The Never Changing Face of Immutability

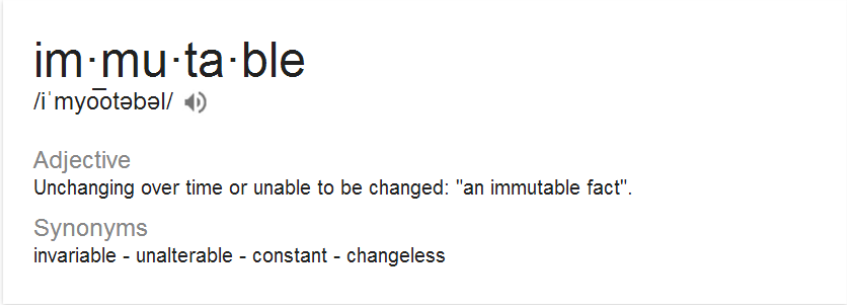
Chris Howe-Jones

15th December 2015

Warning!!

- There will be a Lisp!
- There will be Entomology!
- There will be History!
- 1st law of Clojure talks
- Any talk with Clojure in it must have some entomology

The Never Changing Face of Immutability



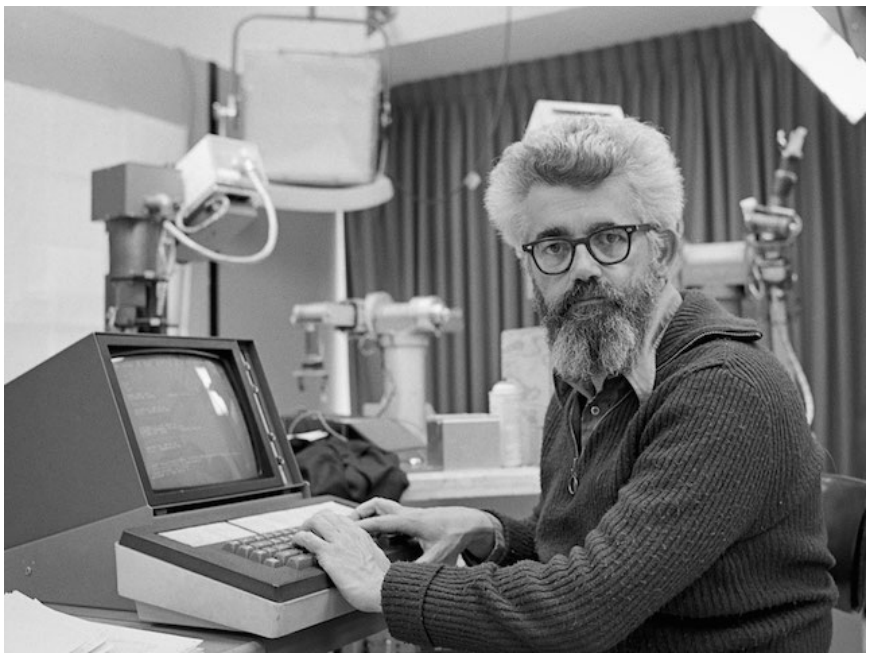
Who am I?

Name: `Chris Howe-Jones`
Job Title: `Technical Navigator`
Twitter: `@agile_geek`
Github: `github.com/chrishowejones`
Blog: `chrishowejones.wordpress.com`

Credentials

- 28 years of pushing data around
- Procedural/OOP/FP
- Architecture & Design
- RAD/Agile/Lean
- CTO

History Lesson



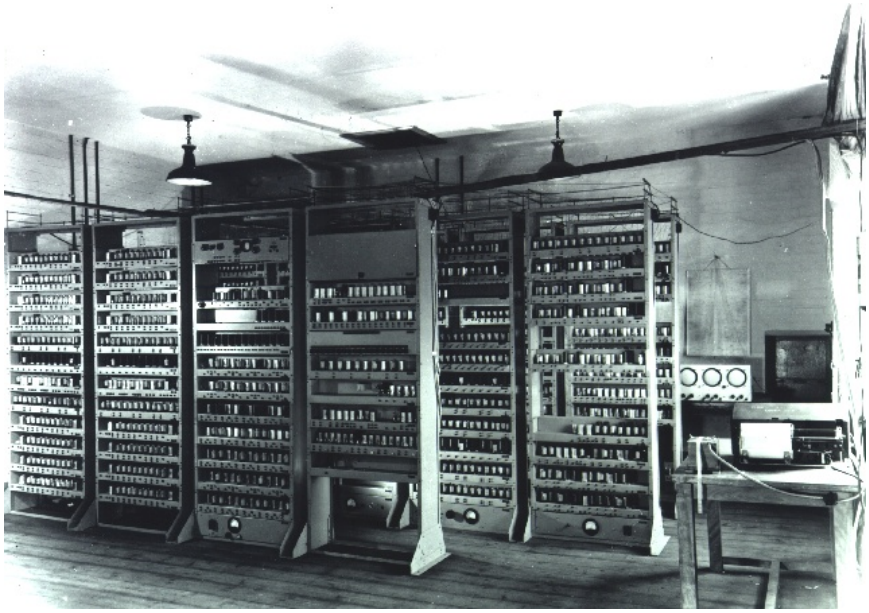
- Who is this?
- John McCarthy
  - developed Lisp
  - influenced design of ALGOL
  - invented GC
  - created term AI
  - first to suggest publicly the idea of utility computing
  - credited with developing an early form of time-sharing

Once upon a time..

Book Keeping

- List of entries in a ledger
- No 'crossing out'!

Dawn of Computing



- Math
- Transient storage
- EDSAC - Electronic Delay Storage Automatic Calculator
- Cambridge 1949 - early general purpose electronic programmable computer (ENIAC 1946 was 1st)
- Storage - mecury delay lines, derated vacuum tubes for logic
- n 1950, M. V. Wilkes and Wheeler used EDSAC to solve a differential equation relating to gene frequencies in a paper by Ronald Fisher. This represents the first use of a computer for a problem in the field of biology.
- In 1951, Miller and Wheeler used the machine to discover a 79-digit prime – the largest known at the time.
- In 1952, Sandy Douglas developed OXO, a version of noughts and crosses (tic-tac-toe) for the EDSAC, with graphical output to a VCR97 6" cathode ray tube. This may well have been the world's first video game.

60's-90's

- Spot the expense?
- Memory
- Tape
- Disk

21st Century

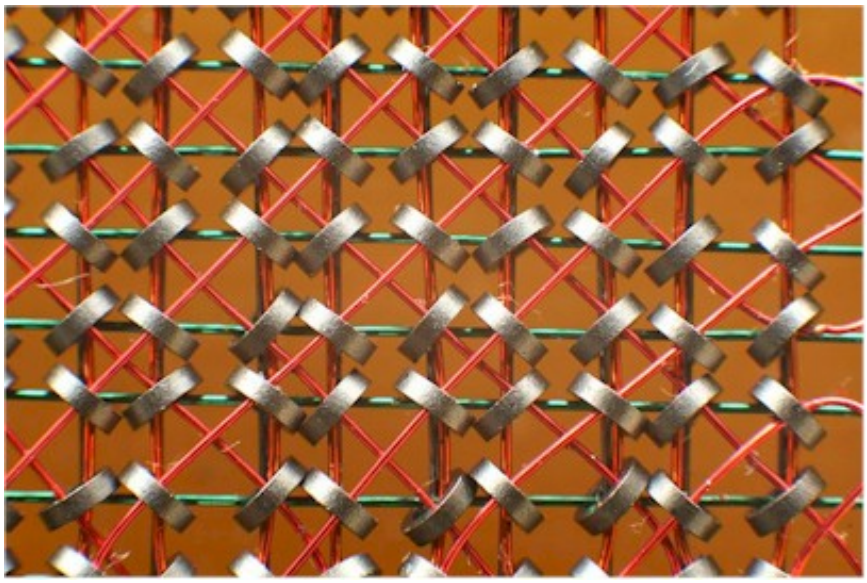
Spot the expense?

- Developers

Cheap resources: SSD/Disk, Memory, CPU

And..

In place computing



- Update data in place
- Reuse expensive real estate
- Magnetic core memory 1955-75
- Core uses tiny magnetic toroids (rings), the cores, through which wires are threaded to write and read information.
- Each core represents one bit of information.
- Magnetized in 2 directions (clockwise/counterclockwise) to represent 1 or 0

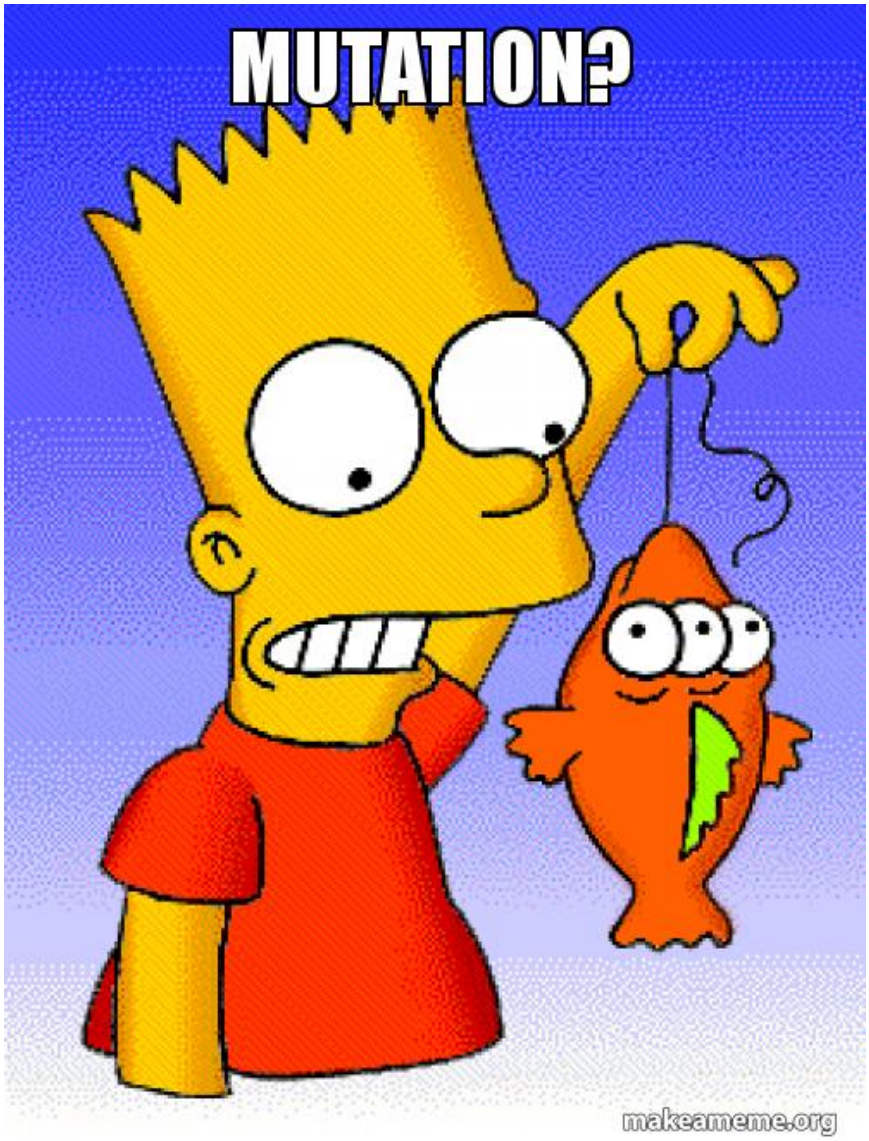
RDBMS

- Data updated
- Values overwritten
- Reuse memory and disk
- Disk pack - invented 1965
- IBM Engineers - Thomas G. Leary and R. E. Pattison
- Probably about 50MB on this one.

Result?

In place oriented programming (PLOP) relies on...

Mutation



Which leads to..

complect

transitive verb

Definition of COMPLECT

Popularity: Bottom 20% of words

obsolete
: INTERTWINE, EMBRACE; *especially* : to plait together : INTERWEAVE

Complect

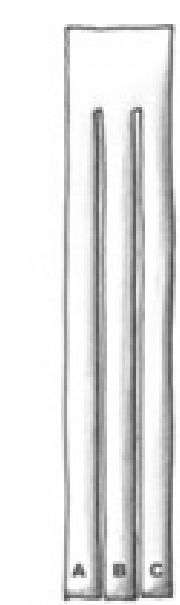


fig. 1

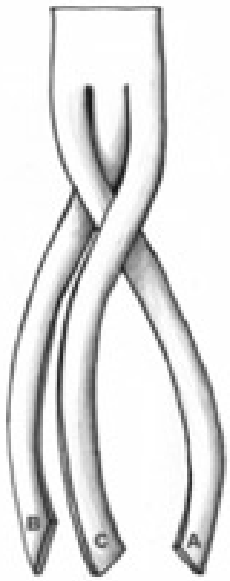


fig. 2

fig. 3

fig. 4

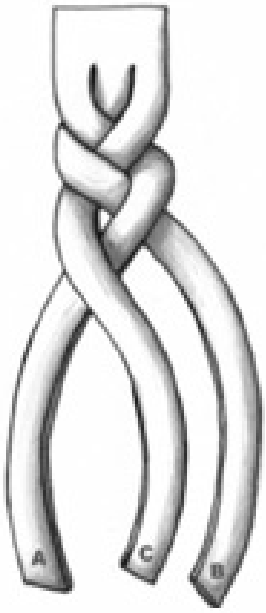


fig. 5

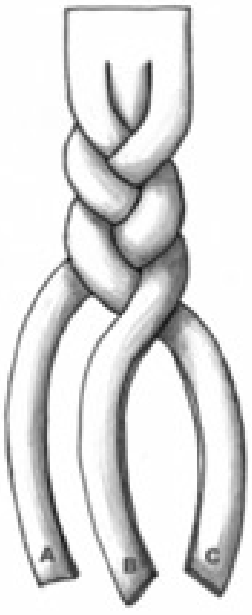


fig. 6

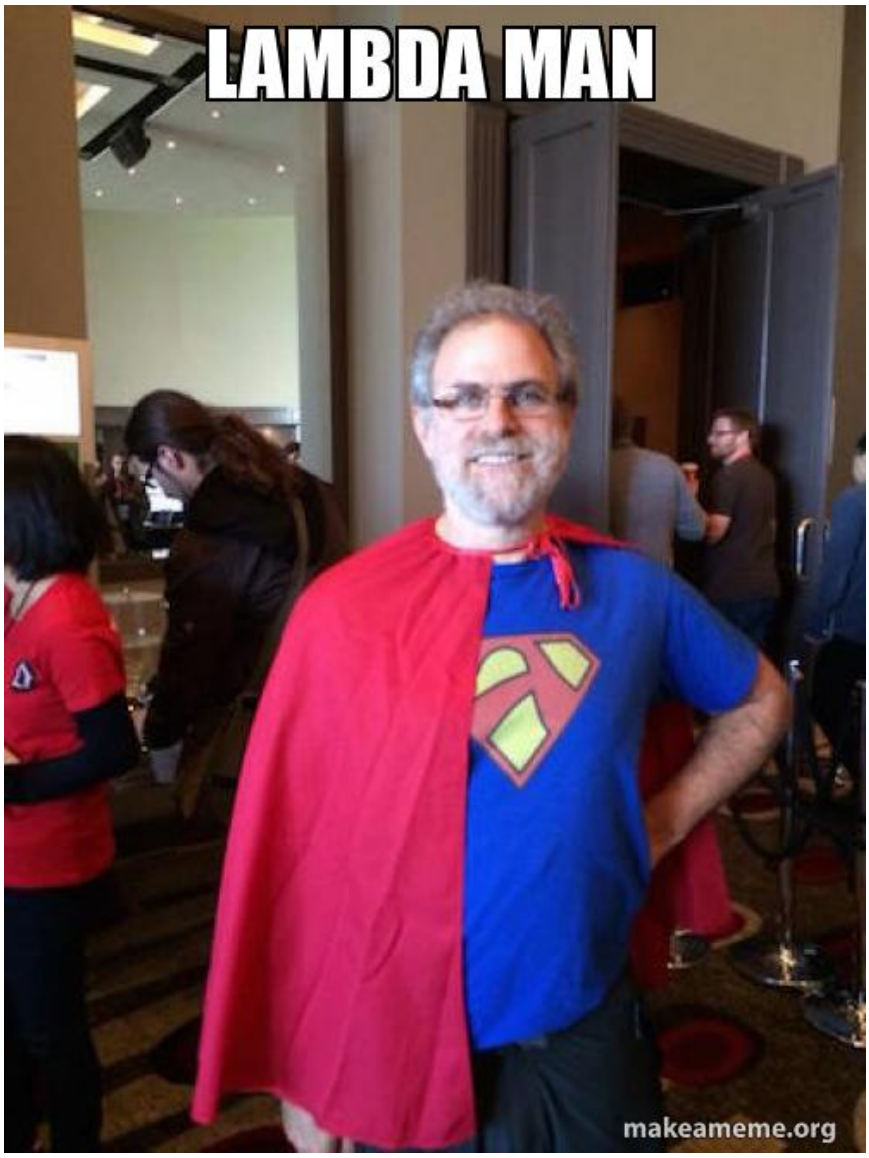
- Complecting Identity & Value
- Especially RDBMS, OOP
- Pessimistic concurrency strategies

What's changed?

`./historical_cost_graph5.gif`

- Computing capacity has increased by a million fold!

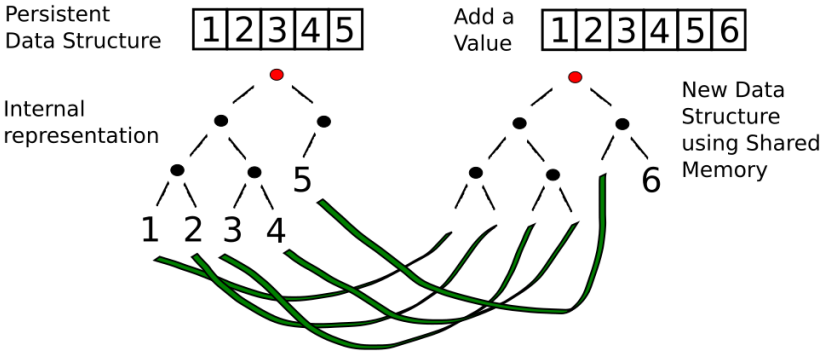
Immutability (and values) to the rescue!



Values

- Values are generic
- Values are easy to fabricate
- Drives reuse
- Values aggregate to values
- Distributable

Isn't copying values inefficient?



- Structural sharing
- For example in Clojure:
  - persistent bit-partitioned vector trie
  - 32 node tries
  - Wide shallow trees

What does it look like?

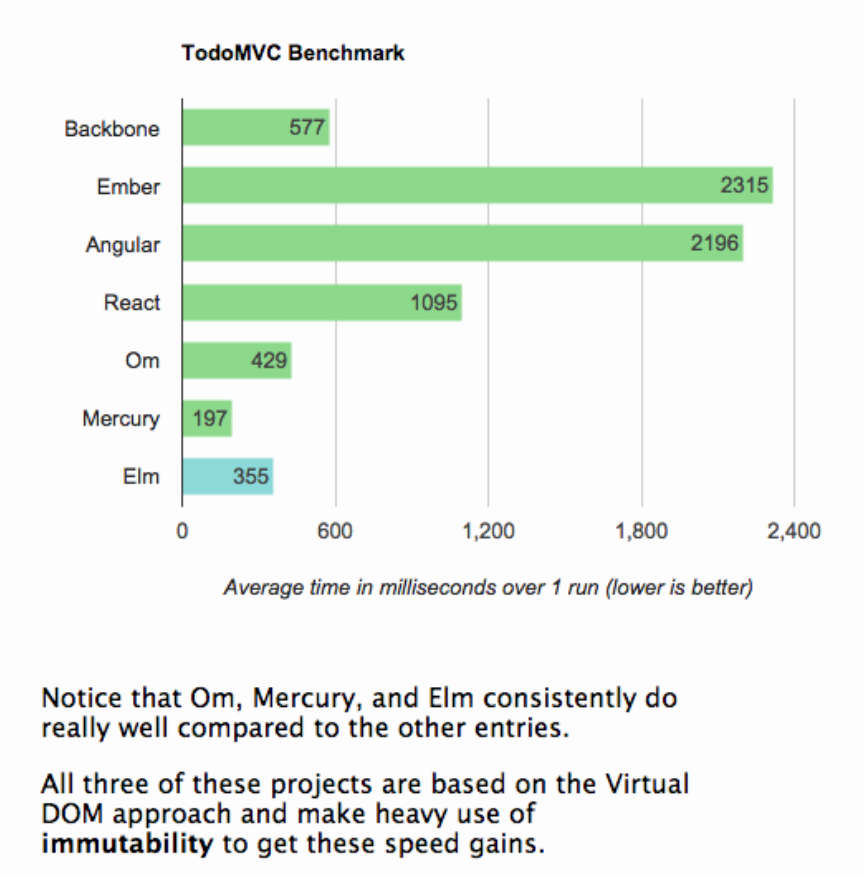
- Immutable by default
- Explicit state change
- Database as a value
- Make state change obvious
- Pass a snapshot of the database as a value
  - always remote
- Lack of Basis from database
  - consistency across long term conversations
  - what does update mean?

ClojureScript on the client

```
(def initial-state
  {:event {:event/name "" :event/speaker ""} :server-state nil})

(defn- event-form
  [ui-channel {:keys [event/name event/speaker] :as event}]
  [:table.table
   [:tr
    [:td [:label "Event name:"]]
    [:td [:input {:type :text
                  :placeholder "Event name..."
                  :defaultValue event/name
                  :on-change (send-value! ui-channel m/->ChangeEventName)}]]]
   [:tr
    [:td [:label "Speaker:"]]
```

```
    [:td [:input {:type :text
                  :placeholder "Speaker..."
                  :defaultValue event/speaker
                  :on-change (send-value! ui-channel m/->ChangeEventSpeaker)}]]]
   [:tr
    [:td
     [:button.btn.btn-success
      {:on-click (send! ui-channel (m/->CreateEvent))}
      "Go"]]]])

(defrecord ChangeEventName [name])

(defrecord ChangeEventSpeaker [speaker])

(defrecord CreateEvent [event])

(defrecord CreateEventResults [body])

(extend-protocol Message
  m/ChangeEventName
  (process-message [{:keys [name]} app]
    (assoc-in app [:event :event/name] name)))
;; redacted for clarity ...

(extend-protocol EventSource
  m/CreateEvent
  (watch-channels [_ {:keys [event]
                      :as app}]
    #{(rest/create-event event)}))

(extend-protocol Message
  m/CreateEventResults
  (process-message [response app]
    (assoc app :server-state (-> response :body))))
```

Efficiency



Notice that Om, Mercury, and Elm consistently do really well compared to the other entries.

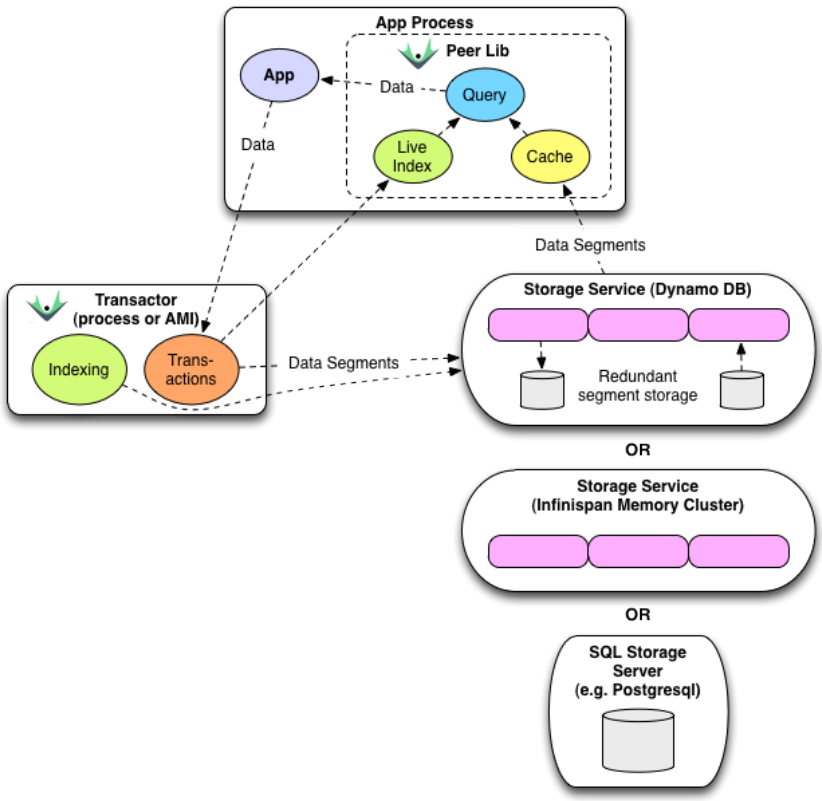
All three of these projects are based on the Virtual DOM approach and make heavy use of **immutability** to get these speed gains.

Clojure on the server

```
(defn- handle-query
  [db-conn]
  (fn [{req-body :body-params}]
    {:body (case (:type req-body)
             :get-events (data/get-events db-conn)
             :create-event (data/create-entity db-conn (:txn-data req-body)))}))


(defn app [dbconn]
  (-> (routes
       (GET "/" [] home-page)
```

```
 (POST "/q" []
       (handle-query dbconn))
 (resources "/"))
(wrap-restful-format :formats [:edn :transit-json])
(rmd/wrap-defaults (-> rmd/site-defaults
                       (assoc-in [:security :anti-forgery] false)))))
```

Datomic for Data



- App get's its own query, comms, memory- Each App is a peer
- Apps are peers
- Transactor broadcasts txns to peers
- Peers cache data locally

Database as a value

Entity	Attribute	Value	Time
Fiona	likes	Ruby	01/06/2015
Dave	likes	Haskell	25/09/2015
Fiona	likes	Clojure	15/12/2015

- Effectively DB is local
- Datalog query language

```
[:find ?e :where [?e :likes "Clojure"]]
```

- Ask connection for database - it returns a value representing the db
- This is because datoms are immutable - new versions thru time
- Can invoke your own code from query engine as data is just normal data structures (lists, maps, etc.)
- Assertions and retractions of facts (Datoms)

Schema

```
 ;;event
 {
  :db/id                 #db/id[:db.part/db]
  :db/ident              :event/name
  :db/cardinality        :db.cardinality/one
  :db/valueType          :db.type/string
  :db/unique             :db.unique/identity
  :db.install/_attribute :db.part/db
  }
 {
  :db/id                 #db/id[:db.part/db]
  :db/ident              :event/description
  :db/cardinality        :db.cardinality/one
  :db/valueType          :db.type/string
  :db.install/_attribute :db.part/db
  }
 {
```

```
  :db/id                 #db/id[:db.part/db]
  :db/ident              :event/location
  :db/cardinality        :db.cardinality/one
  :db/valueType          :db.type/ref
  :db.install/_attribute :db.part/db
  }
...

;;location
 {
  :db/id                 #db/id[:db.part/db]
  :db/ident              :location/postCode
  :db/cardinality        :db.cardinality/one
  :db/valueType          :db.type/string
  :db.install/_attribute :db.part/db
  }
 {
  :db/id                 #db/id[:db.part/db]
  :db/ident              :location/description
  :db/cardinality        :db.cardinality/one
  :db/valueType          :db.type/string
  :db.install/_attribute :db.part/db
  }
...
```

Persistence

```
(defn create-entity
  "Takes transaction data and returns the resolved tempid"
  [conn tx-data]
  (let [had-id (contains? tx-data ":db/id")
        data-with-id (if had-id
                       tx-data
                       (assoc tx-data :db/id #db/id[:db.part/user -1000001]))
        tx @(d/transact conn [data-with-id])]
    (if had-id (tx-data ":db/id")
        (d/resolve-tempid (d/db conn) (:tempids tx)
                          (d/tempid :db.part/user -1000001)))))

(defn get-events [db]
  (d/pull-many db [:*]
```

```
(->> (d/q '{:find [?event-id]
            :where [[?event-id :event/name]]}
          db)
     (map first))))
```

Conclusion?

- Immutability simplifies
- State as function call stack
- Mostly pure functions
  - Easier to test & reason about
- Time as first class concept
- Easier to distribute

Resources

- Rich Hickey talks -
  - 'The Value of Values'
  - 'The Language of the System'
  - 'Simple Made Easy'
  - 'Clojure, Made Simple'

  - 'The Database as a Value'
  - 'The Language of Systems'
- Moseley and Marks - Out of the Tar Pit
- Kris Jenkins
  - 'ClojureScript - Architecting for Scale' (Clojure eXchange 2015)
- History
  - book keeping - double entry. Didn't change in place.
  - 50's, 60's memory expensive resource (dates? picture of large old machine)
  - Swapping instructions in and out of memory - tape -> disk
  - 70's, 80's and 90's secondary storage expensive - rise of RDBMS
  - memory still reasonably expensive
  - In place computing as resources scarce
  - 00's and 2010's disk cheaper, memory very cheap.
  - in parallel the rise of OOP - objects with data and behaviour
- Why immutability?
  - What does mutation bring (picture of three eyed fish from Simpsons _ other pop culture references)
  - Can't stand in same river twice (where is origin of quote?)
  - Complecting the concepts of identity and value particularly OO and RDBMS in trad. use.
  - Issues of concurrency. Complex values are changed underneath you.
  - Optimisations - (dig out graph of Om compared with React.js)
- What does it look like?
  - Examples in:
    * Clojurescript - UI state as a value
    * Clojure - server state as value and a chain of functions
    * Datomic - database as a value - local cache, peer to peer