Datomic Cloud Documentation

Search

What is Datomic?

- Data Model
- Architecture
- Supported Operations
- Programming with Data and EDN

Local Dev and CI

Cloud Setup

- Start a System
- Configure Access
- Get Connected

Tutorial

- Client API
- Assertion
- Read
- Accumulate
- Read Revisited
- Retract
- History

Client API

Videos

- AWS Setup
- Edn
- Datalog
- Datoms
- HTTP Direct
- CLI tools

Schema

- Defining Attributes
- Schema Reference
  Schema Grammar:db/cardinality:db/doc:db/id:db/ident:db/isComponent:db/noHistory:db/unique:db/valueTypeAttribute PredicatesEntity Specs
- Changing Schema
- Data Modeling
- Schema Limits

Transactions

- Processing Transactions
- Transaction Data Reference
- Transaction Functions
- ACID
- Client Synchronization

Query and Pull

- Executing Queries
- Query Data Reference
- Pull
- Index-pull
- Raw Index Access

Time in Datomic

- Log API
- Time Filters

Ions

- Ions Tutorial
- Ions Reference
- Monitoring Ions

Analytics Support

- Configuration
- Connecting
- Metaschema
- SQL CLI
- Troubleshooting

Analytics Tools

- Metabase
- R
- Python
- Jupyter
- Superset
- JDBC
- Other Tools

Operation

- Planning Your System
- Start a System
- AWS Account Setup
- Access Control
- CLI Tools
- Client Applications
- High Availability (HA)
- Howto
- Query Groups
- Monitoring
- Upgrading
- Scaling
- Deleting
- Splitting Stacks

Hide All Examples

Schema Data Reference

This topic defines the grammar for schema data.

Schema Grammar

Syntax Used in Grammar

```
'' literal
"" string
[] = list or vector
{} = map {k1 v1 ...}
() grouping
| choice
? zero or one
+ one or more
```

Schema Map Grammar

```
attr-def       = {':db/ident' keyword
                  ':db/cardinality' cardinality
                  ':db/valueType' type
                  (':db/doc' string)?
                  (':db/unique' unique)?
                  (':db/isComponent' boolean)?
                  (':db/id' n-identifier)?
                  (':db/noHistory' boolean)?
                  (':db.attr/preds' attr-pred+)?
                  (':db.entity/attrs' ident+)?
                  (':db.entity/preds' ent-pred+)?}
n-identifier   =  (identifier | tempid)
identifier     = (eid | lookup-ref | ident)
eid            = nat-int
lookup-ref     = [identifier value]
ident          = keyword
attr-pred      = fully-qualified symbol that name a /predicate/ of a value
ent-pred       = fully-qualified symbol that name a /predicate/ of
                 a database value and an entity id
cardinality    = (':db.cardinality/one' | ':db.cardinality/many)
type           = (':db.type/bigdec' | ':db.type/bigint' | ':db/type/boolean' |
                  ':db.type/double' | ':db.type/float' | ':db.type/instant' |
                  ':db.type/keyword' | ':db.type/long' | ':db.type/ref' |
                  ':db.type/string' | ':db.type/symbol' | ':db.type/tuple' |
```

```
                  ':db.type/uuid' | ':db.type/uri' )
unique          = (':db.unique/identity' | ':db.unique/value')
boolean         = ('true' | 'false')
```

Grammar Notes

The grammar above shows only the attributes that are built-in to Datomic schema. Schema entities can also have attributes that you define.

The grammar shows only the transaction map form. It is also possible to define schema with the more verbose transaction list form, as these forms are semantically equivalent.

The `:db` namespace, and all `:db.*` namespaces, are reserved for use by Datomic. With the exception of `:db/doc`, and `:db/ident`, you should not use the built-in Datomic attributes on your own domain entities.

Because schema is composed of ordinary Datomic data, the schema grammar is a specialization of the transaction grammar. The shared grammar elements `n-identifier`, `identifier`, `eid`, `lookup-ref`, and `ident` are documented there.

`:db/cardinality`

```
{':db/cardinality' cardinality}
cardinality = (':db.cardinality/one' | ':db.cardinality/many)
```

The required `:db/cardinality` attribute specifies whether an attribute associates a single value or a set of values with an entity. It has no default value.

The values allowed for `:db/cardinality` are:

- `:db.cardinality/one` - the attribute is single-valued, it associates a single value with an entity.
- `:db.cardinality/many` - the attribute is multi-valued, it associates a set of values with an entity.

`:db/doc`

```
':db/doc' = string
```

The optional `:db/doc` specifies a documentation string, and can be any string value.

`:db/id`

```
{:db/id' n-identifier}
n-identifier   = (identifier | tempid)
identifier     = (eid | lookup-ref | ident)
eid            = nat-int
lookup-ref     = [identifier value]
ident          = keyword
```

`:db/id` is not an attribute; rather, it is syntactic sugar for specifying the entity identifier in a transaction data map. For example, the following two forms are equivalent:

```
{:db/id "alan"
 :person/name "Alan Turing"}

["alan" :person/name "Alan Turing"]
```

⇧

Entity identifiers is rarely used in schema data, as schema elements are uniquely identified by the mandatory `:db/ident` attribute.

Entity identifiers are much more common in ordinary transaction data, and they are documented in the transaction data reference.

`:db/ident`

```
{':db/ident' keyword}
```

The `:db/ident` attribute specifies a unique programmatic name for an entity. Idents are required for schema entities and are optional for all other entities.

When an entity has an ident, you can use that ident in place of the numeric identifier, e.g.

```
;; assuming that :person/loves is entity id 1007, these are equivalent
["alan" :person/loves "pizza"]
["alan" 1007 "pizza"]
```

Idents should be used for two purposes: to name schema entities and to represent enumerated values. To support these usages, idents are designed to be extremely fast and always available. All idents associated with a database are stored in memory in every Datomic compute node.

These characteristics also imply situations where idents should *not* be used:

- Idents should not be used as unique names or ids on ordinary domain entities. Such entity names should be implemented with a domain-specific attribute that is a unique identity.
- Idents should not be used as names for test data. (Your real data will not have such names, and you don't want test data to behave differently than the real data it simulates.)

Idents can be used instead of entity ids in the following API calls:

- as the sole argument to entity
- in the e, a, and v positions of assertions and retractions passed to transact and with.
- in the e, a, and v positions of a query.

Allowable Values

The allowable value of `:db/ident` is a namespaced keyword with the form `:<namespace>/<name>`. It is possible to define a name without a namespace, as in `:<name>`, but a namespace is preferred in order to avoid naming collisions. Namespaces can be hierarchical, with segments separated by ".", as in `:<namespace>.<nested-namespace>/<name>`.

The `:db` namespace, and all `:db.*` namespaces, are reserved for use by Datomic. With the exception of `:db/doc` and `:db/ident`, you should not use the built-in Datomic attributes on your own domain entities.

Use only letters, numbers, hyphens, and underscores in `:db/ident` values.

`:db/isComponent`

```
{':db/isComponent' boolean}
```

The optional `:db/isComponent` attribute specifies that an attribute whose :db/valueType is `:db.type/ref` refers to a sub-component of the entity to which the attribute is applied. When you retract an entity with `:db.fn/retractEntity`, all sub-components are also retracted.

Omitting `:db/isComponent` for an entity is semantically equivalent to setting it to `false`.

`:db/noHistory`

```
{':db/noHistory' boolean}
```

Description and Use Cases

Specifies a boolean value indicating whether past values of an attribute should not be retained. Defaults to false. The purpose of `:db/noHistory` is to conserve storage, not to make semantic guarantees about removing information. The effect of `:db/noHistory` happens in the background, and some amount of history may be visible even for attributes with `:db/noHistory` set to true.

`db/noHistory` is often used for high churn attributes along with attributes that you do not require a history of.

`:db/unique`

```
{':db/unique' unique}
unique         = (':db.unique/identity' | ':db.unique/value')
```

The `:db/unique` attribute specifies a uniqueness constraint for the values of an attribute.

In order to add a uniqueness constraint to an attribute, both of the following must be true:

- The attribute must have a `:db/cardinality` of `:db.cardinality/one`.
- If there are values present for that attribute, they must be unique in the set of *current* database assertions.

Adding a unique constraint does not change history, therefore historical databases may contain non-unique values. Code that expects to find a unique value may find multiple values when querying against history.

All entities in a database have an internal key, the *entity id*. You can use `:db/unique` to define an attribute to represent an external key. An entity may have any number of external keys. External keys must be single attributes, multi-attribute keys are not supported.

`:db.unique/identity`

Unique identity is specified through an attribute with `db.unique` set to `db.unique/identity`. Values of the attribute must be unique to each entity and upsert is enabled; attempts to insert a duplicate value for a temporary entity id will cause all attributes associated with that temporary id to be merged with the entity already in the database.

The tutorial creates an `:inv/sku` that is unique:

```
{:db/ident :inv/sku
 :db/valueType :db.type/string
 :db/unique :db.unique/identity
 :db/cardinality :db.cardinality/one}
```


Unique identity is appropriate whenever you want to assert a database-wide unique identifier for an entity. Common examples include emails, account names, product codes/skus, and UUIDs.

If a transaction specifies a unique identity for a temporary id, and that unique identity already exists in the database, then that temporary id will resolve to the *existing* entity in the system. This upsert behavior makes it possible for transactions to work with domain identities, without ever having to specify Datomic entity ids.

It is legal for a single entity to have multiple different unique attributes, e.g. you might decide that people have both unique emails and unique government identifiers. However, note that this creates the possibility of conflict, which will result in a conflict anomaly. (Conflict will occur if a transaction tries to upsert a tempid into two *different* existing entities. As an example. if entity 42 has the unique email"johndoe@example.com", and entity 43 has the unique account number 1007, then a transaction cannot claim that a new tempid is both John Doe and account 1007.)

Uniqueness can be declared on attributes of any value type, including references (`:db.type/ref`). Only (`:db.cardinality/one`) attributes can be declared unique.

Datomic does provide a mechanism to declare composite uniqueness constraints in Composite Tuples; however, you can also implement them (or any arbitrary functional constraint) via transaction functions.

`:db.unique/value`

Unique value is specified through an attribute with `db/unique` set to `db.unique/value`. A unique value represents an attribute-wide value that can be asserted only once.

Unique values have the same semantics as unique identities, with one critical difference: Attempts to assert a new tempid with a unique value already in the database will cause a conflict anomaly.

:db/valueType

> **Note:** Symbols require 480-8770 or later and a compatible base schema.

```
{':db/valueType' type}
type           = (':db.type/bigdec' | ':db.type/bigint' | ':db/type/boolean' |
                  ':db.type/double' | ':db.type/float' | ':db.type/instant' |
                  ':db.type/keyword' | ':db.type/long' | ':db.type/ref' |
                  ':db.type/string' | ':db.type/symbol' | ':db.type/tuple' |
                  ':db.type/uuid' | ':db.type/uri')
```

The `:db/valueType` attribute specifies the type of value that can be associated with an attribute. The type is one of the keywords in the table below.

`:db/valueType` cannot be updated after an attribute is created.

Value Type	Description	Java equivalent	Example
:db.type/bigdec	arbitrary precision decimal	`java.math.BigDecimal`	1.0M
:db.type/bigint	arbitrary precision integer	`java.math.BigInteger`	7N
:db.type/boolean	boolean	`boolean`	true
:db.type/double	64-bit IEEE 754 floating point number	`double`	1.0
:db.type/float	32-bit IEEE 754 floating point number	`float`	1.0
:db.type/instant	instant in time	`java.util.Date`	#inst "2017-09-16T11:43:32.450-00:00"
:db.type/keyword	namespace + name	N/A	:yellow
:db.type/long	64 bit two's complement integer	`long`	42
:db.type/ref	reference to another entity	N/A	42
:db.type/string	Unicode string	`java.lang.String`	"foo"
:db.type/symbol	symbol	N/A	foo
:db.type/tuple	tuples of scalar values	N/A	[42 12 "foo"]
:db.type/uuid	128-bit universally unique identifier	`java.util.UUID`	#uuid "f40e770e-9ad5-11e7-abc4-cec278b6b50a"
:db.type/uri	Uniform Resource Identifier (URI)	`java.net.URI`	https://www.datomic.com/details.html

Notes on Value Types

- Keywords are interned for efficiency.
- Instances are stored as the number of milliseconds since the epoch.
- Strings are limited to 4096 characters.
- BigDecimals are limited to 1024 digit precision.
- BigIntegers are limited to a bit length of 8192.
- Symbols map to the symbol type in languages that support them, e.g. clojure.lang.Symbol in Clojure

Tuples

> **Note:** Tuples require 480-8770 or later and a compatible base schema.

A tuple is a collection of 2-8 scalar values, represented in memory as a Clojure vector. There are three kinds of tuples:

- Composite tuples are derived from other attributes of the same entity. Composite tuple types have a `:db/tupleAttrs` attribute, whose value is 2-8 keywords naming other attributes.
- Heterogeneous fixed length tuples have a `:db/tupleTypes` attribute, whose value is a vector of 2-8 scalar value types.
- Homogeneous variable length tuples have a `:db/tupleType` attribute, whose value is a keyword naming a scalar value type.

The following types are considered scalar types suitable for use in a tuple:

```
:db.type/bigdec :db.type/bigint :db.type/boolean :db.type/double
:db.type/instant :db.type/keyword :db.type/long :db.type/string
:db.type/symbol :db.type/ref :db.type/uri :db.type/uuid
```

⇧

String values within a tuple are limited to 256 characters.

`nil` is a legal value for any slot in a tuple. This facilitates using tuples in range searches, where `nil` sorts lowest.

Datomic includes the query helpers tuple and untuple to support storing same values both in a tuple and in separate attributes.

Composite Tuples

Composite tuples are applicable:

- when a domain entity has a multi-attribute key
- to optimize a query that joins more than one high-population attribute on the same entity

For example, consider the domain of course registrations, modeled with the following entity types:

- courses represent a course, e.g. Algebra II
- semesters represent a period in time when a course is run, e.g. "Fall 2019"
- students can take courses in particular semesters

```
[{:db/ident :student/first
  :db/valueType :db.type/string
  :db/cardinality :db.cardinality/one}
 {:db/ident :student/last
  :db/valueType :db.type/string
  :db/cardinality :db.cardinality/one}
 {:db/ident :student/email
  :db/valueType :db.type/string
  :db/cardinality :db.cardinality/one
  :db/unique :db.unique/identity}
 {:db/ident :semester/year
  :db/valueType :db.type/long
  :db/cardinality :db.cardinality/one}
 {:db/ident :semester/season
  :db/valueType :db.type/keyword
  :db/cardinality :db.cardinality/one}
 {:db/ident :semester/year+season
  :db/valueType :db.type/tuple
  :db/tupleAttrs [:semester/year :semester/season]
  :db/cardinality :db.cardinality/one
  :db/unique :db.unique/identity}
 {:db/ident :course/id
  :db/valueType :db.type/string
  :db/unique :db.unique/identity
  :db/cardinality :db.cardinality/one}
 {:db/ident :course/name
  :db/valueType :db.type/string
  :db/cardinality :db.cardinality/one}]
```

⇧

A *registration* entity is a unique combination of a student, semester, and course. In Datomic schema:

```
{:db/ident :reg/course
 :db/valueType :db.type/ref
```

```
 :db/cardinality :db.cardinality/one}
{:db/ident :reg/semester
 :db/valueType :db.type/ref
 :db/cardinality :db.cardinality/one}
{:db/ident :reg/student
 :db/valueType :db.type/ref
 :db/cardinality :db.cardinality/one}
```

⇧

A given course/semester/student combination is unique in the database. To model this, you can create a composite tuple whose `:db/tupleAttrs` are

```
{:db/ident :reg/course+semester+student
 :db/valueType :db.type/tuple
 :db/tupleAttrs [:reg/course :reg/semester :reg/student]
 :db/cardinality :db.cardinality/one
 :db/unique :db.unique/identity}
```

⇧

With this composite installed, Datomic's unique identity will ensure that all assertions about a semester/course/student combination resolve to the same entity.

Composite attributes are entirely managed by Datomic–you never assert or retract them yourself. Whenever you assert or retract any attribute that is part of a composite, Datomic will automatically populate the composite value.

Given a database with the courses and semesters schema, add some seed data:

```
[{:semester/year 2018
  :semester/season :fall}
 {:course/id "BIO-101"}
 {:student/first "John"
  :student/last "Doe"
  :student/email "johndoe@university.edu"}]
```

⇧

Now if you register John for Bio 101 in the fall of 2018 by transacting:

```
[{:reg/course [:course/id "BIO-101"]
  :reg/semester [:semester/year+season [2018 :fall]]
  :reg/student [:student/email "johndoe@university.edu"]}]
```

⇧

Datomic will also add the composite tuple datom:

```
#datom[20736789299855447 83 [6324390882967637 44112406506373204 64110323992363094] 13194139533320 true]]
```

Note that your entity IDs will differ from those in the example above

If the current value of an entity does not include all attributes of a composite, the missing attributes will be nil. For example, given a composite 4-tuple :reg/course+semester+student+grade that also includes a student's grade, the assertions above would cause Datomic to populate:

```
#datom[20736789299855447 83 [6324390882967637 44112406506373204 64110323992363094 nil] 13194139533320 true]]
```

Note that nil sorts lower than all other values, so tuples with trailing nils can be useful for range queries.

If you retract all constituents of a composite, Datomic will retract the composite. For example, transacting:

```
[[:db/retract 20736789299855447 :reg/course [:course/id "BIO-101"]]
 [:db/retract 20736789299855447 :reg/semester [:semester/year+season [2018 :fall]]]
 [:db/retract 20736789299855447 :reg/student [:student/email "johndoe@university.edu"]]]
```

⇧

will cause Datomic to retract the composite:

```
#datom[20736789299855447 83 [6324390882967637 44112406506373204 64110323992363094] 13194139533321 false]]
```

Again, note that you will need to substitute the entity IDs from your initial transaction to replicate this example in your system.

Adding Composites to Existing Entities

Adding a composite tuple to a database that contains existing data using those attributes will **not** immediately generate values for the new tuple. The composite tuple will be created the next time any of the composite member attributes are transacted. This includes "no-op" transactions of the same attribute value. This design allows you to add composite tuples in a systematic and paced manner, so as not to overwhelm a running system.

An example helper function can be found in the day of datomic cloud examples. The function will cycle through all values for a given attribute, reasserting them, with the specified batch size and pause between batches. You can use this, or something like it, to systematically add composite tuples once you've created the necessary schema attribute(s).

Heterogeneous Tuples

Heterogeneous tuples have a `:db/tupleTypes` attribute, with a value specified as a vector of 2-8 scalar types.

For example, you could model a location in a 2D game with the following tuple attribute:

```
{:db/ident :player/location
 :db/valueType :db.type/tuple
 :db/tupleTypes [:db.type/long :db.type/long]
 :db/cardinality :db.cardinality/one}
```

⇧

You can then explicitly assert a player's location with a vector of the appropriate tuple types:

```
(def data [{:player/handle "Argent Adept"
            :player/location [100 0]}])
```

Homogeneous Tuples

Homogeneous tuples provide variable length composites of a single attribute type. The `:db.type` of a homogeneous tuple is specified by the keyword attribute `:db/tupleType`.

Datomic itself includes a good example of homogeneous tuples in the definitions of the other tuple types. Both `:db/tupleTypes` and `:db/tupleAttrs` are declared as homogeneous tuples of `:db/tupleType :db.type/keyword`:

```
;; built in to Datomic
[{:db/ident :db/tupleAttrs
  :db/valueType :db.type/tuple
  :db/tupleType :db.type/keyword
  :db/cardinality :db.cardinality/one}
 {:db/ident :db/tupleTypes
  :db/valueType :db.type/tuple
  :db/tupleType :db.type/keyword
  :db/cardinality :db.cardinality/one}]
```

⇧

Attribute Predicates

> **Note:** Attribute Predicates require 480-8770 or later and a compatible base schema.

You may want to constrain an attribute value by more than just its storage/representation type. For example, an email address is not just a string, but a string with a particular format. In Datomic, you can assert *attribute predicates* about an attribute. Attribute predicates are asserted via the `:db.attr/preds` attribute, and are fully-qualified symbols that name a *predicate* of a value. Predicates return *true* (and only *true*) to indicate success. All other values indicate failure and are reported back as transaction errors.

Inside transactions, Datomic will call all attribute predicates for all attribute values, and abort a transaction if any predicate fails.

For example, the following function validates that a `user-name` has a particular length:

```
(ns datomic.samples.attr-preds)

(defn user-name?
  [s]
  (<= 3 (count s) 15))
```

⇧

To install the `user-name?` predicate, add a `db.attr/preds` value to an attribute, e.g.

```
{:db/ident :user/name,
 :db/valueType :db.type/string,
 :db/cardinality :db.cardinality/one,
 :db.attr/preds 'datomic.samples.attr-preds/user-name?}
```

⇧

A transaction that includes an invalid `user-name` will result in an *incorrect* anomaly that includes:

- the entity id
- the attribute name
- the attribute value
- the name of the failed predicate
- the predicate return in `:db.error/pred-return`

For example, the string "This-name-is-too-long" is not a valid `user-name?` and will cause an anomaly like:

```
{:cognitect.anomalies/category
 :cognitect.anomalies/incorrect,
 :cognitect.anomalies/message
 "Entity 42 attribute :user/name value This-name-is-too-long
failed pred datomic.samples.attr-preds/user-name?"
 :db.error/pred-return false}
```

⇧

Attribute predicates must be on the classpath of a process that is performing a transaction.

Attribute predicates can be asserted or retracted at any time, and will be enforced starting on the transaction after they are asserted. Asserting or retracting an attribute predicate has no effect on attribute values that already exist in the database.

Attribute predicates can `cancel` the transaction directly.

Entity Specs

> **Note:** Entity Specs require 480-8770 or later and a compatible base schema.

You may want to ensure properties of an entity being asserted, for example

- required keys
- the creation of related tuples
- satisfaction of properties that cut across attributes and the db

An *entity spec* is a Datomic entity having one or more of

- (usually) a `:db/ident`
- `:db.entity/attrs` naming required attributes
- `:db.entity/preds` naming entity predicates

You can then ensure an entity spec by asserting the `:db/ensure` attribute for an entity. For example, the following transaction data ensures entity "new-account-1" with entity spec `:new-account`

```
{:db/id "new-account-1"
 :db/ensure :new-account
 ;; other data that must be a valid :new-account
 }
```

`:db/ensure` is a virtual attribute. It is not added in the database; instead it triggers checks based on the named entity.

Entity predicates must be on the classpath of a process that is performing a transaction.

Entity specs can be asserted or retracted at any time, and will be enforced starting on the transaction after they are asserted. Asserting or retracting an entity spec has no effect on entities that already exist in the database.

Entity specs and `:db/ensure` are *not* analogous to traditional SQL constraints:

- Specs are more flexible, enforcing arbitrary shapes on particular entities without imposing an overall structure on the database.
- Specs can do more, allowing arbitrary functions of an entity and a database value.
- Specs must be requested explicitly per transaction+entity, and enforcement is never automatic or retroactive.

Required Attributes

The `:db.entity/attrs` attribute is a multi-valued attribute of keywords, where each keyword names a required attribute.

For example, the following transaction data creates a spec that requires a `:user/name` and `:user/email`:

```
{:db/ident :user/validate
 :db.entity/attrs [:user/name :user/email]}
```

The `:user/validate` entity can then be used in later transaction to ensure that all required attribute are present. For example, the following transaction would fail

```
{:user/name "John Doe"
 :db/ensure :user/validate}
```

When a required attribute is missing, Datomic will throw an anomaly whose `ex-data` includes the failing entity and the name of the spec, e.g.:

```
{:cognitect.anomalies/category
 :cognitect.anomalies/incorrect,
 :cognitect.anomalies/message
 "Entity 42 missing attributes [:user/email] of by :user/validate"}
```

Entity Predicates

The `:db.entity/preds` attribute is a multi-valued attribute of symbols, where each symbol names a predicate of database value and entity id. Inside a transaction, Datomic will call all predicates, and abort the transaction if any predicate returns a value that is not `true`.

For example, given the following predicate:

```
(ns datomic.samples.entity-preds
  (:require [datomic.api :as d]))

(defn scores-are-ordered?
  [db eid]
  (let [m (d/pull db [:score/low :score/high] eid)]
    (<= (:score/low m) (:score/high m))))
```

⇧

You could install the predicate on a guard entity:

```
{:db/ident :score/guard
 :db.entity/attrs [:score/low :score/high]
 :db.entity/preds 'datomic.samples.entity-preds/scores-are-ordered?}
```

⇧

Given an invalid entity that requests the `:score/guard`:

```
{:score/low 100
 :score/high 20
 :db/ensure :score/guard}
```

⇧

Datomic will throw an anomaly whose `ex-data` names the failing predicate:

```
{:cognitect.anomalies/category
 :cognitect.anomalies/incorrect,
 :cognitect.anomalies/message
 "Entity 42 failed pred datomic.samples.entity-preds/scores-are-ordered? of spec :score/guard"
 :db.error/pred-return false}
```

⇧

Datomic will report the value returned by the failing predicate under the `:db.error/pred-return` key. This can be used to report more information about what went wrong.

`cancel` can be used directly from entity predicates to cancel the transaction.

Next: Changing Schema