Datomic Cloud Documentation

Search

- Home ›
- Query and Pull ›
- Executing Queries
- Support
- Forum

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
- Changing Schema
- Data Modeling
- Schema Limits

Transactions

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

Query and Pull

- Executing Queries
  Querying a DatabaseqqseqUnificationList Form vs. Map FormWork with Data Structures, Not StringsTimeoutClause OrderQuery Caching
- 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

Hide All Examples

Executing Queries

Day of Datomic Cloud goes over query concepts, with examples on Github.

Querying a Database

To query a database, you must first obtain a connection and a database value. The example below shows a simple query using the Synchronous API.

```
;; get db value
(def db (d/db conn))

;; query
(d/q {:query [:find ?release-name
              :where [_ :release/name ?release-name]]
      :args [db]})

;; result
#{["Osmium"]
  ["Hela roept de akela"]
  ["Ali Baba"]
  ["The Power of the True Love Knot"]
  ...}
```

⇧

The arguments to `d/q` are documented in the Query Data Reference.

q

`datomic.client.ap/q` is the primary entry point for Datomic query.

`q` Performs the query described by query and args, and returns a collection of tuples.

- `:query` - The query to perform: a map, list, or string. Complete description.
    - `:find` - specifies the tuples to be returned.
    - `:with` - is optional, and names vars to be kept in the aggregation set but not returned
    - `:in` - is optional. Omitting ':in …' is the same as specifying ':in $'
    - `:where` - limits the result returned
- `:args` - Data sources for the query, e.g. database values retrieved from a call to db, and/or rules.

qseq

`qseq` is a variant of `q` that pulls and xforms lazily as you consume query results.

`datomic.client.ap/qseq` utilizes the same arguments and grammar as q.

`qseq` is primarily useful when you know in advance that you do not need/want a realized collection. i.e. you are only going to make a single pass (or partial pass) over the result data.

Item transformations such as `pull` are deferred until the seq is consumed. For queries with pull(s), this results in:

- Reduced memory use and the ability to execute larger queries.
- Lower latency before the first results are returned.

The returned seq object efficiently supports `count`.

Unification

Unification occurs when a variable appears in more than one data pattern. In the following query, *?e* appears twice:

```
;;which 42-year-olds like what?
[:find ?e ?x
 :where [?e :age 42] [?e :likes ?x]]
```

⇧

Matches for the variable *?e* must *unify*, i.e. represent the same value in every clause in order to satisfy the set of clauses. So a matching *?e* must have both *:age* 42 and *:likes* for some *?x*:

```
[[fred pizza], [ethel sushi]]
```

List Form vs. Map Form

Queries written by humans typically are a list, and the various keyword arguments are inferred by position. For example, the query

```
[:find ?e
 :in $ ?fname ?lname
 :where [?e :user/firstName ?fname]
        [?e :user/lastName ?lname]]
```

⇧

has one `:find` argument, three `:in` arguments, and two `:where` arguments.

While most people find the positional syntax easy to read, it makes extra work for programmatic readers and writers, which have to keep track of what keyword is currently "active" and interpret tokens accordingly. For such cases, queries can be specified more simply as maps. The query above becomes:

```
{:find [?e]
 :in [$ ?fname ?lname]
 :where [[?e :user/firstName ?fname]
         [?e :user/lastName ?lname]]}
```

⇧

Work with Data Structures, Not Strings

Two features of Datalog queries make them immune to many of the SQL-injection style attacks to which many other DBMSs are vulnerable:

- Datalog queries are composed of data structures, rather than strings, which obviates the need to do string interpolation, sanitization, escaping, etc.
- The query API is parameterized with data sources. In many cases, this feature obviates the need to include user-provided data in the query itself. Instead, you can pass user data to a parameterized query as its own data source.

You should avoid building queries by reading in a string that has been built up by concatenation or interpolation. Doing so gives up the security and simplicity of working with native data structures.

The example below shows the contrast between good and bad practice.

```
;; parameterized query: "The Beatles" is a data source
(def query '[:find ?e
             :in $ ?name
             :where [?e :artist/name ?name]])
(d/q query db "The Beatles")

;; NEVER DO THIS: string interpolation into a hard-coded query
(def query (format "[:find ?e
                     :where [?e :artist/name \"%s\"]]" "The Beatles"))
(d/q query db)
```

⇧

Timeout

You can configure a query to abort if it takes too long to run using Datomic's timeout functionality.

The example below lists all movies in the database by genre, but will likely fail due to the 1msec timeout.

```
(d/q {:query '[:find ?movie-genre
               :where [_ :movie/genre ?movie-genre]]
      :timeout 1
      :args [db]})
```

⇧

You will likely see something like `ExceptionInfo Datomic Client Timeout clojure.core/ex-info (core.clj:4739)`.

Note: Timeout is approximate. It is meant to protect against long running queries, but is not guaranteed to stop after precisely the duration specified.

The timeout is passed as an argument to the q API. Specifying timeout requires use of the 1-arity version.

Clause Order

To minimize the amount work the query engine must do, query authors should put the most restrictive or narrowing `:where` clauses first, and then proceed on to less restrictive clauses.

As an example, consider the following two queries looking for Paul McCartney's releases. The first `:where` clause begins with a data pattern (`[?release :release/name ?name]`) that is not at all selective, forcing the query engine to consider **all** the releases in the database:

```
;; query
[:find ?name
 :in $ ?artist
 :where [?release :release/name ?name]
        [?release :release/artists ?artist]]

;; inputs
db, mccartney
```

```
;; result
[["McCartney"] ["Another Day / Oh Woman Oh Why"] ["Ram"] ...]
```

⇧

The following equivalent query reorders the `:where` clause, leading with a much more selective pattern (`[?release :release/artists ?artist]`) that is limited in this context to the single `?artist` passed in.

```
;; query
[:find ?name
 :in $ ?artist
 :where [?release :release/artists ?artist]
        [?release :release/name ?name]]

;; inputs and result same as above
```

⇧

The second query runs 50 times faster on the mbrainz dataset.

Query Caching

Datomic processes maintain an in-memory cache of parsed query representations. Caching is based on equality of the query argument to `q`. To take advantage of caching, programs should

- Use parameterized queries (that is, queries with multiple inputs) instead of building dynamic queries.
- When building dynamic queries, use a canonical approach to naming and ordering such that equivalent queries will be equal.

In the example below, the parameterized query for artists will be cached on first use and can be reused any number of times:

```
(def query '[:find ?e
             :in $ ?name
             :where [?e :artist/name ?name]])

;; first use compiles and caches the query
(d/q query db "The Beatles")

;; subsequent uses find query in the cache
(d/q query db "The Who")
```

⇧

A semantically equivalent query with different names will be separately compiled and cached:

```
;; not an identical query, ?artist-name instead of ?name
(def query '[:find ?e
             :in $ ?artist-name
             :where [?e :artist/name ?artist-name]])
```

⇧