*Clojure ♥ C**

Building a sane ecosystem around Cassandra 1.2+ for Clojure

Max Penet

@***mpenet*** on Twitter & GitHub


cassandra

- Originally borrows many elements from Dynamo and the data model from Google's Big Table
- Durable
- No single point of failure
- Linear scalability
- Fault tolerance
- "Simple" to operate
- Great performance across the board

- Secondary Indexes
- Tunable Consistency

and …

- Very expressive (not horrible) query language

Data model crash course

At a Low level

Column Family -> Row-Key {column-name: column-value, ...}

Users['mpenet']{'location': 'SoftShake', 'name': 'mpenet'}

CQL abstracts over this

- Simpler Data Model
- Primary component of the Primary Key in a row becomes the row-key (or partition key)
- Rows are clustered by the remaining columns of the primary key
- Other columns can be indexed separately from the PK

```
CREATE TABLE users (
  user_name varchar PRIMARY KEY,
  password varchar,
  gender varchar,
  session_token varchar,
  state varchar,
  birth_year bigint
);
```

users['mpenet']{password: '1234', gender: 'male', ...}

```
CREATE TABLE timeline (
  user_id varchar,
  tweet_id uuid,
  author varchar,
  body varchar,
  PRIMARY KEY (user_id, tweet_id)
);
```

timeline['@mpenet']{[12345, "author"]: 'Max Penet', [12345, "body"]: 'Oh no!'}

No join or subqueries but we can have collection column values (with indexing in the near future)

- *List*
- *Set*
- *Map*

Users['mpenet']{'hashtags': ['#clojure', '#cassandra']}

And C* data model, performance characteristics encourage heavy denormalisation anyway.

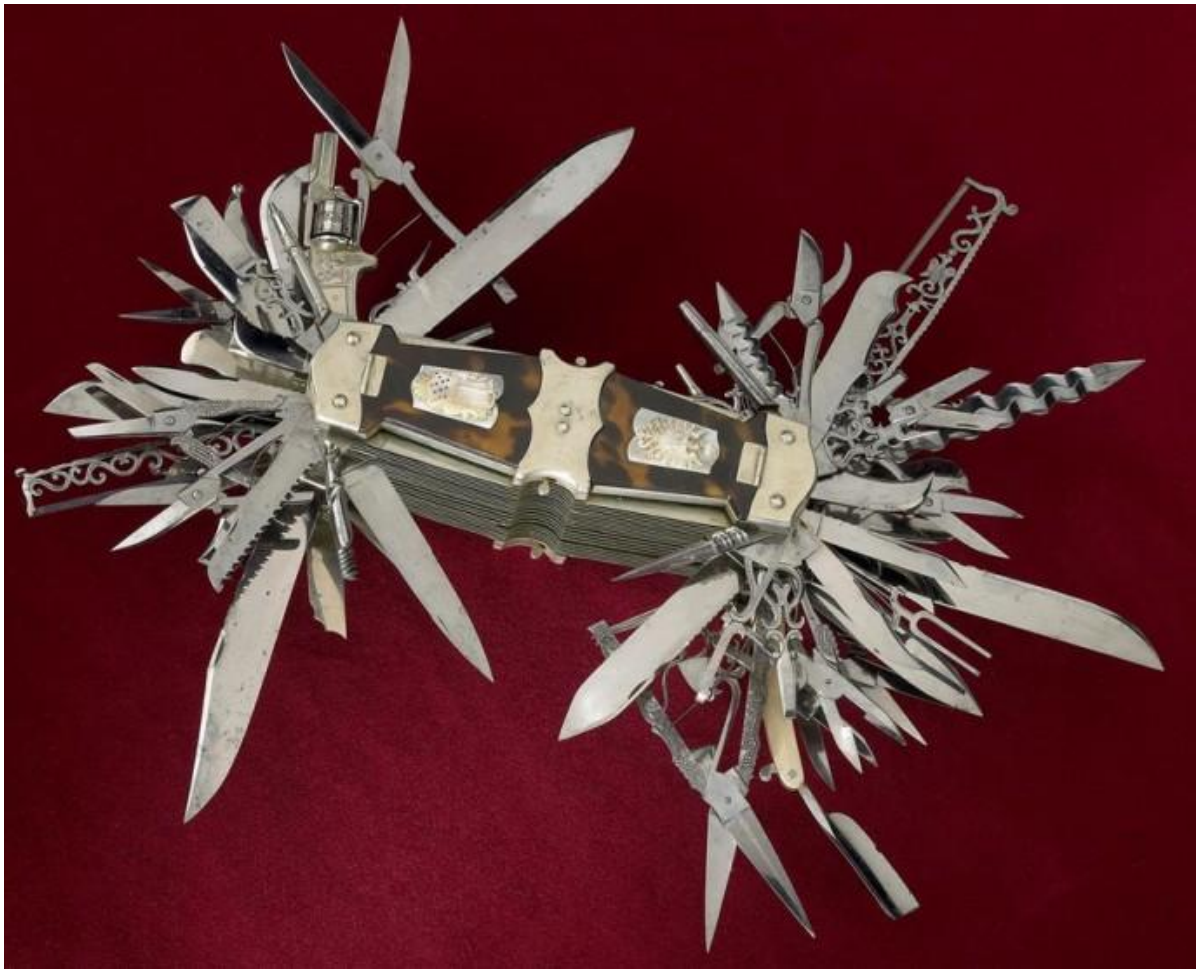
“***Clojure*** is a dynamic programming language that targets the Java Virtual Machine”

- clojure.org

What else?

- A modern LISP
- (Pragmatic) functional purity
- Concurrency primitives

- Complete java interop, meaning access to the entire java ecosystem



- Very much data oriented from the language core to the community, persistent data structures
- Interactive development via the REPL
- Powerful tooling (emacs/vim, eclipse, you name it...)
- Vibrant and helpful community that produces very high quality libraries
- IRC freenode #clojure

- Cascalog
- Storm
- High quality clients for any datastore you could think of
- Incanter

And …

- ClojureScript

Clojure crash course

```
(def num 2)

(defn increment [x]
  (+ 1 x))

(increment num)
>> 3
```

Data types (oversimplified) overview

[1 2 3]	*Vector*
‘(1 2 3)	*List*
{:foo 1, :bar 2}	*Map*
#{1 2 3 “bar”}	*Set*

Cassandra + Clojure history

Pre 1.2: *Thrift based clients*

- clj-hector (Hector wrapper)
- Netflix Astyanax (via interop)
- casyn (native clojure client)
- Cassaforte (alpha)

And tons of half working/finished clients

Enter C* 1.2

- New Binary protocol designed with CQL3 in mind
- Notifications (topology, status, schema changes)
- Each connection can handle more requests
- Tons of server side improvements (VNodes, compaction, compression, query tracing, etc…)
- Atomic batches

And...

CQL3

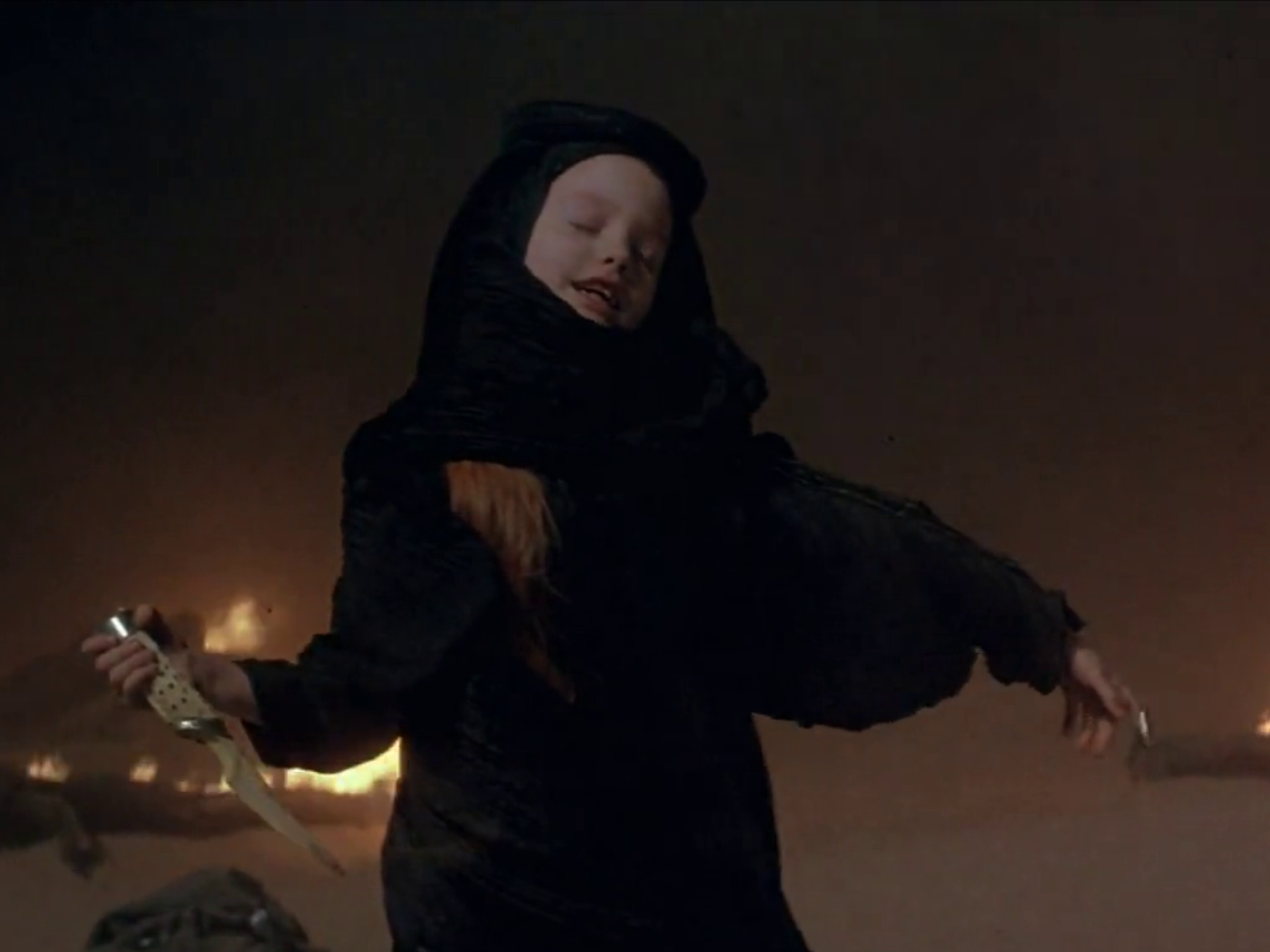
- Abstracts over the pre 1.2 (low level) storage
- SQL like
- Introduction of collection types (map, list, set)
- Single language for cli exploration and in app querying
- Hides the powerful but “ugly” (wide rows)
- Querying still requires knowledge of the low level bits, but it's less hairy

Driver/Client dev perspective

- Obsolescence: client APIs modeled around Thrift clients
- New protocol requires rewrite of foundations of the drivers

But thrift is still supported, kind of...

Then…



Alia

- A wrapper to the *java-driver* made by Datastax
- All the new toys from 1.2+
- Asynchronous architecture (netty)
- Robustness
- Very up to date
- Feature complete

Goals

- Minimal API
- Support everything the wrapped driver can do, do not shadow anything, avoid indirection
- Idiomatic to Clojure folks, modern Clojure features support

- Up to date (43 releases, branches ready for cassandra/master features)
- Performant
- Integrate with clojure popular interfaces for asynchronous workflows (Lamina, core. async)

- Policies: retry, reconnection, load balancing
- UUIDs operations
- Different coding styles: dynamic bindings vs argument parameters
- Enhanced and clojure friendly debugging: *ExceptionInfo*
- All cluster level and query execution level options: *hosts, tracing, metrics, policies, consistency, pooling, custom executors, compression, auth, SSL, JMX* etc…

Demo: the basics

```
(ns softshake.simple-alia-demo
  (:require [qbits.alia :as alia]))

;; Defines a cluster
(def cl (alia/cluster "127.0.0.1" :port 1234))

;; Creates a session connected to a specific keyspace
(def s (alia/connect cl "demo-keyspace"))

;; Synchronous queries
(alia/execute s "SELECT * FROM users;")

;; Prepared statements
(def ps (alia/prepare s "SELECT * FROM users WHERE id = ?;"))
(alia/execute s ps :values ["mpenet"])
```

Raw CQL queries

Enter **Hayt**

- Avoid fragmentation
- Leverage clojure strength
- Similar API to other query DSL
- Highly composable
- Query as/is data
- Decoupled (compiler, DSL api, utils)
- Up to date (11 releases, c* master ready)

- Support every single CQL3 feature
- Compile as raw or prepared statement
- Don’t get in the way of advanced users
- Performant
- Community support (17 PRs)

Used by Alia, Cassaforte (ClojureWerkz), casyn, hopefully more in the future.

Select

```
(select :users
        (columns :firstname :age :id :email)
        (where {:firstname "John"
                :age [> 18]
                :tech [:in ["#cassandra" "#clojure"]]})
        (order-by [:age :desc])
        (limit 25))
```

```
SELECT firstname, age, id, email
  FROM users
  WHERE firstname = 'John' AND age > 18 AND tech IN ('#cassandra', '#clojure')
  ORDER BY age desc
  LIMIT 25;
```

Insert

```
(insert :users
        (values {:username "mpenet"
                 :age 37})
        (using :ttl 3600))
```

```
INSERT INTO users (age, username)
  VALUES (37, 'mpenet')
  USING TTL 3600;
```

Update

```
(update :users
        (where {:id "mpenet"})
        (set-columns {:firstname "Max"
                      :lastname "Penet"}))
```

```
UPDATE users
  SET firstname = "Max", lastname = "Penet"
  WHERE id = "mpenet";
```

Meh?

I said ***composable*** and ***query as data***

Data under the hood: Statement & Clauses return maps

```
(select :users
        (columns :firstname :age :id :email)
        (where {:firstname "John"
                :age [> 18]
                :tech [:in ["#cassandra" "#clojure"]]})
        (order-by [:age :desc])
        (limit 25))


{:select :users
 :columns [:firstname :age :id :email]
 :where {:firstname "John"
         :age [> 18]
         :tech [:in ["#cassandra" "#clojure"]]}
 :order-by [[:age :desc]]
 :limit 25}
```

Entire clojure.core API is at your fingertips for query manipulation.

assoc, assoc-in, conj, merge, concat etc...

Nothing new to learn really

```clojure
(def user-select (select :users (where {:id "foo"})))

;; or let's change the where clause
(conj user-select (where {:id "mpenet" :age 37}))

;; or using hash-map functions
(assoc user-select :where {:id "bar"})


;; deep updates
(assoc-in user-select [:where :id] "mpenet")


;; etc
(conj user-select
      (where {:id "doe"})
      (order-by [:age :where])
```

Compilation

```
;; To obtain the end result it needs to be passed to one of these 2 functions:

(->raw (select :users (where {:firstname "John" :lastname "Doe"})))
>> SELECT * FROM users WHERE firstname = "John" AND lastname = "Doe";


(->prepared (select :users (where {:firstname "John" :lastname "Doe"})))
>> ["SELECT * FROM users WHERE firstname = ? AND lastname = ?;"  ["John" "Doe"]]
```

Advanced features

- Possibility to generate prepared statements from ->raw
- DIY DSL facade
- Values encoding/escaping: ByteBuffer/blob, Date/timestamp, etc…
- Extensible
- And tons more...

Really feature complete!

Statements: *alter-column-family, alter-keyspace, alter-table, alter-user, batch, create-index, create-keyspace, create-table, create-trigger, create-user, delete, drop-index, drop-keyspace, drop-table, drop-trigger, drop-user, grant, insert, list-perm, list-users, revoke, select, truncate, update, use-keyspace*

Clauses: *add-column, allow-filtering, alter-column, column-definitions, columns, counter, custom, drop-column, if-exists, index-name, limit, logged, only-if, order-by, password, perm, queries, recursive, rename-column, resource, set-columns, superuser, user, using, values, where, with*

Functions: *as, ascii->blob, bigint->blob, blob->ascii, blob->bigint, blob->boolean, blob->counter, blob->decimal, blob->double, blob->float, blob->inet, blob->int, blob->text, blob->timestamp, blob->timeuuid, blob->uuid, blob->varchar, blob->varint, boolean->blob, count*, count1, counter->blob, cql-fn, cql-raw, date-of, decimal->blob, distinct*, double->blob, float->blob, inet->blob, int->blob, max-timeuuid, min-timeuuid, now, text->blob, timestamp->blob, timeuuid->blob, token, ttl, unix-timestamp-of, uuid->blob, varchar->blob, varint->blob, writetime*

Utils: *list-type, map-type, set-type, ?*

Tight integration of ***Hayt*** into ***Alia***.

- Alia will accept Hayt queries (maps) anywhere it accepts raw queries
- Compiles as necessary
- Query (string) caching via *clojure/core.memoize* (tunable, togglable)

```
(alia/prepare (select :users (where {:id ?})))
(alia/execute (select :users (where {:lastname "Penet"})))
```

Getting Fancy with
alia/lazy-query

Origins from clojure.core

Lazy, chunked sequences:

```
;; lazy infinite sequence of numbers
(def nums (range))

;; just realize/take the first 20
(take 20 nums)
```

The docstring

"Takes a query (hayt, raw or prepared) and a query modifier fn that receives the last query and last chunk/response and returns a new query or nil.

The first chunk will be the original query result, then for each subsequent chunk the query will be the result of last query modified by the modifier fn unless the fn returns nil, which would causes the iteration to stop."

What ?!

Can be used to generate an iterator over all the rows of a table, having full control over the size of chunks/partitions you will receive, how subsequent queries will be performed and when termination happens.

```
;; For instance here we will represent the items from
;; $1 to $1000 by chunks/partitions of 10 items.

(alia/lazy-query
  (select :items
          (limit 10)
          (where {:price 1}))
  (fn [query coll]
    (let [last-price (-> coll last :price)]
      ;; When the value for :price of the last item in the collection is
      ;; smaller than 1000 continue
      (when (< last-price 1000)
        ;; To continue we then modify the query for the next chunk
        ;; by modifying its `where` clause with the price + 1
        (merge query (where {:price (inc last-price)}))))))
```

lazy-query is not limited to Hayt queries but Hayt makes it trivial to use.

Asynchronous execution

Two ways to do asynchronous queries

Lamina from @**ztellman**

qbits.alia/execute-async

- Takes the same arguments as ***qbits.alia/execute*** + optionally error and success callbacks
- Returns a Lamina ***result-channel***

This subscribes 2 callbacks and returns a result channel immediately

```
(execute-async (select :foo)
               :success (fn [result] ...)
               :error (fn [error] ...))
```

lamina pipelines

```
(ns soft-shake.lamina
  (:require [lamina.core :as l]
            [qbits.alia :as alia]
            [qbits.hayt :refer :all]))

(run-pipeline (alia/execute-async (select :users (where {:id "mpenet"})))
 {:error-handler (fn [ex] (log-or-something ex))}
 #(alia/execute-async (select :tweets (where {:user-token (:token %)})))
 #(do-something-synchronously %))
```

clojure/core.async from Rich Hickey, Timothy Baldridge and contributors

Heavily inspired by Hoare's ***CSP***,
"Communicating Sequential Processes"
& the more recent ***Go channels/goroutines***.

Oversimplified description

- A ***channel*** is a queue/buffer
- Producers/consumers take/put on channels, blocking or non blocking
- ***go*** blocks that make this seems synchronous thanks to macro magic that rewrites this as a state machine and saves us from callback pyramids
- Growing api around these concepts

qbits.alia/execute-chan

- Takes the same arguments as ***qbits.alia/execute***
- Returns an empty clojure.core.async ***channel*** that will receive the query result at some point in the future, or an exception

```clojure
(ns soft-shake.core-async
  (:require
     [clojure.core.async :as async :refer :all]
     [qbits.hayt :refer :all]
     [qbits.alia :as alia]))


;; This returns a channel that will receive the eventual result or exception
(alia/execute-chan (select :foo))

;; To pull a value from it synchronously
(<!! (alia/execute-chan (select :foo)))

;; To pull a value from it asynchronously
(take! (alia/execute-chan (select :foo))
          (fn [result] ...))

;; In a go block (similar to a goroutine)
(go
 (let [foos (<! (alia/execute-chan (select :foo)))
       bars (<! (alia/execute-chan (select :bar)))]
   (concat foos bars)))
```

It gets interesting when you need to use many of these and use the equivalent of Go ***select***, ***alts!*** in clojure talk, to grab results as they are realized.

```
(let [queries [(alia/execute-chan (select :foo))
               (alia/execute-chan (select :bar))
               (alia/execute-chan (select :baz))]]
  (go
   (loop [queries queries                ;; the list of queries remaining
          query-results '()]             ;; where we'll store our results
     ;; If we are done with our queries return them, it's the exit clause
     (if (empty? queries)
       query-results
       ;; else wait for one query to complete (first completed first served)
       (let [[result channel] (alts!! queries)]
         (println "Received result: "  result " from channel: " channel)
         (recur
          ;; we remove the channel that just completed from our
          ;; pending queries collection
          (remove #{channel} queries)

          ;; and finally we add the result of this query to our
          ;; bag of results
          (conj query-results result)))))))
```

Differences with Lamina ***result-channel***

What's coming ***next***

Type checked queries for Hayt via clojure/core.typed

Support for Cassandra 2.1+,
query paging, custom types, ...

Questions?

Thanks!

https://github.com/mpenet/alia

https://github.com/mpenet/hayt

https://github.com/clojure/core.async

https://github.com/ztellman/lamina