CHARON: A Secure Cloud-of-Clouds System for Storing and Sharing Big Data — *Supplemental Material* —

Ricardo Mendes, Tiago Oliveira, Vinicius Cogo, Nuno Neves, and Alysson Bessani

1 LEASE PROTOCOL CORRECTNESS

In this section, we present the correctness proofs for the Byzantine-resilient composite lease and for all base lease implementations.

1.1 Byzantine-Resilient Composite Lease

In order to prove the mutual exclusion in the access of some resources, there is the need to precisely define what it means for a process to hold a lease for a given resource.

Definition 1. *A correct client c is said to hold the lease at a given time t for $T' > 0$ time units if it obtains T' as response when executing the lease(T) operation in a quorum of base lease objects.*

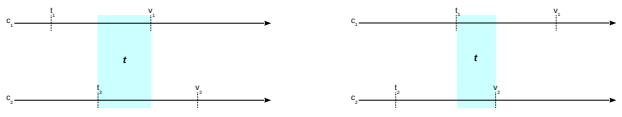
With this definition, we proceed to prove the properties of lease algorithm presented in the paper.

We use the function $\mathrm{C}()$ to prove some of these properties. This function maps the local clock values in the processes to the (global) real-time clock value. Note that the processes do not have access to this function, this is only a theoretical device to reason about the correctness of the protocols.

Theorem 1 (Mutual Exclusion). *There are never two correct clients with a lease.*

Proof. Assume this is false: there exists a global real-time instant t in which two clients c_1, c_2 both hold a lease. We will prove that this assumption leads to a contradiction. Let t_1 (resp. t_2) be the local time in which the *lease(T)* operation returned the value T' to c_1 (resp. c_2), i.e., the moment the client starts holding the lease. Let also t_{start1} (resp. t_{start2}) be the local time in which the *lease(T)* operation is called by the c_1 (resp. c_2). The moment c_1 assumes the lease has expired is $v_1 = t_{start1} + T'$ (resp. c_2 and $v_2 = t_{start2} + T'$).

Within these definitions, the existence of t requires that either $\mathrm{C}(t_2) \leq \mathrm{C}(v_1)$ or $\mathrm{C}(t_1) \leq \mathrm{C}(v_2)$.



$\mathrm{C}(t_2) \leq \mathrm{C}(v_1)$ $\quad$ $\mathrm{C}(t_1) \leq \mathrm{C}(v_2)$

Figure 1. Mutual exclusion proof illustration.

Since the invocation of *lease(T)* precedes its return and it precedes the moment in which the lease expires in all base lease objects, we have:

$$\mathrm{C}(t_{start1}) < \mathrm{C}(t_1) < \mathrm{C}(v_1) \tag{1}$$

$$\mathrm{C}(t_{start2}) < \mathrm{C}(t_2) < \mathrm{C}(v_2) \tag{2}$$

Assume the left case presented in Figure 1. In this case, since each base lease object guarantees *mutual exclusion*, the client c_2 will only be able to acquire a lease when the lease held by c_1 has already expired in at least $n - f$ base objects, i.e, $\mathrm{C}(t_1) + T' < \mathrm{C}(t_2)$. Given this condition together with Equations 1 and 2 we have:

$$\begin{aligned} &\mathrm{C}(t_1) + T' < \mathrm{C}(t_2) \implies \\ &\mathrm{C}(t_{start1}) + T' < \mathrm{C}(t_2) \implies \\ &\mathrm{C}(t_{start1} + T') < \mathrm{C}(t_2) \implies \\ &\mathrm{C}(v_1) < \mathrm{C}(t_2) \end{aligned} \tag{3}$$

Equation 3 contradicts the left case of Figure 1 for the existence of t. The same approach have to be used, assuming that c_2 obtains the lease before c_1, to prove that the other condition is also impossible. □

Theorem 2 (Obstruction-freedom). *A correct client that attempts to obtain a lease without contention will succeed in acquiring it.*

Proof. Client c starts by trying to obtain $n - f$ successful leases from the base objects (Lines 5–8). It will obtain the lease from all correct base lease objects since, (1) there is no other client holding the same lease, (2) no other client is trying to obtain the lease at the same time, and (3) each base lease object must guarantee the "*Obstruction-freedom*" property. The client c acquires the lease and returns it after obtaining $n - f$ successful responses from base objects. □

Theorem 3 (Time-boundedness). *A correct client that acquires a lease will hold it for at most T time units, unless the lease is renewed.*

Proof. Assume that it is false: there is a case in which a client c holds a lease for $T' > T$ time units. We will prove that this assumption results in a contradiction. Let t_1 be the local time

ALGORITHM 1: Object storage lease by client c.

```
1  function objStorageLease(leaseDuration) begin
2      L ⟵ cloud.list();
3      cTime ⟵ cloud.getTime();
4      lease_id ←"lease-"+c+"-"+leaseDuration;
5      foreach lease-cid-t ∈ L :
         lastModified(lease-cid-t)+t > cTime ∧ verify(lease-cid-t)
         do
6          if cid ≠ c then return nok ;
7          else return performLease(lease_id, lease-cid-t) ;
8      return performLease(lease_id, ⊥);
9  function performLease(lease_id, oldLease) begin
10     cloud.write(lease_id, sign(lease_id));
11     L ⟵ cloud.list();
12     if ∃lease-cid-t∈ L : cid ≠ c ∧ lastModified(lease-cid-
         t)+t> cTime ∧ verify(lease-cid-t))
         then
13         cloud.delete(lease_id));
14         return nok;
15     if (oldLease ≠⊥) ∧ (oldLease ≠ lease_id) then
         cloud.delete(oldLease) ;
16     return ok;
```

lease(T) returns the value T' to c_1, i.e., the moment the client holds the lease. Let also t_{start} be the local time in which the *lease(T)* operation is called by c_1.

Given that c_1 holds a lease for $T' > T$ time units and since the invocation of *lease(T)* precedes its return ($t_{start} < t_1$), we have:

$$\begin{aligned} T' = T - (t_1 - t_{start}) > T \implies \\ -(t_1 - t_{start}) > 0 \implies \\ t_1 - t_{start} < 0 \implies \\ t_1 < t_{start} \end{aligned} \tag{4}$$

The result of Equation 4 contradicts the causal definition of t_{start} and t_1. □

1.2 Storage Services

In this subsection we will prove the correctness of the cloud storage based lease object protocol. The implementation is presented in Algorithm 1.

Lemma 1. *An entry e created with the operation write(e, s) and not removed, will appear in the result of later list() operations executed on the same cloud.*

Proof. The *write(e, s)* operation is only completed when the entry is created/written in the cloud (Line 10 of Algorithm 1). Since we assume that all services used to implement all base lease objects provide at least read-after-write consistency, it means that if a client tries to list the entries in the leases container of some storage cloud after the *write(e, s)* operation, it must return the entry e with the content s written before. □

To prove the properties mentioned before we need to define what it means for a client to hold a lease.

Definition 2. *A correct client c is said to hold the lease at a given time t if an entry $e \equiv$ lease-c-T containing sign(e, K_{r_c}) with lastTimeModified(e)+T $> t$ appears in list() result when this operation is executed in a cloud storage service.*

Now, the safety and liveness properties for Algorithm 1 will be proved.

Theorem 4 (Mutual Exclusion). *There are never two correct clients with a lease.*

Proof. Let us assume this is false: there is a time t in which two correct clients c_1 and c_2 hold the lease. We will prove that this assumption leads to a contradiction. If both c_1 and c_2 hold the lease, both *lease-c*$_1$*-T*$_1$ and *lease-c*$_2$*-T*$_2$ with *lastTimeModified(lease-c*$_1$*-T*$_1$*)+T*$_1 > t$ and *lastTimeModified(lease-c*$_2$*-T*$_2$*)+T*$_2 > t$ are returned in *list()* from a cloud storage service. Algorithm 1 and Lemma 1 state that it can only happen if both c_1 and c_2 wrote valid lease entries (Line 10) and did not remove them (Line 13). In order for this to happen, both c_1 and c_2 must see only their lease entries in their second *list()* on the cloud (Line 11).

Two situations may arise when c_1 and c_2 acquire write leases: (1) either c_1 (resp. c_2) writes its lease entry before c_2 (resp. c_1) lists the lease entries the second time or (2) c_1 (resp. c_2) writes its lease entry while c_2 (resp. c_1) is executing its second *list()*.

In the first situation, when c_2 (resp. c_1) lists the leases container to discover what were the written leases, it will see both lease entries and thus remove *lease-c*$_2$*-T*$_2$ (resp. *lease-c*$_1$*-T*$_1$), releasing the lease (Lines 11–14). Situation (2) is more complex because the start and finish of each phase of the algorithm must be analysed. Consider the case in which c_1 finishes writing its lease entry (Line 10) after c_2 executes the second list (Line 11). Clearly, in this case c_2 may or may not see *lease-c*$_1$*-T*$_1$ in Line 12. However, we can say that the second list of c_1 will see *lease-c*$_2$*-T*$_2$ since it is executed after c_1 lease entry is written, which happens, only after c_2 starts its second list, and consequently after its lease entry is written. It means that the condition of Line 12 will be true for c_1, and it will remove *lease-c*$_1$*-T*$_1$. The symmetric case (c_2 finishes writing its lease file after c_1 executes the second list) also holds.

In both situations we have a contradiction, i.e., there is no execution and time in which two correct clients hold the lease. □

Theorem 5 (Obstruction-freedom). *A correct client that attempts to obtain a lease without contention will succeed in acquiring it.*

Proof. When there is no other valid lease entry in the cloud (i.e., the first list in Line 2 does not return any lease entry), c will execute the procedure *performLease(lease_id, ⊥)* in Line 8 and will write *lease-c-T* on the cloud storage service. This lease entry will be the only valid entry read on the second list (the condition of Line 12 will not hold) since (1) no other valid lease entry is available on the clouds since no other client is trying to acquire the lease and (2) Lemma 1 states that if the lease entry was written, it will be available to read. After this, c acquires the lease by returning *ok* (Line 16). □

Theorem 6 (Time-boundedness). *A correct client that acquires a lease will hold it for at most T time units, unless the lease is renewed.*

Proof. Let us assume this is false: a client that acquires a lease hold it for $T' > T$ time units without renewing it. We will prove that this assumption leads to a contradiction. Let $t_1 =$ *lastTimeModified(e)* be the moment in which the client acquires the lease (i.e. starts holding it). This is the exact moment cloud executes the *write(e,s)* operation (Line 10 of Algorithm 1), where

ALGORITHM 2: Augmented queue lease by client c.

```
1  function queueLease(leaseDuration) begin
2      lease_id ⟵ "lease-"+c+nonce+"-"+sign("lease-"+c+nonce);
3      L ⟵ queue.list();
4      if ∃lease-cid-N-s ∈ L : cid ≠ c ∧ verify(s) then return nok;
5      return performLease(L, lease_id, leaseDuration);
6  procedure performLease(L, lease_id, leaseDuration) begin
7      queue.add(lease_id, leaseDuration);
8      L2 ⟵ queue.list();
9      for 0 ≤ i ≤ size(L2) do
10         if L2[i] ≡lease-cid-nonce-s ∧ verify(s) then
11             if L2[i] = lease_id then
12                 queue.delete(L);
13                 return ok;
14             else if cid ≠ c then
15                 queue.delete(lease_id);
16                 return nok;
17     return nok;
```

ALGORITHM 3: NoSQL database lease by client c.

```
1  function noSqlDBLease(leaseDuration) begin
2      res ⟵ db.query("key", EQ, "lease");
3      cTime ⟵ db.getTime();
4      if (res ≠⊥) ∧ (res.cId ≠ c) ∧ (res.expirationTime < cTime) ∧ verify(res) then return nok;
5      expTime ⟵ cTime + leaseDuration;
6      succeed ⟵ db.testAndSetItem(res, ⟨"lease", expTime, c, sign("lease"+expTime+c)⟩);
7      return succeed;
```

e is the lease entry with the content s. If the client holds the lease for $T' > T$ time units then exists an instant v_1 such that $v_1 - t_1 > T$.

Algorithm 1 (Line 5) and Definition 2 state that a client holds a lease at instant t_1 if an entry e is returned in a *list()* operation and *lastTimeModified(e)*$+T > t_1$. According with this, we have:

$$\begin{gathered} lastTimeModified(e) + T > v_1 \implies \\ t_1 + T > v_1 \implies \\ T > v_1 - t_1 \end{gathered} \tag{5}$$

We obtained a contradiction between Equation 5 and the causal definition of v_1. □

1.3 Augmented Queues

This subsection presents the correctness proofs for the lease object algorithm based on augmented queues (presented in Algorithm 2).

Lemma 2. *An entry e, created with the operation add(e, T) at instant t_{add}, will appear as result of later operations list() at instant t_{list} if the condition $t_{add} < t_{list} < t_{add} + T$ holds.*

Proof. The *add(e, T)* operation is only finished when the entry is created in the queue (Line 7 of Algorithm 2). Since the cloud ensures the entry e to be available in the queue for T time units after the *add(e, T)* operation occurs, if a client lists the entries in the queue at that interval, e must be returned in the result. □

To prove the properties mentioned before we are obligate to define what it means for a client to hold a lease.

Definition 3. *A correct client c is said to hold the lease at a given time t if an entry lease-c-T-s is the valid lease entry with the lowest index in the result of list() when this operation is executed in a cloud queue service.*

With this definition, we are now able to prove the safety and liveness of Algorithm 2.

Theorem 7 (Mutual Exclusion). *There are never two correct clients with a lease.*

Proof. We will prove this theorem by contradiction. Assume that there is global real-time t in which two correct clients c_1 and c_2 hold the lease.

If both c_1 and c_2 hold the lease we have that both *lease-*c_1*-*T_1 and *lease-*c_1*-*T_2 are returned in the second *list()* at instant t from a queue service. Algorithm 2 and Lemma 2 state that it can only happen if both c_1 and c_2 wrote valid lease entries (Line 7). In this case, both c_1 and c_2 must see their lease entries as the valid ones with the lowest index in their second *list()* on the cloud (Lines 8–11).

This is impossible since, independently of the order in which the cloud queue executes the lease entry creation requests from clients c_1 and c_2, only one of the entries will be at the head of the queue, being that entry the one with the lowest index in later *list()* invocations. Given this, there is no execution and time in which two correct clients hold the same lease. □

Theorem 8 (Obstruction-freedom). *A correct client that attempts to obtain a lease without contention will succeed in acquiring it.*

Proof. When there is no valid lease entries in the queue, c will execute *performLease(L, lease_id, time)* (Line 5) since the condition of Line 4 will not hold. In this procedure the entry *lease-c-nonce-s* will be created on the queue service.

On the second *list()* the only valid lease entry obtained will be the one just written. Due of that, the conditions of Lines 10–11 will hold since (1) no other valid lease entry is available on the queue because no other client is trying to acquire the lease and (2) Lemma 2 states that if the lease entry was written it will be available to subsequent *list()* operations. At this point, c will acquire the lease after returning *ok* (Line 13). □

Theorem 9 (Time-boundedness). *A correct client that acquires a lease will hold it for at most T time units, unless the lease is renewed.*

Proof. Definition 3 defines that a client c holds a lease e if it is the valid lease entry with the lowest index in the result of the operation *list()*.

Since the cloud queue services ensure that an entry e created with the operation *add(e, T)* will be present in the queue for at most T time units, then a lease entry inserted by this operation will also be returned as result of the *list()* operation for at most T time units. □

1.4 NoSQL Databases

In this subsection we will prove the correctness of the Amazon DynamoDB based lease object implementation. The pseudo-code is presented in Algorithm 3.

Lemma 3. *An entry e with the key "lease" created with a successful operation testAndSetItem(e′, e) and not removed will appear as the result of later operations query("key", EQ,"lease").*

Proof. As explained before, the *testAndSetItem(e′, e)* operation only is succeeded if the entry is created / updated in DynamoDB service (Line 6 of Algorithm 3). If a client tries to read the available lease entry by executing the *query("key", EQ,"lease")* operation after a successful *testAndSetItem(e′, e)* operation occurs, clearly the cloud must return e as result. □

To prove the properties mentioned before we have to define carefully what it means for a client to hold a lease.

Definition 4. *A correct client c is said to hold the lease at a given time t if a valid lease entry e with e.key ="lease", e.cId = c and t < e.expirationTime is stored in the DynamoDB cloud service.*

With this definition, we are now able to prove the safety and liveness of Algorithm 2.

Theorem 10 (**Mutual Exclusion**). *There are never two correct clients with a lease.*

Proof. Assume this is false: there is global real-time t in which two correct clients c_1 and c_2 hold the lease.

If both c_1 and c_2 hold the lease, accordingly with Lemma 3 and Definition 4, we have that both must successful execute the *testAndSetItem(e′, e)* operation (Line 6).

Since this operation is atomic, either if c_1 or c_2 execute this operation first, only one of the two will be succeed in inserting its lease entry e in the cloud service, being it the one that obtains success in the operation. When the other executes that operation, the lease entry present in the cloud will differ from the one it provides as e' and then its lease entry will not be inserted. Consequently, we have a contradiction. □

Theorem 11 (**Obstruction-freedom**). *A correct client that attempts to obtain a lease without contention will succeed in acquiring it.*

Proof. When there is no contention in acquiring the lease and there is no valid lease entries in DynamoDB, c will obtain $res = \perp$ when executing the *query("key", EQ, "lease")* (Line 2). After this, c will try to insert the lease entry e in the service by executing *testAndSetItem(⊥,e)* (Line 6). Since there is no contention in acquiring the lease, this operation will succeed and c will acquire the lease. □

Theorem 12 (**Time-boundedness**). *A correct client that acquires a lease will hold it for at most T time units, unless the lease is renewed.*

Proof. We will prove this theorem by contradiction: there is an instant t in which a client holds a lease for more than $T' > T$ time units without renew it. Let t_{start} and t_1 be the moment in which the *lease(T)* operation is called and the moment in which the client starts holding the lease, i.e. inserts the lease entry e in Amazon DynamoDB, respectively. Let also $v_1 = t_{start} + T$ be the expiration time of that lease. If the client holds the lease for $T' > T$ time units then we have t such that $t - t_1 > T$.

Since the start of execution of the *lease(T)* operation precedes the acquisition of the lease and its expiration ($t_{start} < t_1 < v_1$) and since Algorithm 3 and Definition 4 state that a client holds a lease in instant t only if $t < v_1$, we have:

ALGORITHM 4: Transactional data base lease c.

```
1  function bdTransLease(leaseDuration) begin
2      trans ⟵ ds.beginTransaction();
3      res ⟵ ds.lookUp("lease", trans);
4      cTime ⟵ ds.getTime();
5      if (res ≠ ⊥) ∧ (res.cId ≠ c) ∧ (res.expirationTime <
         cTime) ∧ verify(res.sign) then
6          ds.abort(trans);
7          return nok;
8      return
         performLease((cTime + leaseDuration), (res = ⊥), trans);
9  procedure performLease(expTime, isRenew, trans) begin
10     lease_id ⟵ ⟨
         "lease", c, expTime, sign("lease"+c+expTime)⟩;
11     if isRenew then ds.update(lease_id, trans) ;
12     else ds.insert(lease_id, trans) ;
13     return ds.commit(trans);
```

$$\begin{gathered} t < v_1 \implies t < t_{start} + T \implies \\ t < t_1 + T \implies \\ t - t_1 < T \end{gathered} \tag{6}$$

The result of this equation contradicts the causal definition of t. □

1.5 Transactional Databases

This subsection presents the correctness proofs of the Algorithm 4, which describes the base lease objects based on transactional databases.

Lemma 4. *An entry e with the key "lease" created with an operation insert(e, trans) and a subsequent commit(trans), if not removed, will appear as result of later operations lookUp("lease",trans′) with trans′ = trans ∨ trans′ ≠ trans.*

Proof. As described before, the *insert(e′, trans)* inserts the entry e in the Google Datastore if, when the subsequent commit occurs, there is not other entry with the same key of e (i.e. "lease"). If the subsequent *commit(trans)* operation succeeds it means that the entry was created in the service (Line 13 of Algorithm 4). Due of this, if the entry e was not removed it must appear as result of later executions of the *lookUp("lease",trans′)* operation either if *trans′* is the transaction used to create the entry e or not. □

To prove the safety and liveness of the Algorithm 4, we need first to define what it means for a client c to hold a lease.

Definition 5. *A correct client c is said to hold the lease at a given time t if a valid lease entry e with e.key ="lease", e.cId = c and e.expirationTime > t is stored in Google Datastore.*

With this definition, we are able to prove the properties for the base lease implementation presented in Algorithm 4.

Theorem 13 (**Mutual Exclusion**). *There are never two correct clients with a lease.*

Proof. Assume this is false: there is global real-time t in which two correct clients c_1 and c_2 hold the lease.

If both c_1 and c_2 hold the lease, accordingly with Definition 5 and Lemma 4, we have that both must obtain success in the

commit(trans) operation after the insertion of the entry e (Line 12 and 13).

Since the transaction *trans* is started before the *lookUp("lease", trans)* (at Line 2) and it is only committed after the insertion of the entry e in the service (Line 13), this makes these two operation to be executed in an atomic way. Due of this, there are never two clients executing the *lookUp("lease-", trans)* and *insert(e,trans)* operations concurrently.

In this way, when the *commit(trans)* operation of c_1 (resp. c_2) is executed, if there is no other client that holds a valid lease, it will succeed and c_1 (resp. c_2) acquires the lease. At its turn, when c_2 (resp. c_1) executes its *commit(trans)* operation, according with Lemma 4, there will be in the Google Datastore the lease entry previously written by c_1 (resp. c_2). Since the *insert(e, trans)* needs the service to not have any entry with the same key of e, the subsequent *commit(trans)* operation will not succeed, resulting in c_2 (resp. c_1) to failing in acquiring the lease.

We have a contradiction because there is no possible execution and real-time t in which two correct clients hold the lease. □

Theorem 14 (**Obstruction-freedom**). *A correct client that attempts to obtain a lease without contention will succeed in acquiring it.*

Proof. Since there is no contention in acquiring the lease, the cloud service will not have any valid lease entries and then, when c executes the *lookUp("lease", trans)* operation, it will obtain $res = \perp$ (Line 3).

After this, it will execute the procedure *performLease(expTime, isRenew, trans)* at Line 8 of the Algorithm 4 with $isRenew = false$ (since $res \neq \perp$). In this procedure, client c will call the operation *insert(e, trans)* and will commit the transaction *trans*, being e the lease entry. Since *trans* started before the *lookUp("lease", trans)* operation and there is no contention in acquiring the lease, the *commit(trans)* operation must succeed. According with the Definition 5 it means that c holds the lease. □

Theorem 15 (**Time-boundedness**). *A correct client that acquires a lease will hold it for at most T time units, unless the lease is renewed.*

Proof. This proof is similar to the one from Theorem 12. We will also prove this theorem by contradiction: there is an instant t in which a client holds a lease for more than $T' > T$ time units without renew it. As in that proof, let t_{start} and t_1 be the moment in which the *lease(T)* operation is called and the moment in which the lease entry e is inserted in the Google Datastore, respectively. The expiration time of the lease is given by $v_1 = t_{start} + T$. Assuming client holds the lease for $T' > T$ time units then we have t such that $t - t_1 > T$.

Due to the fact that the call of the *lease(T)* operation precedes the acquisition of the lease and its expiration time ($t_{start} < t_1 < v_1$) and since Algorithm 4 and Definition 5 state that a client only holds a lease at instant t if $t < v_1$, we can infer:

$$t < v_1 \implies t < t_{start} + T \implies t < t_1 + T \implies t - t_1 < T \quad (7)$$

The result of this equation contradicts the causal definition of t. □

2 FS-Biobench: A File System Benchmark from Bioinformatics Workflows

Benchmarking file systems is complex and certainly has some pitfalls and limitations [1], [2], for instance choosing non-representative workflows.

In this appendix, we describe a file system benchmark, FS-Biobench, which simulates the I/O of representative tasks commonly executed in the interactions between bioinformatics researchers and data repositories.

2.1 Bioinformatics Workflows

Obtaining human genomes at affordable cost and time is a reality with high-throughput sequencing methods (i.e., the Next Generation Sequencing—NGS [3]). In fact, such approach is already changing the basic structures of bioinformatics workflows. Earlier algorithms focused on working with small sequences, while more recent proposals are required to work with whole genome sequences, or even large sets of genomes. In this section, we present the eight workflows considered in our benchmark.

w1_genotyping. Genotyping obtains the genomic variations of an individual compared with a reference genome [4]. Each entry for a genotyped variation contains its identifier, the chromosome and position where it is found and the individual's genotype (e.g., `rs10219424 11 124565388 CA`). This last field contains two letters, where one is inherited from the father's chromosome and one from the mother's.

Each entry sizes 27 bytes, and the number of genotyped variations depend on the platform used. For example, genotyping files from the 23AndMe platform have more than 960k variations in approximately 24MB each. In the `w1_genotyping` workflow, we sequentially write a text file containing one genotyped variation entry per line, up to the number of entries specified by the user.

w2_sequencing. DNA sequencing obtains all nucleotides (A, C, G, and T letters) from portions or the entire genome of an individual [5]. However, this process does not directly create a contiguous DNA sequence. Sequencing output usually is divided in several large text files in the FASTQ format containing unaligned DNA reads, which are composed of 50–1000 characters each. Each read entry in these files is composed of 4 lines: a comment line about the sequence, the read sequence, a comment line about the read quality, and the read quality, for example:

```
@SRR067577.2766/1
TATATTGGTCAGGCTGCCTCAGGCGATCCACCCGCCTCAGCCTCC
+
IFIHHGHIHGIGHHEGGGDEDGEGEDEDD@DD7DD@BB@>=B###
```

In the `w2_sequencing` workflow, we sequentially write a text file containing read entries composed of those four lines. The workflow receives the DNA read size S and the number of sequenced reads as arguments from users. Each entry sizes $(2 \times S) + 23$ bytes. For example, an entry for reads with 100 characters will size 223 bytes. The final output size will depend also on the number of DNA reads it will contain. For example, the sequencing output of an entire human genome with 30× coverage sizes approximately 180GB.

Output files from workflows `w1_genotyping` and `w2_sequencing` are considered raw data for several other bioinformatics workflows since they are the most basic data set one can obtain from biological samples.

w3_prospection. Finding the appropriate data samples for a study is important since large quantities of samples (up to 50k) are

often necessary to obtain valid results. Large data storages have an extremely important role in this prospection since they are responsible for providing the access to data samples.

MIABIS [6] is a data model proposed as the minimal data set needed for sharing biobank samples and data. The `w3_prospection` workflow implements the MIABIS data model in XML files and simulates a query for selecting appropriate sample collections, by country and disease, for a study. Considering the 330 European biobanks and more than 700 sample collections registered in the BBMRI-ERIC catalogue [7], the biobanks XML file in our workflow sizes 256kB (900B per entry) and the XML with sample collection entries sizes approximately 850kB (1400B per entry). The output file from `w3_prospection` workflow contains a list of prospected samples that sizes 4kB.

w4_alignment. Read alignment finds the chromosome and the position in a reference genome where a DNA read is found [8]. Researchers align all DNA reads from a sequenced genome (e.g., from workflow `w2_sequencing`) before starting several more complex analyses (e.g., `w7_annotation`). Thus, the input of workflow `w4_alignment` is composed of sequencing data and the aligned reads are stored in a format called Sequence Alignment/Map (SAM). Each entry sizes approximately $(2 \times S) + 134$ bytes, where S is the read size. It is bigger than a FASTQ entry because it contains almost the same information plus the alignment result.

Workflow `w4_alignment` sequentially reads the sequencing input file, and writes one SAM entry for each read. However, not all DNA reads are successfully aligned because there are either unknown sequences with no good match or ambiguous sequences that could be aligned to more than one chromosome and position. In our benchmark, for each GB of input, we write approximately 960MB to the output.

w5_assembly. Genome assembly obtains the contiguous whole genome sequence from a sequencing data file (e.g., from workflow `w2_sequencing`) [9]. Each DNA read is first aligned to a genome reference, similarly to workflow `w4_alignment`, and finally the resulting fragments are merged in a single contiguous sequence. The entire assembled sequence is written in a text file only at the end of workflow `w5_assembly` because the merging step needs all aligned sequences containing each position to decide on the best match. A human genome in this text format (i.e., FASTA) sizes approximately 3GB.

Output files from workflows `w4_alignment` and `w5_assembly` contribute to several complex analyses because knowing where each DNA read is positioned in a genome allows researchers to compare the segments with other studied sequences with well-known functions.

w6_gwas. Genome Wide Association Studies (GWAS) correlate genomic variations and traits by comparing cases (e.g., diagnosed patients) and controls (e.g., healthy people) [10]. Variations that are much more frequent in one group than in the other, become variations of interest. GWAS' main goal is to find good evidences that the presence or absence of a specific set of variations accelerates or inhibits the development of some disease, which should be further investigated by other studies.

In `w6_gwas`, we sequentially read two genotyping files (e.g., from `w1_genotyping`) with approximately 24MB each one, and create an intermediate file in the Variant Call Format (VCF). Each line in the VCF file contains the chromosome, the position and identification of the genomic variation, the reference and variant letters, and the genotype of each individual included in this study. It sizes approximately $(4 \times I) + 28$, where I is the number of individuals included in the GWAS. The next step reads the VCF file, and for each genomic variation simulates the calculation of odds ratio of an individual having the mutation in his genome and contracting the studied disease. Additionally, the significance of odds ratios is calculated. These two variables are written in a second intermediate file that contains one line per genomic variation and each entry sizes approximately 37 bytes. The last step of this workflow writes a file with the same size as a graph in the bitmap format (420kB), which correlates the genome positions with the negative logarithmic of the significance (i.e., a Manhattan plot).

w7_annotation. Attesting the presence or absence of disease-related genes in a genome is important for suggesting the predisposition to an individual contract a disease [11]. Annotating a genome, with biological information related to genomic data, accelerates the reports from the personalized medicine context, which are auxiliary methods to medicine rather than diagnoses.

The `w7_annotation` workflow sequentially reads a sequencing data file (e.g., from the `w2_sequencing`), aligns all DNA reads similarly to `w4_alignment`, and writes a SAM file. The second step of this workflow iterates the SAM file searching for genomic variations and writes a VCF file. The third step reads the VCF file, annotates the genomic variations with known biological information, and write them to a new VCF file which is called annotated VCF. In our workflow implementation, this annotated VCF file sizes approximately 268MB.

w8_methylation. The last workflow comes from the epigenetics area, which studies phenotype changes that are not caused by modifications in the DNA sequence (the genotype). Such changes in gene expression can be caused by external factors, such as environmental conditions. A DNA methylation is a natural chemical process that change the expression of genes, for good (i.e., development) and for bad (i.e., diseases) [12]. Diverse studies are analyzing the effect of methylation in human development and aging.

The `w8_methylation` workflow sequentially reads a special sequencing data file, called Whole Genome Bi-sulfite Sequencing (WGBS), aligns all DNA reads similarly to `w4_alignment`, and writes a SAM file. The second step estimates the genotype and methylation status, and creates an intermediate file containing all this information, which sizes approximately 30MB. The final step of this workflow writes a list of candidate residues for differential methylation, which sizes 1MB in our case.

Workflows `w6_gwas`, `w7_annotation` and `w8_methylation` are considered complex workflows because they have more than one internal step, and extract meaningful results from genomes.

2.2 The FS-Biobench

The FS-Biobench is a file system benchmark based on the previously described bioinformatics workflows, and some of its design aspects are noteworthy. First, we focus on I/O tasks rather than on processing ones, which reflects the choice of replacing common CPU-intensive tasks by *dummy* processes that consume data as fast as possible. Second, we read and write synthetic data since our processing tasks ignore the meaning of data, but we use real file formats and sizes. Third, no input data is stored in the

local file system since it would only compromise benchmarking the target FS. Fourth, no reference genomes or databases are used in our benchmark because there is no real processing and this data would be obtained from the local file system or the internet at run time. Fifth, our benchmark consumes only one file per workflow step at time (e.g., a single coded genome), because reading more than one file concurrently would mostly increase the time for reading the files without contributing to evaluate the target FS.

The FS-Biobench was implemented using the Python programming language (version 3.2 or newer) and is publicly available as a free open-source software in the GitHub (https://github.com/vvcogo/fs-biobench) under the GNU general public license (GPL) version 3.0. FS-Biobench has at least two known limitations inherited from the macrobenchmark approach [1]: representativeness and isolation.

Users of FS-Biobench must carefully address three potential pitfalls: the standard deviation in results, the cache dirtiness, and the cache warm-up period [2]. First, network transfers and disk accesses may result in high standard deviation values. The less multi-tenancy the system has, the lower the standard deviation is. Possible solutions to this problem include guaranteeing FS-Biobench is running alone in the system, and executing our benchmark more times to reduce the impact of outliers. Second, several file systems use or implement cache mechanisms to reduce the latency of reads. If the cache already contains all required files, every read operation will cause a cache hit—biasing the latency results. FS-Biobench provides users an option to configure a command that cleans the cache between the execution of any two workflows. Third, the cache warm-up period happens when read requests cause several cache misses at the beginning of execution, until some cache hits are achieved. It may misguide the results' interpretation because there are two separated phases: the warm-up and the dirty cache, which cannot be considered as one. Similarly to the previous case, one may use the configurable inter-workflow command available in FS-Biobench to clean the cache and analyze the file system behavior during the cache warm-up period. Or, if it is the case, one may configure the mentioned command to make the cache dirty before logging the execution time.

REFERENCES

[1] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer, "Benchmarking file system benchmarking: It *IS* rocket science," in *HotOS*, 2011.

[2] A. Traeger, N. Joukov, C. P. Wright, and E. Zadok, "A nine year study of file system and storage benchmarking," *ACM Trans. on Storage*, vol. 4, no. 2, pp. 25–80, may 2008.

[3] E. R. Mardis, "The impact of next-generation sequencing technology on genetics," *Trends in genetics*, vol. 24, no. 3, pp. 133–141, 2008.

[4] D. G. Wang *et al.*, "Large-scale identification, mapping, and genotyping of single-nucleotide polymorphisms in the human genome," *Science*, vol. 280, no. 5366, pp. 1077–1082, 1998.

[5] D. A. Wheeler *et al.*, "The complete genome of an individual by massively parallel DNA sequencing," *Nature*, vol. 452, no. 7189, pp. 872–876, 2008.

[6] L. Norlin, M. N. Fransson, M. Eriksson, R. Merino-Martinez, M. Anderberg, S. Kurtovic, and J.-E. Litton, "A minimum data set for sharing biobank samples, information, and data: MIABIS," *Biopreservation and biobanking*, vol. 10, no. 4, pp. 343–348, 2012.

[7] H.-E. Wichmann *et al.*, "Comprehensive catalog of European biobanks," *Nature biotechnology*, vol. 29, no. 9, pp. 795–797, 2011.

[8] A. Hatem, D. Bozdağ, A. E. Toland, and Ü. V. Çatalyürek, "Benchmarking short sequence mapping tools," *BMC bioinformatics*, vol. 14, no. 1, p. 184, 2013.

[9] P. Flicek and E. Birney, "Sense from sequence reads: methods for alignment and assembly," *Nature methods*, vol. 6, pp. S6–S12, 2009.

[10] M. I. McCarthy, G. R. Abecasis, L. R. Cardon, D. B. Goldstein, J. Little, J. P. Ioannidis, and J. N. Hirschhorn, "Genome-wide association studies for complex traits: consensus, uncertainty and challenges," *Nature Reviews Genetics*, vol. 9, no. 5, pp. 356–369, 2008.

[11] D. G. MacArthur *et al.*, "A systematic survey of loss-of-function variants in human protein-coding genes," *Science*, vol. 335, no. 6070, pp. 823–828, 2012.

[12] H. Heyn *et al.*, "Distinct DNA methylomes of newborns and centenarians," *Proc. of the National Academy of Sciences*, vol. 109, no. 26, pp. 10 522–10 527, 2012.