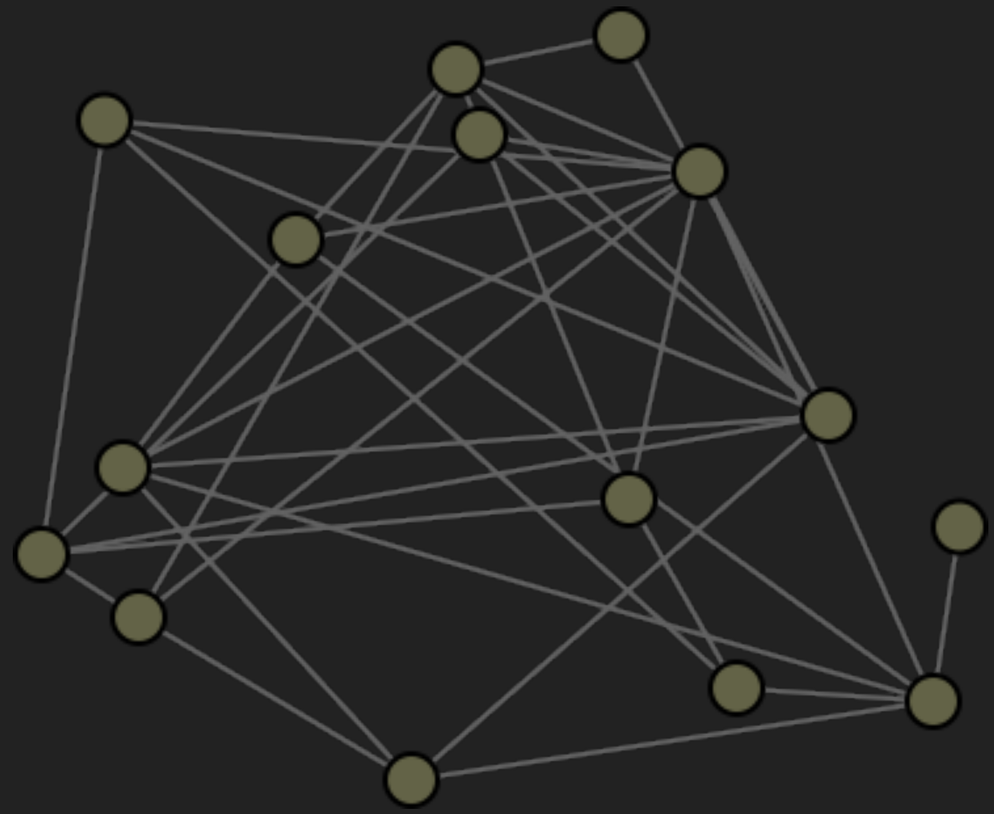


PAULA GEARON

GRAPH DATABASES

TRIPLES

Many ways of labeling the same things

- Entity – Attribute – Value
- Subject – Predicate – Object
- Source – Edge – Target
- Object – Property – Value

TABLES AS TRIPLES

id	first-name	last-name	age	child
1	“Luke”	“Skywalker”	19	NULL
2	“Anakin”	“Skywalker”	41	1

```
:node1 :first-name “Luke”
:node1 :last-name  “Skywalker”
:node1 :age        19
:node2 :first-name “Anakin”
:node2 :last-name  “Skywalker”
:node2 :age        41
:node2 :child      :node1
```

OBJECTS AS TRIPLES

```
[
 { “first-name”: “Luke”,
   “last-name”: “Skywalker”,
   “age”: 19 },
 { “first-name”: “Anakin”,
   “last-name”: “Skywalker”,
   “age”: 41,
   “child”:
     {“first-name”: “Luke”… } }
]
```

```
:node1 :first-name “Luke”
:node1 :last-name  “Skywalker”
:node1 :age        19

:node2 :first-name “Anakin”
:node2 :last-name  “Skywalker”
:node2 :age        41
:node2 :child      :node1
```

BASIC STRUCTURE

BASIC STRUCTURE

MULTIPLE VALUES

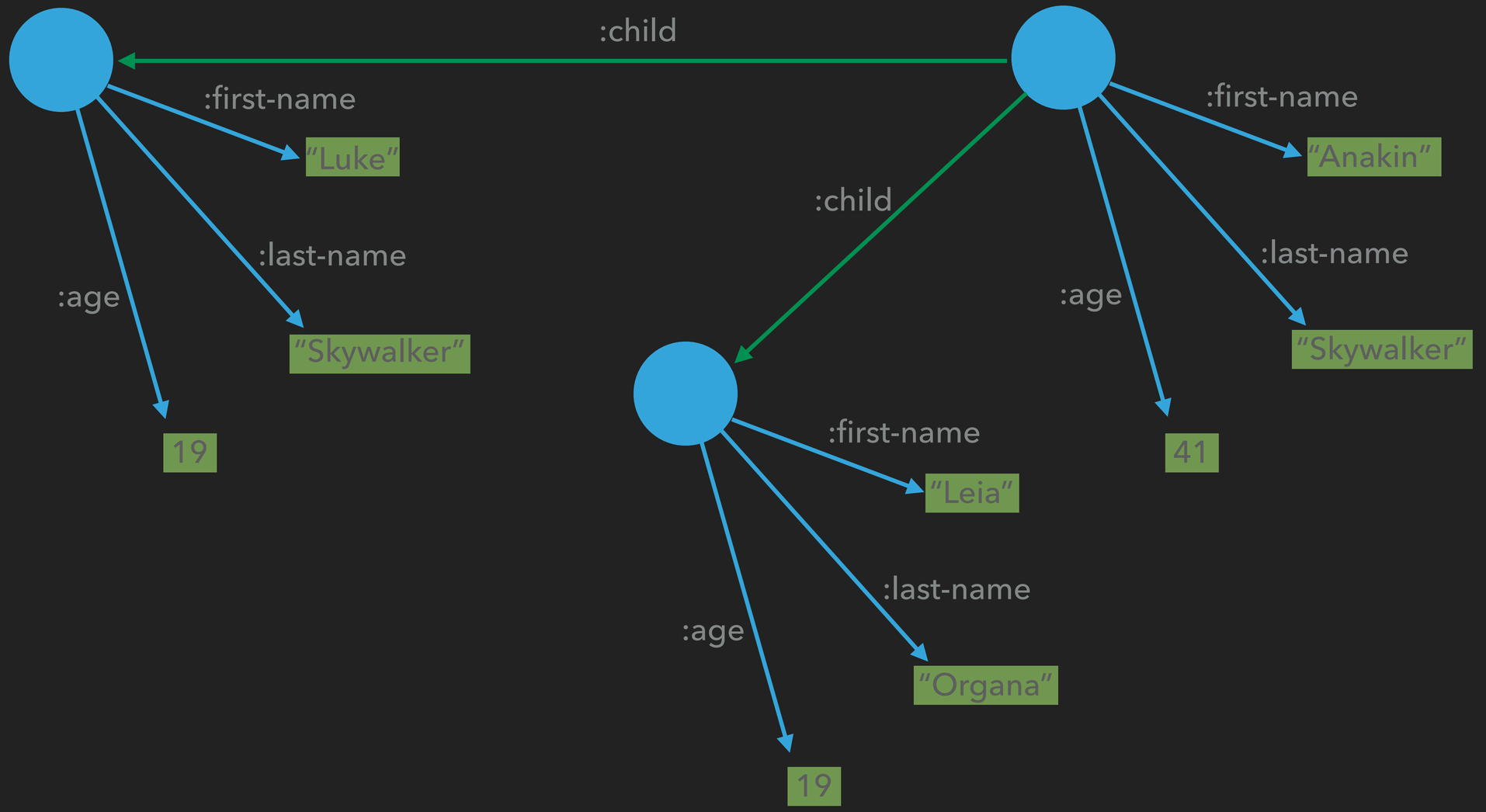
```
{“id”: 1,
 “first-name”: “Luke”,
 “last-name”:  “Skywalker”,
 “age”: 19 },
{“id: 2,
 “first-name”: “Anakin”,
 “last-name”: “Skywalker”,
 “age”: 41,
 “child”: [1, 3]},
{“id”: 3,
 “first-name”: “Leia”,
 “last-name”:  “Organa”,
 “age”: 19 }
```

```
:node1 :first-name “Luke”
:node1 :last-name  “Skywalker”
:node1 :age        19

:node2 :first-name “Anakin”
:node2 :last-name  “Skywalker”
:node2 :age        41
:node2 :child      :node1
:node2 :child      :node3

:node3 :first-name “Leia”
:node3 :last-name  “Organa”
:node3 :age        19
```

BASIC STRUCTURE



GRAPHS

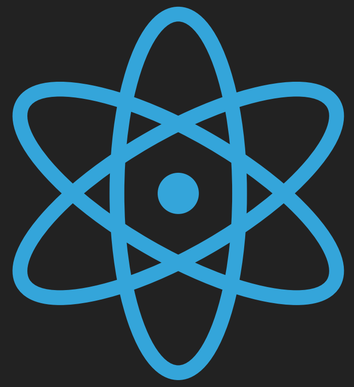
- Provides:
  - Contexts
  - Grouping

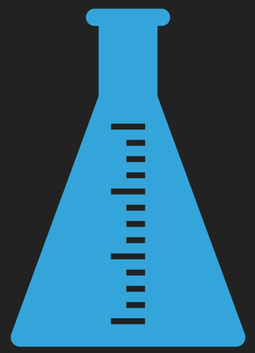
Example:

  - Movie Graph
    Darth Vader: Scenes, Lines, Interactions
  - Game Graph
    Darth Vader: Controls, Abilities

METADATA

			Transaction ID	Graph	Timestamp
:node1	:first-name	“Luke”	1	:movie	2024-04-12T14:00:00Z
:node1	:last-name	“Skywalker”	1	:movie	2024-04-12T14:00:00Z
:node1	:age	19	1	:movie	2024-04-12T14:00:00Z
:node2	:first-name	“Anakin”	1	:movie	2024-04-12T14:00:00Z
:node2	:last-name	“Skywalker”	1	:movie	2024-04-12T14:00:00Z
:node2	:age	41	1	:movie	2024-04-12T14:00:00Z
:node2	:child	:node1	1	:movie	2024-04-12T14:00:00Z
:node3	:first-name	“Leia”	2	:movie	2024-04-12T14:10:00Z
:node3	:last-name	“Organa”	2	:movie	2024-04-12T14:10:00Z
:node3	:age	19	2	:movie	2024-04-12T14:10:00Z





DATA

LAB 1

READ JSON

Load `src/lab1/json.clj`

READ JSON

```
(defn read-json
  [filename]
  ;; TODO read a JSON file, with keyword keys
  )
```

READ JSON

```
(defn read-json
  [filename]
  (json/read-str (slurp filename) :key-fn keyword))
```

LINK TRIPLES

```
(defn link->triples
  [nodes link-index {:keys [source target value] :as link}]
  ;; TODO convert a link to a seq of triples
  ;; the nodes are provided since links connect nodes
  )
```

LINK TRIPLES - OPAQUE NODES

```
(defn link->triples
  [nodes link-index {:keys [source target value] :as link}]
  (let [source-node (keyword (str "node" source))
        target-node (keyword (str "node" target))]
    [[source-node :interacts-with target-node]
     [target-node :interacts-with source-node]]))
```

LINK TRIPLES - NAMED NODES

```
(defn link->triples
  [nodes link-index {:keys [source target value] :as link}]
  (let [source-id (node-id (get nodes source))
        target-id (node-id (get nodes target))]
    [[source-id :interacts-with target-id]
     [target-id :interacts-with source-id]]))
```

LINK TRIPLES - OPAQUE NODES

```
(defn link->triples
  [nodes link-index {:keys [source target value] :as link}]
  (let [source-node (node-id (get nodes source))
        target-node (node-id (get nodes target))
        connection (keyword (str "link-" link-index))]
    [[source-node :interacts-with target-node]
     [target-node :interacts-with source-node]
     [connection :type :connection]
     [connection :end source-node]
     [connection :end target-node]
     [connection :count value]]))
```

NODE TRIPLES

```
(defn node->triples
  [node-index {:keys [name value colour] :as node}]
  ;; TODO convert a node to a seq of triples
  )
```

NODE TRIPLES - OPAQUE NODES

```
(defn node->triples
  [node-index {:keys [name value colour] :as node}]
  (let [id (keyword (str "node" node-index))]
    [[id :name (capitalize-name name)]
     [id :value value]
     [id :color colour]]))
```

NODE TRIPLES - NAMED NODES

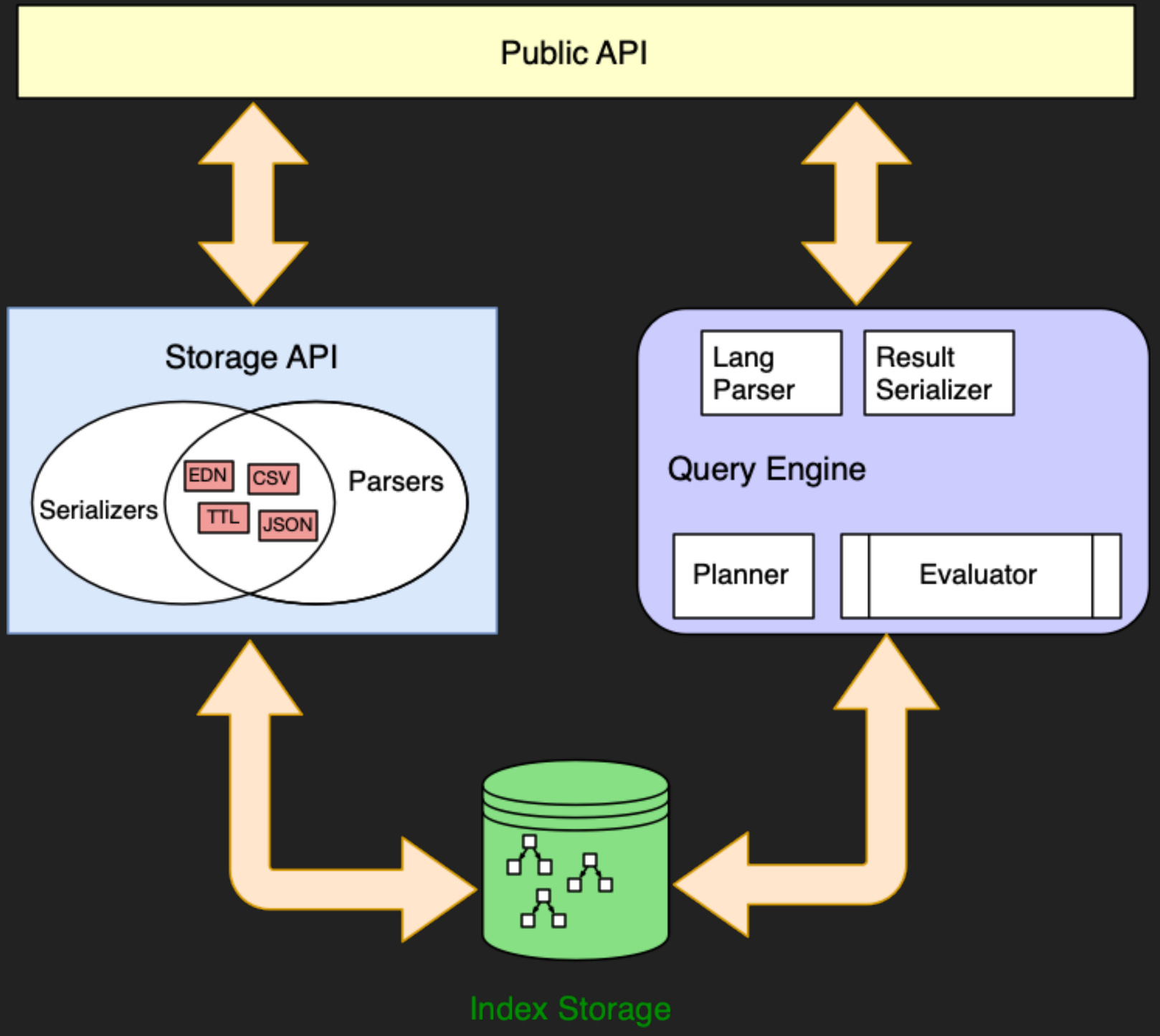
```
(defn node->triples
  [node-index {:keys [name value colour] :as node}]
  (let [id (node-id node)]
    [[id :name (capitalize-name name)]
     [id :value value]
     [id :color colour]]))
```

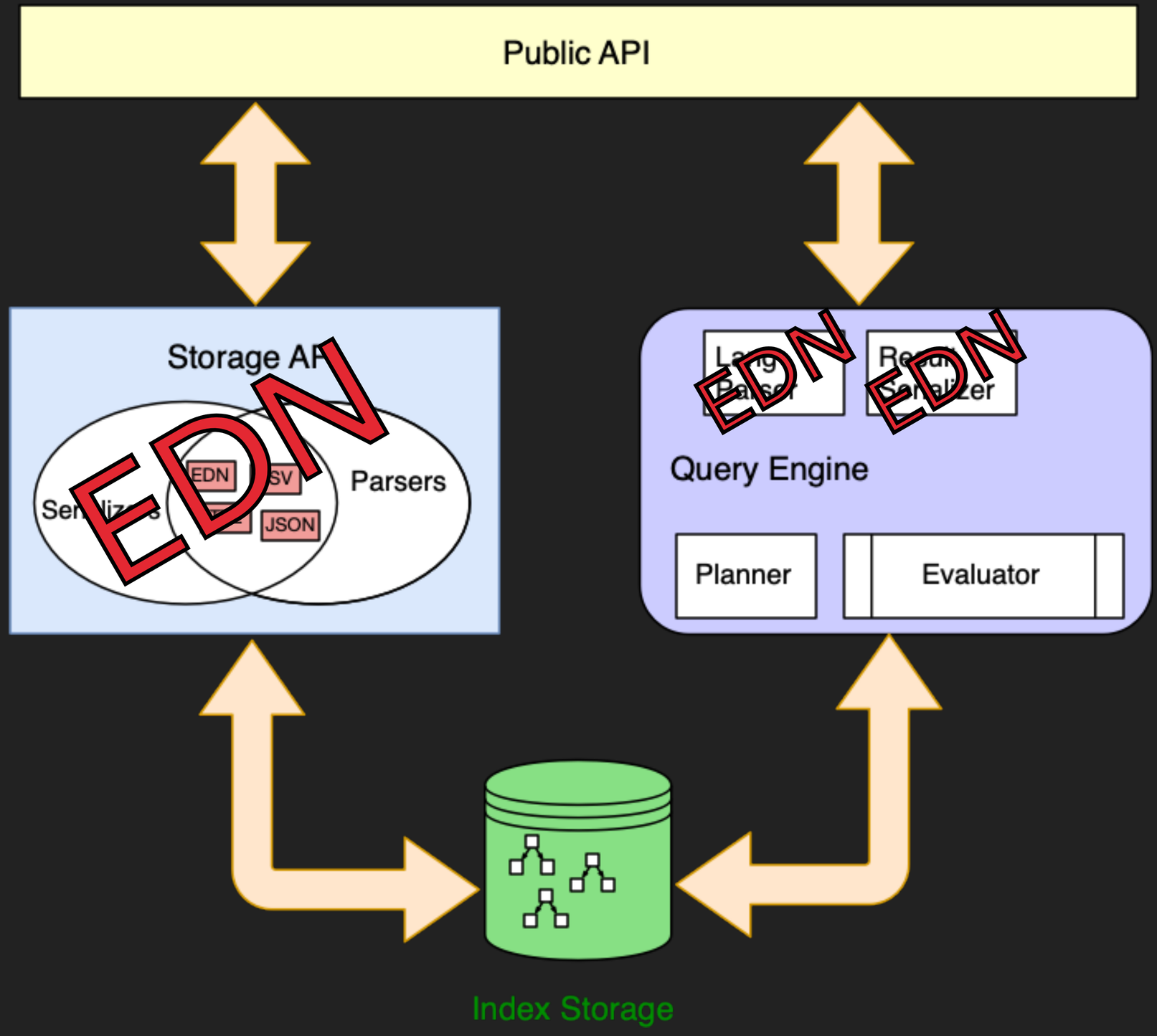
CSV TRIPLES

```
(defn obj->triples
  "Convert a map to triples"
  ([object] (obj->triples object :app-id))
  ([object key-field]
   (let [id (keyword (str "g" (get object key-field)))]
     ;; TODO convert a node to a seq of triples
     )))
```

CSV TRIPLES

```
(defn obj->triples
  "Convert a map to triples"
  ([object] (obj->triples object :app-id))
  ([object key-field]
   (let [id (keyword (str "g" (get object key-field)))]
     (keep (fn [[k v]]
             (when (seq v) [id k v]))
           object))))
```


Public API
Storage API
Serializers
EDN
CSV
TTL
JSON
Parsers
Lang Parser
Result Serializer
Query Engine
Planner
Evaluator
Index Storage


Public API
Storage API
EDN
EDN
Parsers
JSON
EDN
EDN
Query Engine
Planner
Evaluator
Index Storage

STORAGE

- Standard database problem
- Just Triples
  - Triples plus Index
- Just Index - triples implied

STORAGE

- Standard database problem
- Just Triples — *early Jena*
  - Triples plus Index — *Datascript*
- Just Index - triples implied — *Asami*

API

- Import/Export
  - via File formats
    - Convert: CSV, TSV, JSON, XML, Parquet…
    - Native Graph: JSON-LD, Turtle, N3, TriG, RDF-XML, GraphML, DOT

API

- Query Languages (text):
  - SPARQL
  - Cypher
  - Datomic
  - GraphQL
- API
  - Gremlin
  - other…

Not fully graph

QUERY ENGINE

- Query Operations
  - Pattern Matching
  - Joins
  - Filtering
  - Path Analysis
  - Other:
    - subqueries, aggregates, scalar functions, coalescence…

QUERY ENGINE

- Query Operations
  - Pattern Matching
  - Joins
  - Filtering
  - Path Analysis
  - Other:
    - subqueries, aggregates, scalar functions, coalescence…

Example:
All triples with property **:first-name**

```
:node1 :first-name “Luke”
:node2 :first-name “Anakin”
:node3 :first-name “Leia”
```

QUERY ENGINE

- Query Operations
  - Pattern Matching
  - Joins
  - Filtering
  - Path Analysis
  - Other:
    - subqueries, aggregates, scalar functions, coalescence…

Example:
All :first-names of people aged 19

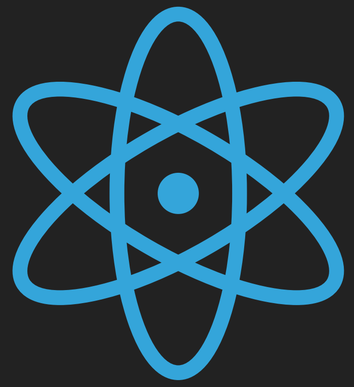
```
:node1  :age 19 | :node1  :first-name “Luke”
:node3  :age 19 | :node3  :first-name “Leia”
```

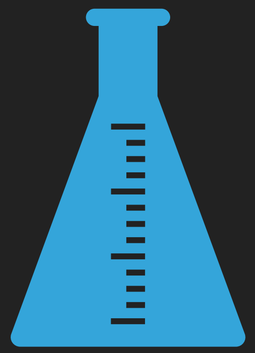
QUERY ENGINE

- Query Operations
  - Pattern Matching
  - Joins
  - Filtering
  - Path Analysis
  - Other:
    - subqueries, aggregates, scalar functions, coalescence…

QUERY ENGINE

- Update Operations
  - Create
    - Combines query operations with import
  - Delete
    - Combines query operations with removal (new operation)
- There is no need for in-place modification in a graph





DATA

LAB 2

Load `src/lab2/datascript.clj`

```
(defn read-triples
  [filename]
  (edn/read-string (slurp filename)))
```

eval:

```
(def triples (read-triples “../../graphs/starwars.edn"))
```

```
(defn triples->datoms
  [triples]
  (let [idents (set (map first triples))
        ids (zipmap idents (range -1 (- (inc (count idents))) -1))
        ident-dec (map
                    (fn [[ident id]] [:db/add id :db/ident ident])
                    ids)
        data (map (fn [[e a v]] [:db/add e a v]) triples)]
    (concat ident-dec data)))

(defn insert-triples
  [conn triples]
  (d/transact conn (triples->datoms triples)))
```

eval:

```
(def conn (create-db))
(insert-triples (create-db) triples)
```

eval:

```
(d/q '[:find ?name :where [:yoda :name ?name]] @conn)

(d/q '[:find ?character ?name :where [?character :name ?name]] @conn)

(d/q '[:find ?character ?name :where [?character :color “#000000"]
                                     [?character :name ?name]]
     @conn)
```

Load `src/lab2/asami.clj`

```
(defn read-triples
  [filename]
  (edn/read-string (slurp filename)))
```

eval:

```
(def triples (read-triples “../../graphs/starwars.edn"))
```

ASAMI

```
(defn create-db []
  (d/connect "asami:mem://dbname"))
```

eval:

```
(def conn (a/create-db))
```

```
(defn triples->datoms
  [triples]
  (let [idents (set (map first triples))
        ids (zipmap idents (range -1 (- (inc (count idents))) -1))
        ident-dec (map
                    (fn [[ident id]] [:db/add id :db/ident ident])
                    ids)
        data (map (fn [[e a v]] [:db/add e a v]) triples)]
    (concat ident-dec data)))

(defn insert-triples
  [conn triples]
  (d/transact conn (triples->datoms triples)))
```

eval:

```
(def tx (insert-triples conn triples))
```

eval:

```
(d/q '[:find ?name :where [:yoda :name ?name]] conn)

(d/q '[:find ?character ?name :where [?character :name ?name]] conn)

(d/q '[:find ?character ?name :where [?character :color “#000000"]
                                     [?character :name ?name]]
     conn)
```

SIMPLE

- Just store flat files of triples
  - cvs, n3, EDN
  - Early Jena did this

INDEXED

- Store a record, build indexes
  - Datascript, Datomic
- Make the index the data
  - Asami, Mulgara

INDEXED

- Store a record, build indexes
  - Redundant data: uses more space
  - Can store large blocks with a triple (e.g. reference to an entire record)
    - examples: Datascript, Datomic
- Make the index the data
  - Less redundancy
  - Adding data to each triple makes indexes larger
    - examples: Asami, Mulgara

MEMORY

- Flat triples
  - Process linearly
- Indexed - Maps
  - Map + triples (Datascript)
  - Nested map (Asami)
- Tree maps: space efficient, ordered
- Hashmaps: faster, unordered

INDEXED STORAGE: EXISTING

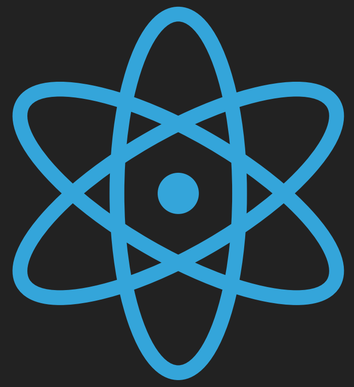
- Libraries
  - Redis, Outboard, BerkeleyDB, RocksDB
- Databases
  - SQLLite, Hypersonic, MySQL, Postgres

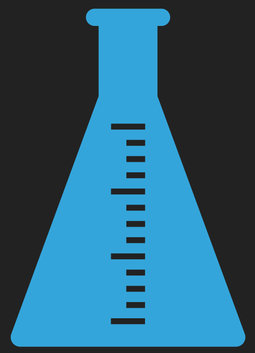
- Store:
  - Statements
  - Groups of statements

DIY

- Manages blocks of data
  - Triple blocks
  - Other blocks (strings, records, etc)

- Can build these out of existing indexes
  - Datomic uses 3rd party indexes for DIY management
  - Asami is designed for this





DATA

LAB 3

Load `src/lab3/flat.clj`

```
(defn load-data
  [filename]
  (edn/read-string (slurp filename)))
```

eval:

```
(def data (load-data "../../graphs/starwars.edn"))
```

INDEXED TRIPLES

Load `src/lab3/indexed.clj`

eval:

```
(def i (assoc-in {} [:yoda :interacts-with :luke] :luke))
(assoc-in i [:yoda :interacts-with :obi-wan] :obi-wan)
```

INDEXED GRAPH

```
(defn add-to-index
  [index [_ _ c :as triple]]
  (assoc-in index triple c))

(defprotocol Index
  (add [this triple])
  (match [this pattern]))

(defrecord IndexedGraph [spo pos osp]
  Index
  (add [this [s p o :as triple]]
    (IndexedGraph. (add-to-index spo triple)
                   (add-to-index pos [p o s])
                   (add-to-index osp [o s p]))))
```

…

eval:

```
(def fi0 (IndexedGraph. {} {} {}))
(def fi1 (add fi0 [:yoda :interacts-with :luke]))
(def fi2 (add fi1 [:yoda :interacts-with :obi-wan])
```

LOAD TO INDEXED GRAPH

```
(defn load-data
  [filename]
  (let [data (edn/read-string (slurp filename))]
    (reduce add (IndexedGraph. {} {} {}) data)))
```

eval:

```
(def data (load-data "../../graphs/starwars.edn"))
```

PATTERN MATCHING

- Find triples according to a pattern
- A single pattern match in the WHERE clause of query languages:
  - Datomic: `[?person :name “LUKE”]`
  - SPARQL: `{?person :name “LUKE”}`
  - Cypher: `(p:Person {name: ‘LUKE’})`

- Cypher is a little different, as it is looking for nodes with an identifier, not an edge

PATTERN MATCHING

- Finding node-to-node edges
- A single pattern match in the WHERE clause of query languages:

  - Datomic: `[:darth-vader :interacts-with ?character]`
  - SPARQL: `{:darth-vader :interacts-with ?character}`
  - Cypher:

```
(:Person {id: 'darth-vader'})-[:INTERACTS_WITH]->(character)
```

PATTERN MATCHING

- “Binds” a variable to a list of values
- Each “Binding” has one value for each variable in the binding

```
[:yoda ?property ?value]
      {?property :name, ?value “Yoda”}
      {?property :scenes, ?value 46}
      {?property :color, ?value “#9ACD32”}
      {?property :interacts-with, ?value :darth-vader}
      {?property :interacts-with, ?value :r2-d2}
      {?property :interacts-with, ?value :luke}
      ...
      ...
```

PATTERN MATCHING: IMPLEMENTATION

- Filter for: `[?person :name “Luke”]`
  - subject = ???
    predicate = `:name`
    object = “Luke”

```
(defn select-by-p-o
  [data predicate object]
  (filter
    (fn [[s p o]] (and (= p predicate) (= o object)))
    data))
```

PATTERN MATCHING: INDEXED

- Depends on index structure

- Map:

  {subject → {predicate → #{object}}}
  {predicate → {object → #{subject}}}
  {object → {subject → #{predicate}}}

INDEXING

- Index with metadata

  {subject → {predicate → {object → statement/metadata}}}
  {predicate → {object → {subject → statement/metadata}}}
  {object → {subject → {predicate → statement/metadata}}}

DATOMIC INDEXING

- EAVT
  AEVT
  AVET
  VAET
- {subject → {predicate → {object → metadata}}}
  {predicate → {subject → {object → metadata}}}
  {predicate → {object → {subject → metadata}}}
  {object → {predicate → {subject → metadata}}}

subject = entity
predicate = attribute
object = value
metadata = transactionId

DATOMIC INDEXING

- EAVT
  AEVT
  AVET
  VAET
- {subject → {predicate → {object → metadata}}}
  {predicate → {subject → {object → metadata}}}
  {predicate → {object → {subject → metadata}}}
  {object → {predicate → {subject → metadata}}}

subject = entity
predicate = attribute
object = value
metadata = transactionId

INDEX SELECTION

[?subject ?predicate ?object]	{subject → {predicate → #{object}}}
[:yoda ?property ?value]	{subject → {predicate → #{object}}}
[?character :name ?name]	{predicate → {object→ #{subject}}}
[?entity ?property 42]	{object → {subject→ #{predicate}}}
[?vader :name “Darth Vader”]	{predicate → {object→ #{subject}}}
[:r2-d2 ?relatedBy :yoda]	{object → {subject→ #{predicate}}}
[:darth-vader :color ?color]	{subject → {predicate → #{object}}}
[:yoda :name “Yoda”]	{subject → {predicate → #{object}}}

FILTERING

- e.g. Find all characters whose name starts with “Darth”
  - Start by getting name triples: `[?person :name ?name]`
  - Then filter:

SPARQL:

```
{?person :name ?name FILTER strStarts(?name, “Darth”)}
```

Datomic:

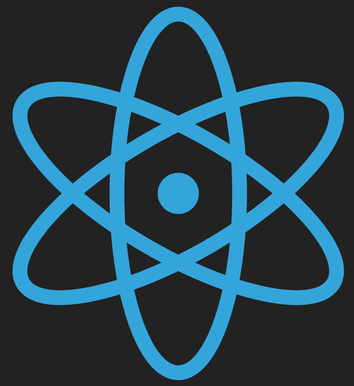
```
[?person :name ?name] (str/starts-with? ?name, “Darth”)
```

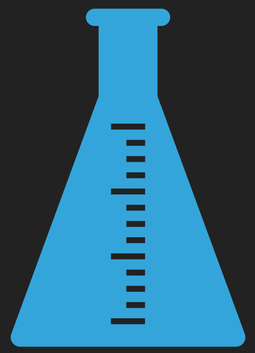
Cypher:

```
MATCH (person:Character) WHERE person.name STARTS WITH 'Darth'
```

PATTERN MATCHING VS FILTERING

- Both identify required triples
- **Matching:** “conceptually” tries to match graph edges
- **Matching:** takes advantage of the triple-pattern for fast lookups
- **Filtering:** iterates over triples
- SQL
  - Matching an filtering are syntactically identical
  - Existence of indexes indicates which should be chosen
- Some SPARQL filters ***can*** be reimplemented using indexes, like SQL





DATA

LAB 4

Load `src/lab4/flat.clj`

```
(defn load-data
  [filename]
  (edn/read-string (slurp filename)))
```

eval:

```
(def data (load-data "../../graphs/starwars.edn"))
```

MATCHING ON FLAT TRIPLES

```
(defn variable? [x] (and (symbol? x) (= \? (first (name x)))))

(defn element-match [p e] (or (variable? p) (= p e)))

(defn match
  [data pattern]
  (filter (fn [triple] (every? #(element-match (nth pattern %)
                                               (nth triple %))
                               (range 3)))
          data))
```

eval:

```
(match data '[?yoda :name "Yoda"])
(match data '[?character :color "#000000"])
(match data '[:yoda :interacts-with ?character])
```

BINDING A MATCH

```
(defn bind
  [data pattern]
  (keep (fn [triple] (reduce (fn [bndg n]
                               (let [p (nth pattern n)]
                                 (if (variable? p)
                                   ;; map the variable to the value
                                   (assoc bndg p (nth triple n))
                                   (if (= p (nth triple n))
                                     ;; pattern element = triple element
                                     bndg
                                     ;; elements not equal: return nil
                                     (reduced nil)))))
                             {} (range 3)))
        data))
```

eval:

```
(bind data '[?yoda :name "Yoda"])
(bind data '[?character :color "#000000"])
(bind data '[:yoda :interacts-with ?character])
```

BINDING VIA VECTORS

```
(defn bind2
  [data pattern]
 (keep (fn [triple] (reduce (fn [bndg n]
                              (let [p (nth pattern n)]
                                (if (variable? p)
                                  (conj bndg (nth triple n))
                                  (if (= p (nth triple n))
                                    bndg
                                    (reduced nil)))))
                            [] (range 3)))
       data)))
```

eval:

```
(def color-pattern '[?character :color "#000000"])
(def vars (vec (filter variable? color-pattern)))
(def black-characters (bind2 data color-pattern))
(map #(zipmap vars %) black-characters)
```

LOAD TO INDEXED GRAPH

Load `src/lab4/indexed.clj`

```
(defn load-data
  [filename]
  (let [data (edn/read-string (slurp filename))]
    (reduce add (IndexedGraph. {} {} {}) data)))
```

eval:

```
(def data (load-data "../../graphs/starwars.edn"))
```

RAW ACCESS TO INDEXED GRAPH

eval:

```
(get-in (:spo data) [:yoda :name])
(get (:osp data) "#191970")
(let [idx (get (:osp data) "#191970")]
  (for [[s pm] idx p (keys pm)] [s p]))
```

VARIABLES AND PATTERNS

```
(defn variable? [x] (and (symbol? x) (= \? (first (name x)))))

(def ? '?)
(def x 'x)

(defn normalize [_ pattern] (mapv #(if (variable? %) ? x) pattern))
```

eval:

```
(def name-pattern '[?character :name ?name])
(vec (filter variable? name-pattern))
(normalize nil name-pattern)
```

MATCHING ON INDEXES

```
(defmulti get-from-index normalize)
(defmethod get-from-index [x x x] [{idx :spo} [s p o]]
  (let [os (get-in idx [s p])] (if (get os o) [[]] [])))
(defmethod get-from-index [x x ?] [{idx :spo} [s p o]]
  (map vector (keys (get-in idx [s p]))))
(defmethod get-from-index [x ? x] [{idx :osp} [s p o]]
  (map vector (keys (get-in idx [o s]))))
(defmethod get-from-index [x ? ?] [{idx :spo} [s p o]]
  (let [edx (idx s)] (for [[p om] edx o (keys om)] [p o])))
(defmethod get-from-index [? x x] [{idx :pos} [s p o]]
  (map vector (keys (get-in idx [p o]))))
(defmethod get-from-index [? x ?] [{idx :pos} [s p o]]
  (let [edx (idx p)] (for [[o sm] edx s (keys sm)] [s o])))
(defmethod get-from-index [? ? x] [{idx :osp} [s p o]]
  (let [edx (idx o)] (for [[s pm] edx p (keys pm)] [s p])))
(defmethod get-from-index [? ? ?] [{idx :spo} [s p o]]
  (for [[s pom] idx [p om] pom o (keys om)] [s p o]))
```

MATCHING ON INDEXED GRAPH

```
(defprotocol Index
  (add [this triple])
  (match [this pattern]))

(defrecord IndexedGraph [spo pos osp]
  Index
  …
  (match [this pattern]
    {:vars (vec (filter variable? pattern))
     :bindings (get-from-index this pattern)}))
```

eval:

```
(match data '[?yoda :name "Yoda"])
(match data '[?character :color “#000000"])

(def rabe (match data '[:rabe ?attribute ?value]))
  (map #(zipmap (:vars rabe) %) (:bindings rabe))
```

eval:

```
(def names (match data '[?character :name ?name]))
(filter (fn [[?character ?name]] (str/starts-with? ?name “Darth"))
        (:bindings names))
```

BUILDING A FILTER

eval:

```
(def fltr '(str/starts-with? ?name "Darth"))
(def vars (:vars names))
(def filter-fn (eval `(fn [~vars] ~fltr)))
(filter filter-fn (:bindings names))
```

FILTER FUNCTION

```
(defn do-filter
  [match-result filter-expr]
  (let [vars (:vars match-result)
        data (:bindings match-result)
        fltr (first filter-expr)
        filter-fn (eval `(fn [~vars] ~fltr))]
    {:vars vars
     :bindings (filter filter-fn data)}))
```

eval:

```
(-> data
    (match '[?character :name ?name])
    (do-filter '[(str/starts-with? ?name "Darth")]))
```

JOINS

- 2 or more pattern matches
- Can expand the number of variables in a binding
- Joins on shared variables
  - No shared variables means a cross product

JOINS

```
[?character :color “#000000”] [?character :name ?name]
```

```
[?character :color “#000000”]
{?character :darth-vader}
{?character :kylo-ren}
```

```
[?character :name ?name]
{?character :darth-vader, ?name “Darth Vader”}
{?character :r2d2, ?name “R2-D2”}
...
{?character :kylo-ren, ?name “Kylo Ren”}
{?character :finn, ?name “Finn”}
...
...
```

JOINS

```
[?character :color “#000000”] [?character :name ?name]
```

```
[?character :color “#000000”]
{?character :darth-vader}
{?character :kylo-ren}
```

```
[?character :name ?name]
{?character :darth-vader, ?name “Darth Vader”}
{?character :r2d2, ?name “R2-D2”}
...
{?character :kylo-ren, ?name “Kylo Ren”}
{?character :finn, ?name “Finn”}
...
...
```

JOINS

```
[?character :color “#000000”] [?character :name ?name]
```

Results:

```
{:vars [?character ?name]
 :bindings [
   {?character :darth-vader, ?name “Darth Vader”}
   {?character :kylo-ren, ?name “Kylo Ren”}
 ]
}
```

JOINS

```
[?character :color “#000000”] [?character :name ?name]
```

```
[?character :color “#000000”]
{?character :darth-vader}
{?character :kylo-ren}
```

JOINS

```
[?character :color “#000000”] [?character :name ?name]
```

```
[?character :color “#000000”]
{?character :darth-vader}
{?character :kylo-ren}
```

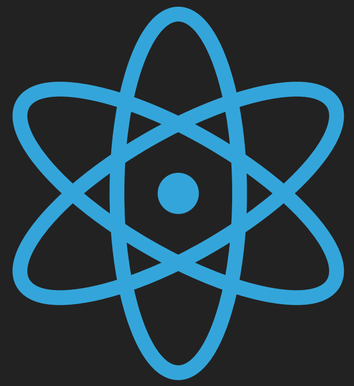
```
[:darth-vader :name ?name]
{?name “Darth Vader”}
```

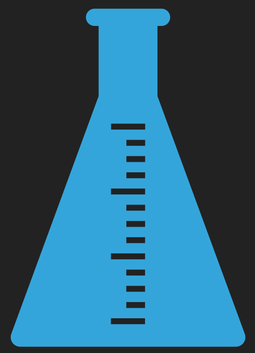
JOINS

```
[?character :color “#000000”] [?character :name ?name]
```

```
[?character :color “#000000”]
{?character :darth-vader}
{?character :kylo-ren}
```

```
[:kylo-ren :name ?name]
{?name “Kylo Ren”}
```





DATA

LAB 5

Load `src/lab5/indexed.clj`

```
(defn load-data
  [filename]
  (edn/read-string (slurp filename)))
```

eval:

```
(def data (load-data "../../graphs/starwars.edn"))
```

STARTING A JOIN

eval:

```
(def color-pattern '[?character :color “#000000"])
(def name-pattern '[?character :name ?name])

(def color-result (match data color-pattern))
(def vars (:vars color-result))
```

eval:

```
(for [binding (:bindings color-result)]
  binding)

(for [binding (:bindings color-result)]
  (rewrite-pattern vars binding '[?character :name ?name]))
```

MATCH ON REWRITTEN PATTERN

eval:

```
(for [binding (:bindings color-result)]
  (match data (rewrite-pattern vars
                               binding
                               '[?character :name ?name])))

(for [binding (:bindings color-result)
      new-values (:bindings
                  (match data
                    (rewrite-pattern vars
                                     binding
                                    '[?character :name ?name])))]
 (vec (concat binding new-values)))


(join data color-result '[?character :name ?name])
```

JOIN RESULT

eval:

```
(join data color-result '[?character :name ?name])
```