Testing stateful, concurrent, and async systems using test.check

Eric Normand

PurelyFunctional.tv

Outline

- Example-based testing is inadequate
- Generating test data
- Generating sequential tests
- Adding parallelism

A stateful example

- A key-value database
  - Operations:

```
(db/create)

(db/clear!  db)
(db/store!  db k v)
(db/delete! db k)
(db/fetch   db k)
(db/size    db)
```

For the video and transcript of this presentation, click here:

https://lispcast.com/testing-stateful-and-concurrent-systems-using-test-check/

Let's test it!

DB should contain a key/value after storing

```
(deftest store-contains
)
```

DB should contain a key/value after storing

```
(deftest store-contains
  (let [db (db/create)
k “a”
v “b”]
                                    ))
```

DB should contain a key/value after storing

```
(deftest store-contains
  (let [db (db/create)
        k “a”
        v “b”]
    (db/store! db k v)
                                  ))
```

DB should contain a key/value after storing

```
(deftest store-contains
  (let [db (db/create)
        k “a”
        v “b”]
(db/store! db k v)
        (db/fetch db k)  ))
```

DB should contain a key/value after storing

```
(deftest store-contains
  (let [db (db/create)
        k “a”
        v “b”]
    (db/store! db k v)
    (is (= v (db/fetch db k)))))
```

store! should overwrite old values

```
(deftest store-overwrite
```

)

store! should overwrite old values

```
(deftest store-overwrite
  (let [db (db/create)
        k  “a”
        v1  “b”
        v2  “c”]
                                        ))
```

store! should overwrite old values

```
(deftest store-overwrite
  (let [db (db/create)
        k “a”
        v1 “b”
        v2 “c”]
    (db/store! db k v1)
    (db/store! db k v2)
                                  ))
```

store! should overwrite old values

```
(deftest store-overwrite
  (let [db (db/create)
        k “a”
        v1 “b”
        v2 “c”]
(db/store! db k v1)
(db/store! db k v2)
          (db/fetch db k)  ))
```

store! should overwrite old values

```
(deftest store-overwrite
  (let [db (db/create)
        k “a”
        v1 “b”
        v2 “c”]
(db/store! db k v1)
(db/store! db k v2)
(is (= v2 (db/fetch db k)))))
```

DB should be empty after clearing

```
(deftest clear-empty
```

```
)
```

DB should be empty after clearing

```
(deftest clear-empty
  (let [db (db/create)
k “a”
v “b”]
                                    ))
```

DB should be empty after clearing

```
(deftest clear-empty
  (let [db (db/create)
        k “a”
        v “b”]
    (db/store! db k v)
    (db/clear! db)
                                ))
```

DB should be empty after clearing

```
(deftest clear-empty
  (let [db (db/create)
        k “a”
        v “b”]
    (db/store! db k v)
    (db/clear! db)
               (db/size db)  ))
```

DB should be empty after clearing

```
(deftest clear-empty
  (let [db (db/create)
        k “a”
        v “b”]
    (db/store! db k v)
    (db/clear! db)
    (is (zero? (db/size db)))))
```

I don't want you to feel bad, but . . .

you should feel bad about these tests.

Not *guilty*, but scared.

How big is our system?

- How many strings are there?
- How many unicode characters are there?
- How many key-value pairs are there?
- How many operations are there?
- How many pairs of operations are there?
- How many triples of operations are there?

The *description* of the database is small.

Let's set up some generators

```
(def gen-key   gen/string)
(def gen-value gen/string)
```

```
> (gen/sample gen-key 20)

("" "Ï" "Û" "Ù" "þ7Ä" "Ì" "§" "þ8zÈäè" "" "Õ" "¢
öU÷¸" "W\b^è÷¯Ð\\" "\ngz|µþÒW." "ño" "ô>,
ß¡wA,\r!" ";ÊÑ²ãÔ9" "þèIà0TzJÜ\bi" "ó¥è¬#À&ö\\\"
ÈjF#" "u=?" "´ö")
```

Some easy properties...

DB should contain a key/value after storing

```
(defspec store-contains 100
```

```
)
```

DB should contain a key/value after storing

```
(defspec store-contains 100
  (prop/for-all [k gen-key
                 v gen-value]

                ))
```

DB should contain a key/value after storing

```
(defspec store-contains 100
  (prop/for-all [k gen-key
                 v gen-value]
(let [db (db/create)]

                          )))
```

DB should contain a key/value after storing

```
(defspec store-contains 100
  (prop/for-all [k gen-key
                 v gen-value]
(let [db (db/create)]
  (db/store! db k v)
                        )))
```

DB should contain a key/value after storing

```
(defspec store-contains 100
  (prop/for-all [k gen-key
                 v gen-value]
(let [db (db/create)]
  (db/store! db k v)
  (= v (db/fetch db k)))))
```

store! should overwrite old values

```
(defspec store-overwrite 100
```

)

store! should overwrite old values

```
(defspec store-overwrite 100
  (prop/for-all [k  gen-key
                 v1 gen-value
                 v2 gen-value]


                                ))
```

store! should overwrite old values

```
(defspec store-overwrite 100
  (prop/for-all [k  gen-key
                 v1 gen-value
                 v2 gen-value]
(let [db (db/create)]

                                 )))
```

store! should overwrite old values

```
(defspec store-overwrite 100
  (prop/for-all [k  gen-key
                 v1 gen-value
                 v2 gen-value]
(let [db (db/create)]
  (db/store! db k v1)
  (db/store! db k v2)
                          )))
```

store! should overwrite old values

```
(defspec store-overwrite 100
  (prop/for-all [k  gen-key
                 v1 gen-value
                 v2 gen-value]
(let [db (db/create)]
  (db/store! db k v1)
  (db/store! db k v2)
                          )))
```

store! should overwrite old values

```
(defspec store-overwrite 100
  (prop/for-all [k  gen-key
                 v1 gen-value
                 v2 gen-value]
(let [db (db/create)]
  (db/store! db k v1)
  (db/store! db k v2)
  (= v2 (db/fetch db k)))))
```

DB should be empty after clearing

```
(defspec clear-empty 100
  (prop/for-all [k gen-key
                 v gen-value]
(let [db (db/create)]
  (db/store! db k v)
  (db/clear! db)
  (zero? (db/size db)))))
```

```
{:result false,
 :test-var “store-contains",
 :failing-size 28,
 :num-tests 29,
:fail ["æ]qÜ\"Ì¹±W¿þZ¢Ëµgä>Å¨" “õZãºí®"],
:shrunk {:total-nodes-visited 139,
         :depth 33,
         :result false,
         :smallest ["æ" ""]},
:seed 1489522410083}
```

Can we describe the behavior in one go?

1. Build a simple, pure model

- A key-value database is like a *hash map*.

2. Reify the operations and make generators

```
(def gen-clear  (gen/return [:clear!]))
(def gen-size   (gen/return [:size]))
(def gen-store  (gen/tuple (gen/return :store!)
                           gen-key
                           gen-value))
(def gen-delete (gen/tuple (gen/return :delete!)
                           gen-key))
(def gen-fetch  (gen/tuple (gen/return :fetch)
                           gen-key))

(def gen-ops (gen/vector
              (gen/one-of [gen-clear  gen-store
                           gen-delete gen-fetch
                           gen-size])))
```

```
> (gen/sample gen-ops)

([]
 [[:clear!]]
 []
 [[:fetch "wº"] [:clear!]]
 [[:clear!] [:fetch "*QZü"] [:clear!] [:fetch “¤'"]]
 [[:size] [:size] [:delete! “K]¨j"]]
 []
 [[:fetch “t"]]
 [[:fetch "§6"] [:size] [:size] [:clear!]]
 [[:fetch "P/7¹"] [:store! "Þ=" ""] [:delete! "Â"]
  [:store! "B" "¬Ê'y"]])
```

3. Make 2 "runners"

```
(defn db-run [db ops]
```

)

3. Make 2 "runners"

```
(defn db-run [db ops]
  (doseq [[op k v] ops]
```

))

3. Make 2 "runners"

```
(defn db-run [db ops]
  (doseq [[op k v] ops]
    (case op
```



```
)))
```

3. Make 2 "runners"

```
(defn db-run [db ops]
  (doseq [[op k v] ops]
    (case op
      :clear!  (db/clear!  db)
```

)))

3. Make 2 "runners"

```
(defn db-run [db ops]
  (doseq [[op k v] ops]
    (case op
      :clear!  (db/clear!  db)
```

)))

3. Make 2 "runners"

```
(defn db-run [db ops]
  (doseq [[op k v] ops]
    (case op
      :clear!  (db/clear!  db)
      :size    (db/size    db)

                              )))
```

3. Make 2 "runners"

```
(defn db-run [db ops]
  (doseq [[op k v] ops]
    (case op
      :clear!  (db/clear!  db)
      :size    (db/size    db)
      :store!  (db/store!  db k v)
      :delete! (db/delete! db k)
      :fetch   (db/fetch   db k))))
```

3. Make 2 "runners"

```
(defn hm-run [db ops]
  (reduce
    (fn [hm [op k v]]
                    )
db ops))
```

3. Make 2 "runners"

```
(defn hm-run [db ops]
  (reduce
    (fn [hm [op k v]]
      (case op
                  ))
db ops))
```

3. Make 2 "runners"

```
(defn hm-run [db ops]
  (reduce
    (fn [hm [op k v]]
      (case op
        :clear!  {}


                 ))
db ops))
```

3. Make 2 "runners"

```
(defn hm-run [db ops]
  (reduce
    (fn [hm [op k v]]
      (case op
        :clear!  {}
        :size    hm

                 ))
db ops))
```

3. Make 2 "runners"

```
(defn hm-run [db ops]
  (reduce
    (fn [hm [op k v]]
      (case op
        :clear!  {}
        :size    hm
        :store!  (assoc  hm k v)

                 ))
db ops))
```

3. Make 2 "runners"

```
(defn hm-run [db ops]
  (reduce
    (fn [hm [op k v]]
      (case op
        :clear!  {}
        :size    hm
        :store!  (assoc  hm k v)
        :delete! (dissoc hm k)
                 ))
db ops))
```

3. Make 2 "runners"

```
(defn hm-run [db ops]
  (reduce
    (fn [hm [op k v]]
      (case op
        :clear!  {}
        :size    hm
        :store!  (assoc  hm k v)
        :delete! (dissoc hm k)
        :fetch   hm))
db ops))
```

4.Define your property

```
(defspec hash-map-equiv 100
  (prop/for-all [ops gen-ops]



                        ))
```

4.Define your property

```
(defspec hash-map-equiv 100
  (prop/for-all [ops gen-ops]
    (let [hm (hm-run {} ops)
          db (db/create)]

                    )))
```

4.Define your property

```
(defspec hash-map-equiv 100
  (prop/for-all [ops gen-ops]
    (let [hm (hm-run {} ops)
          db (db/create)]
      (db-run db ops)
                  )))
```

4.Define your property

```
(defspec hash-map-equiv 100
  (prop/for-all [ops gen-ops]
    (let [hm (hm-run {} ops)
          db (db/create)]
      (db-run db ops)
      (equiv? db hm))))
```

4.Define your property

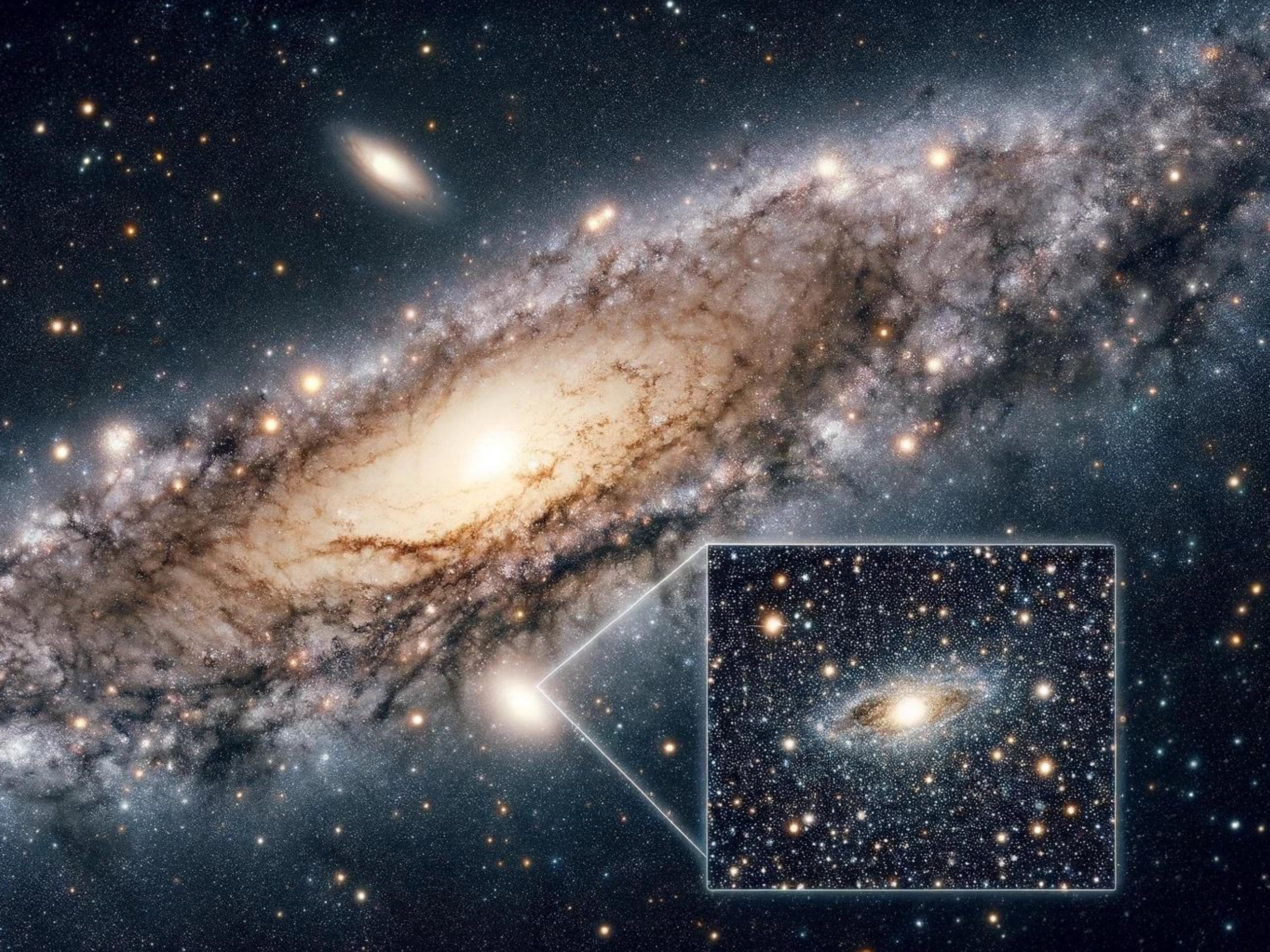
```
(defn equiv? [db hm]
                )

(defspec hash-map-equiv 100
  (prop/for-all [ops gen-ops]
    (let [hm (hm-run {} ops)
          db (db/create)]
      (db-run db ops)
      (equiv? db hm))))
```

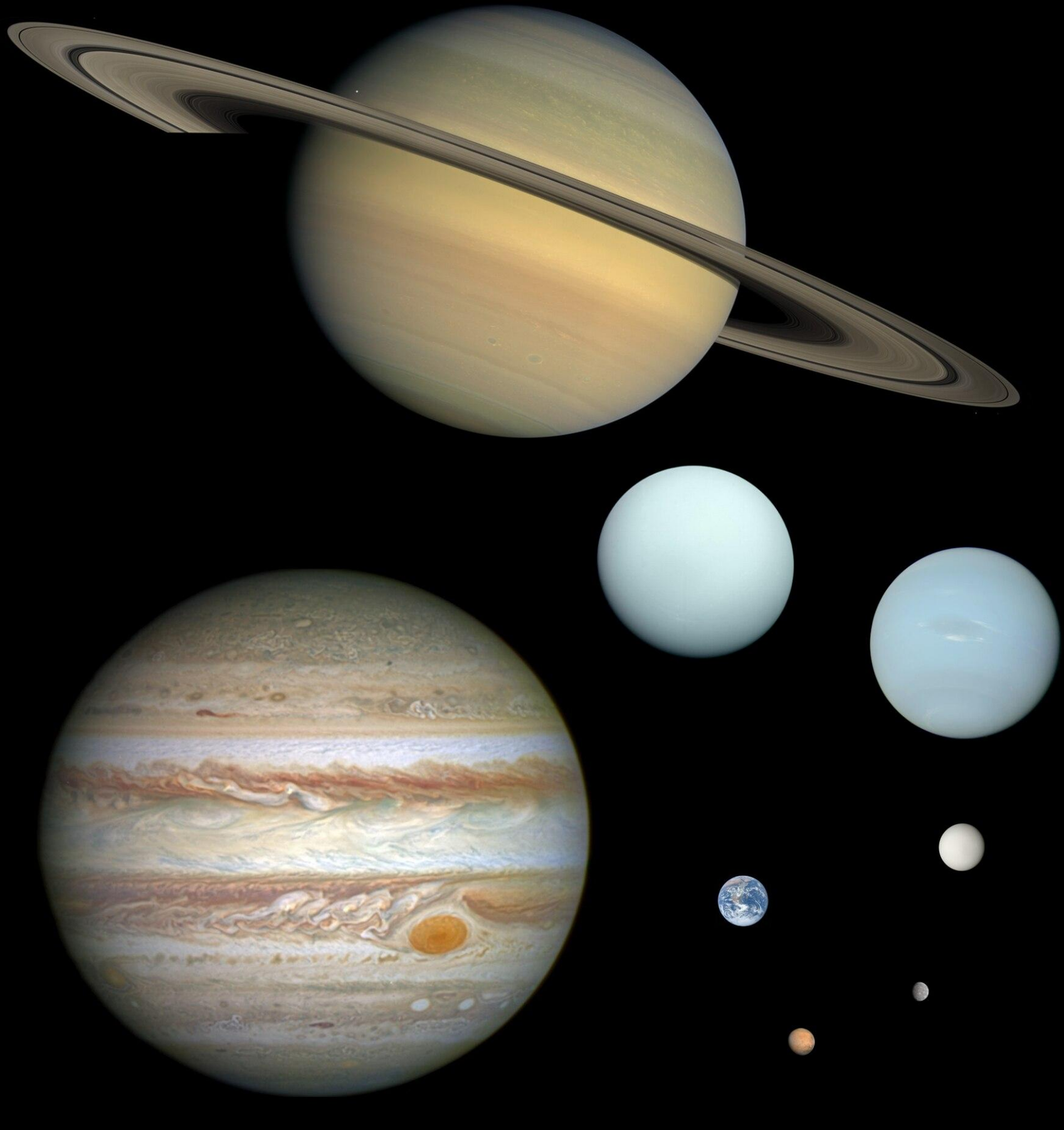
4.Define your property

```
(defn equiv? [db hm]
  (and (= (count hm) (db/size db))
       ))

(defspec hash-map-equiv 100
  (prop/for-all [ops gen-ops]
    (let [hm (hm-run {} ops)
          db (db/create)]
      (db-run db ops)
      (equiv? db hm))))
```

4. Define your property

```
(defn equiv? [db hm]
  (and (= (count hm) (db/size db))
       (every? (fn [[k v]]
                 (= v (db/fetch db k)))
               hm)))

(defspec hash-map-equiv 100
  (prop/for-all [ops gen-ops]
    (let [hm (hm-run {} ops)
          db (db/create)]
      (db-run db ops)
      (equiv? db hm))))
```





Encourage collisions

```
(def gen-clear  (gen/return [:clear!]))
(def gen-size   (gen/return [:size]))

(defn gen-store [keys]
 (gen/tuple (gen/return :store!)   (gen/elements keys) gen-value))

(defn gen-delete [keys]
  (gen/tuple (gen/return :delete!) (gen/elements keys)))

(defn gen-fetch [keys]
  (gen/tuple (gen/return :fetch)   (gen/elements keys)))

(defn gen-ops* [keys]
  (gen/vector
    (gen/one-of [gen-clear          (gen-store keys)
                 (gen-delete keys) (gen-fetch keys)
                 gen-size])))

(def gen-ops (gen/let [keys (gen/not-empty (gen/vector gen-key))]
               (gen-ops* keys))
```

```
> (gen/sample gen-ops)

([]
 []
 [[:fetch ""] [:size]]
 [[:fetch "w?"] [:clear!]]
 [[:delete! “Z¿"]]
 [[:fetch "ü"] [:fetch "ü"] [:size] [:clear!]]
 []
 [[:store! ")Á@" "k"] [:store! ")Á@" "c"] [:fetch "G,"]
  [:clear!]]
 [[:clear!]]
 [[:store! "¸w·§" "ý}"] [:clear!] [:fetch " `¢þ"]
  [:fetch "Ï*]p®_"] [:delete! " `¢þ"]])
```

```
> (apply max (map count (gen/sample gen-ops 100)))
91

> (apply max (map count (gen/sample gen-ops 1000)))
96

> (Math/pow 5 91)
4.038967834731581E63
```

49 operations

```
{:result false,
 :test-var "hash-map-equiv",
 :seed 1489523387287,
 :failing-size 26,
 :num-tests 27,
 :fail [[[:delete! "9kàµ¾l!9ÞÀglÐÅF¨æy8ì)"] [:size] [:delete! "
\re¯Ú²<×/Ho^|üç6lÉÊ"] [:clear!] [:size] [:clear!] [:delete! "t_®cWuPû"] [:size] [:clear!] [:fetch
"B·7{ÎÔ"] [:clear!] [:size] [:delete! "¥?t·í\\Â\fZ"] [:clear!] [:clear!] [:delete! "Ê5Þí§
uÉVzÓâH½ëì¥W#6"] [:size] [:size] [:size] [:delete! "ku"] [:delete! "Me±àÜJzCw ²²¾Yþ¬£~£ü¡ÁQW\tU"]
[:delete! "9kàµ¾l!9ÞÀglÐÅF¨æy8ì)"] [:delete! "ËKssy¸Îe>°Æµ¶qí3
sAËx¹\fø\\tdB§¦"] [:size] [:store! "V%tCÀx0û{½¼ô5·zØUy¦\nåöþ§,òUCdGÁBl" "²ìÒ¿¸Ø
1Ö¹z¥²\fl\">GÌ?2 ¹3¢ÚÔ<¹°A·ç"] [:clear!] [:clear!] [:clear!] [:fetch "è²Ú'ÃË·\\Ý÷-Þ(äkÿ2bß'FdóN²O
£ÂH×"] [:delete! "Z¹5ýuá`&v"] [:store! "¥?t·í\\Â\fZ" ""] [:store! "ËKssy¸Îe>°Æµ¶qí3
sAËx¹\fø\\tdB§¦" "\rUmfûR¢)2õ´Ùß rÄ«ûbåÙ¶~±9"] [:delete! "ö§åRÁj÷9Èê³ÈxÎÂÐX³RB\""] [:delete!
"t_®cWuPû"] [:clear!] [:delete! "®âl¾Tùwl°|Çi"] [:store! "ÏPtDá\r\"jR
Îm]AÜ°SÊ¨É?kõfØ«WÌX·\tÚ~ßCm0hÉ" "ïIô£·¨²¬x\btatøýóS)¶h)É9FÕ&"] [:fetch "e#ðE"] [:fetch "t_®cWuPû"]
[:delete! "(¼"] [:clear!] [:fetch "ö§åRÁj÷9Èê³ÈxÎÂÐX³RB\""] [:fetch "è²Ú'ÃË·\\Ý÷-Þ(äkÿ2bß'FdóN²O
£ÂH×"] [:delete! "Z¹5ýuá`&v"] [:size] [:store! "úöYø
g¶Ò\r@¹òÕï" "3£>°Sn"] [:delete! "Oc®ªÄia"] [:store! "}Üòò\r" "8ª$8,QøÛN+,Ø£kÐ)°ñ4½THA{öNó÷qñ\\FÇ
W"] [:store! "5BÑþ9¿³'<TáÓBföQ" "×½Ut*Æ#B»GÑÝX¢taN'tºí"]]],
 :shrunk {:total-nodes-visited 361,
          :depth 162,
          :result false,
          :smallest [[[:store! "ø" ""]]]}}
```

But what about race conditions?

Run it in multiple threads

```
(defn run-in-thread [db ops]
  (.start (Thread. (fn []
                     (db-run db ops)))))
```

```
(defn thread-run [db ops-sequences]
  (run! #(run-in-thread db %) ops-sequences))
```

Wait for them all to finish

```
(defn run-in-thread [db ops]
  (let [done (promise)]
    (.start (Thread. (fn []
                       (db-run db ops)
                       (deliver done :done!))))
    done))
```

Wait for them all to finish

```
(defn run-in-thread [db ops]
  (let [done (promise)]
    (.start (Thread. (fn []
                       (db-run db ops)
                       (deliver done :done!))))
    done))

(defn thread-run [db ops-sequences]
  (let [threads (map #(run-in-thread db %)
                     ops-sequences)]
    (dorun threads)
    (run! deref threads))
```

Start all threads at once

```
(defn run-in-thread [latch db ops]
  (let [done (promise)]
    (.start (Thread. (fn []
                       @latch
                       (db-run db ops)
                       (deliver done :done!))))
    done))
```

Start all threads at once

```
(defn run-in-thread [latch db ops]
  (let [done (promise)]
    (.start (Thread. (fn []
                       @latch
                       (db-run db ops)
                       (deliver done :done!))))
    done))

(defn thread-run [db ops-sequences]
  (let [latch (promise)
        threads (map #(run-in-thread latch db %)
                     ops-sequences)]
    (dorun threads)
    (deliver latch :go!)
    (run! deref threads)))
```

Test against the model

```
(defspec hash-map-equiv 100
```

```
)
```

Test against the model

```
(defspec hash-map-equiv 100
  (prop/for-all [ops-a gen-ops
                 ops-b gen-ops]
```

```
))
```

Test against the model

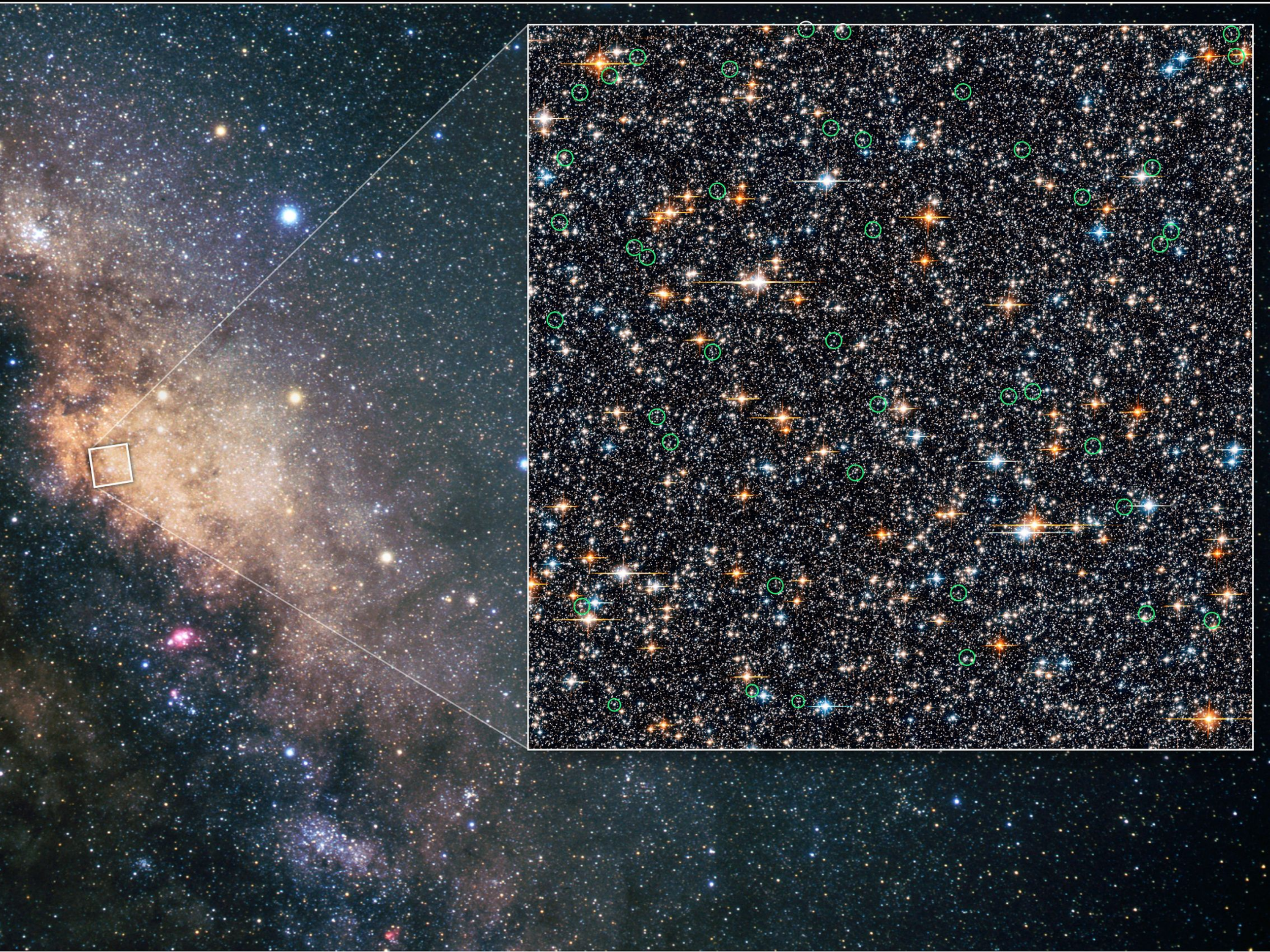
```
(defspec hash-map-equiv 100
  (prop/for-all [ops-a gen-ops
                 ops-b gen-ops]
    (let [ops (concat ops-a ops-b)
          ]

    )))
```

Test against the model

```
(defspec hash-map-equiv 100
  (prop/for-all [ops-a gen-ops
                 ops-b gen-ops]
    (let [ops (concat ops-a ops-b)
          hm  (hm-run {} ops)
          db  (db/create)]

      )))
```

Test against the model

```
(defspec hash-map-equiv 100
  (prop/for-all [ops-a gen-ops
                 ops-b gen-ops]
    (let [ops (concat ops-a ops-b)
          hm  (hm-run {} ops)
          db  (db/create)]
(thread-run db [ops-a ops-b])
(equiv? db hm))))
```



Encourage collisions across threads

```
(defn gen-ops-sequences [n]
)
```

Encourage collisions across threads

```
(defn gen-ops-sequences [n]
  (gen/let [keys (gen/not-empty
                  (gen/vector gen-key))]
```

Encourage collisions across threads

```
(defn gen-ops-sequences [n]
  (gen/let [keys (gen/not-empty
                  (gen/vector gen-key))]
    (apply gen/tuple
           (repeat n (gen-ops* keys)))))
```

Collisions

```
(defspec hash-map-equiv 100
  (prop/for-all [[ops-a ops-b]
                 (gen-ops-sequences 2)]
```

```
))
```

Collisions

```
(defspec hash-map-equiv 100
  (prop/for-all [[ops-a ops-b]
                 (gen-ops-sequences 2)]
    (let [ops (concat ops-a ops-b)
          hm  (hm-run {} ops)
          db  (db/create)]
(thread-run db [ops-a ops-b])
(equiv? db hm))))
```

Time

A

B

DB

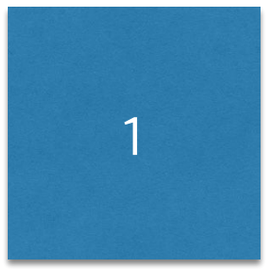
1

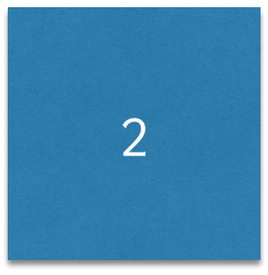
1

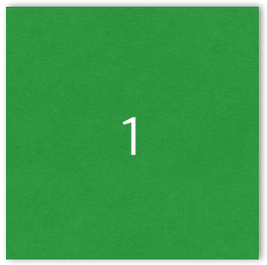
?

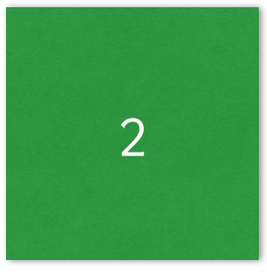
2

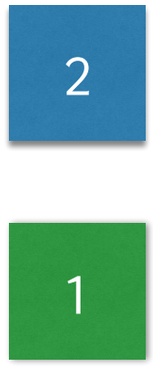
2











Time

A

1

2

B

1

2

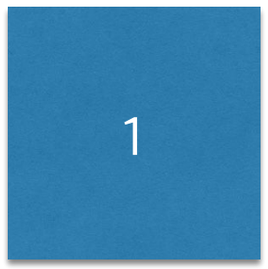
DB

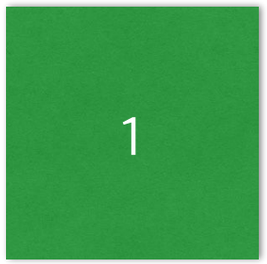
1

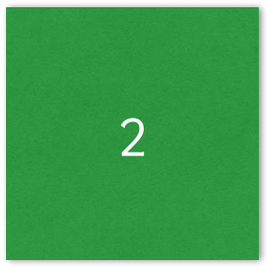
2

1

2







Time

A

1

2

B

1

2

DB

1

2

1

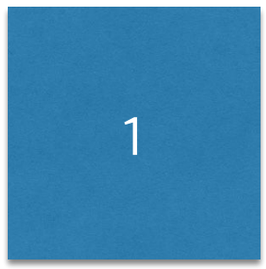
2

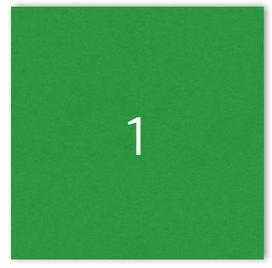
Time

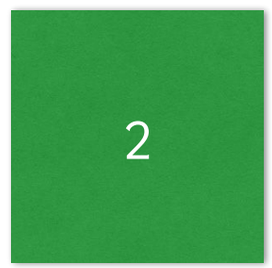
A

B

DB





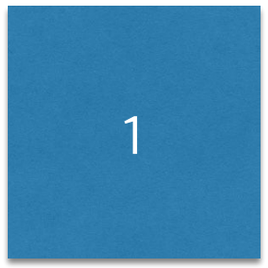


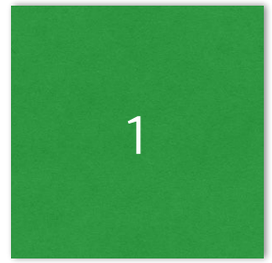
Time

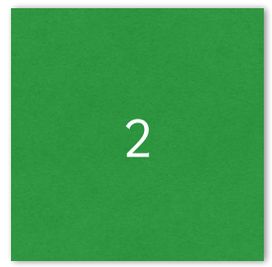
A

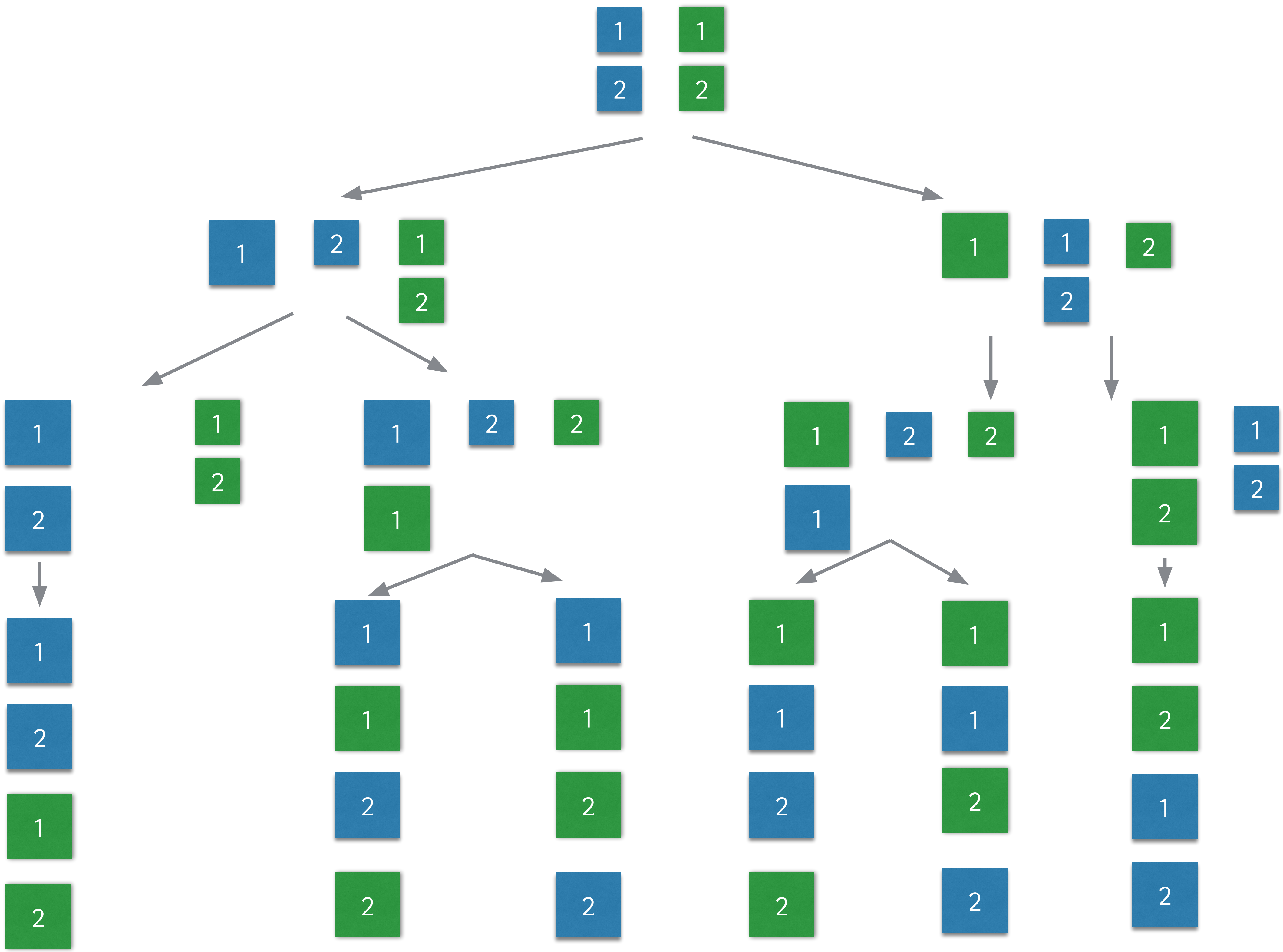
B

DB









1
1
2
2
1
2
1
2
1
1
2
2
1
2
1
2
1
2
2
1
1
2
2
1
1
1
2
2
1
2
1
2
1
1
2
2
1
1
2
2
1
1
2
2
1
1
2
2
1
2
1
2

Possible interleavings

```
(defn children [{:keys [sequence threads]}]
  (for [[k [v & thread]] threads]
    {:sequence (conj sequence v)
     :threads (if thread
                (assoc  threads k thread)
                (dissoc threads k))}))

(defn branch? [x]
  (-> x :threads not-empty))

(defn possible-interleavings [& sequences]
  (let [threads (into {} (map vector (range) sequences))]
    (->>
     (tree-seq branch? children {:sequence [] :threads threads})
     (remove branch?)
     (map :sequence))))
```

Equivalent to some possible interleaving

```
(defspec hash-map-equiv 100
  (prop/for-all [[ops-a ops-b]
                 (gen-ops-sequences 2)]
    (let [ops-i (possible-interleavings ops-a
                                        ops-b)
          ]
                                              )))
```

Equivalent to some possible interleaving

```
(defspec hash-map-equiv 100
  (prop/for-all [[ops-a ops-b]
                 (gen-ops-sequences 2)]
    (let [ops-i (possible-interleavings ops-a
                                        ops-b)
          db (db/create)]
      (thread-run db [ops-a ops-b])

                                          ))))
```

Equivalent to some possible interleaving

```
(defspec hash-map-equiv 100
  (prop/for-all [[ops-a ops-b]
                 (gen-ops-sequences 2)]
    (let [ops-i (possible-interleavings ops-a
                                        ops-b)
          db (db/create)]
      (thread-run db [ops-a ops-b])
      (some? #(equiv? db %)
        (map #(hm-run {} %) ops-i))))))
```

Repeatability (every?)

```
(defspec hash-map-equiv 100
  (prop/for-all [[ops-a ops-b]
                 (gen-ops-sequences 2)]
    (let [ops-i (possible-interleavings ops-a
                                        ops-b)]
      (every?
        (for [_ (range 10)]
          (let [db (db/create)]
            (thread-run db [ops-a ops-b])
            (some? #(equiv? db %)
              (map #(hm-run {} %) ops-i)))))))))
```

A

```
[:store! “a” “b”]
[:store! “a” “c”]
```

B

```
[:fetch “”]
[:delete! “”]
[:delete! “”]
[:size]
[:delete! “”]
[:fetch “”]
[:fetch “”]
[:size]
[:fetch “”]
…
[:delete! “a”]
```

Timing

```
(def gen-sleep (gen/tuple (gen/return :sleep)
                          (gen/choose 1 100)))

(defn gen-ops* [keys]
  (gen/vector
    (gen/one-of [gen-sleep
                 gen-size
                 (gen-fetch  keys)
                 (gen-store  keys)
                 (gen-delete keys)
                 gen-clear])))
```

DB Runner

```
(defn db-run [db ops]
  (doseq [[op k v] ops]
    (case op
      :sleep   (Thread/sleep k)
      :clear!  (db/clear!  db)
      :size    (db/size    db)
      :store!  (db/store!  db k v)
      :delete! (db/delete! db k)
      :fetch   (db/fetch   db k))))
```

Hash map runner

```
(defn hm-run [db ops]
  (reduce
    (fn [hm [op k v]]
      (case op
      :sleep   hm
      :clear!  {}
      :size    hm
      :store!  (assoc  hm k v)
      :delete! (dissoc hm k)
      :fetch   hm))
db ops))
```

A

```
[:store! “a” “b”]
[:store! “a” “c”]
```

B

```
[:sleep 66]
[:delete! “a”]
```

```
(defspec store-fetch 100
  (check/for-all [k gen-key
                  v gen-value]
    (let [db (db/create)]
      (db/store! db k v)
      (db/fetch db k)
      (= v (db/fetch db k)))))

(defspec store-store 100
  (check/for-all [k gen-key
                  v1 gen-key
                  v2 gen-key]
    (let [db (db/create)]
      (db/store! db k v1)
      (db/store! db k v2)
      (= v2 (db/fetch db k)))))

(defspec store-clear 100
  (check/for-all [k gen-key
                  v gen-value]
    (let [db (db/create)]
      (db/store! db k v)
      (db/clear! db)
      (zero? (db/size db)))))

(defspec store-delete 100
  (check/for-all [k gen-key
                  v gen-value]
    (let [db (db/create)]
      (db/store!  db k v)
      (db/delete! db k)
      (zero? (db/size db)))))

(defspec store-size 100
  (check/for-all [k gen-key
                  v gen-value]
    (let [db (db/create)]
      (db/store!  db k v)
      (db/size db)
      (= 1 (db/size db)))))

(defspec fetch-fetch 100
  (check/for-all [k1 gen-key
                  k2 gen-key]
    (let [db (db/create)]
      (db/fetch db k1)
      (db/fetch db k1)
      (zero? (db/size db)))))

(defspec fetch-store 100
  (check/for-all [k gen-key
                  v1 gen-key
                  v2 gen-key]
    (let [db (db/create)]
      (db/fetch db k)
      (db/store! db k v2)
      (= v2 (db/fetch db k)))))

(defspec fetch-clear 100
  (check/for-all [k gen-key
                  v gen-value]
    (let [db (db/create)]
      (db/fetch! db k)
      (db/clear! db)
      (zero? (db/size db)))))

(defspec fetch-delete 100
  (check/for-all [k gen-key
                  v gen-value]
    (let [db (db/create)]
      (db/fetch!  db k)
      (db/delete! db k)
      (zero? (db/size db)))))

(defspec fetch-size 100
  (check/for-all [k gen-key
                  v gen-value]
    (let [db (db/create)]
      (db/fetch  db k)
      (db/size db)
      (zero? (db/size db)))))

(defspec fetch-fetch 100
  (check/for-all [k1 gen-key
                  k2 gen-key]
    (let [db (db/create)]
      (db/fetch db k1)
      (db/fetch db k1)
      (zero? (db/size db)))))

(defspec fetch-store 100
  (check/for-all [k gen-key
                  v1 gen-key
                  v2 gen-key]
    (let [db (db/create)]
      (db/fetch db k)
      (db/store! db k v2)
      (= v2 (db/fetch db k)))))

(defspec fetch-clear 100
  (check/for-all [k gen-key
                  v gen-value]
    (let [db (db/create)]
      (db/fetch! db k)
      (db/clear! db)
      (zero? (db/size db)))))

(defspec fetch-delete 100
  (check/for-all [k gen-key
                  v gen-value]
    (let [db (db/create)]
      (db/fetch!  db k)
      (db/delete! db k)
      (zero? (db/size db)))))

(defspec fetch-size 100
  (check/for-all [k gen-key
                  v gen-value]
    (let [db (db/create)]
      (db/fetch  db k)
      (db/size db)
      (zero? (db/size db)))))

(defspec fetch-fetch 100
  (check/for-all [k1 gen-key
                  k2 gen-key]
    (let [db (db/create)]
      (db/fetch db k1)
      (db/fetch db k1)
      (zero? (db/size db)))))

(defspec fetch-store 100
  (check/for-all [k gen-key
                  v1 gen-key
                  v2 gen-key]
    (let [db (db/create)]
      (db/fetch db k)
      (db/store! db k v2)
      (= v2 (db/fetch db k)))))

(defspec fetch-clear 100
  (check/for-all [k gen-key
                  v gen-value]
    (let [db (db/create)]
      (db/fetch! db k)
      (db/clear! db)
      (zero? (db/size db)))))

(defspec fetch-delete 100
  (check/for-all [k gen-key
                  v gen-value]
    (let [db (db/create)]
      (db/fetch!  db k)
      (db/delete! db k)
      (zero? (db/size db)))))

(defspec fetch-size 100
  (check/for-all [k gen-key
                  v gen-value]
    (let [db (db/create)]
      (db/fetch  db k)
      (db/size db)
      (zero? (db/size db)))))

(defspec fetch-fetch 100
  (check/for-all [k1 gen-key
                  k2 gen-key]
    (let [db (db/create)]
      (db/fetch db k1)
      (db/fetch db k1)
      (zero? (db/size db)))))

(defspec fetch-store 100
  (check/for-all [k gen-key
                  v1 gen-key
                  v2 gen-key]
    (let [db (db/create)]
      (db/fetch db k)
      (db/store! db k v2)
      (= v2 (db/fetch db k)))))

(defspec fetch-clear 100
  (check/for-all [k gen-key
                  v gen-value]
    (let [db (db/create)]
      (db/fetch! db k)
      (db/clear! db)
      (zero? (db/size db)))))

(defspec fetch-delete 100
  (check/for-all [k gen-key
                  v gen-value]
    (let [db (db/create)]
      (db/fetch!  db k)
      (db/delete! db k)
      (zero? (db/size db)))))

(defspec fetch-size 100
  (check/for-all [k gen-key
                  v gen-value]
    (let [db (db/create)]
      (db/fetch  db k)
      (db/size db)
      (zero? (db/size db)))))
```

```
(def gen-key   gen/string)
(def gen-value gen/string)

(def gen-clear  (gen/return [:clear!]))
(def gen-size   (gen/return [:size]))
(defn gen-store [keys]
  (gen/tuple (gen/return :store!)  (gen/elements keys) gen-value))
(defn gen-delete [keys]
  (gen/tuple (gen/return :delete!) (gen/elements keys)))
(defn gen-fetch [keys]
  (gen/tuple (gen/return :fetch) (gen/elements keys)))
(def gen-sleep (gen/tuple (gen/return :sleep) (gen/choose 1 10)))

(defn gen-ops* [keys]
  (gen/vector
    (gen/one-of [gen-sleep
                 gen-size
                 (gen-fetch  keys)
                 (gen-store  keys)
                 (gen-delete keys)
                 gen-clear])))

(defn gen-ops-sequences [n]
  (gen/let [keys (gen/not-empty (gen/vector gen-key))]
    (apply gen/tuple (repeat n (gen-ops* keys)))))

(defn db-run [db ops]
  (doseq [op ops]
    (match [op]
      [[:sleep   n  ]] (Thread/sleep n)
      [[:clear!     ]] (db/clear!  db)
      [[:size       ]] (db/size    db)
      [[:store!  k v]] (db/store!  db k v)
      [[:delete! k  ]] (db/delete! db k)
      [[:fetch   k  ]] (db/fetch   db k))))

(defn hm-run [db ops]
  (reduce
    (fn [db op]
      (match [op]
        [[:sleep   n  ]] db
        [[:clear!     ]] (hm-clear!  db)
        [[:size       ]] (hm-size    db)
        [[:store!  k v]] (hm-store!  db k v)
        [[:delete! k  ]] (hm-delete! db k)
        [[:fetch   k  ]] (hm-fetch   db k)))
    db ops))

(defn equiv? [db hm]
  (and (= (count hm) (db/size db))
       (every? (fn [[k v]]
                 (= v (db/fetch db k)))
               hm)))

(defn run-in-thread [latch db ops]
  (let [done (promise)]
    (.start (Thread. (fn []
                       @latch
                       (db-run db ops)
                       (deliver done :done!))))
   done))

(defn thread-run [db ops-sequences]
  (let [latch (promise)
        threads (map #(run-in-thread latch db %) ops-sequences)]
    (dorun threads)
    (deliver latch :go!)
    (run! deref threads)))

(defn children [{:keys [sequence threads]}]
  (for [[k [v & thread]] threads]
    {:sequence (conj sequence v)
     :threads (if thread
                (assoc threads k thread)
                (dissoc threads k))}))

(defn branch? [x]
  (-> x :threads not-empty))

(defn possible-interleavings [& sequences]
  (let [threads (into {} (map vector
                              (range (count sequences))
                              sequences))]
    (->>
     (tree-seq branch? children {:sequence [] :threads threads})
     (remove branch?)
     (map :sequence))))

(defspec hash-map-equiv
  100
  (check/for-all [[ops-s ops-a ops-b] (gen-ops-sequences 3)]
    (let [ops-i (possible-interleavings ops-a ops-b)
          hm (hm-run (hm-create) ops-s)]
      (prn ops-s ops-a ops-b)
      (every? identity
              (for [_ (range 10)]
                (let [db (db/create)]
                  (db-run db ops-s)
                  (thread-run db [ops-a ops-b])
                  (some? (map #(equiv? db (hm-run hm %)) ops-i))))))))
```

Eric Normand

LispCast

Follow Eric on:

Eric Normand

@EricNormand

lispcast.com

eric@lispcast.com