Datomic Cloud Documentation

Search

What is Datomic?

Local Dev and CI

Cloud Setup

Tutorial

Client API

Videos

- HTTP Direct
- CLI tools

Schema

- Defining Attributes
- Schema Reference
- 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

Tech Notes

- Turning Off Unused Resources
- Reserved Instances
- Lambda Provisioned Concurrency

Best Practices

Troubleshooting

FAQ

Examples

Releases

Glossary

Hide All Examples

Datomic Data Model

A Datomic database is a set of immutable atomic facts called datoms. A database contains no tables; rather, there is a universal schema of user-defined attributes. Any entity can possess any attribute.

Datomic datalog queries automatically use multiple indexes to support a variety of access patterns. In addition to supporting query, these indexes support identity and uniqueness, an accumulate-only time model, and lookup refs.

Day of Datomic Cloud contains a video overview of Datomic's Information Model.

Database

In Datomic, a *database value* is a set of *datoms* and is often abbreviated as *database* or *db*. A db is a point-in-time, immutable value and will never change. If you use the same db for several queries, you will know the answers are based upon exactly the same data from a single point in time.

Datoms

A *datom* is an atomic fact that represents the addition or retraction of a relation between an entity, an attribute, a value, and a transaction. A datom is expressed as a five-tuple:

- an entity id (E)
- an attribute (A)
- a value for the attribute (V)
- a transaction id (Tx)
- a boolean (Op) indicating whether the datom is being added or retracted

Example Datom

E	42
A	:user/favorite-color
V	:blue
Tx	1234
Op	true

Entities

An *entity* is a set of *datoms* that are all about the same E.

Example Entity

An entity can be visualized as a table:

E	A	V	Tx	Op
42	:user/favorite-color	:blue	1234	true
42	:user/first-name	"John"	1234	true
42	:user/last-name	"Doe"	1234	true
42	:user/favorite-color	:green	4567	true
42	:user/favorite-color	:blue	4567	false

Point-In-Time Entity Example

A point-in-time (as-of) view of an entity considers only datoms that whose *Op* is `true` as of a certain `Tx`. In the example above, John no longer prefers `:blue` as-of 4567, so the point-in-time view as-of 4567 is

E	A	V	Tx	Op
42	:user/first-name	"John"	1234	true
42	:user/last-name	"Doe"	1234	true
42	:user/favorite-color	:green	4567	true

Map View Example

It is often convenient to consider a point-in-time view as only a three-tuple with *Tx* and *Op* elided:

E	A	V
42	:user/first-name	"John"
42	:user/last-name	"Doe"
42	:user/favorite-color	:green

This three-tuple view is very similar to a programming language object where the *E* is analogous to *this* or *self*. The *map view* of an entity at a particular point in time captures this information more compactly, using the reserved pseudo-attribute name `:db/id` for *E*:

```
{:db/id 42
 :user/favorite-color :green
 :user/first-name "John"
 :user/last-name "Doe"}
```

Universal Schema

In a relational database, you must specify a table schema that enumerates in advance the attributes (columns) an entity can have. By contrast, Datomic requires only that you specify the properties of individual attributes. Any entity can then have any attribute. Because all datoms are part of a single relation, this is called a *universal schema*.

For example, consider storing an inventory database in Datomic. All inventory items have a unique string identifier, so you create an `:inv/id` attribute. In addition, you create other named attributes, specifying the types and cardinalities of each.

You can then store various inventory items in the database, each with different attributes, as shown in the following table.

E	A	V
7	:inv/id	"SKU-1234"
7	:inv/color	:green
8	:inv/id	"SKU-5678"
8	:inv/watts	60000

E	A	V
8	:doc/url	"http:…"

Notice that, other than `:inv/id`, entities 7 and 8 have entirely disjoint attributes.

Defining Schema

Each Datomic database has a schema that describes the set and kind of attributes that can be associated with your domain entities.

A schema only defines the characteristics of the attributes themselves. It does not define which attributes can be associated with which entities. Decisions about which attributes apply to which entities are made by your application.

This gives applications a great degree of freedom to evolve over time. For example, an application that wants to model a person as an entity does not have to decide up front whether the person is an employee or a customer. It can associate a combination of attributes describing customers and attributes describing employees with the same entity. An application can determine whether an entity represents a particular abstraction, customer or employee, simply by looking for the presence of the appropriate attributes.

There are two kinds of attributes in Datomic:

- **Domain attributes** - describe aspects of your domain data. You use domain attributes to describe the data about your domain entities.
- **Schema attributes** - describe aspects of the schema itself. Schema attributes are built-in and cannot be extended. You use schema attributes to define your domain attributes.

For more information see the schema documentation.

Indexes

Indexes are named by the order in which datom components are considered in the sort. For example, the index that sorts by (E)ntity, then (A)ttribute, then (V)alue, then (T)ransaction is named EAVT.

Datomic's indexes automatically support multiple styles of data access: row-oriented, column-oriented, document-oriented, K/V, and graph:

- The E-leading index EAVT supports efficient queries for details about particular entities, analogous to a traditional relational database.
- The A-leading index AEVT supports efficient queries for a single attribute across all entities, analogous to a column store.
- The V-leading index VAET supports efficient queries for references between entities, analogous to a graph database.
- The combination of EAVT and VAET supports arbitrary nested navigation among entities. This is like a document database but vastly more flexible. You can navigate in any direction at any time, instead of being limited to the containment hierarchy you selected when storing the document.

Taking advantage of the indexes requires no effort on the part of application developers:

- Indexes are all managed automatically, and there are no configuration knobs to consider when storing data.
- The appropriate indexes are selected automatically by the query engine, and there are no indexing options to consider when writing a query.

Datomic indexes are covering indexes. This means the index actually contains the datoms, rather than just a pointer to them. So, when you find datoms in the index, you get the datoms themselves, not just a reference to where they live. This allows Datomic to very efficiently access datoms through its indexes.

For more information, see the Indexes Documentation.

Time Model

Datomic's time model is applied automatically as transactions are recorded, ensuring that the following properties hold.

- Every transaction is assigned a transaction id that is unique within the database.
- Transactions are fully serialized, and transaction ids increase over time, so transaction ids are a total ordering of all transactions.
- Every datom in the database includes the id of the transaction that added that datom. Therefore transaction ids also act as a total ordering of all datoms.
- Transaction ids are themselves entity ids, so transactions can have datoms that describe them. Datomic assigns a `:db/txInstant` for every transaction, so every datom knows the wall clock time it was added.
- Transactions can have other attributes. For example, a system might record domain-specific provenance information associated with each transaction.

For example, imagine setting the count of an item in an inventory database in one transaction, then updating that count later. In addition to recording the item counts, Datomic also records the transactions that made the changes, when the changes happened, plus any additional provenance information you choose to include. This might be used to, for example, record that a particular transaction corrects another transaction that was made in error

Time Model

The example below shows that *Tx* acts as a global ordering of transactions. It also shows using a custom attribute to add audit information to a transaction: the `:correction/for` shows that *Tx* 1235 is fixing a mistake recorded in *Tx* 1234.

E	A	V	Tx	Op
42	:election/winner	"Dewey"	1234	true
42	:election/winner	"Dewey"	1235	false
42	:election/winner	"Truman"	1235	true
1235	:correction/for	1234	1235	true

Datomic's time model allows complete recovery of the data "as of" any point in time, and it also allows queries that look at differences between two or more points in time.

For more information, see the Datomic Time Model Documentation.

Datalog

Datomic's query and rules system is an extended form of Datalog. Datalog is a deductive query system, typically consisting of:

- A database of facts

- A set of rules for deriving new facts from existing facts
- a query processor that, given some partial specification of a fact or rule:
  - finds all instances of that specification implied by the database and rules
  - i.e. all the matching facts

Typically a Datalog system would have a global fact database and set of rules. Datomic's query engine instead takes databases (or other data sources) as fact sources and rule sets as inputs.

Datomic Datalog is simple, declarative, and logic-based.

Simple

Datalog is simple. The basic component of Datalog is a clause, which is a list that either begins with the name of a rule, or is a data pattern. These clauses can contain variables (symbols beginning with a `?`). The query engine finds all combinations of values of the variables that satisfy all of the clauses. There is no complex syntax to learn.

Declarative

Like SQL and other good query languages, Datalog is declarative. That is, you specify what you want to know and not how to find it. Declarative programs are:

- More *evident* - it is easier to tell what their purpose is, both for programmers and stakeholders.
- More readily optimized - the query engine is free to reorder and parallelize operations to a degree not normally taken on by application programs.
- Simpler - and thus, more robust.

Logic-based

Even SQL, while fundamentally declarative, still includes many operations that go beyond the query itself, like specifying joins explicitly. Because Datalog is based upon logical implication, joins are implicit, and the query engine figures out when they are needed.

Identity and Uniqueness

Datomic provides a number of tools for dealing with identity and uniqueness.

- When you create a datom, Datomic automatically assigns it a database-unique number called an entity id.
- The built-in :db/ident attribute can be used for programmatic names.

- The built-in :db/unique attribute can be added at any time. The uniqueness properties of an existing attribute also can be changed at any time (so long as the uniqueness properties are valid for the data). The :db/unique attribute can be used in the following ways:

  - `:db.unique/identity` allows transactions to work with domain notions of identity instead of entity ids. For example a primary contact email or SKU number can be modeled with a `:db.unique/identity` attribute.
  - `:db.unique/value` enforces a single holder of an identifying key.

- The `:db.type/uuid` value type is efficient for globally unique identifiers.
- Lookup-refs represent a lookup on a unique attribute as an attribute-value pair.

An *entity identifier* is any one of the three ways that Datomic can uniquely identify an entity:

- an entity id (`:db/id`)
- an ident (`:db/ident`)
- a lookup ref.

Unless otherwise specified, Datomic APIs that refer to entities take any these kinds of entity identifiers as arguments.

Lookup Refs

In many databases, entities have unique identifiers in the problem domain like an email address or an order number. Applications often need to find entities based on these external keys. You can do this with query, but it's easier to use a *lookup ref*. A lookup ref is a pair containing an attribute identifier and a value. It identifies the entity with the given unique attribute value. For example, this lookup ref:

```
[:person/email "joe@example.com"]
```

identifies the entity with the *:person/email* value "joe@example.com".

You can use lookup refs to retrieve entities using the *pull* API and to retrieve datoms using the *datoms* API. You cannot use them in the body of a query, use datalog clauses instead.

You can also use lookup refs to refer to existing entities in transactions, avoiding extra lookup code:

```
{:db/id [:person/email "joe@example.com"]
 :person/loves :pizza}
```

This transaction asserts that the entity with the value "joe@example.com" for the *:person/email* attribute also loves pizza.

Lookup refs have the following restrictions:

- The specified attribute must be defined as either *:db.unique/value* or *:db.unique/identity*.
- When used in a transaction, the lookup ref is evaluated against the specified attribute's index as it exists before the transaction is processed, so you cannot use a lookup ref to lookup an entity being defined in the same transaction.
- Lookup refs cannot be used in the body of a query though they can be used as inputs to a parameterized query.

Accumulate Only

Datomic is accumulate-only. Information accumulates over time, and change is represented by accumulating the new, not by modifying or removing the old. For example, "removing" occurs not by taking something away, but by adding a retraction.

This is in stark contrast with the Create/Read/Update/Delete (CRUD) paradigm:

CRUD	Datomic
Create	Assert
Read	Read

CRUD	Datomic
Update	Accumulate
Delete	Retract

- *Assertions* are granular statements of fact.
- *Reads* are always performed against an immutable database value at a particular point in time. Time is globally ordered in a database by ACID transactions.
- New transactions only *Accumulate* new data. Existing datoms never change.
- *Retractions* state that an assertion no longer holds at some later point in time. The original assertion remains unchanged.

Assert/Read/Accumulate/Retract (ARAR) should be pronounced doubled and in a pirate voice "Ar Ar Ar Ar".

Note that accumulate-only is **not** the same as append-only. Datomic is not an append-only system, and does not have the performance characteristics associated with append-only systems.