- Proving the correctness of our design

- Proving the correctness of our design
- Writing a key-value store

- Proving the correctness of our design
- Writing a key-value store
- Supporting bloc on non-POSIX systems