Keeping Simple Things Simple in Data Processing

Elango Cheran
April 15, 2016

Functional Programming

- Separate data and functions
- “Methods” => OOP Objects
  - Methods => functions tied to data => complected => :-(

Functional Programming and Object Oriented Programming

- Are these notions compatible?
- How many functional languages do not embrace OOP?
- How many languages overall do not embrace OOP?

Data Oriented Programming

- Represent and manipulate data, unadorned
    - Promote plain data structures
    - Promote common data operations
    - Don’t complect with type hierarchies, objects, etc.
- Any language can do this

Tradeoffs of Being Data Oriented

- Costs of plain data
  - Static typing - type safety (compiler checking)
  - Objects do “encapsulation”
  - Are you willing to sacrifice *safety*??

Tradeoffs of Being Data Oriented

- Gains of plain data
  - Simplicity and power
  - Are you willing to lose all that *power*??
    - Do most people know what they’re missing? (blub)
- Tradeoffs
  - No right/wrong, but a choice based on a value system

My Assertion

- Static typing is great when…
  - The requirements are known ahead of time and don’t change
  - The extra value given by static types is more than it costs
  - Examples: banks, airplanes
  - The abstractions become powerful but have more inertia

My Assertion

- Dynamic typing is great when…
  - Requirements change and grow
  - Flexibility and power give benefits via concise code
  - Examples: most everything
  - Code can organically become DSL-like but requires strong discipline
  - Disclaimer: you always want schemas around the data passed between components

The gap

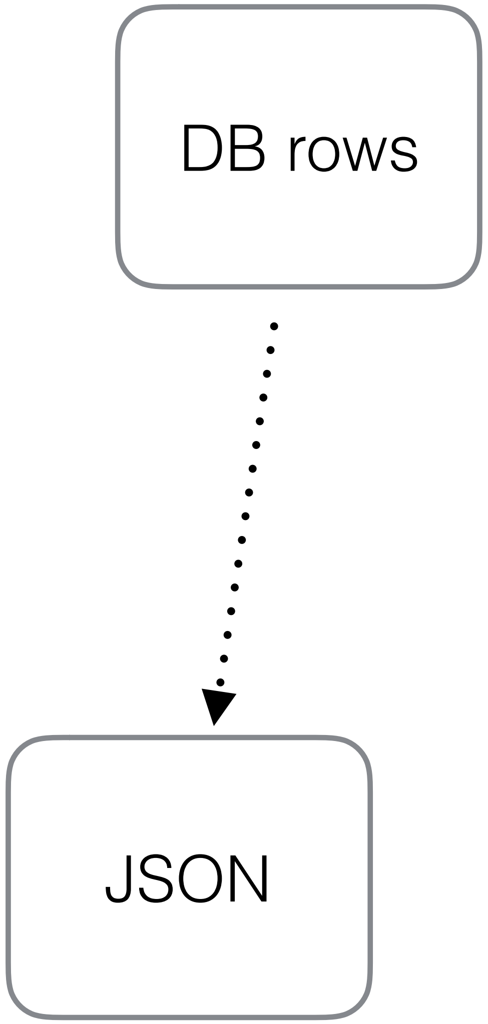
- Most people understand static typing
- Most people understand object oriented programming
- Most people don’t understand the simplicity that they’re missing
    - (Or the complexity that they’re living with, but that’s hard to show without a comparison)

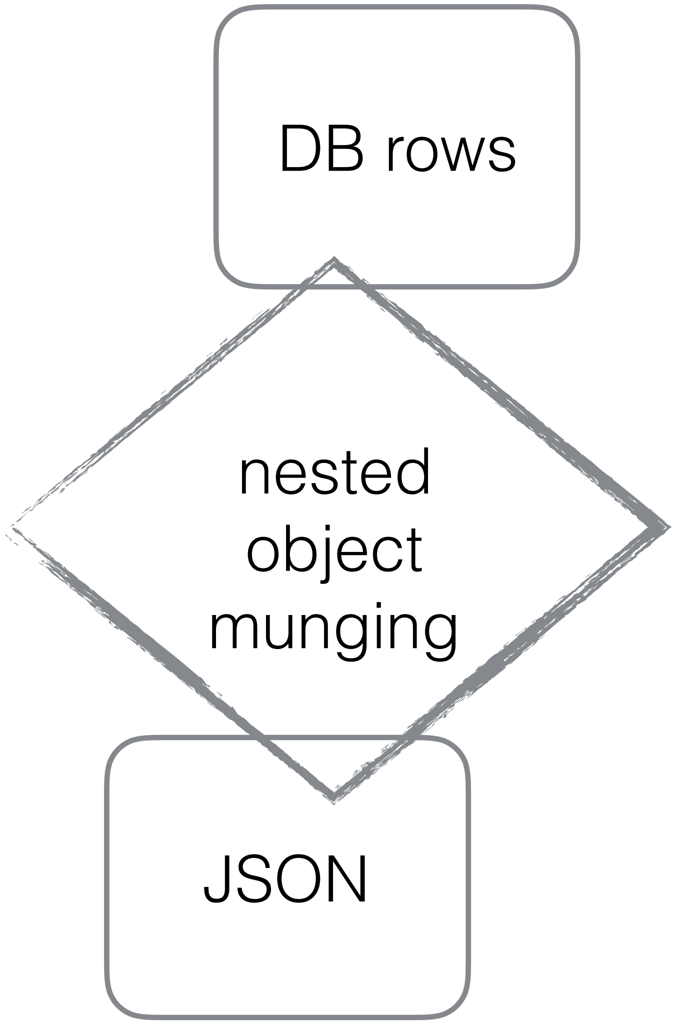
Clojure, simplicity, and data

- Eschews OOP paradigm
  - Namespaces are used to organize data and functions
  - No extra hoops to use data or functions
    - Instantiation, access modifiers, type hierarchies, etc.
- Everything can be treated as plain data structures
  - Maps - objects/beans/structs/records/case classes
  - Vectors + maps = JSON, but Clojure/EDN is more practical and extensible

Example 1 - serving up DB data to a webpage

- You can't reach into a DB directly from a webpage
- Need to run a web server
- Fetch results from tables (SQL), serve up as JSON for JS libraries
- Each monitoring metric stored in a separate table with separate schema and custom column names
- Need to convert DB rows to JSON


DB rows
JSON



DB rows

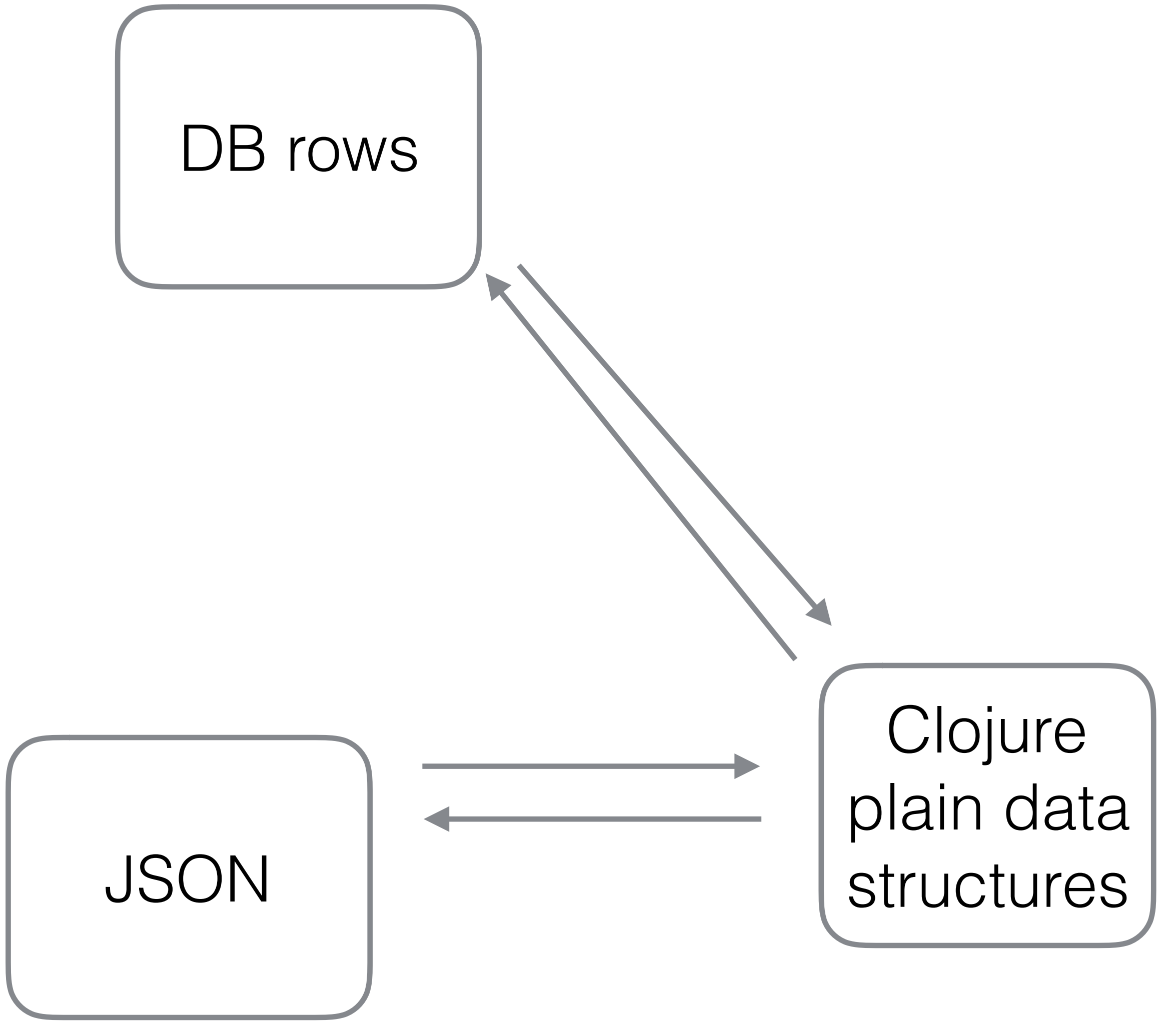
nested object munging

JSON

Example 1 - serving up DB data to a webpage

Clojure:

```
(defn rows->json
  "take the seq of maps that clojure jdbc gives you
as a query result and format as json"
  [rows]
  (let [new-rows (map normalize-row rows)]
    (json/generate-string new-rows)))
```


DB rows
JSON
Clojure plain data structures

Example 2 - dealing with nested IDL schemas (Avro/Protobuf)

- Scenario: have to deal with many, many nested Protobuf + Avro messages
- Avro is the most interesting part
    - Avro was designed for Hadoop, is the defacto schema/serialization format
    - Writing / printing serialized data is difficult

Example 2 - dealing with nested IDL schemas (Avro/Protobuf)

- My pet project: create a function to take an Avro Schema object and return N random records adhering to the Schema

- Allow Schema to contain nesting, enum types, array types, and union types

- Logic impl:

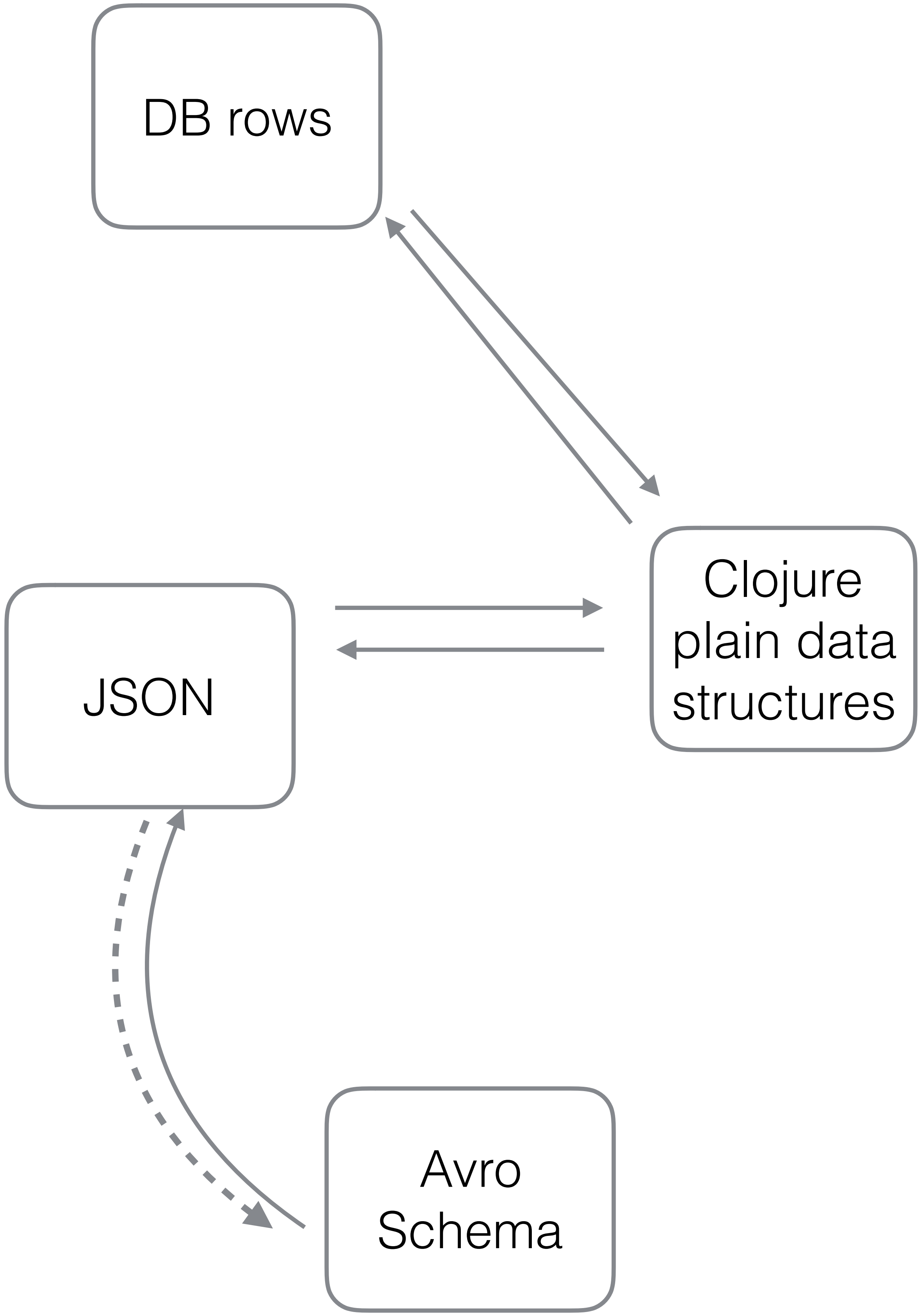
    - Schema obj -> JSON (via toString) to get structure

    - Convert JSON -> nested map

    - Nested map to smaller Clojure data structure to simplify

    - Clojure data structure + Avro library -> serialized binary

    - Clojure data structure ->
      generators of Clojure data structures ->
      generators of serialized Avro binary

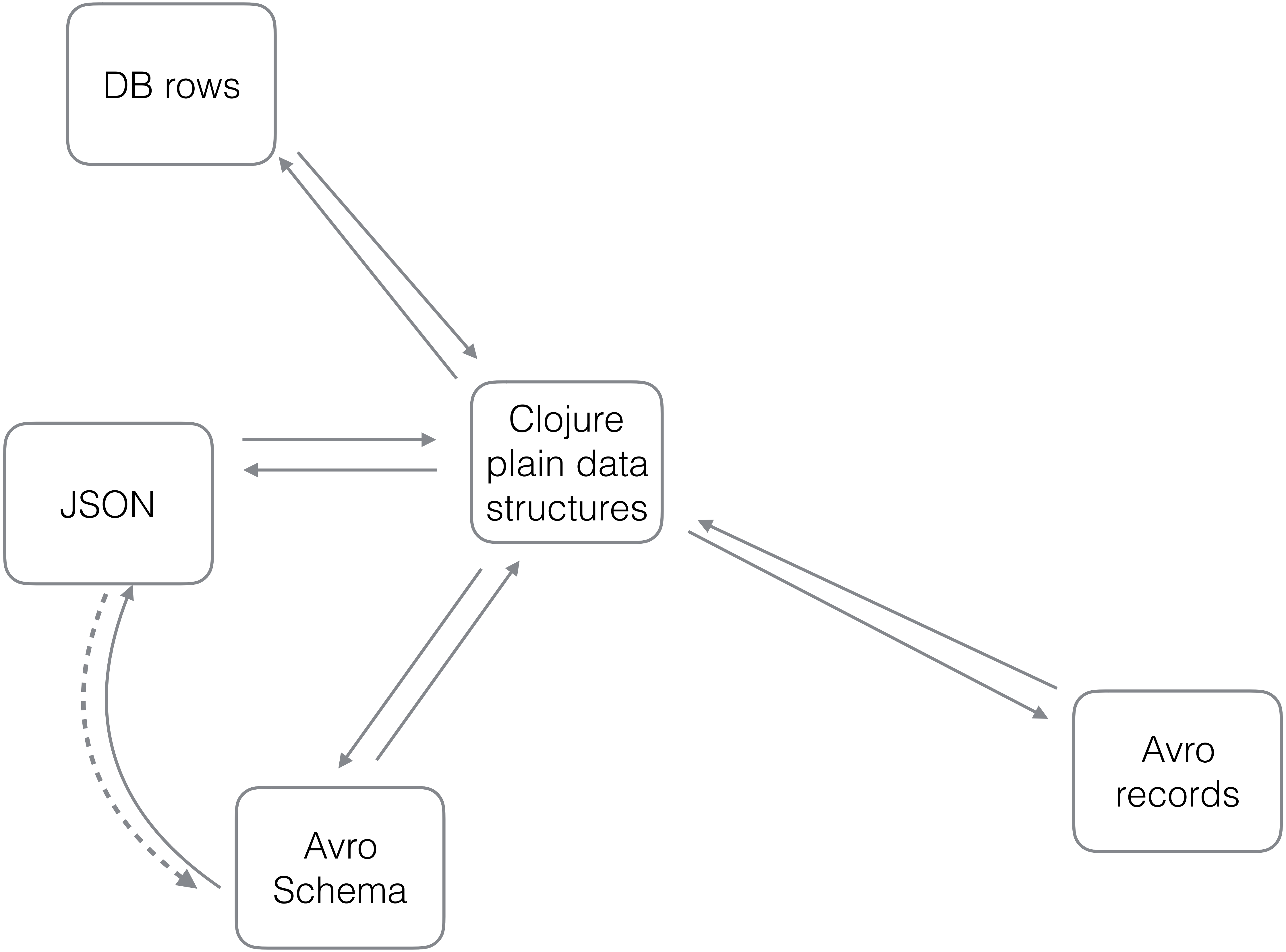


DB rows

Clojure plain data structures

JSON

Avro Schema



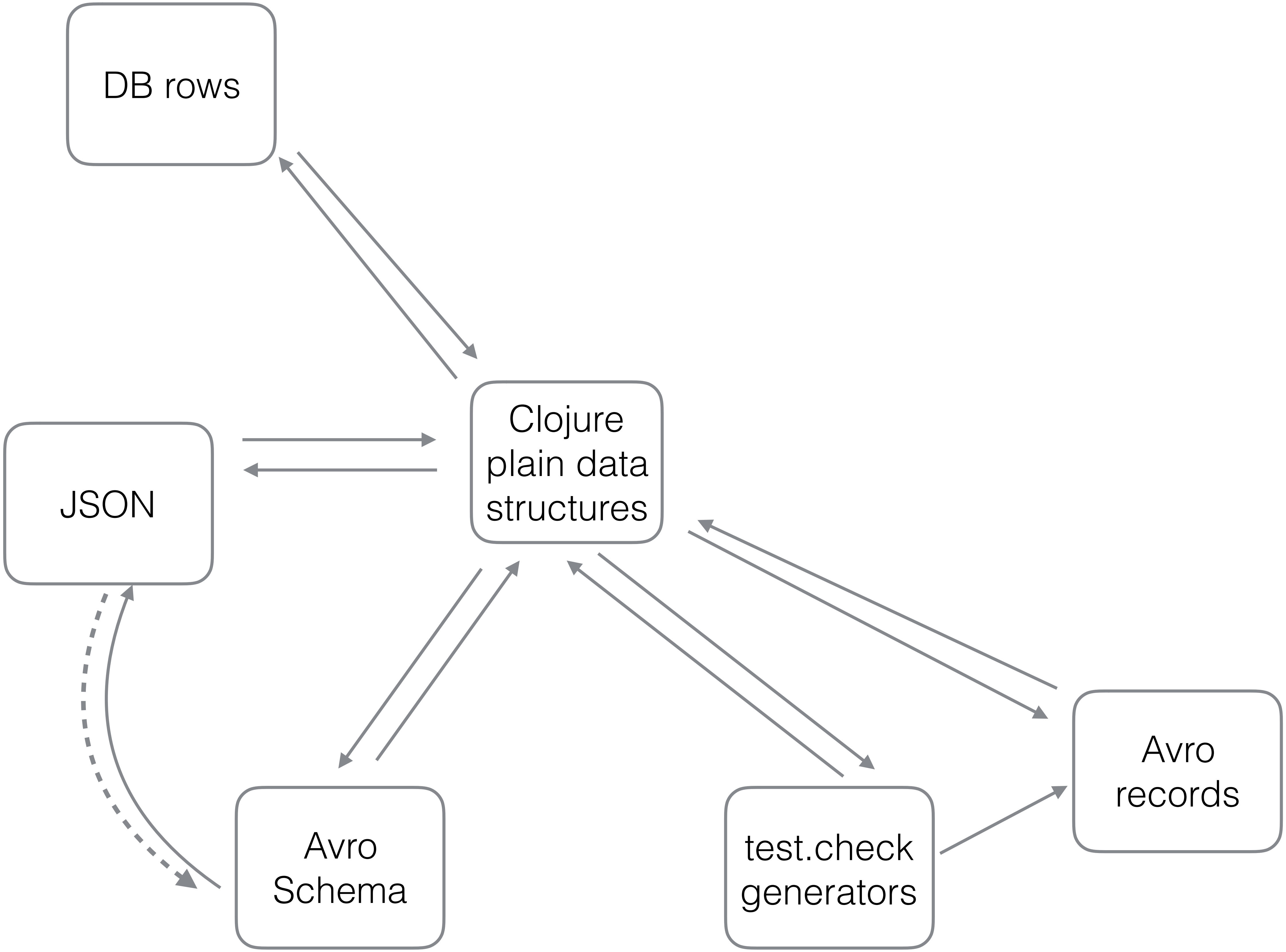
DB rows

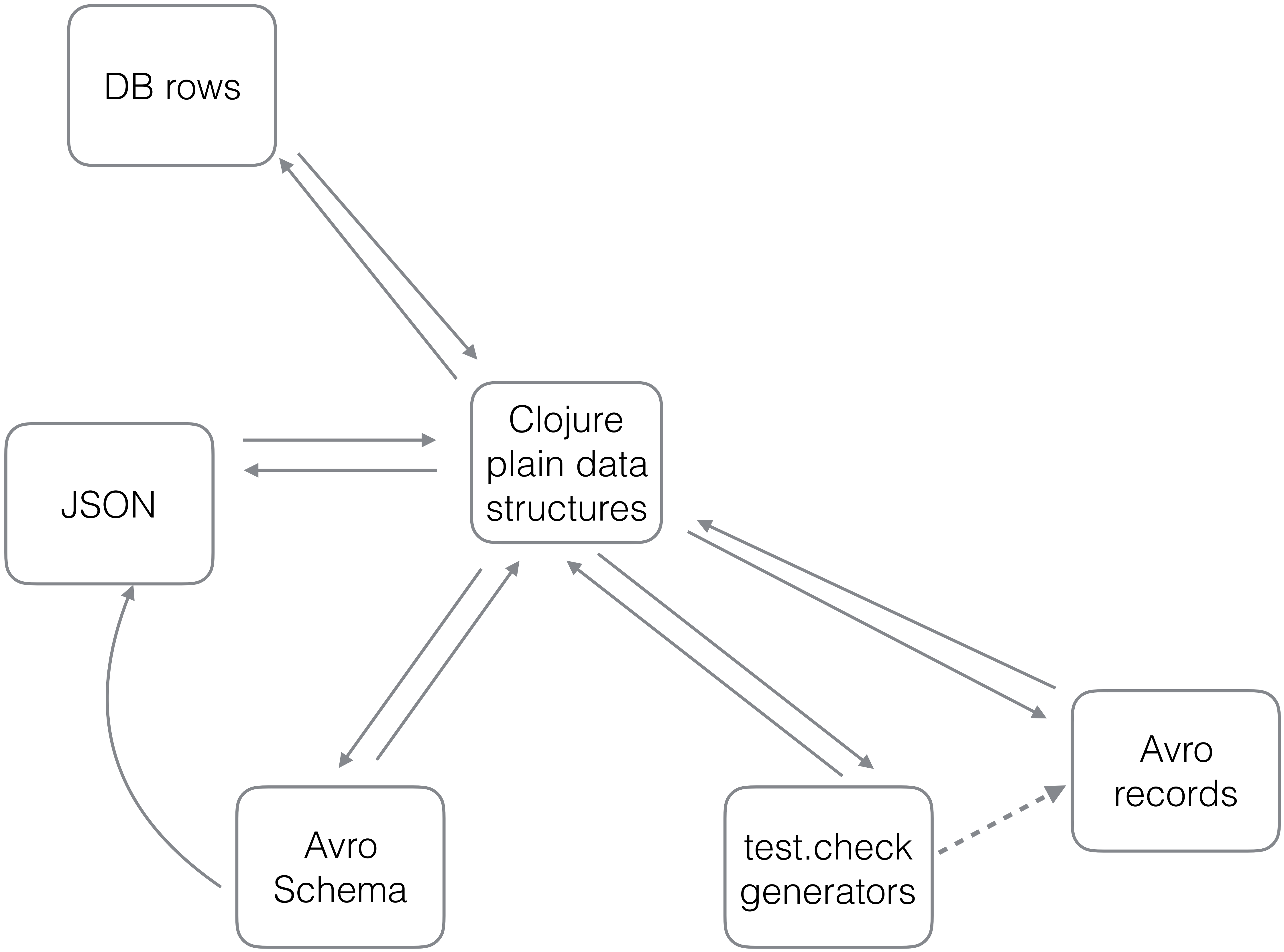
JSON

Clojure plain data structures

Avro Schema

Avro records


DB rows
JSON
Clojure plain data structures
Avro Schema
test.check generators
Avro records



DB rows

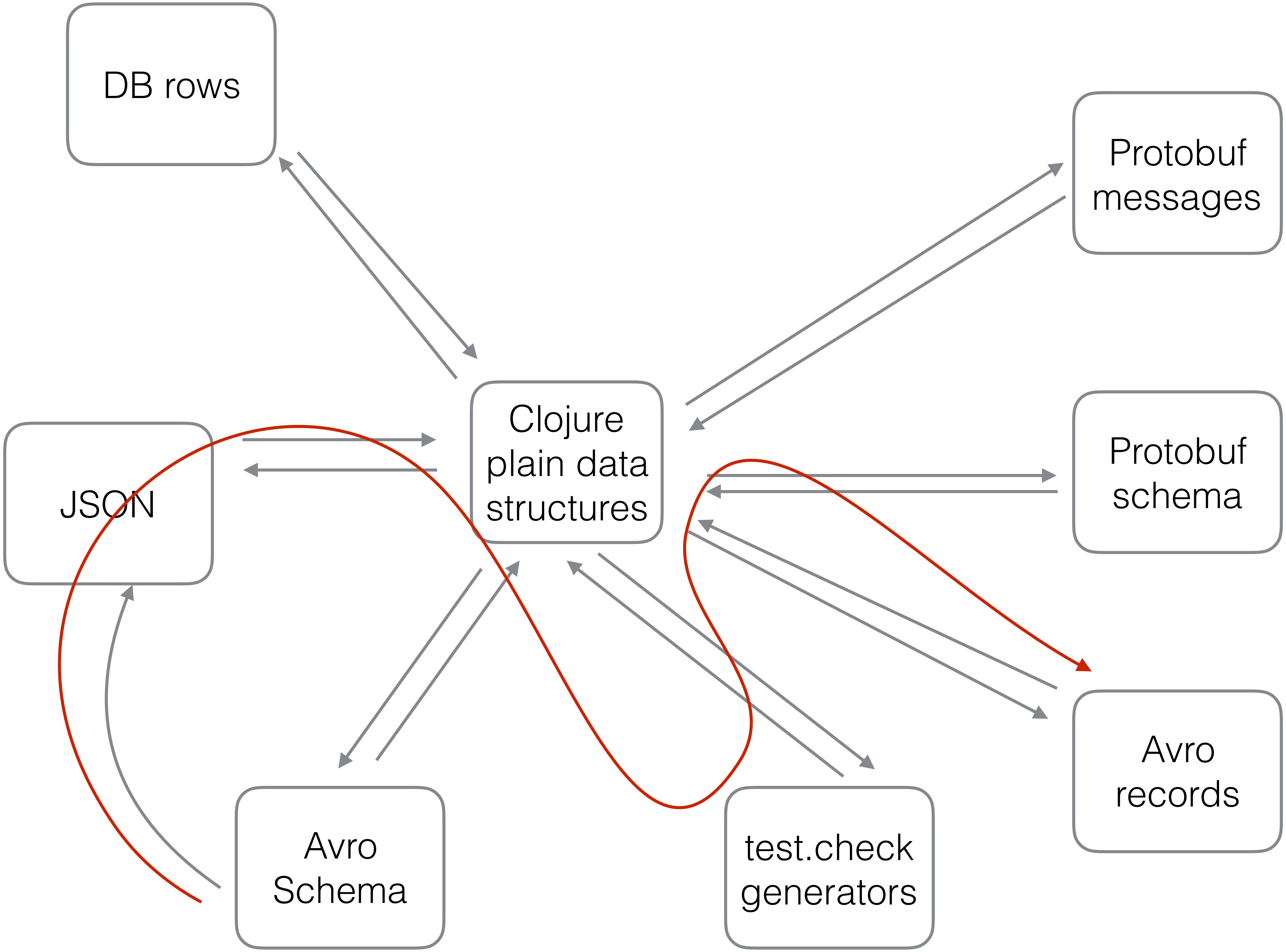
JSON

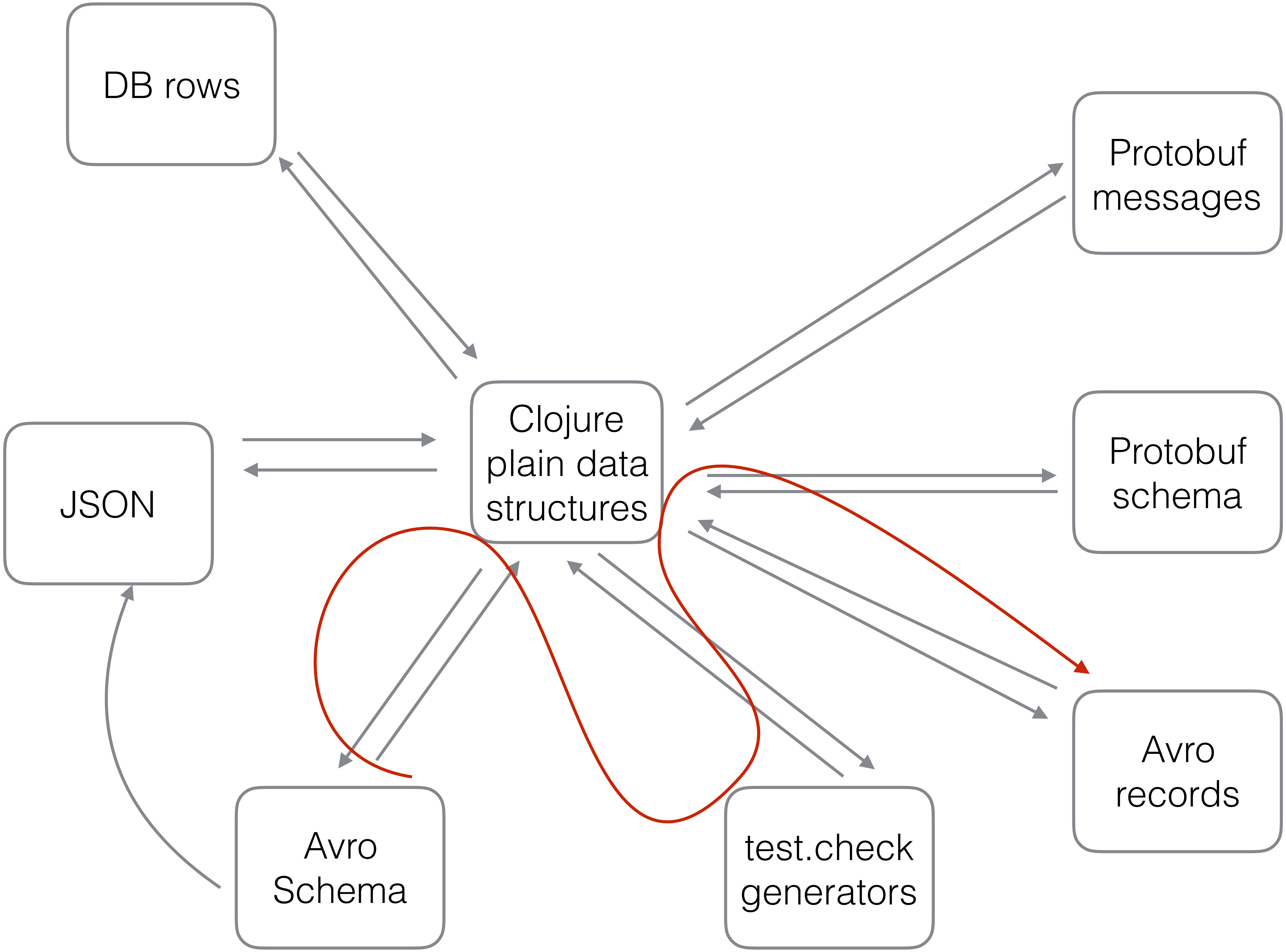
Clojure plain data structures

Avro Schema

test.check generators

Avro records


DB rows
Protobuf messages
JSON
Clojure plain data structures
Protobuf schema
Avro records
Avro Schema
test.check generators


DB rows
Protobuf messages
JSON
Clojure plain data structures
Protobuf schema
Avro records
Avro Schema
test.check generators

Example 2 - dealing with nested IDL schemas (Avro/Protobuf)

```
(defn record-gen-with-overrides
  "given an Avro schema, returns a generator of Avro records as Clojure maps
(unserialized)
the override-map is a map of vectors (of strings) to generators.  the vector of
strings (map key) indicates the nesting of the field, and the generators (map value)
will be used instead of default generators to generate values in that field"
  [schema override-map]
  (let [json-str (.toString schema)
        json-map (json/parse-string json-str)
        schema-clj-map (schema->clj-map json-map)
        new-schema-clj-map (override-schema-clj-map schema-clj-map override-map)
        clj-record-gen (schema-clj-map->generator new-schema-clj-map)]
    clj-record-gen))
```

Example 3 - configuration

- Scenario: you want to handle configurations intelligently
- Different configurations for: (production, qa, dev) and (unit test, integration test)
- You want unit test configs to just override a few settings from default dev settings

Example 3 - configuration

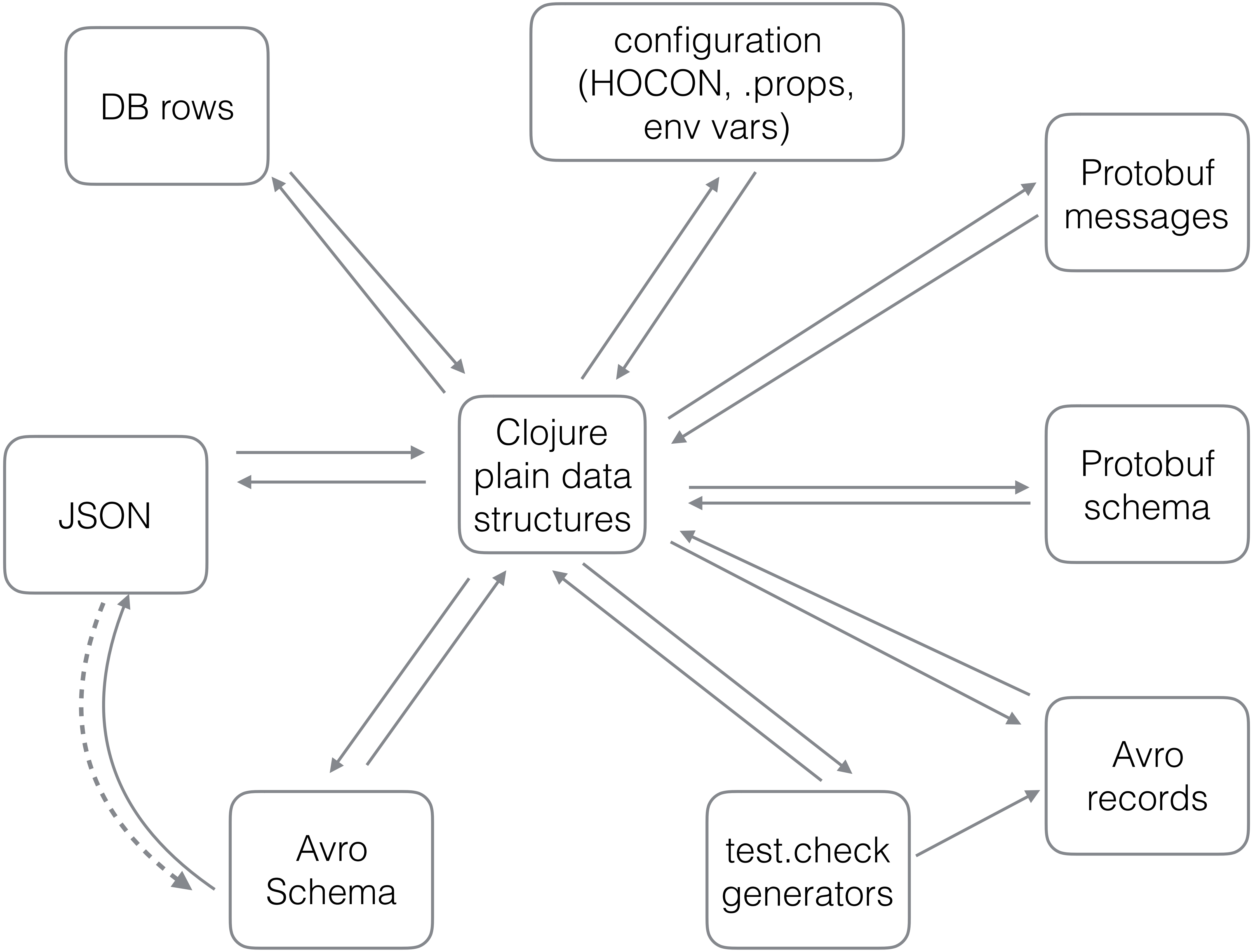
- One way I've done configuration is using HOCON (Typesafe Config)
  - Config objects are immutable (yay!)
  - Config is an object in a class hierarchy with interfaces, etc.
  - Operations:
    - merge - withFallbackConfig
    - no real way to assoc-in
- Difficult to create a Config from literal data (ex: literal Map)

HOCON Example

```
val INPUT_FILTER_MAP_CONFIG_PREFIX = "myapp.input-filter-map"

val INPUT_FILTER_MAP = {
  val pathPrefix = INPUT_FILTER_MAP_CONFIG_PREFIX + "."
  Map(
    (pathPrefix + "key") -> Seq(
      "val1",
      “val2”,
      ...
    ).asJava,
    ...
  )
  }


def reformatConfig(config: Config): Config = {
  val newInputFilterMap = INPUT_FILTER_MAP
  val newInputFilterMapConfig = ConfigFactory.parseMap(newInputFilterMap)
  val newConfig = newInputFilterMapConfig.withFallback(config)
  newConfig
}
```

Example 3 - configuration

- The secret: configuration is data, too
  - Represent everything as maps
  - Use merge, merge-with, assoc, assoc-in, update, update-in, …
  - This is what the many Clojure config libraries do (environ, et al.)



DB rows

configuration (HOCON, .props, env vars)

Protobuf messages

JSON

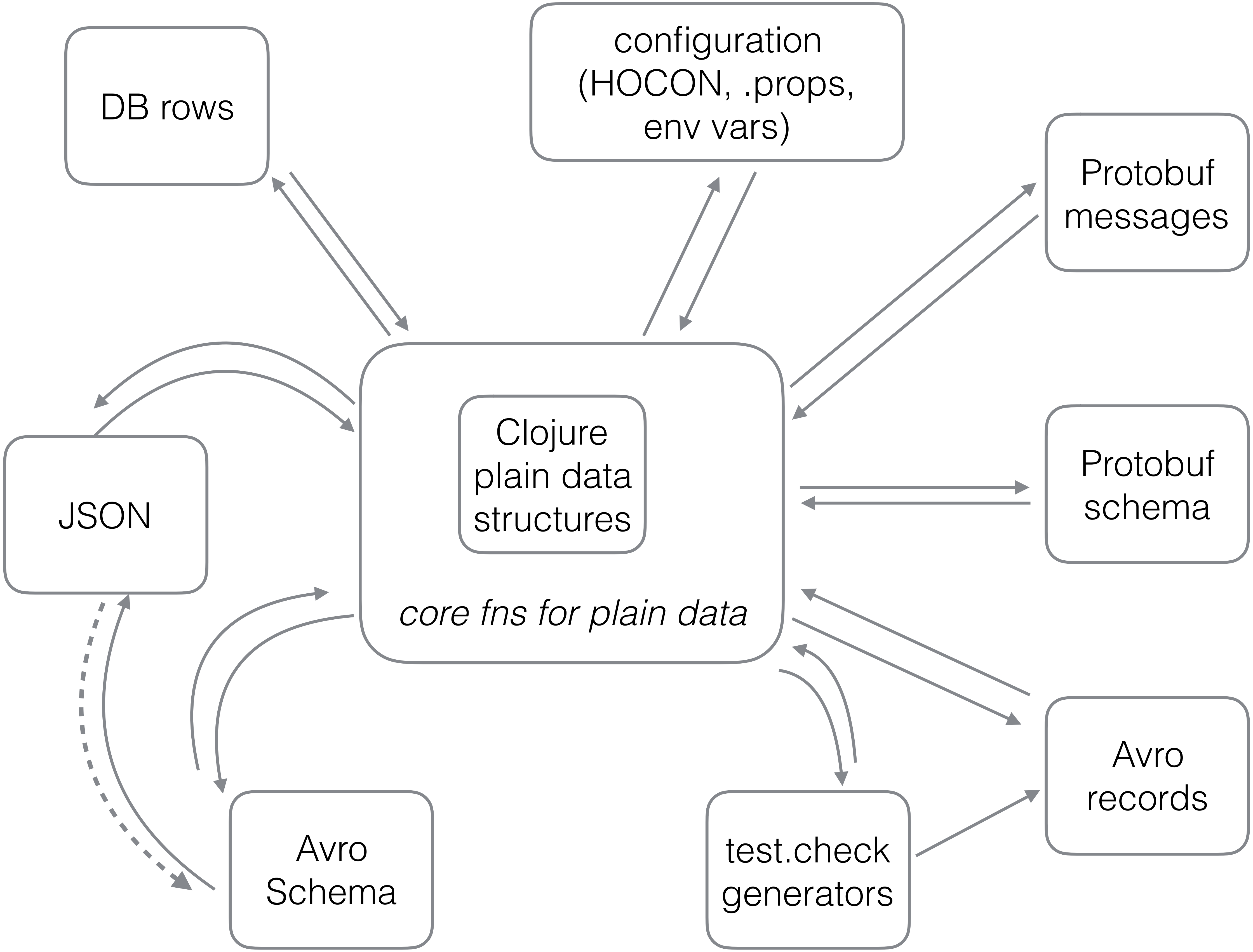
Clojure plain data structures

Protobuf schema

Avro Schema

test.check generators

Avro records



DB rows

configuration (HOCON, .props, env vars)

Protobuf messages

JSON

Clojure plain data structures

core fns for plain data

Protobuf schema

Avro records

Avro Schema

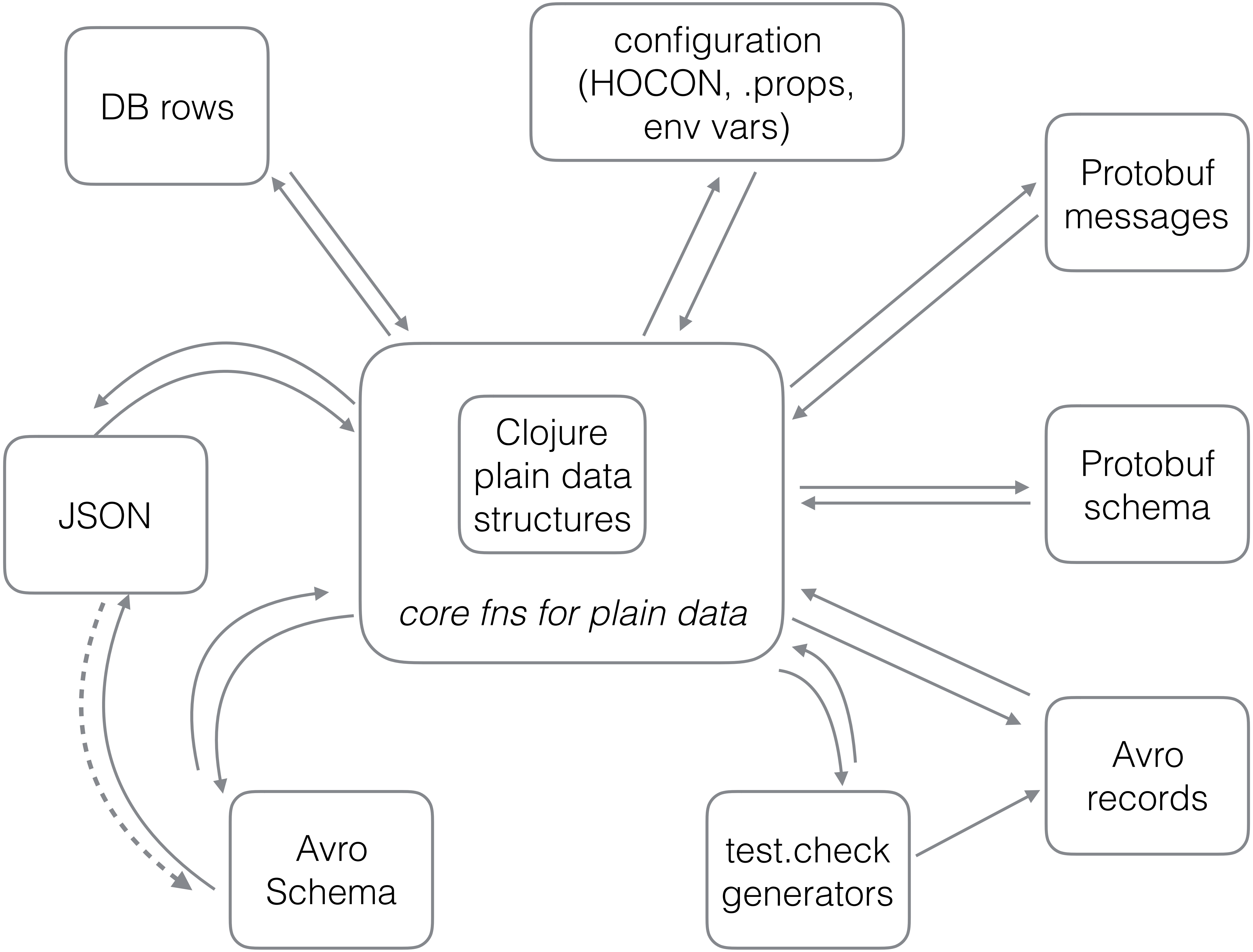
test.check generators

Example 4 - testing

```
repl> parse(""" { "numbers" : [1, 2, 3, 4] } """)
res: JObject(List((numbers,JArray(List(JInt(1), JInt(2), JInt(3), JInt(4))))))

// if writing a test that uses this as the expected value
val expectedJson = JObject(List((numbers,JArray(List(JInt(1), JInt(2), JInt(4),
JInt(3))))))



// you’ll get something like this:
ERROR: expecting [JObject(List((numbers,JArray(List(JInt(1), JInt(2), JInt(4),
JInt(3)))))))] was [JObject(List((numbers,JArray(List(JInt(1), JInt(2), JInt(3),
JInt(4)))))))]
```

Example 4 - testing

```
(def exp-map {"numbers" [1 2 4 3] "letters" ["b"]})

(expect exp-map (json/parse-string " { \"numbers\" : [1, 2, 3, 4], \"letters
\" : [\"a\"] } "))



;at the CLI when running lein test =>

(expect
 exp-map
 (json/parse-string
  " { \"numbers\" : [1, 2, 3, 4], \"letters\" : [\"a\"] } "))

           expected: {"numbers" [1 2 4 3], "letters" ["b"]}
                was: {"numbers" [1 2 3 4], "letters" ["a"]}

           in expected, not actual: {"letters" ["b"], "numbers" [nil nil 4 3]}
           in actual, not expected: {"letters" ["a"], "numbers" [nil nil 3 4]}
```



DB rows

configuration (HOCON, .props, env vars)

Protobuf messages

JSON

Clojure plain data structures

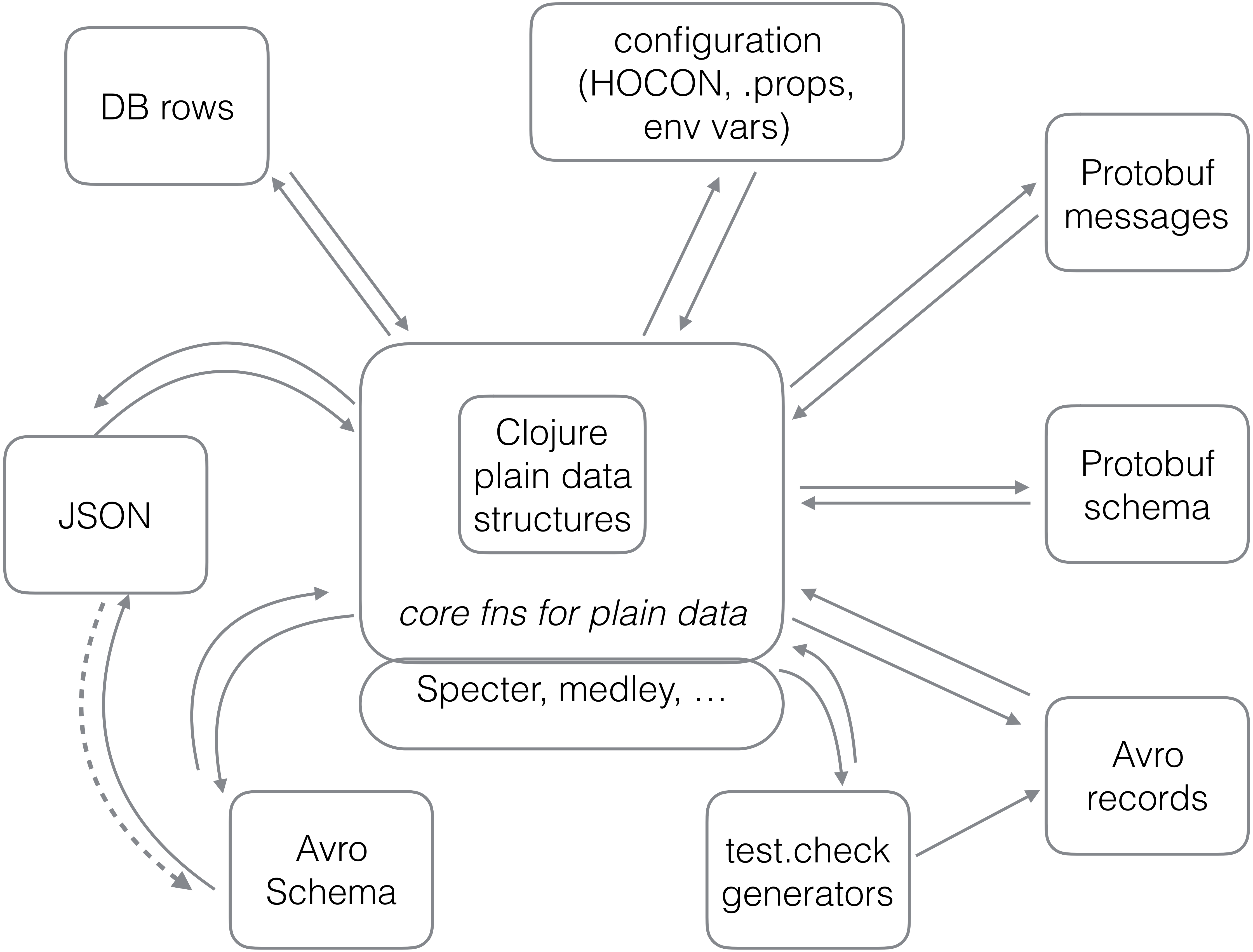
core fns for plain data

Protobuf schema

Avro records

Avro Schema

test.check generators



DB rows

configuration
(HOCON, .props,
env vars)

Protobuf
messages

JSON

Clojure
plain data
structures

core fns for plain data

Protobuf
schema

Specter, medley, …

Avro
records

Avro
Schema

test.check
generators

Example 5 - Spark

- Serializing in Spark -> Kryo
- Must register serializer for every type put into an RDD
- Function values (closures) must have serializable environment
- If you want to deal with Avro data in Spark, you need to create a case class to wrap each Avro generated class

Example 5 - Spark

- Regular Spark serialization - an excerpt

```
kryo.register(classOf[Array[Boolean]])
kryo.register(classOf[Array[Double]])
kryo.register(classOf[Array[Short]])

// more stuff copied from common Kryo registrator
kryo.register(classOf[Array[(_,_)]])        // why doesn't chill handle this?
kryo.register(classOf[Array[Object]])       // why doesn't chill handle this?
kryo.register(classOf[Array[scala.collection.Iterable[_]]])
kryo.register(classOf[scala.collection.immutable.::[_]])
kryo.register(Class.forName("scala.collection.immutable.Nil$"))
```

Example 5 - Spark

- In Clojure, if you use plain data structures, Flambo has your Kryo needs covered
    - Clojure's pr / pr-str is a "serializer", read is a "deserializer"
    - Nippy is a more efficient de-/serializer for Clojure data, with other benefits
    - Flambo (Spark), Datasplash (Dataflow), etc. already include Nippy on your behalf