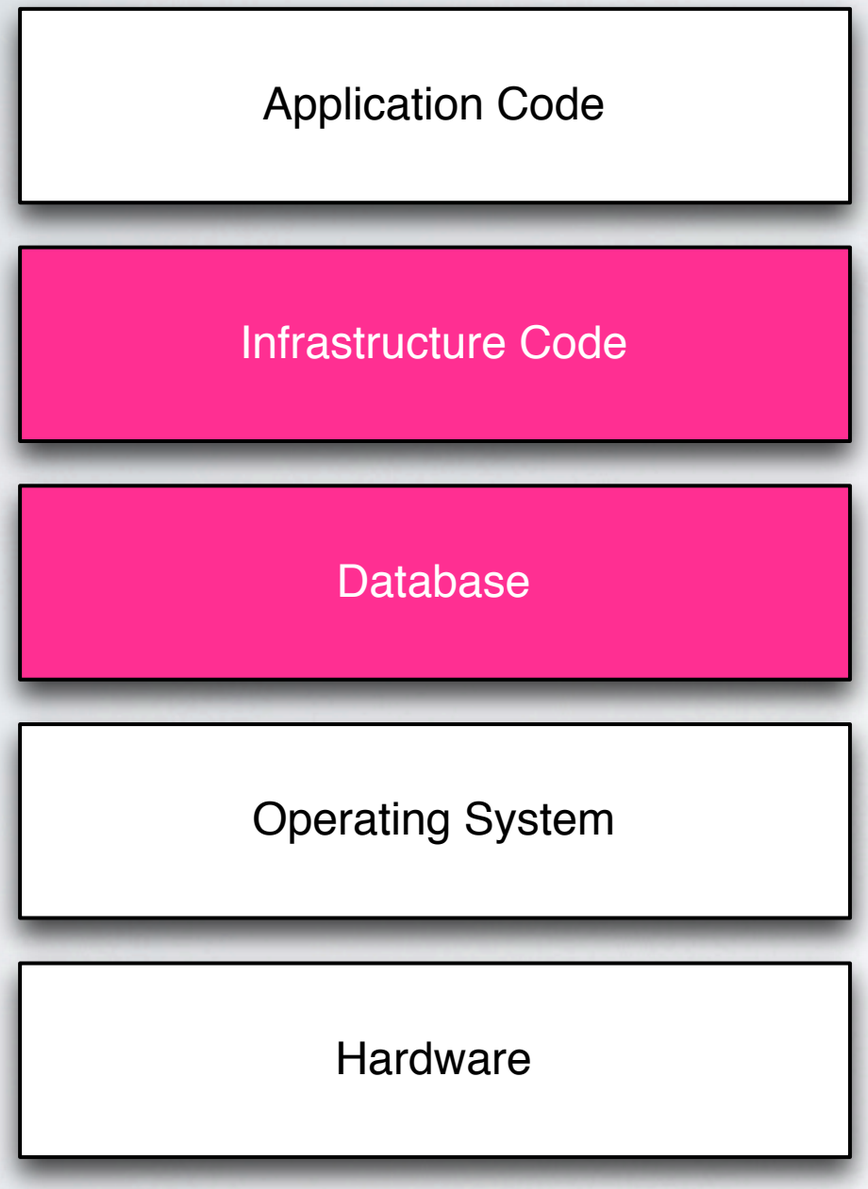
BUILDING A DATA PLATFORM WITH DATOMIC

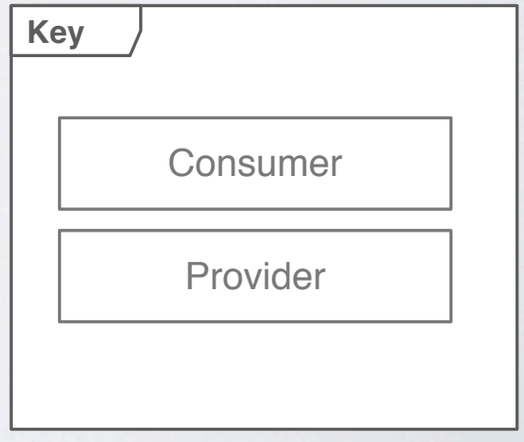
Antonio Andrade

Clojure/conj 2013

WHAT IS A DATA PLATFORM



Software system that defines a **data representation**, abstracts its **persistence** and **retrieval,** and provides tools to **manipulate** that data.



BACKGROUND

- Software Architect at Appian
- Build your own software
- Customers build their own applications and data models
- Highly dynamic system. Everything happens at runtime
- **Can't** make too **many modeling** decisions early on

WHERE DOES APPIAN POWER COME FROM?

- Transactional
- Rich data modeling: Arbitrary tree structures
- Data manipulation and query power: APL variant
- Data consistency: Multiple queries on data snapshot
- Performance. Ultra high-speed, in-memory, column-oriented data store
- Programming model: Embedded programming language

ROADMAP

- Data models & schemas
- Datomic data model
- A self-describing schema definition
- Techniques for versioning and data migration
- Enforcing data constraints
- Reified Transactions

DATA MODELS

- Describe the **structure** of the *objects* represented by a system and the **relationship** between those *objects*.

- Always exists, either implicitly or explicitly.

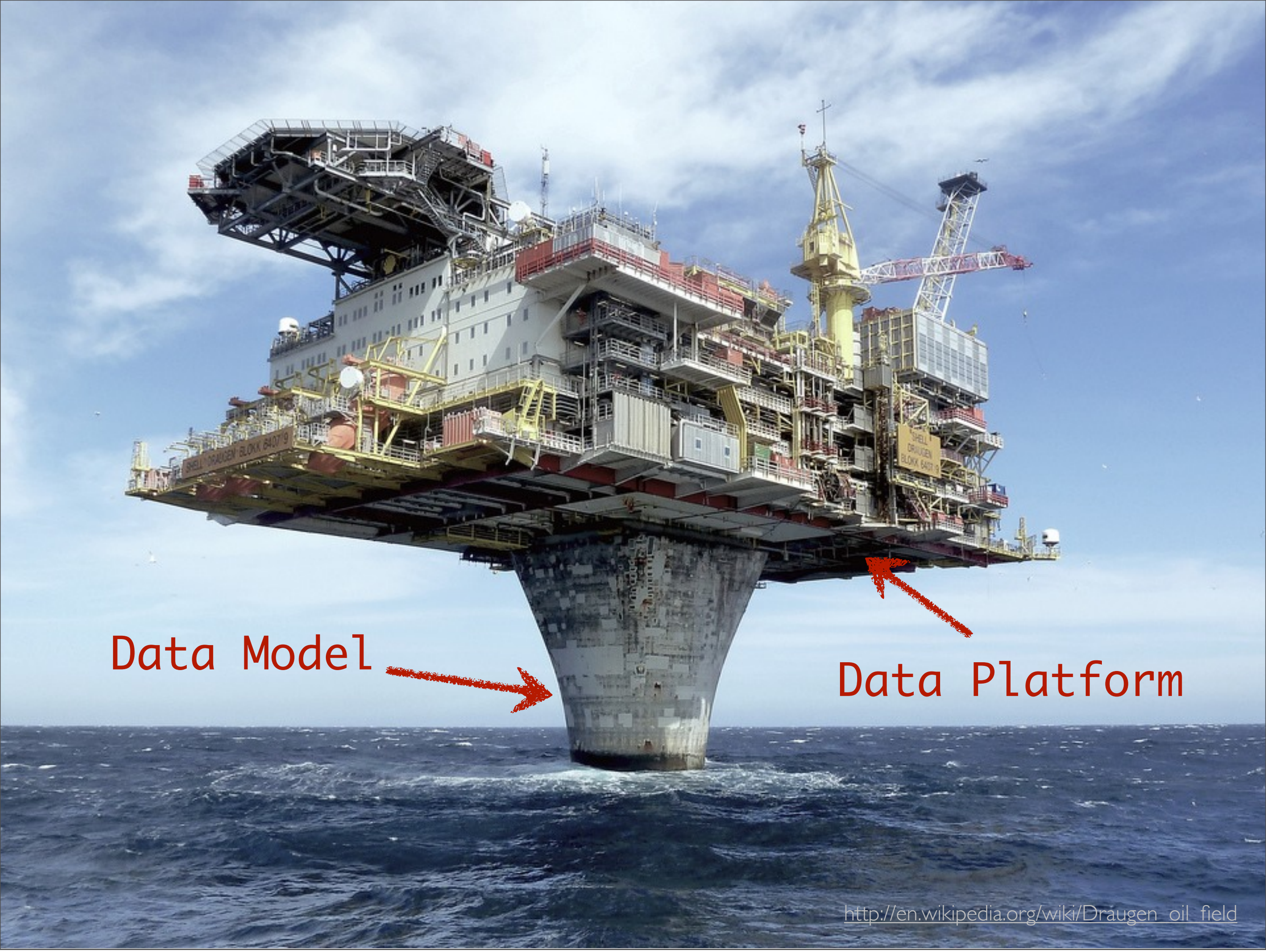
- Really valuable as a communication tool and documentation.

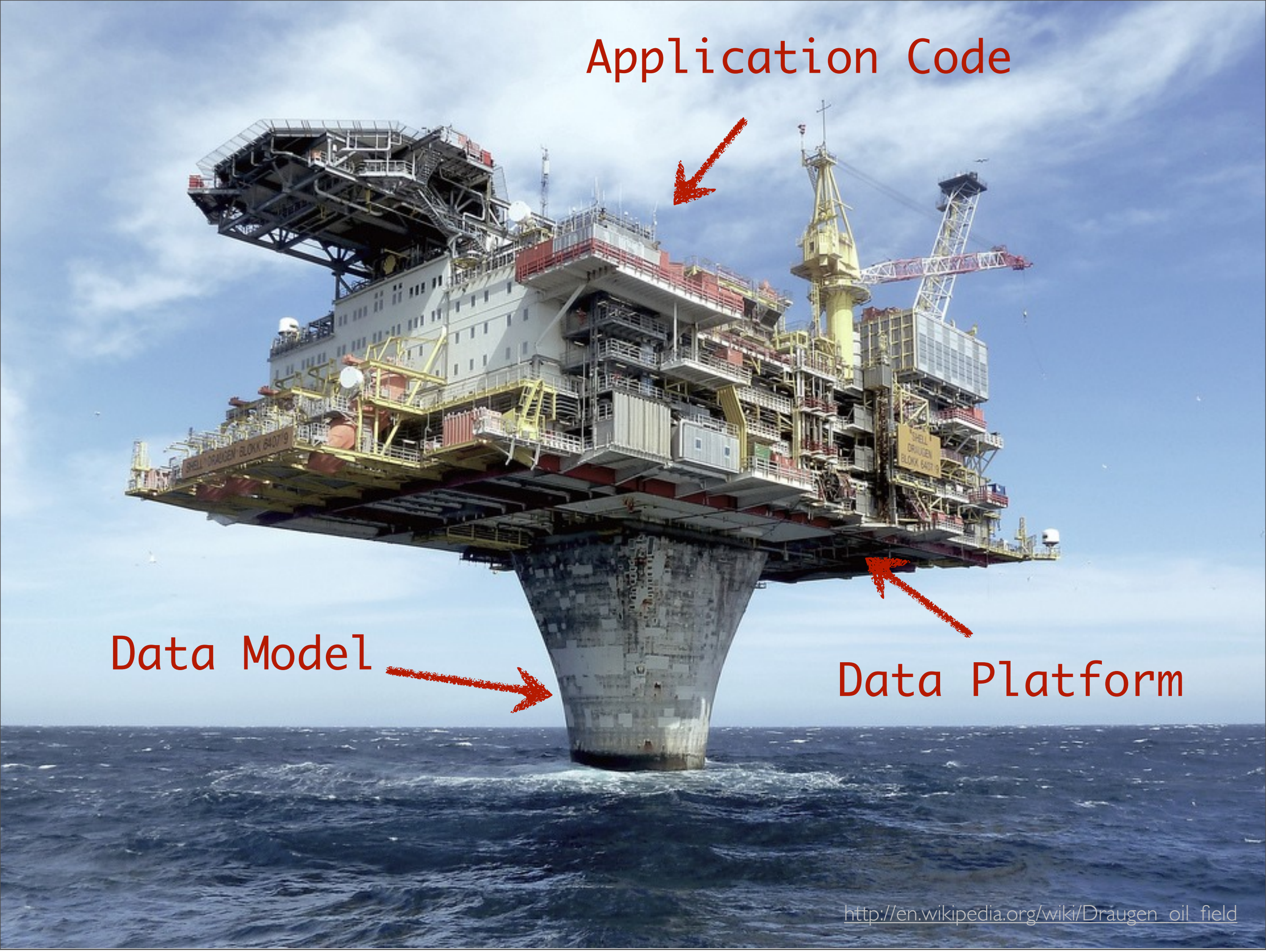
- A schema is a *language* used to describe them.

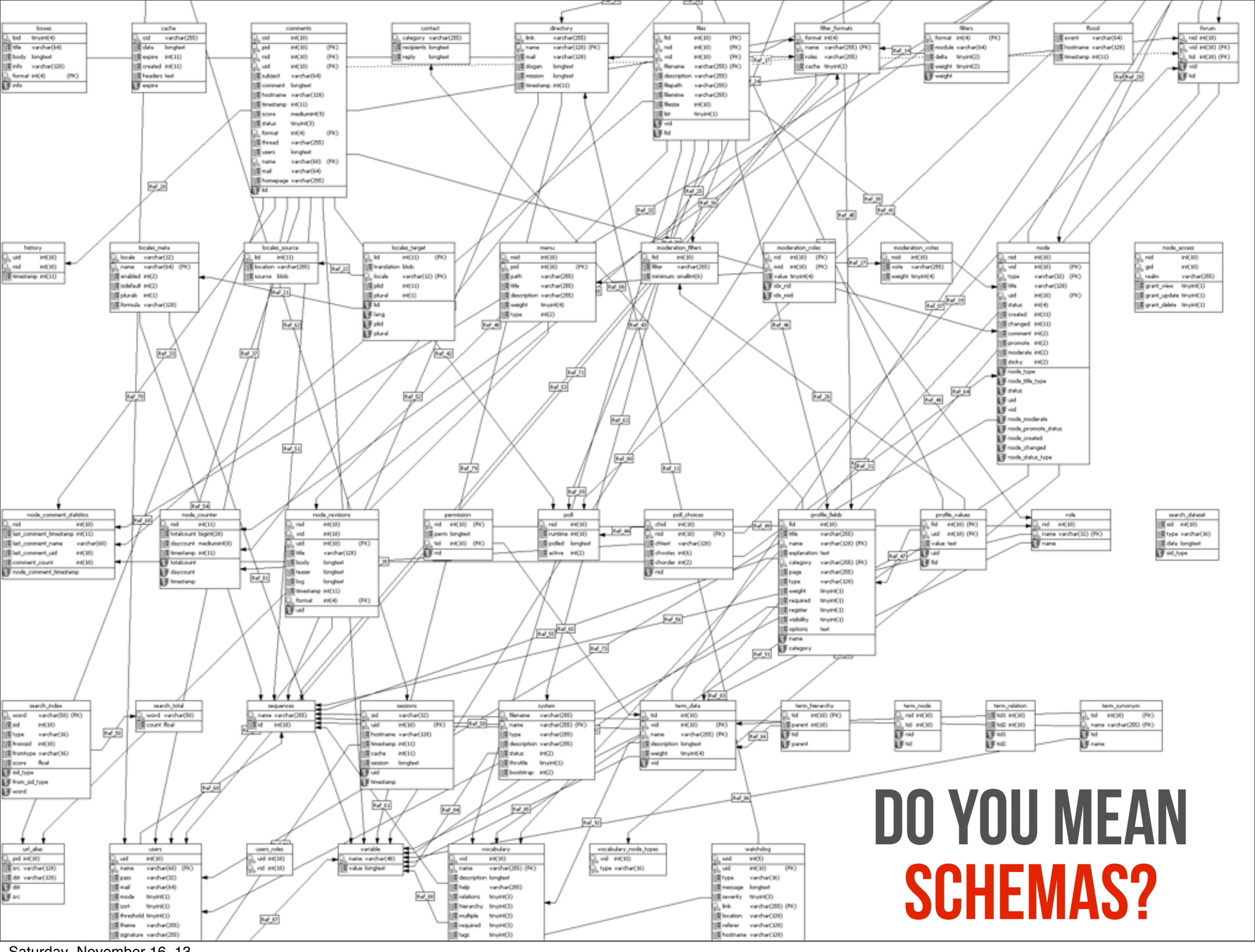
DATA MODELS ARE AT THE CORE OF DATA PLATFORMS


http://en.wikipedia.org/wiki/Draugen_oil_field


Data Model
http://en.wikipedia.org/wiki/Draugen_oil_field


Data Model
Data Platform
http://en.wikipedia.org/wiki/Draugen_oil_field


Application Code
Data Model
Data Platform
http://en.wikipedia.org/wiki/Draugen_oil_field



DO YOU MEAN SCHEMAS?

SCHEMAS

- Good for performance
- Great for documentation. New hires appreciate it
- Good for enforcing data contracts
- Good as Reflection APIs on highly dynamic systems
- Good for code generation
- ***Really bad*** **when structure needs to change**


Need to upgrade?
http://en.wikipedia.org/wiki/Draugen_oil_field

THE KEY IS ADAPTABILITY

- Application code should be able to retrieve the data in whatever shape it's needed.

- A system should not require live data migration to reshape the structure of the data when the structure needs to evolve.

- Avoid systems that can only serve efficient queries retrieving data in the shape they were originally stored in.

DATOM

THE DATOMIC DATA MODEL

BASIC API

```
;; require API and create alias
(require '[datomic.api :as d])

;; get a connection to the database
(def uri "datomic:mem://db-name")
(def conn (d/connect uri))

;; write (datoms) into the database
(d/transact conn
  [DATOMS])
```

BASIC API

```
;; require API and create alias
(require '[datomic.api :as d])

;; get a connection to the database
(def uri "datomic:mem://db-name")
(def conn (d/connect uri))

;; write (datoms) into the database
(d/transact conn
  [DATOMS])
```

BASIC API

```
;; require API and create alias
(require '[datomic.api :as d])

;; get a connection to the database
(def uri "datomic:mem://db-name")
(def conn (d/connect uri))

;; write (datoms) into the database
(d/transact conn
  [DATOMS])
```

DATOM == FACT

[+/- entity attribute value tx]

```
[[:db/add #db/id[:db.part/user -1]
  :person/name “John”]
 [:db/add #db/id[:db.part/user -1]
  :person/age 25]]
```

DATOM == FACT

[+/- entity attribute value tx]

```
[[:db/add #db/id[:db.part/user -1]
  :person/name “John”]
 [:db/add #db/id[:db.part/user -1]
  :person/age 25]]
```

DATOM == FACT

[+/- entity attribute value tx]

```
[[:db/add #db/id[:db.part/user -1]
  :person/name “John”]
 [:db/add #db/id[:db.part/user -1]
  :person/age 25]]
```

DATOM == FACT

[+/- entity attribute value tx]

```
[[:db/add #db/id[:db.part/user -1]
  :person/name “John”]
 [:db/add #db/id[:db.part/user -1]
  :person/age 25]]
```

DATOM == FACT

[+/- entity attribute value tx]

```
[[:db/add #db/id[:db.part/user -1]
  :person/name “John” ]
 [:db/add #db/id[:db.part/user -1]
  :person/age 25]
 [:db/add #db/id[:db.part/tx]
  :db/txInstant (System/currentTimeMillis)]]
```

SHORT-HAND NOTATION

```
[{:db/id #db/id[:db.part/user]
  :person/name “John”
  :person/age 25}]
```

ATTRIBUTES ARE FACTS!

```
[{:db/id #db/id[:db.part/db]
  :db/ident :person/name
  :db/valueType :db.type/string
  :db/cardinality :db.cardinality/one
  :db.install/_attribute :db.part/db}]
```

Attributes are data and can be installed on the fly

FREE ASSOCIATIONS

```
[{:db/id #db/id[:db.part/user]
  :person/name “John”
  :car/top-speed 200
  :computer/keyboard “DAS”
  :city/population 250000}]
```

What’s this entity schema / data model?

HOW TO **DEFINE** A HIGHER-LEVEL **SCHEMA** AND USE IT TO DEFINE **ITSELF**

CORE DATA MODEL: DATATYPE

ATTRIBUTE NAME	ATTRIBUTE TYPE	CARDINALITY
:dt/dt	ref	one
:dt/namespace	string	one
:dt/name	string	one
:db/parent	ref	one
:dt/list	ref	one
:dt/component	ref	one
:db/fields	ref	many

:dt/dt

```
[{:db/id #db/id[:db.part/db]
  :db/ident :dt/dt
  :db/valueType :db.type/ref
  :db/cardinality :db.cardinality/one
  :db/doc
    "A reference to the data type of this
entity. Entities with this attribute
are known as Typed-Entities."
  :db.install/_attribute :db.part/db}]
```

≈ .getClass

:dt/namespace

```
[{:db/id #db/id[:db.part/db]
  :db/ident :dt/namespace
  :db/valueType :db.type/string
  :db/cardinality :db.cardinality/one
  :db/doc
    "Data types have qualified names.
This is the data type namespace."
  :db.install/_attribute :db.part/db}]
```

```
≈ #(-> % .getPackage .getName)
```

:dt/name

```
[{:db/id #db/id[:db.part/db]
  :db/ident :dt/name
  :db/valueType :db.type/string
  :db/cardinality :db.cardinality/one
  :db/doc
    "The local part or unqualified name
of a data type."
  :db.install/_attribute :db.part/db}]
```

≈ .getSimpleName

:dt/parent

```
[{:db/id #db/id[:db.part/db]
  :db/ident :dt/parent
  :db/valueType :db.type/ref
  :db/cardinality :db.cardinality/many
  :db/doc "A reference to the data type
that this data type inherits fields from."
  :db.install/_attribute :db.part/db}]
```

≈ .getSuperClass

:dt/list

```
[{:db/id #db/id[:db.part/db]
  :db/ident :dt/list
  :db/valueType :db.type/ref
  :db/cardinality :db.cardinality/one
  :db/doc “Dynamically-generated
reference to a list data type. Provides
support for multi-dimensional values.”
  :db.install/_attribute :db.part/db}]
```

```
≈  #(-> (Array/newInstance % 0) .getClass)
```

:dt/component

```
[{:db/id #db/id[:db.part/db]
  :db/ident :dt/component
  :db/valueType :db.type/ref
  :db/cardinality :db.cardinality/one
  :db/doc Only populated if this is a
list data type.”
  :db.install/_attribute :db.part/db}]
```

≈ .getComponentType

:dt/fields

```
[{:db/id #db/id[:db.part/db]
  :db/ident :dt/fields
  :db/valueType :db.type/ref
  :db/cardinality :db.cardinality/many
  :db/doc
    "References to the data type fields,
themselves data types."
  :db.install/_attribute :db.part/db}]
```

≈ .getFields

DATATYPE ≈ CLASS

ATTRIBUTE	IS ANALOGOUS TO
:dt/dt	`.getClass`
:dt/namespace	`#(-> % .getPackage .getName)`
:dt/name	`.getSimpleName`
:db/parent	`.getSuperClass`
:dt/list	`#(-> (Array/newInstance % 0) .getClass)`
:dt/component	`.getComponentType`
:db/fields	`.getFields`

PERSON INSTANCE

```
[{:db/id #db/id[:db.part/user]

   :person/name “John”
   :person/age 25}]
```

PERSON INSTANCE

```
[{:db/id #db/id[:db.part/user]
  :dt/dt :person.datatype/entity
  :person/name “John”
  :person/age 25}]
```

A **Typed** Entity

```
dt/dt ≈ .getClass
```

Assume `:person.datatype/entity` *had been defined*

PERSON DATATYPE

```
[{:db/id :person.datatype/entity
  :dt/dt :dt/dt
  :dt/namespace "customer"
  :dt/name "Person"
  :dt/fields [:dt/dt
              :person/name
              :person/age]}]
```

Also a **Typed** Entity

```
dt/dt ≈ .getClass
```

DATATYPE DATATYPE

```
;; The dt/dt entity is both the definition
;; of a data type and an instance of itself
[{:db/id :dt/dt
  :dt/dt :dt/dt
  :dt/namespace "system"
  :dt/name "Datatype"
  :dt/fields [:dt/dt
    :dt/namespace :dt/name :dt/parent
    :dt/list :dt/component :dt/fields]}]
```

Also a **Typed** Entity

```
dt/dt ≈ .getClass
```

DATATYPE DATATYPE

```
;; The dt/dt entity is both the definition
;; of a data type and an instance of itself
[{:db/id :dt/dt
  :dt/dt :dt/dt
  :dt/namespace "system"
  :dt/name "Datatype"
  :dt/fields [:dt/dt
    :dt/namespace :dt/name :dt/parent
    :dt/list :dt/component :dt/fields]}]
```

DATATYPE DATATYPE

```
;; The dt/dt entity is both the definition
;; of a data type and an instance of itself
[{:db/id :dt/dt
  :dt/dt :dt/dt
  :dt/namespace "system"
  :dt/name "Datatype"
  :dt/fields [:dt/dt
    :dt/namespace :dt/name :dt/parent
    :dt/list :dt/component :dt/fields]}]
```

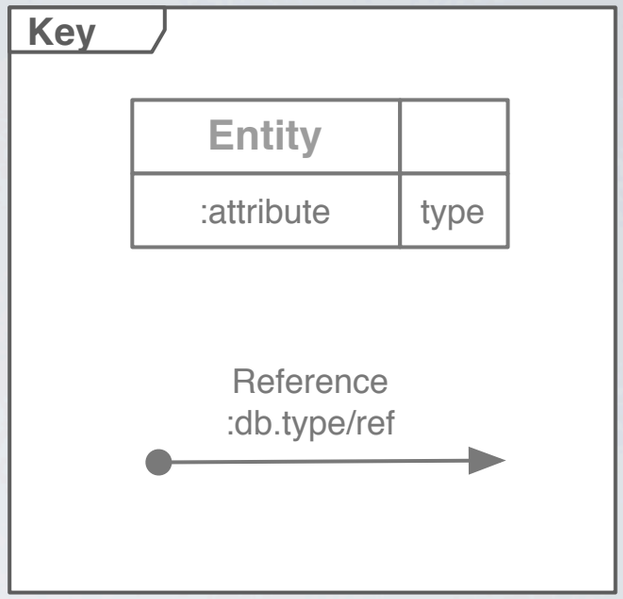
DATATYPE DATATYPE

```
;; The dt/dt entity is both the definition
;; of a data type and an instance of itself
[{:db/id :dt/dt
  :dt/dt :dt/dt
  :dt/namespace "system"
  :dt/name "Datatype"
  :dt/fields [:dt/dt
    :dt/namespace :dt/name :dt/parent
    :dt/list :dt/component :dt/fields]}]
```

DATATYPE DATATYPE

```
;; The dt/dt entity is both the definition
;; of a data type and an instance of itself
[{:db/id :dt/dt
  :dt/dt :dt/dt
  :dt/namespace "system"
  :dt/name "Datatype"
  :dt/fields [:dt/dt
    :dt/namespace :dt/name :dt/parent
    :dt/list :dt/component :dt/fields]}]
```

META-MODEL



3

Datatype Datatype	
:dt/dt	ref
:dt/namespace	string
:dt/name	string
:dt/parent	ref
:dt/list	ref
:dt/component	ref
:dt/fields	ref[]

2

Person Datatype	
:dt/dt	ref
:dt/namespace	string
:dt/name	string
:dt/fields	ref[]

1

Person Instance	
:dt/dt	ref
:person/name	string
:person/age	long
:person/weigth	long

META-MODEL

3

Datatype Datatype	
:dt/dt	ref
:dt/namespace	string
:dt/name	string
:dt/parent	ref
:dt/list	ref
:dt/component	ref
:dt/fields	ref[]

3

Datatype Datatype	
:dt/dt	ref
:dt/namespace	string
:dt/name	string
:dt/parent	ref
:dt/list	ref
:dt/component	ref
:dt/fields	ref[]

2

Person Datatype	
:dt/dt	ref
:dt/namespace	string
:dt/name	string
:dt/fields	ref[]

DATATYPE HIERARCHIES

```
[{:db/id #db/id[:db.part/user]
  :dt/dt :dt/dt
  :dt/namespace "customer"
  :dt/name "Student"
  :dt/parent :person.datatype/entity
  :dt/fields [:student/school]}]
```

```
:dt/parent  ≈  .getSuperClass
```

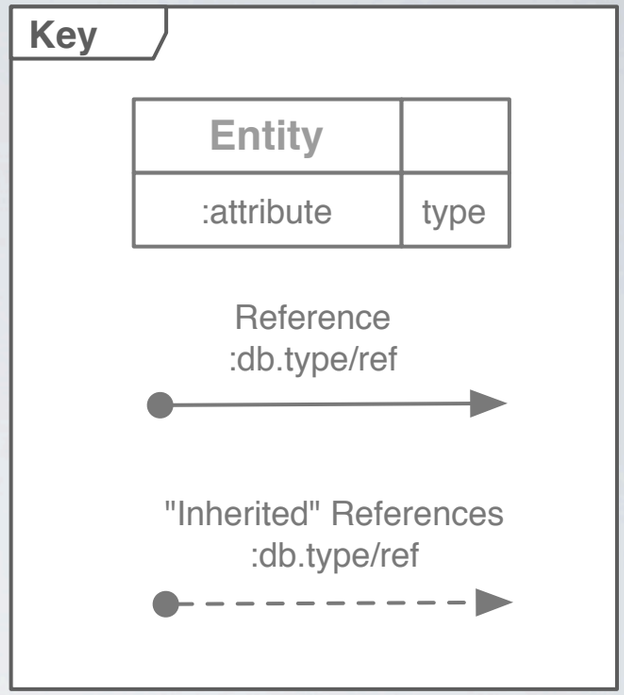
DATATYPE HIERARCHIES

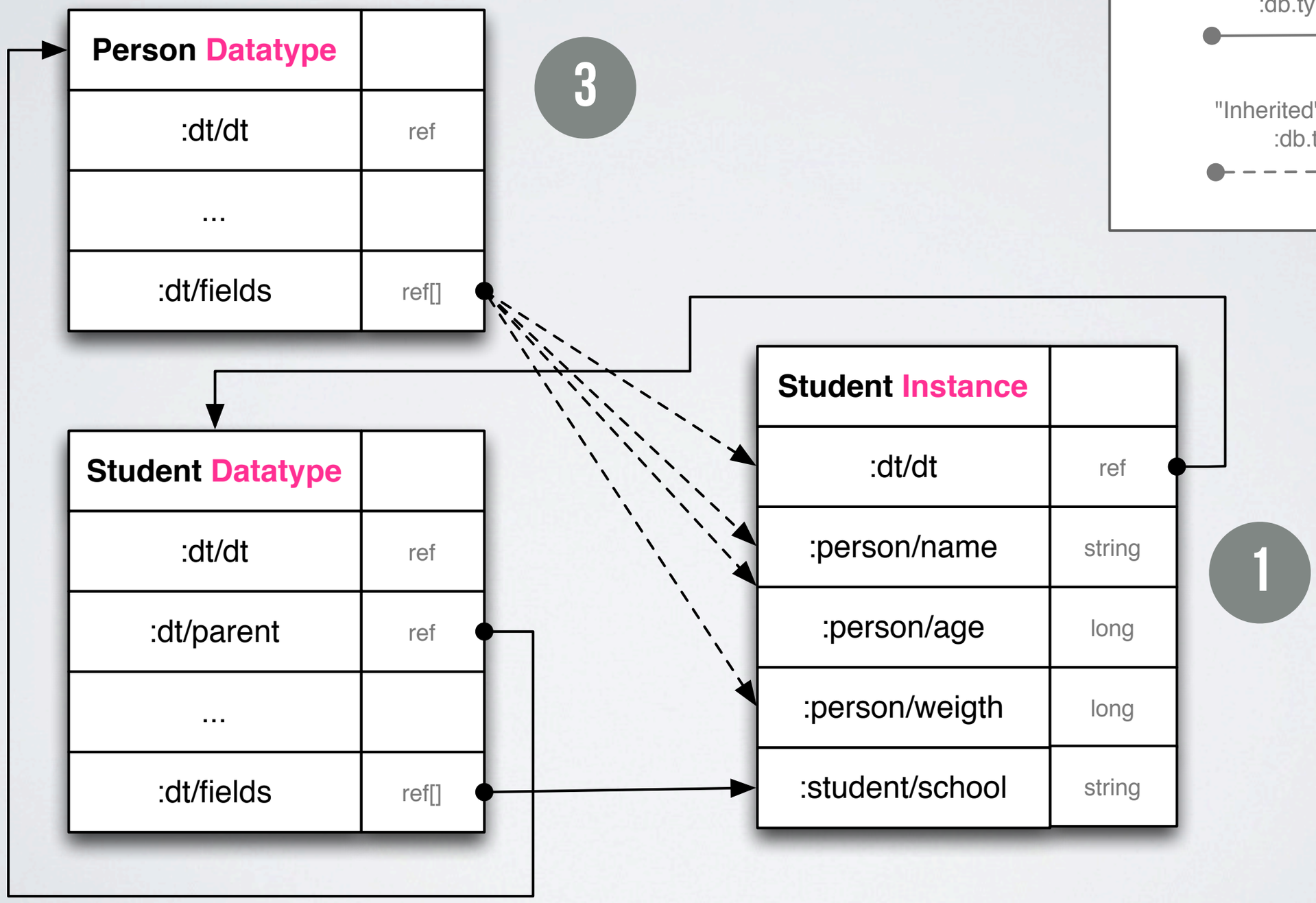
```
[{:db/id #db/id[:db.part/user]
  :dt/dt :dt/dt
  :dt/namespace "customer"
  :dt/name "Student"
  :dt/parent :person.datatype/entity
  :dt/fields [:student/school]}]
```

```
:dt/parent  ≈  .getSuperClass
```

Assume ***`:student/school`*** *had been defined*

DATATYPE HIERARCHIES





MULTI-DIMENSIONAL VALUES

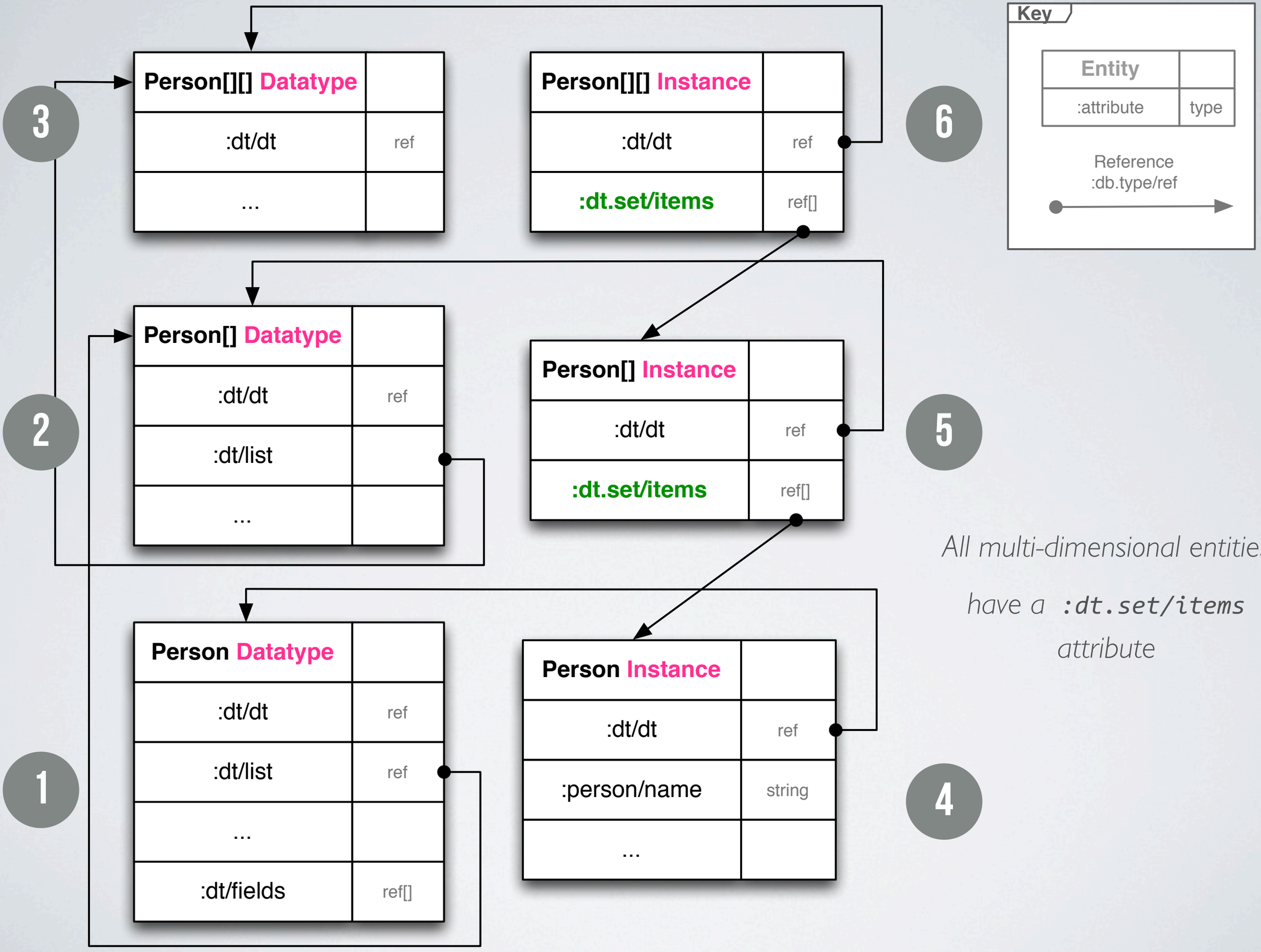
```
 ;; e.g. Person[]
[{:db/id #db/id[:db.part/user -1]
  :dt/dt :dt/dt
  :dt/component :person.datatype/entity
  :dt/_list :person.datatype/entity
  :dt/fields [:dt.set/items]}
 ;; e.g. Person[][]
 {:db/id #db/id [:db.part/user]
  :dt/dt :dt/dt
  :dt/component #db/id[:db.part/user -1]
  :dt/_list #db/id[:db.part/user -1]
  :dt/fields [:dt.set/items]}]
```

All references in Datomic are bidirectional so these refs are, in fact, redundant.

MULTI-DIMENSIONAL VALUES

```
 ;; e.g. Person[]
[{:db/id #db/id[:db.part/user -1]
  :dt/dt :dt/dt
  :dt/component :person.datatype/entity
  :dt/_list :person.datatype/entity
  :dt/fields [:dt.set/items]}
 ;; e.g. Person[][]
{:db/id #db/id [:db.part/user]
  :dt/dt :dt/dt
  :dt/component #db/id[:db.part/user -1]
  :dt/_list #db/id[:db.part/user -1]
  :dt/fields [:dt.set/items]}]
```

You can use temporary ids to link entities before they are transacted.



All multi-dimensional entities

have a `:dt.set/items`

attribute

ANY TYPE / VARIANT TYPE

```
[{:db/id #db/id[:db.part/db]
    :db/ident :dt/any
    :db/valueType :db.type/ref
    :db/cardinality :db.cardinality/one
    :db/doc “Built-in types enumeration”
    :db.install/_attribute :db.part/db}
 {:db/id #db/id[:db.part/db]
  :db/ident :dt.any/string
  :db/valueType :db.type/string
  :db/cardinality :db.cardinality/one
  :db.install/_attribute :db.part/db}
 ...
]
```

ANY TYPE / VARIANT TYPE

```
[{:db/id :dt/any
  :dt/dt :dt/dt
  :dt/namespace "system"
  :dt/name "Any"
  :dt/fields [:dt/dt
              :dt/any
              :dt.any/string
              :dt.any/boolean
              :dt.any/long
              ...
              :dt.any/ref]}]
```

Only one of the **`dt.any/*`**
attributes is present
at runtime for any given
"Any" entity

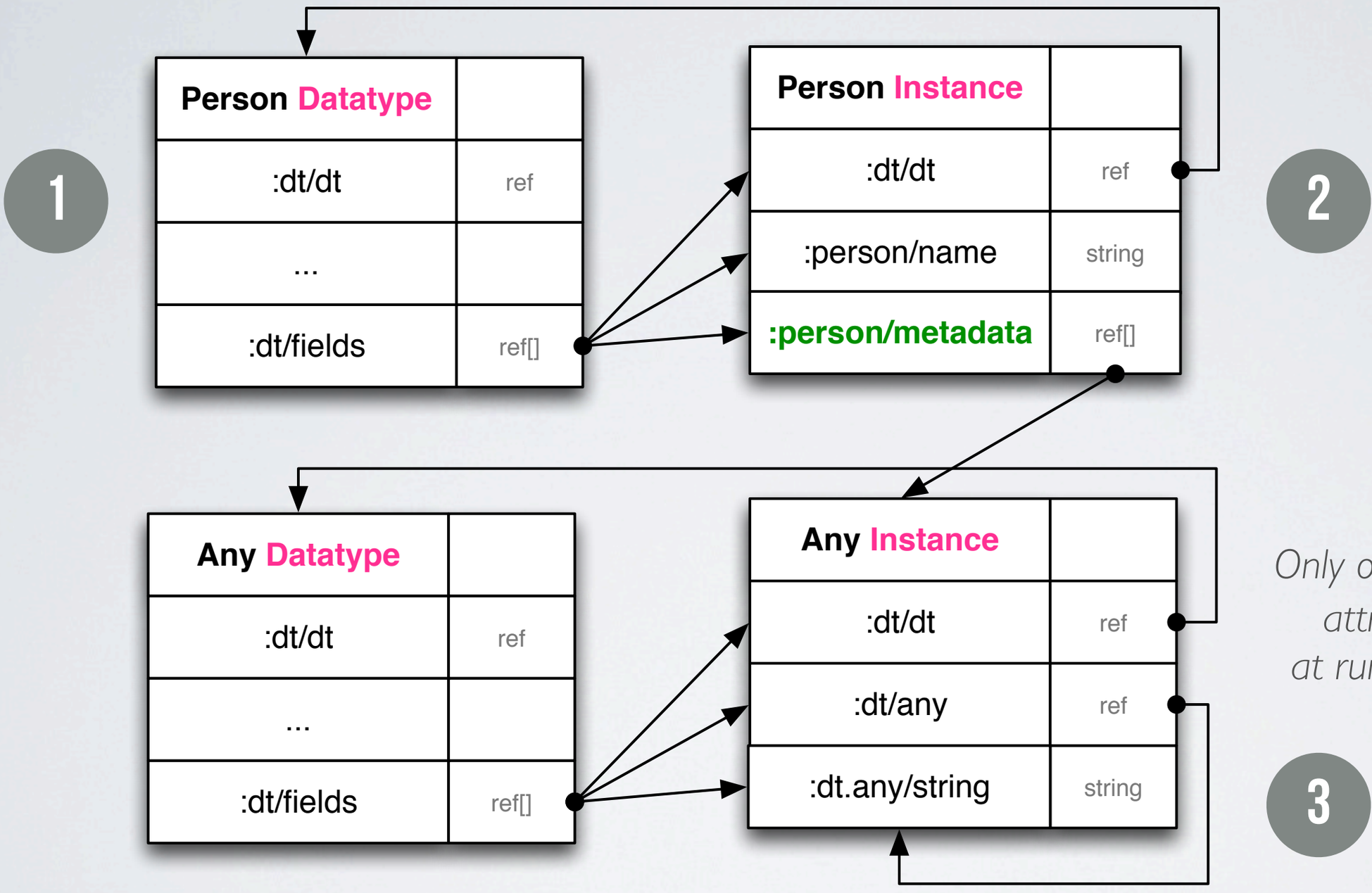
TYPED REFERENCES

```
[{:db/id #db/id[:db.part/db]
  :db/ident :person/metadata
  :db/valueType :db.type/ref
  :dt.meta/valueType :dt/any
  :db/cardinality :db.cardinality/many
  :db.install/_attribute :db.part/db}]

[{:db/id :person.datatype/entity
  :dt/fields [:person/metadata]}]
```

Assume `:dt.meta/valueType` *had been defined*

ANY TYPE / VARIANT TYPE



Only one of the ***`dt.any/*`*** *attributes is present at runtime for any given "Any" entity*

3

HOW TO DEAL WITH CHANGES IN SCHEMAS

PERSON DATATYPE v2

```
[{:db/id :person.datatype/entity
  :dt/fields [:person/weight]}]
```

Transactions can add individual values for multi-valued attributes.

EXISTING PERSON INSTANCE

```
[{:db/id #db/id[:db.part/user]
  :dt/dt :person.datatype/entity
  :person/name “John”
  :person/age 25}]
```

all existing entities
are still valid!

PERSON DATATYPE v3

```
[{:db/id #db/id[:db.part/db]
  :db/ident :health/insurance
  :db/valueType :db.type/string
  :db/cardinality :db.cardinality/one
  :dt.meta/required true
  :db.install/_attribute :db.part/db}]
```

Assume `:dt.meta/required` *had been defined*

PERSON DATATYPE v3

```
[{:db/id :person.datatype/entity
  :dt/fields [:health/insurance]}]
```

health insurance is NOT optional

EXISTING PERSON INSTANCE

```
[{:db/id #db/id[:db.part/user]
  :dt/dt :person.datatype/entity
  :person/name “John”
  :person/age 25}]
```

existing entities are
NO LONGER “valid”!

HOW TO FIX IT?

1 DATABASE FUNCTIONS

```
[{:db/id :health/insurance
  :dt.meta/migration
    (d/function
     '{:lang :clojure
      :params [db entity]
      :code [[
        :db/add entity
        :health/insurance "PPACA"]]})}]
```

Must eventually return a list of Datoms

Database functions!

1

DATABASE FUNCTIONS

```
[{:db/id #db/id[:db.part/db]
  :db/ident :dt.meta/migration
  :db/valueType :db.type/fn
  :db/cardinality :db.cardinality/one
  :db.install/_attribute :db.part/db}]
```

Datomic first-class functions

2

DECIDE WHEN TO VALIDATE

2

DECIDE WHEN TO VALIDATE

- Enforce that all required attributes include a migration function at the time the attributes is created.

2 DECIDE WHEN TO VALIDATE

- Enforce that all required attributes include a migration function at the time the attributes is created.

- Enforce constraints when **new** entities are created.

2 DECIDE WHEN TO VALIDATE

- Enforce that all required attributes include a migration function at the time the attributes is created.

- Enforce constraints when **new** entities are created.

- How? wrap `(d/transact)` and perform these checks.

3

DECIDE WHEN TO MIGRATE

3

DECIDE WHEN TO MIGRATE

- At query time, **but** cannot be used as a rule.

3

DECIDE WHEN TO MIGRATE

- At query time, **but** cannot be used as a rule.
- At transaction time, **but** queries could return "invalid" data.

3

DECIDE WHEN TO MIGRATE

- At query time, **but** cannot be used as a rule.
- At transaction time, **but** queries could return "invalid" data.
- At attribute access time, **but** restricted scope.

3

DECIDE WHEN TO MIGRATE

- At query time, **but** cannot be used as a rule.
- At transaction time, **but** queries could return "invalid" data.
- At attribute access time, **but** restricted scope.
- As a background processing task, **but** data would be eventually consistent.

3

DECIDE WHEN TO MIGRATE

- At query time, **but** cannot be used as a rule.
- At transaction time, **but** queries could return "invalid" data.
- At attribute access time, **but** restricted scope.
- As a background processing task, **but** data would be eventually consistent.
- Manually, driven by application code.

HOW TO ENFORCE ARBITRARY DATA CONSTRAINTS

ENUMS

```
[{:db/id #db/id[:db.part/db]
  :db/ident :person/gender
  :db/valueType :db.type/ref
  :db/cardinality :db.cardinality/one
  :db.install/_attribute :db.part/db}

 {:db/id #db/id[:db.part/user]
  :db/ident :person.gender/female}

 {:db/id #db/id[:db.part/user]
  :db/ident :person.gender/male}]
```

ENUMS

```
[[:db/add #db/id[:db.part/user]
  :person/gender :car/top-speed]]
```

will succeed

HOW TO FIX IT?

```
[{:db/id #db/id [:db.part/user]
    :db/ident :add-fk
    :db/fn (d/function
             '{:lang :clojure
               :params [db e a v]
               :code (let [;; get value's symbolic keyword
                           ident (if (keyword? v) v (d/ident db v))
                           ;; get enum's constraint
                           aent (d/entity db a)
                           aname (:db/ident aent)
                           enum-ns (:enum/ns aent)
                           ;; find all possible enum idents
                           allowed (if (nil? enum-ns)
                                     (throw (Exception. (str "Cannot check fk
constraint. " aname " has no :enum/ns attribute")))
                                     (d/q '[:find ?ident
                                            :in $ ?enum-ns
                                            :where
                                            [_ :db/ident ?ident]
                                            [(namespace ?ident) ?ns]
                                            [(= ?enum-ns ?ns)]]
                                          db enum-ns))]
                       ;; enforce constraint
                       (if (contains? allowed [ident])
                         [[:db/add e a v]]
                         (throw (Exception. (str v " is not one of " allowed))))))}]
```

1 give it a name

2 add hint to enum values

look up valid enum values 3

4 fail transaction

ENUMS

```
[[:db/add-fk #db/id[:db.part/user]
  :person/gender :car/top-speed]]
```

will FAIL

REIFIED TRANSACTIONS

AUDIT TRAIL?

```
[[db/retract 12345 :health/insurance]]
```

TRANSACTIONS ARE FACTS!

```
[[db/retract 12345 :health/insurance

  [:db/add #db/id[:db.part/tx]
    :audit/source username-id]]
```

Assume `:audit/source` *had been defined*

THANK YOU

Antonio Andrade

Clojure/conj 2013