Datomic: History of an entity

2013-10-29

Originally published 2013-01-20, at my personal blog. Edited and optimized.

Full runnable example code for this article can be found here.

A short primer

Datomic is an append-only database. You store facts, such as Bob's e-mail is A, and Bob's e-mail is B. When you stated a fact, is in storage forever. But you can retrieve the value of the database - the index - of any point in time. When you ask this database/index for Bob's e-mail, you'll get the one as of that point in time. This index is maintained automatically for you by Datomic.

Another efficient operation is datomic is to query over full list of facts that has been added over time. This is what we'll be using in this article. In this special database/index, you'll get all values Bob's e-mail has ever had, and information about the transaction it correlates to. All facts in datomic are added to the database via transactions.

The data

We're going to assume a database value with an user that has an e-mail, a name, an age and an address. I'll skip the boring parts and show just the transaction values that we're assuming.

```
(require '[datomic.api :as d])

(defn get-inserted-entity
  [tempid tx-res]
  (d/entity
    (:db-after tx-res)
    (d/resolve-tempid (:db-after tx-res) (:tempids tx-res) tempid)))

(defn transact-entity
  [datomic-conn tempid tx]
  (get-inserted-entity tempid @(d/transact datomic-conn tx)))

(let [user-tempid (d/tempid :db.part/user)]
  (def user
    (transact-entity
     datomic-conn
     user-tempid
     [[:db/add user-tempid :user/email "quentin@test.com"]
      [:db/add user-tempid :user/name "Quentin Test"]])))

(transact-entity
 datomic-conn
 (:db/id user)
 [[:db/add (:db/id user) :user/name "Quentin F. Test"]])

(transact-entity
 datomic-conn
 (:db/id user)
 [[:db/add (:db/id user) :user/age 27]
  [:db/add (:db/id user) :user/address "Some Road 155, Norway"]])

(transact-entity
 datomic-conn
 (:db/id user)
 [[:db/add (:db/id user) :user/email "quentin@mycompany.com"]])
(def db (d/db datomic-conn))
```

Now we have an entity with some changes over time applied to it. Let's extract some history!

A list of before/after entities

This example shows a very crude method of getting the full user entity as of before and after the transactions above. We won't actually use it to reach our end goal of building an UI of a list of changes, but it's a useful first step.

```
(->>
 ;; This query finds all transactions that touched a particular entity
 (d/q
  '[:find ?tx
    :in $ ?e
    :where
    [?e _ _ ?tx]]
  (d/history db)
  (:db/id user))
 ;; The transactions are themselves represented as entities. We get the
 ;; full tx entities from the tx entity IDs we got in the query.
 (map #(d/entity (d/db datomic-conn) (first %)))
 ;; The transaction entity has a txInstant attribute, which is a timestmap
 ;; as of the transaction write.
 (sort-by :db/txInstant)
 ;; as-of yields the database as of a t. We pass in the transaction t for
 ;; after, and (dec transaction-t) for before. The list of t's might have
 ;; gaps, but if you specify a t that doesn't exist, Datomic will round down.
 (map
  (fn [tx]
    {:before (d/entity (d/as-of db (dec (d/tx->t (:db/id tx)))) (:db/id user))
     :after (d/entity (d/as-of db (:db/id tx)) (:db/id user))})))
```

Pretty cool! Now we have a list of maps that contains the full user entity, before and after the individual transactions/changes we did earlier.

This code is not sufficient for our UI, though. We just want to show the changes in the UI, not the full user entity before/after the change. We could do manual diffing, but it's better to leverage Datomic's query engine and also the fact that the transaction knows which attributes that actually changed, and rewrite the code so we don't have to do any diffing at all.

Leveraging queries

Here's some code that does exactly what we want to achieve.

```
(->>
 ;; This query finds all tuples of the tx and the actual attribute that
 ;; changed for a specific entity.
 (d/q
  '[:find ?tx ?a
    :in $ ?e
    :where
    [?e ?a _ ?tx]]
  (d/history db)
  (:db/id user))
 ;; We group the tuples by tx - a single tx can and will contain multiple
 ;; attribute changes.
 (group-by (fn [[tx attr]] tx))
 ;; We only want the actual changes
 (vals)
 ;; Sort with oldest first
 (sort-by (fn [[tx attr]] tx))
 ;; Creates a list of maps like '({:the-attribute {:old "Old value" :new "New value"}})
 (map
  (fn [changes]
    {:changes (into
               {}
               (map
                (fn [[tx attr]]
                  (let [tx-before-db (d/as-of db (dec (d/tx->t tx)))
                        tx-after-db (d/as-of db tx)
                        tx-e (d/entity tx-after-db tx)
                        attr-e-before (d/entity tx-before-db attr)
                        attr-e-after (d/entity tx-after-db attr)]
                    [(:db/ident attr-e-after)
                     {:old (get
                            (d/entity tx-before-db (:db/id user))
                            (:db/ident attr-e-before))
                      :new (get
                            (d/entity tx-after-db (:db/id user))
                            (:db/ident attr-e-after))}]))
                changes))
     :timestamp (->> (ffirst changes)
                     (d/entity (d/as-of db (ffirst changes)))
                     :db/txInstant)})))
```

This gives us exactly what we need! We now have a sorted list of maps, where the map represents a single transaction. We also added the timestamp as of the transaction occurred.

There is no actual UI rendering code here, but it should be obvious how to use this data structure to implement a UI for it.

Here's the full output, assuming we ran the transactions at the beginning of this post.

```
;; Output
({:changes
  {:user/name {:old nil, :new "Quentin Test"},
   :user/email {:old nil, :new "quentin@test.com"}},
  :timestamp #inst "2013-10-28T18:50:50.207-00:00"}
 {:changes {:user/name {:old "Quentin Test", :new "Quentin F. Test"}},
  :timestamp #inst "2013-10-28T18:50:51.637-00:00"}
 {:changes
  {:user/age {:old nil, :new 27},
   :user/address {:old nil, :new "Some Road 155, Norway"}},
  :timestamp #inst "2013-10-28T18:50:53.117-00:00"}
 {:changes
  {:user/email
   {:old "quentin@test.com", :new "quentin@mycompany.com"}},
  :timestamp #inst "2013-10-28T18:50:54.643-00:00"})
```

Closing thoughts

As mentioned, transactions are themselves represented with an entity. You can add add extra attributes to a transaction to annotate it with extra values, such as which logged in user that performed the change. Then it's just a matter of reading it out from the tx entity like we read out the timestamp of the tx in the code above.

This article demonstrates one of the many superpowers you get when your database doesn't delete old data. We don't have to do anything special to ensure we don't update in place, Datomic handles it all for us. Also note how all the querying code is completely referentially transparent. The database is represented as an immutable value, and can be memoized.