The Cozo Database Manual

Release 0.1.0

Ziyang Hu

Oct 22, 2022

CONTENTS

1 Getting started **1**
- 1.1 Starting Cozo . . . 1
- 1.2 The query API . . . 1
- 1.3 Running queries . . . 2
- 1.4 Building . . . 3

2 Queries **5**
- 2.1 Inline rules . . . 5
- 2.2 Fixed rules . . . 9
- 2.3 Query options . . . 9

3 Stored relations and transactions **11**
- 3.1 Stored relations . . . 11
- 3.2 Chaining queries . . . 13
- 3.3 Triggers . . . 13

4 System ops **15**
- 4.1 Explain . . . 15
- 4.2 Ops on stored relations . . . 15
- 4.3 Monitor and kill . . . 16
- 4.4 Maintenance . . . 16

5 Datatypes **17**
- 5.1 Value types . . . 17
- 5.2 Value literals . . . 18
- 5.3 Column types . . . 18

6 Query execution **19**
- 6.1 Stratification . . . 19
- 6.2 Magic set rewrites . . . 20
- 6.3 Semi-naïve evaluation . . . 20
- 6.4 Ordering of atoms . . . 20
- 6.5 Relations as indices . . . 21
- 6.6 Early stopping . . . 21

7 Functions **23**
- 7.1 Equality and Comparisons . . . 23
- 7.2 Boolean functions . . . 24
- 7.3 Mathematics . . . 24
- 7.4 String functions . . . 26
- 7.5 List functions . . . 27

7.6 Binary functions . . . 28
7.7 Type checking and conversions . . . 29
7.8 Random functions . . . 30
7.9 Regex functions . . . 31
7.10 Timestamp functions . . . 32

8 Aggregations **33**
8.1 Meet aggregations . . . 33
8.2 Ordinary aggregations . . . 34

9 Utilities and algorithms **37**
9.1 Utilities . . . 37
9.2 Connectedness algorithms . . . 38
9.3 Pathfinding algorithms . . . 39
9.4 Community detection algorithms . . . 41
9.5 Centrality measures . . . 42
9.6 Miscellaneous . . . 43

Index **45**

CHAPTER

ONE

GETTING STARTED

Cozo is distributed as a single executable. To get started, download the executable for your platform and uncompress it. After decompression, you may also need to give it executable permission by `chmod +x ./cozo` on Unix-based systems.

The pre-compiled distributions of Cozo support Linux, Mac and Windows. As the toolchain on Windows is very different from UNIX-based systems, the Windows build hasn't received as much attention as the other builds, and may suffer from inferior performance and Windows-specific bugs. For Windows users, we recommend running Cozo under WSL[1] if possible, especially if your workload is heavy.

1.1 Starting Cozo

Run the `cozo` command in a terminal:

```
./cozo PATH_TO_DATA_DIRECTORY
```

If `PATH_TO_DATA_DIRECTORY` does not exist, it will be created. Cozo will then start a web server and bind to address `127.0.0.1` and port `9070`. These two can be customized: run the executable with the `-h` option to learn how.

To stop Cozo, type `CTRL-C` in the terminal, or send `SIGTERM` to the process with e.g. `kill`.

1.2 The query API

Queries are run by sending HTTP POST requests to the server. By default, the API endpoint is `http://127.0.0.1:9070/text-query`. The structure of the expected JSON payload is:

```
{
    "script": "<COZOSCRIPT QUERY STRING>",
    "params": {}
}
```

`params` should be an object of named parameters. For example, if you have `params` set up to be `{"num": 1}`, then `$num` can be used anywhere in your query string where an expression is expected. Always use `params` instead of constructing query strings yourself when you have parametrized queries.

[1] https://learn.microsoft.com/en-us/windows/wsl/install

1.2.1 Security

Cozo is currently designed to run in a trusted environment and be used by trusted clients, therefore it does not come with elaborate authentication and security features. If you must access Cozo remotely, you are responsible for setting up firewalls, encryptions and proxies yourself.

As a guard against users carelessly binding Cozo to any address other than `127.0.0.1` and potentially exposing content to everyone on the Internet, in this case, Cozo will refuse to start unless you also set up the environment variable `COZO_AUTH`. With the variable set, Cozo will then require all queries to provide the content of the set variable in the HTTP header field `x-cozo-auth` for verification. Please note that this "security measure" is not considered sufficient for any purpose and is only a last defence when every other security measure that you are responsible for setting up fails.

1.3 Running queries

1.3.1 Making HTTP requests

As Cozo has a web-based API, it is accessible by all languages that are capable of making web requests. The structure of the API is also deliberately kept minimal so that no dedicated clients are necessary. The return values of requests are JSON when requests are successful, or text descriptions when errors occur, so a language only needs to be able to process JSON to use Cozo.

1.3.2 JupyterLab

JupyterLab[2] is a web-based notebook interface in the python ecosystem heavily used by data scientists and is the recommended "IDE" of Cozo.

First, install JupyterLab by following the install instructions of the project. Then install the pycozo library by running:

```
pip install "pycozo[pandas]"
```

Now, open the JupyterLab web interface, start a Python 3 kernel, and in a cell run the following magic command[3]:

```
%load_ext pycozo.ipyext_direct
```

If you need to connect to Cozo using a non-default address or port, or you require an authentication string, you need to run the following magic commands as well:

```
%cozo_host http://<BIND_ADDRESS>:<PORT>
%cozo_auth <YOUR_AUTH_STRING>
```

Now you can execute cells as you usually do in JupyterLab, and the content of the cells will be sent to Cozo and interpreted as CozoScript. Returned relations will be formatted as Pandas dataframe[4].

The above sets up the notebook in the Direct Cozo mode, where cells are default interpreted as CozoScript. You can still execute python code by starting the first line of a cell with the `%%py`. There is also an Indirect Cozo mode, started by:

```
%load_ext pycozo.ipyext
```

[2] https://jupyterlab.readthedocs.io/en/stable/
[3] https://ipython.readthedocs.io/en/stable/interactive/magics.html
[4] https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html

In this mode, only cells with the first line content `%%cozo` are interpreted as CozoScript. Other cells are interpreted in the normal way (by default, python code). Which mode you use depends on your workflow. We recommend the Indirect mode if you have lots of post-processing and visualizations.

When a query is successfully executed, the result will be bound to the python variable `_` as a Pandas dataframe (this is a feature of Jupyter notebooks: the Cozo extension didn't do anything extra).

There are a few other useful magic commands:

- `%cozo_run_file <PATH_TO_FILE>` runs a local file as CozoScript.
- `%cozo_run_string <VARIABLE>` runs variable containing string as CozoScript.
- `%cozo_set <KEY> <VALUE>` sets a parameter with the name `<KEY>` to the expression `<VALUE>`. The set parameters will be used by subsequent queries.
- `%cozo_set_params <PARAM_MAP>` replace all parameters by the given expression, which must evaluate to a dictionary with string keys.
- `%cozo_clear` clears all set parameters.
- `%cozo_params` returns the parameters currently set.

1.3.3 The Makeshift JavaScript Console

The Python and JupyterLab ecosystem is rather heavy-weight. If you are just testing out or running Cozo in an environment that only occasionally requires manual queries, you may be reluctant to install them. In this case, you may find the Makeshift JavaScript Console helpful.

As Cozo is running an HTTP service, we assume that the browser on your local machine can reach its network. We recommend Firefox[5], Chrome[6], or any Chromium-based browser for best display.

If Cozo is running under the default configuration, navigate to `http://127.0.0.1:9070`. You should be greeted with a mostly empty page telling you that Cozo is running. Now open the Developer Console (Firefox console[7] or Chrome console[8]) and switch to the "Console" tab. Now you can execute CozoScript by running:

```
await run("<COZOSCRIPT>")
```

The returned tables will be properly formatted. If you need to pass in parameters, provide a second parameter with a JavaScript object. If you need to set an auth string, modify the global variable `COZO_AUTH`.

The JavaScript Console is not as nice to use as Jupyter notebooks, but we think that it provides a much better experience than hand-rolled CLI consoles, since you can use JavaScript to manipulate the results.

1.4 Building

If for some reason the binary distribution does not work for you, you can build Cozo from source, which is straightforward. First, clone the Cozo git repo (you need to pass the `--recursive` flag so that submodules are also cloned), then you need to install the Rust toolchain[9] for your system. You also need a C++17 compiler. After these preparations, run `cargo build --release` in the root of the cloned repo, wait for potentially a long time, and you will find the compiled binary in `target/release` if everything goes well. You can pass `-F jemalloc` to the `cargo build` command to indicate that you want to compile and use jemalloc as the memory allocator for the RocksDB storage backend, which, depending on your workload, can make a difference in performance.

[5] https://www.mozilla.org/en-US/firefox/new/
[6] https://www.google.com/chrome/
[7] https://firefox-source-docs.mozilla.org/devtools-user/browser_console/index.html
[8] https://developer.chrome.com/docs/devtools/console/javascript/
[9] https://www.rust-lang.org/tools/install

CHAPTER

TWO

QUERIES

The Cozo database system is queried using the CozoScript language. At its core, CozoScript is a Datalog[10] dialect supporting stratified negation and stratified recursive meet-aggregations. The built-in utilities and algorithms (mainly graph algorithms) further empower CozoScript for much greater ease of use and much wider applicability.

A query consists of one or many named rules. Each named rule conceptually represents a relation or a table with rows and columns. The rule named `?` is called the entry to the query, and its associated relation is returned as the result of the query. Each named rule has associated with it a rule head, which names the columns of the relation, and a rule body, which specifies the content of the relation, or how the content should be computed.

In CozoScript, relations (stored relations or relations defined by rules) abide by the *set semantics*, meaning that even if a rule may compute a row multiple times, it will occur only once in the output. This is in contradistinction to SQL.

There are two types of named rules in CozoScript: *inline rules* distinguished by using `:=` to connect the head and the body, and *fixed rules* distinguished by using `<~` to connect the head and the body. You may think that *constant rules* with `<-` constitute a third type, written as:

```
const_rule[a, b, c] <- [[1, 2, 3], [4, 5, 6]]
```

but this is merely syntax sugar for the fixed rule of the `Constant` utility:

```
const_rule[a, b, c] <~ Constant(data: [[1, 2, 3], [4, 5, 6]])
```

2.1 Inline rules

An example of an inline rule is:

```
hc_rule[a, e] := rule_a['constant_string', b], rule_b[b, d, a, e]
```

The rule body of an inline rule consists of multiple *atoms* joined by commas, and is interpreted as representing the *conjunction* of these atoms.

[10] https://en.wikipedia.org/wiki/Datalog

2.1.1 Atoms

Atoms come in various flavours. In the example above:

```
rule_a['constant_string', b]
```

is an atom representing a *rule application*: a rule named `rule_a` must exist in the same query and have the correct arity (2 here). Each row in the named rule is then *unified* with the bindings given as parameters in the square bracket: here the first column is unified with a constant string, and unification succeeds only when the string completely matches what is given; the second column is unified with the *variable* `b`, and as the variable is fresh at this point (meaning that it first appears here), the unification will always succeed and the variable will become *bound*: from this point take on the value of whatever it was unified with in the named relation.

When a bound variable is used again later, for example in `rule_b[b, d, a, e]`, the variable `b` was bound at this point, this unification will only succeed when the unified value is the same as the previously unified value. In other words, repeated use of the same variable in named rules corresponds to inner joins in relational algebra.

Another flavour of atoms is the *stored relation*. It may be written similarly to a rule application:

```
:stored_relation[bind1, bind2]
```

with the colon in front of the stored relation name to distinguish it from rule application. Written in this way, you must give as many bindings to the stored relation as its arity, and the bindings proceed by argument positions, which may be cumbersome and error-prone. So alternatively, you may use the fact that columns of a stored relation are always named and bind by name:

```
:stored_relation{col1: bind1, col2: bind2}
```

In this case, you only need to bind as many variables as you use. If the name you want to give the binding is the same as the name of the column, you may use the shorthand notation: `:stored_relation{col1}` is the same as `:stored_relation{col1: col1}`.

Expressions are also atoms, such as:

```
a > b + 1
```

Here `a` and `b` must be bound somewhere else in the rule, and the expression must evaluate to a boolean, and act as a *filter*: only rows where the expression evaluates to true are kept.

You can also use *unification atoms* to unify explicitly:

```
a = b + c + d
```

for such atoms, whatever appears on the left-hand side must be a single variable and is unified with the right-hand side. This is different from the equality operator ==, where both sides are merely required to be expressions. When the left-hand side is a single *bound* variable, it may be shown that the equality and the unification operators are semantically equivalent.

Another form of *unification atom* is the explicit multi-unification:

```
a in [x, y, z]
```

here the variable on the left-hand side of `in` is unified with each item on the right-hand side in turn, which in turn implies that the right-hand side must evaluate to a list (but may be represented by a single variable or a function call).

2.1.2 Head and returned relation

Atoms, as explained above, corresponds to either relations (or their projections) or filters in relational algebra. Linked by commas, they, therefore, represent a joined relation, with named columns. The *head* of the rule, which in the simplest case is just a list of variables, then defines whichever columns to keep, and their order in the output relation.

Each variable in the head must be bound in the body, this is one of the *safety rules* of Datalog. Not all variables appearing in the body need to appear in the head.

2.1.3 Multiple definitions and disjunction

For inline rules only, multiple rule definitions may share the same name, with the requirement that the arity of the head in each definition must match. The returned relation is then the *disjunction* of the multiple definitions, which correspond to *union* in SQL. *Intersect* in SQL can be written in CozoScript into a single rule since commas denote conjunction. In complicated situations, you may instead write disjunctions in a single rule with the explicit `or` operator:

```
rule1[a, b] := rule2[a] or rule3[a], rule4[a, b]
```

For completeness, there is also an explicit `and` operator, but it is semantically identical to the comma, except that it has higher operator precedence than `or`, which in turn has higher operator precedence than the comma.

During evaluation, each rule is canonicalized into disjunction normal form[11] and each clause of the outmost disjunction is treated as a separate rule. The consequence is that the safety rule may be violated even though textually every variable in the head occurs in the body. As an example:

```
rule[a, b] := rule1[a] or rule2[b]
```

is a violation of the safety rule since it is rewritten into two rules, each of which is missing a different binding.

2.1.4 Negation

Atoms in inline rules may be *negated* by putting `not` in front of them, as in:

```
not rule1[a, b]
```

When negating rule applications and stored relations, at least one binding must be bound somewhere else in the rule in a non-negated context: this is another safety rule of Datalog, and it ensures that the outputs of rules are always finite. The unbound bindings in negated rules remain unbound: negation cannot introduce bound bindings to be used in the head.

Negated expressions act as negative filters, which is semantically equivalent to putting `!` in front of the expression. Since negation does not introduce new bindings, unifications and multi-unifications are converted to equivalent expressions and then negated.

[11] https://en.wikipedia.org/wiki/Disjunctive_normal_form

2.1.5 Recursion and stratification

The body of an inline rule may contain rule applications of itself, and multiple inline rules may apply each other recursively. The only exception is the entry rule `?`, which cannot be referred to by other rules.

Self and mutual references allow recursion to be defined easily. To guard against semantically pathological cases, recursion cannot occur in negated positions: the Russell-style rule `r[a] := not r[a]` is not allowed. This requirement creates an ordering of the rules, since negated rules must evaluate to completion before rules that apply them can start evaluation: this is called *stratification* of the rules. In cases where a total ordering cannot be defined since there exists a loop in the ordering required by negation, the query is then deemed unstratifiable and Cozo will refuse to execute it.

Note that since CozoScript allows unifying fresh variables, you can still easily write programs that produce infinite relations and hence cannot complete through recursion, but that are still accepted by the database. One of the simplest examples is:

```
r[a] := a = 0
r[a] := r[b], a = b + 1
?[a] := r[a]
```

It is up to the user to ensure that such programs are not submitted to the database, as it is not even in principle possible for the database to rule out such cases without wrongly rejecting valid queries. If you accidentally submitted one, you can refer to the system ops section for how to terminate long-running queries. Or you can give a timeout for the query when you submit.

2.1.6 Aggregation

CozoScript supports aggregations, as does SQL, which provides a very useful extension to pure relational algebra. In CozoScript, aggregations are specified for inline rules by applying aggregation operators to variables in the rule head, as in:

```
?[department, count(employee)] := :personnel{department, employee}
```

here we have used the `count` operator familiar to all SQL users. The semantics is that any variables in the head without aggregation operators are treated as *grouping variables*, similar to what appears in a `GROUP BY` clause in SQL, and the aggregation is applied using the grouping variables as keys. If you do not specify any grouping variables, then you get at most one row as the return value.

As we now understand, CozoScript follows relational algebra with set semantics. With the introduction of aggregations, the situation is a little bit more complicated, as aggregations are applied to the relation resulting from the body of the rule using bag semantics, and the resulting relation of the rule, after aggregations are applied, follows set semantics. The reason for this complication is that if aggregations are applied with set semantics, then the following query:

```
?[count(employee)] := :personnel{employee}
```

does not do what you expect: it either returns a row with a single value 1 if there are any matching rows, or it returns nothing at all if the stored relation `:personnel` is empty. Though semantically sound, this behaviour is not useful at all. So for aggregations, we opt for bag semantics, and the query does what one expects.

If a rule has several definitions, they must have identical aggregations applied in the same positions, otherwise, the query will be rejected. The reason is that in complicated situations the semantics is ambiguous and counter-intuitive if we do allow it.

Existing database systems do not usually allow aggregations through recursion, since in many cases, it is difficult to give useful semantics to such queries. In CozoScript we allow aggregations for self-recursion for a limited subset of aggregation operators, the so-called *semi-lattice aggregations*, such as the following example shows:

```
shortest_distance[destination, min(distance)] := route{source: 'A', destination,␣
↪distance}
shortest_distance[destination, min(distance)] :=
    shortest_distance[existing_node, prev_distance], # recursion
    route{source: existing_node, distance: route_distance},
    distance = prev_distance + route_distance
?[destination, min_distance] := shortest_distance[destination, min_distance]
```

this query computes the shortest distances from a node to all nodes using the `min` aggregation operator.

Concerning stratification, if a rule has aggregations in its head, then any rule that contains it in an atom must be in a higher stratum, unless that rule is the same rule (self-recursion) and all aggregations in its head are semi-lattice aggregations.

Consult the dedicated chapter for the aggregation operators available.

2.2 Fixed rules

The body of a fixed rule starts with the name of the utility or algorithm being applied, then takes a specified number of named or stored relations as its *input relations*, followed by *options* that you provide. The following query is a calculation of PageRank:

```
?[] <~ PageRank(:route[], theta: 0.5)
```

In the above example, the relation `:route` is the single input relation expected. Input relations may be stored relations or relations resulting from rules. Each utility/algorithm expects specific shapes of input relations, for example, PageRank expects the first two columns of the relation to denote the source and destination of links in a graph. You must consult the documentation for each utility/algorithm to understand its API. In fixed rules, bindings for input relations are usually omitted, but sometimes if they are provided they are interpreted and used in case-specific ways, for example in the DFS algorithm bindings can be used to construct an expression for testing the termination condition. In the example given above, `theta` is an option of the algorithm, which is required by the API to be an expression evaluating to a constant. Each utility/algorithm expects specific types for the options; some options have default values and may be omitted.

Each fixed rule has a determinate output arity, deduced from the specific utility/algorithm being applied and the options given. Usually, you omit the bindings in the rule head, as we do above, but if you do provide bindings, the arities must match.

In terms of stratification, each fixed rule lives in its own stratum: it is evaluated after all rules it depends on are completely evaluated, and all rules depending on the output relation of a fiex rule start evaluation only after complete evaluation of the fixed rule. In particular, unlike inline rules, there is no early termination even if the output relation is for the entry rule.

2.3 Query options

Each query can have query options associated with it:

```
?[name] := :personnel{name}

:limit 10
:offset 20
```

In the example, `:limit` and `:offset` are query options, with familiar meanings from SQL. All query options start with a single colon `:`. Queries options can appear before or after rules, or even sandwiched between rules. Use this freedom for better readability.

Several query options deal with transactions for the database. Those will be discussed in the chapter on stored relations and transactions. Here we explain query options that exclusively affect the query itself.

`:limit <N>`

Limit output relation to at most <N> rows. If possible, execution will stop as soon as this number of output rows is collected.

`:offset <N>`

Skip the first <N> rows of the returned relation.

`:timeout <N>`

Abort if the query does not complete within <N> seconds. Seconds may be specified as an expression so that random timeouts are possible.

`:sleep <N>`

If specified, the query will wait for <N> seconds after completion, before committing or proceeding to the next query. Seconds may be specified as an expression so that random timeouts are possible. This is useful for deliberately interleaving concurrent queries to test for complex logic.

`:sort <SORT_ARG> (, <SORT_ARG>)*`

Sort the output relation before applying other options or returning. Specify <SORT_ARG> as they appear in the rule head of the entry, separated by commas. You can optionally specify the sort direction of each argument by prefixing them with + or -, with minus denoting descending sort. As an example, `:sort -count(employee), dept_name` sorts by employee count descendingly first, then break ties with department name in ascending alphabetical order. Note that your entry rule head must contain both `dept_name` and `count(employee)`: aggregations must be done in inline rules, not in output sorting. `:order` is an alias for `:sort`.

`:assert none`

With this option, the query returns nothing if the output relation is empty, otherwise execution aborts with an error. Essential for transactions and triggers.

`:assert some`

With this option, the query returns nothing if the output relation contains at least one row, otherwise, execution aborts with an error. Execution of the query stops as soon as the first row is produced if possible. Essential for transactions and triggers.

CHAPTER

THREE

STORED RELATIONS AND TRANSACTIONS

Persistent databases store data on disk. As Cozo is a relational database, data are stored in *stored relations* on disk, which is analogous to tables in SQL databases.

3.1 Stored relations

We already know how to query stored relations: use the `:relation[...]` or `:relation{...}` atoms in inline or fixed rules. To manipulate stored relations, use one of the following query options:

`:create <NAME> <SPEC>`

Creates a stored relation with the given name and the given spec. The named stored relation must not exist before. If a query is specified, data from the resulting relation is put into the created stored relation. This is the only stored relation-related query option in which a query may be omitted.

`:replace <NAME> <SPEC>`

This is similar to `:create`, except that if the named stored relation exists beforehand, it is completely replaced. The schema of the replaced relation need not match the new one. You cannot omit the query for `:replace`.

`:put <NAME> <SPEC>`

Put data from the resulting relation into the named stored relation. If keys from the data exist beforehand, the rows are simply replaced with new ones.

`:ensure <NAME> <SPEC>`

Ensures that rows specified by the output relation and spec already exist in the database and that no other process has written to these rows at commit since the transaction starts. Useful for ensuring read-write consistency.

`:rm <NAME> <SPEC>`

Remove data from the resulting relation from the named stored relation. Only keys are used. If a row from the resulting relation does not match any keys, nothing happens for that row, and no error is raised.

`:ensure_not <NAME> <SPEC>`

Ensures that rows specified by the output relation and spec do not exist in the database and that no other process has written to these rows at commit since the transaction starts. Useful for ensuring read-write consistency.

You can rename and remove stored relations with the system ops `::relation rename` and `::relation remove`, described in the system op chapter.

3.1.1 Create and replace

The format of <SPEC> is identical for all four ops, whereas the semantics is a bit different.

We first describe the format and semantics for `:create` and `:replace`. A spec is a specification for columns, enclosed in curly braces `{}` and separated by commas:

```
?[address, company_name, department_name, head_count] <- $input_data

:create dept_info {
    company_name: String,
    department_name: String,
    =>
    head_count: Int,
    address: String,
}
```

Columns before the symbol => form the *keys* (actually, a composite key) for the stored relation, and those after it form the *values*. If all columns are keys, the symbol => may be omitted altogether. The order of columns matters in the specification, especially for keys, as data is stored in lexicographically sorted order in trees, which has implications for data access in queries. Each key corresponds to a single value.

In the above example, we explicitly specified the types for all columns. Type specification is described in its own chapter. If the types of the rows do not match the specified types, the system will first try to coerce the values, and if that fails, the query is aborted. You can selectively omit types for columns, and columns with types omitted will have the type `Any?`, which is valid for any value. As an example, if you do not care about type validation, the above query can be written as:

```
?[address, company_name, department_name, head_count] <- $input_data

:create dept_info { company_name, department_name => head_count, address }
```

In the example, the bindings for the output match the columns exactly (though not in the same order). You can also explicitly specify the correspondence:

```
?[a, b, count(c)] <- $input_data

:create dept_info { company_name = a, department_name = b, => head_count = count(c),␣
↪address = b }
```

You *must* use explicit correspondence if the entry head contains aggregation.

Instead of specifying bindings, you can specify an expression to generate values:

```
?[a, b] <- $input_data

:create dept_info { company_name = a, department_name = b, => head_count default 0,␣
↪address default '' }
```

The expression is evaluated once for each row, so for example if you specified one of the UUID-generating functions, you will get a different UUID for each row.

3.1.2 Put, remove, ensure and ensure-not

For `:put`, `:remove`, `:ensure` and `:ensure_not`, you do not need to specify all existing columns in the spec if the omitted columns have a default generator, in which case the generator will be used to generate a value, or the type of the column is nullable, in which case the value is `null`. Also, the order of the columns does not matter, and neither does whether a column occurs in the key or value position. The spec specified when the relation was created will be consulted to know how to store data correctly. Specifying default values does not have any effect and will not replace existing ones.

For `:put` and `:ensure`, the spec needs to contain enough bindings to generate all keys and values. For `:rm` and `:ensure_not`, it only needs to generate all keys.

3.2 Chaining queries

Each script you send to Cozo is executed in its own transaction. To ensure consistency of multiple operations on data, You can define multiple queries in a single script, by wrapping each query in curly braces `{}`. Each query can have its independent query options. Execution proceeds for each query serially, and aborts at the first error encountered. The returned relation is that of the last query. Within a transaction, execution of queries adheres to multi-version concurrency control: only data that are already committed, or written within the same transaction, are read, and at the end of the transaction, any changes to stored relations are only committed if there are no conflicts and no errors are raised.

The `:assert`, `:ensure` and `:ensure_not` query options allow you to express complicated constraints that must be satisfied for your transaction to commit.

3.3 Triggers

Cozo does not have traditional indices on stored relations. You must define your indices as separate stored relations yourself, for example by having a relation containing identical data but in different column order. More complicated and exotic "indices" are also possible and used in practice. At query time, you explicitly query the index instead of the original stored relation.

You synchronize your indices and the original by ensuring that any mutations you do on the database write the correct data to the "canonical" relation and its indices in the same transaction. As doing this by hand for every mutation in your business logic leads to lots of repetitions, is error-prone and a maintenance nightmare, Cozo also supports *triggers* to do it automatically for you.

You attach triggers to a stored relation by running the system op `::relation set_triggers`:

```
::relation set_triggers relation_name

on put { <QUERY> }
on put { <QUERY> } # you can specify as many triggers as you need
on rm { <QUERY> }
on replace { <QUERY> }
```

You can have anything valid query for `<QUERY>`.

The `on put` queries will run when any data is inserted into the relation, which can be triggered by `:put`, `:create` and `:replace` query options. The implicitly defined rules `_new[]` and `_old[]` can be used in the queries, and contain the added rows, and the replaced rows (if any).

The `on rm` queries will run when deletion is triggered by the `:rm` query option. The implicitly defined rules `_new[]` and `_old[]` can be used in the queries, the first rule contains the keys of the rows for deletion, and the second rule contains the rows actually deleted, with both keys and non-keys.

The `on replace` queries will run when `:replace` query options are run. They are run before any `on put` triggers are run for the same stored relation.

All triggers for a relation must be specified together, in the same system op. In other words, `::relation set_triggers` simply replaces all the triggers associated with a stored relation. To remove all triggers from a stored relation, simply pass no queries for the system op.

Besides indices, creative use of triggers abounds, but you must consider the maintenance burden they introduce.

Warning: Do not introduce loops in your triggers. A loop occurs when a relation has triggers which affect other relations, which in turn have other triggers that ultimately affect the starting relation.

CHAPTER

FOUR

SYSTEM OPS

System ops start with a double-colon `::` and must appear alone in a script. In the following, we explain what each system op does, and the arguments they expect.

4.1 Explain

`::explain { <query> }`

A single query is enclosed in curly braces. Query options are allowed but ignored. The query is not executed, but its query plan is returned instead. Currently, there is no specification for the return format, but if you are familiar with the semi-naïve evaluation of stratified Datalog programs subject to magic-set rewrites, the returned data is pretty self-explanatory.

4.2 Ops on stored relations

`::relations`

List all stored relations currently in the database

`::relation columns <REL_NAME>`

List all columns for the stored relation <REL_NAME>.

`::relation remove <REL_NAME> (, <REL_NAME>)*`

Remove stored relations. Several can be specified, joined by commas.

`::relation rename <OLD_NAME> -> <NEW_NAME> (, <OLD_NAME> -> <NEW_NAME>)*`

Rename stored relation <OLD_NAME> into <NEW_NAME>. Several may be specified, joined by commas.

`::relation show_triggers <REL_NAME>`

Display triggers associated with the stored relation <REL_NAME>.

`::relation set_triggers <REL_NAME> <TRIGGERS>`

Set triggers for the stored relation <REL_NAME>. This is explained in more detail in the transactions chapter.

4.3 Monitor and kill

`::running`

Display currently running queries and their IDs.

`::kill <ID>`

Kill a running query specified by `<ID>`. The ID may be obtained by `::running`.

4.4 Maintenance

`::compact`

Run compaction on the database. Running this may make the database smaller on disk and faster for queries, but running the op itself may take some time in the background.

CHAPTER

FIVE

DATATYPES

5.1 Value types

A runtime value in Cozo can be of the following *value-types*:

- `Null`
- `Bool`
- `Number`
- `String`
- `Bytes`
- `Uuid`
- `List`

`Number` can be `Float` (double precision) or `Int` (signed, 64 bits). Cozo will auto-promote `Int` to `Float` when necessary.

`List` can contain any number of mixed-type values, including other lists.

Cozo sorts values according to the above order, e.g. `null` is smaller than `true`, which is in turn smaller than the list `[]`.

Within each type values are *compared* according to logic custom to each type:

- `false < true`;
- `-1 == -1.0 < 0 == 0.0 < 0.5 == 0.5 < 1 == 1.0` (however, see the caveat below);
- Lists are ordered lexicographically by their elements;
- Bytes are compared lexicographically;
- Strings are ordered lexicographically by their UTF-8 byte representations.
- UUIDs are sorted in a way that UUIDv1 with similar timestamps are near each other. This is to improve data locality and should be considered an implementation detail. Depending on the order of UUID in your application is not recommended.

Warning: Because there are two internal number types `Int` and `Float` under the umbrella type `Number`, sorting numbers can be more complex than anticipated.

When sorting, the integer always comes before the equivalent float. For example, `1.0 == 1`, `1.0 >= 1` and `1.0 <= 1` all evaluate to true, but when sorting `1` and `1.0` are two _different_ values and `1` is placed before `1.0`.

This may create problems when applying aggregations since if a grouping key contains both `1.0` and `1`, they are treated as separate group headings. In such cases, it may help to use explicit coercion `to_float` or `round` to coerce all sorted values to the same type.

5.2 Value literals

The standard notations `null` for the type `Null`, `false` and `true` for the type `Bool` are followed.

Besides the usual decimal notation for signed integers, you can prefix a number with `0x` or `-0x` for hexadecimal notation, with `0o` or `-0o` for octal notation, or with `0b` or `-0b` for binary notation. Floating point numbers include the decimal dot, which may be trailing, and may be in scientific notation. All numbers may include underscores `_` in their representation for clarity. For example, `299_792_458` is the speed of light in meters per second.

Strings can be typed in the same way as they do in JSON between double quotes `""`, with the same escape rules. You can also use single quotes `''` in which case the roles of the double quote and single quote are switched. In addition, there is a raw string notation:

```
r___"I'm a raw string with "quotes"!"___
```

A raw string starts with the letter `r` followed by an arbitrary number of underscores, and then a double quote. It terminates when followed by a double quote and the same number of underscores. Everything in between is interpreted exactly as typed, including any newlines.

There is no literal representation for `Bytes` or `Uuid` due to restrictions placed by JSON. You must pass in its Base64 encoding for bytes, or hyphened strings for UUIDs, and use the appropriate functions to decode it. If you are just inserting data into a stored relation with a column specified to contain bytes or UUIDs, auto-coercion will kick in.

Lists are items enclosed between square brackets `[]`, separated by commas. A trailing comma is allowed after the last item.

5.3 Column types

The following *atomic types* can be specified for columns in stored relations:

- `Int`
- `Float`
- `Bool`
- `String`
- `Bytes`
- `Uuid`

There is no `Null` type. Instead, if you put a question mark after a type, it is treated as *nullable*, meaning that it either takes value in the type or is null.

Two composite types are available. A *homogeneous list* is specified by square brackets, with the inner type in between, like this: `[Int]`. You may optionally specify how many elements are expected, like this: `[Int; 10]`. A *heterogeneous list*, or a *tuple*, is specified by round brackets, with the element types listed by position, like this: `(Int, Float, String)`. Tuples always have fixed lengths.

A special type `Any` can be specified, allowing all values except null. If you want to allow null as well, use `Any?`. Composite types may contain other composite types or `Any` types as their inner types.

CHAPTER

SIX

QUERY EXECUTION

Usually, in a database, how queries are executed is usually considered an implementation detail hidden behind an abstraction barrier, which normal users need not care about. The idea is that databases will take advantage of this abstraction barrier by using query optimizers to choose the best query plan, regardless of how the query was originally written. As everyone knows, however, this abstraction barrier is leaky, since bad query execution plans invariably occur and hurt performance. The problem is especially severe when dealing with graphs, since graph traversals generally use a lot more joins than non-graph queries, and the reliability of even the best query optimizer decreases exponentially with the number of joins.

In Cozo we take the pragmatic approach and assume that the user eventually knows (or should know) what is the best way to query their data. This is certainly true for those developers who spend hours "coercing" the query optimizers to use a query plan that the user intends, sometimes in rather convoluted ways. In Cozo, no coercion is necessary since the query execution is completely determined by how the query is written: there is no stats-based query planning involved. In our experience, this saves quite a lot of developer time, since developers eventually learn how to write efficient queries naturally, and after they do, they no longer have to deal with endless "query de-optimizations".

6.1 Stratification

As discussed in the chapter on queries, Cozo sees a query as a set of named rules. Fixed rules are left as they are, and inline rules are converted into disjunctive normal forms. After conversion, all inline rules consist of conjunction of atoms only, and negation only occurs for the leaf atoms.

The next step towards executing the query is *stratifying* the rules. Stratification begins by making a graph of the named rules, with the rules themselves as nodes, and a link is added between two nodes when one of the rules applies the other. This application is through atoms for inline rules, and input relations for fixed rules. Now some of the links are labelled *stratifying*: when an inline rule applies another rule through negation, when an inline rule applies another inline rule that contains aggregations, when an inline rule applies itself and it has non-semi-lattice, when an inline rule applies another rule which is a fixed rule, or when a fixed rule has another rule as an input relation. The strongly connected components of this graph are then determined and tested, and if it found that some strongly connected component contains a stratifying link, the graph is deemed *unstratifiable*, and the execution aborts. Otherwise, Cozo will topologically sort the strongly connected components to determine a *stratification* of the rules: rules within the same stratum are logically executed together, and no two rules within the same stratum can have a stratifying link between them. In this process, Cozo will merge the strongly connected components into as few supernodes as possible while still maintaining the restriction on stratifying links. The resulting strata are then passed on to be processed in the next step.

You can see the stratum number assigned to rules by using the `::explain` system op.

6.2 Magic set rewrites

Within each stratum, the input rules are rewritten using a technique called magic sets. In intuitive terms, this rewriting is to ensure that the query execution does not waste time calculating results that are then simply discarded. As an example, consider:

```
reachable[a, b] := link[a, n]
reachable[a, b] := reachable[a, c], link[c, b]
?[r] := reachable['A', r]
```

Without magic set rewrites, the whole `reachable` relation is generated first, then most of them are thrown away, keeping only those starting from `'A'`. Magic set avoids this problem. How the rewrite proceeds is rather technical, but you can see the results in the output of `::explain`.

The rewritten query is guaranteed to yield the same relation for `?`, and will in general yield fewer intermediate rows.

Currently, the rewrite applies only to inline rules without aggregations. So for the moment being, you may need to manually constrain some of your rules.

6.3 Semi-naïve evaluation

Now each stratum contains either a single fixed rule or a set of inline rules. The single fixed rule case is easy: just run the specific implementation of the rule. In the case of the inline rules, each of the rules is assigned an output relation. Assuming we know how to evaluate each rule given all the relations it depends on, the semi-naïve algorithm can now be applied to the rules to yield all output rows.

The semi-naïve algorithm is a bottom-up evaluation strategy, meaning that it tries to deduce all facts from a set of given facts. By contrast, top-down strategies start with stated goals and try to find proof for the goals. Bottom-up strategies have many advantages over top-down ones when the whole output of each rule is needed, but may waste time generating unused facts if only some of the output is kept. Magic set rewrites are introduced to eliminate precisely this weakness.

6.4 Ordering of atoms

Now we discuss how a single definition of an inline rule is evaluated. We know from the query chapter that the body of the rule contains atoms, and after conversion to disjunctive normal forms, all atoms are linked by conjunction, and each atom can only be one of the following:

- an explicit unification,
- applying a rule or a stored relation,
- an expression, which should evaluate to a boolean,
- a negation of an application.

The first two cases may introduce fresh bindings, whereas the last two cannot. The atoms are then reordered: all atoms that introduce new bindings stay where they are, whereas all atoms that do not introduce new bindings are moved to the earliest possible place where all their bindings are bound. In fact, all atoms that introduce bindings correspond to joining with a pre-existing relation followed by projections in relational algebra, and all atoms that do not correspond to filters. The idea is to apply filters as early as possible to minimize the number of rows before joining with the next relation.

This procedure is completely deterministic. When writing the body of rules, we therefore should aim to minimize the total number of rows generated. A strategy that works almost in all cases is to put the most restrictive atoms which generate new bindings first, as this can make the left relation in each join small.

6.5 Relations as indices

Next, we need to understand how a single atom which generates new bindings is processed.

For the case of unification, it is simple: the right-hand side of the unification, which is an expression with all variables bound, is simply evaluated, and the result is joined to the current relation (as in a `map-cat` operation in functional languages).

For the case of the application of relations, the first thing to understand is that all relations in Cozo are conceptually trees. All the bindings of relations generated by inline or fixed rules, and the keys of stored relations, act as a composite key for the tree. The access complexity is therefore determined by whether a key component is bound. For example, the following application:

```
a_rule['A', 'B', c]
```

with `c` unbound is very efficient, since this corresponds to a prefix scan in the tree with the key prefix `['A', 'B']`, whereas the following application:

```
a_rule[a, 'B', 'C']
```

where `a` is unbound is very expensive, since we must do a full relation scan. On the other hand, if `a` is bound, then this is only a logarithmic-time check.

For stored relations, you need to check its schema for the order of keys to deduce the complexity. The system op `::explain` may also give you some information.

6.6 Early stopping

Within each stratum, rows are generated in a streaming fashion. For the entry rule `?`, if `:limit` is specified as a query option, a counter is used to monitor how many valid rows are already generated. If enough rows are generated, the query stops. Note that this only works when the entry rule is inline, and when you are *not* specifying `:order`.

CHAPTER

SEVEN

FUNCTIONS

Functions can be used in expressions in Cozo. All function arguments in Cozo are immutable. All functions except those having names starting with `rand_` are deterministic.

Internally, all function arguments are partially evaluated before binding variables to input tuples. For example, the regular expression in `regex_matches(var, '[a-zA-Z]+')` will only be compiled once during the execution of the query, instead of being repeatedly compiled for every input tuple.

7.1 Equality and Comparisons

eq(*x*, *y*)

Equality comparison. The operator form is `x == y` or `x = y`. The two arguments of the equality can be of different types, in which case the result is `false`.

neq(*x*, *y*)

Inequality comparison. The operator form is `x != y`. The two arguments of the equality can be of different types, in which case the result is `true`.

gt(*x*, *y*)

Equivalent to `x > y`

ge(*x*, *y*)

Equivalent to `x >= y`

lt(*x*, *y*)

Equivalent to `x < y`

le(*x*, *y*)

Equivalent to `x <= y`

Note: The four comparison operators can only compare values of the same value type. Integers and floats are of the same type `Number`.

---

max(*x*, ...)

Returns the maximum of the arguments. Can only be applied to numbers.

min(*x*, ...)

Returns the minimum of the arguments. Can only be applied to numbers.

7.2 Boolean functions

and(...)

Variadic conjunction. For binary arguments it is equivalent to `x && y`.

or(...)

Variadic disjunction. For binary arguments it is equivalent to `x || y`.

negate(*x*)

Negation. Equivalent to `!x`.

assert(*x*, ...)

Returns `true` if `x` is `true`, otherwise will raise an error containing all its arguments as the error message.

7.3 Mathematics

add(...)

Variadic addition. The binary version is the same as `x + y`.

sub(*x*, *y*)

Equivalent to `x - y`.

mul(...)

Variadic multiplication. The binary version is the same as `x * y`.

div(*x*, *y*)

Equivalent to `x / y`.

minus(*x*)

Equivalent to `-x`.

pow(*x*, *y*)

Raises `x` to the power of `y`. Equivalent to `x ^ y`. Always returns floating number.

mod(*x*, *y*)

Returns the remainder when `x` is divided by `y`. Arguments can be floats. The returned value has the same sign as `x`. Equivalent to `x % y`.

abs(*x*)

Returns the absolute value.

signum(*x*)

Returns `1`, `0` or `-1`, whichever has the same sign as the argument, e.g. `signum(to_float('NEG_INFINITY')) == -1`, `signum(0.0) == 0`, but `signum(-0.0) == -1`. Returns `NAN` when applied to `NAN`.

floor(*x*)

Returns the floor of `x`.

ceil(*x*)

Returns the ceiling of `x`.

round(*x*)

Returns the nearest integer to the argument (represented as Float if the argument itself is a Float). Round halfway cases away from zero. E.g. `round(0.5) == 1.0`, `round(-0.5) == -1.0`, `round(1.4) == 1.0`.

exp(*x*)

Returns the exponential of the argument, natural base.

exp2(*x*)

Returns the exponential base 2 of the argument. Always returns a float.

ln(*x*)

Returns the natual logarithm.

log2(*x*)

Returns the logarithm base 2.

log10(*x*)

Returns the logarithm base 10.

sin(*x*)

The sine trigonometric Func.

cos(*x*)

The cosine trigonometric Func.

tan(*x*)

The tangent trigonometric Func.

asin(*x*)

The inverse sine.

acos(*x*)

The inverse cosine.

atan(*x*)

The inverse tangent.

atan2(*x*, *y*)

The inverse tangent atan2[12] by passing *x* and *y* separately.

sinh(*x*)

The hyperbolic sine.

cosh(*x*)

The hyperbolic cosine.

tanh(*x*)

The hyperbolic tangent.

asinh(*x*)

The inverse hyperbolic sine.

acosh(*x*)

The inverse hyperbolic cosine.

atanh(*x*)

The inverse hyperbolic tangent.

deg_to_rad(*x*)

Converts degrees to radians.

[12] https://en.wikipedia.org/wiki/Atan2

`rad_to_deg`(*x*)

Converts radians to degrees.

`haversine`(*a_lat*, *a_lon*, *b_lat*, *b_lon*)

Computes with the haversine formula[13] the angle measured in radians between two points `a` and `b` on a sphere specified by their latitudes and longitudes. The inputs are in radians. You probably want the next function since most maps measure angles in radians.

`haversine_deg_input`(*a_lat*, *a_lon*, *b_lat*, *b_lon*)

Same as the previous function, but the inputs are in degrees instead of radians. The return value is still in radians. If you want the approximate distance measured on the surface of the earth instead of the angle between two points, multiply the result by the radius of the earth, which is about `6371` kilometres, `3959` miles, or `3440` nautical miles.

Warning: The haversine formula, when applied to the surface of the earth, which is not a perfect sphere, can result in an error of less than one percent.

7.4 String functions

`length`(*str*)

Returns the number of Unicode characters in the string.

Can also be applied to a list or a byte array.

Warning: `length(str)` does not return the number of bytes of the string representation. Also, what is returned depends on the normalization of the string. So if such details are important, apply `unicode_normalize` before `length`.

`concat`(*x*, ...)

Concatenates strings. Equivalent to `x ++ y` in the binary case.

Can also be applied to lists.

`str_includes`(*x*, *y*)

Returns `true` if `x` contains the substring `y`, `false` otherwise.

`lowercase`(*x*)

Convert to lowercase. Supports Unicode.

`uppercase`(*x*)

Converts to uppercase. Supports Unicode.

`trim`(*x*)

Removes whitespace[14] from both ends of the string.

`trim_start`(*x*)

Removes whitespace[15] from the start of the string.

[13] https://en.wikipedia.org/wiki/Haversine_formula
[14] https://en.wikipedia.org/wiki/Whitespace_character
[15] https://en.wikipedia.org/wiki/Whitespace_character

trim_end(*x*)

Removes whitespace[16] from the end of the string.

starts_with(*x*, *y*)

Tests if `x` starts with `y`.

Tip: `starts_with(var, str)` is prefered over equivalent (e.g. regex) conditions, since the compiler may more easily compile the clause into a range scan.

---

ends_with(*x*, *y*)

tests if `x` ends with `y`.

unicode_normalize(*str*, *norm*)

Converts `str` to the normalization[17] specified by `norm`. The valid values of `norm` are `'nfc'`, `'nfd'`, `'nfkc'` and `'nfkd'`.

chars(*str*)

Returns Unicode characters of the string as a list of substrings.

from_substrings(*list*)

Combines the strings in `list` into a big string. In a sense, it is the inverse function of `chars`.

Warning: If you want substring slices, indexing strings, etc., first convert the string to a list with `chars`, do the manipulation on the list, and then recombine with `from_substring`. Hopefully, the omission of functions doing such things directly can make people more aware of the complexities involved in manipulating strings (and getting the *correct* result).

7.5 List functions

list(*x*, ...)

Constructs a list from its argument, e.g. `list(1, 2, 3)`. Equivalent to the literal form `[1, 2, 3]`.

is_in(*el*, *list*)

Tests the membership of an element in a list.

first(*l*)

Extracts the first element of the list. Returns `null` if given an empty list.

last(*l*)

Extracts the last element of the list. Returns `null` if given an empty list.

get(*l*, *n*)

Returns the element at index `n` in the list `l`. This function will raise an error if the access is out of bounds. Indices start with 0.

maybe_get(*l*, *n*)

Returns the element at index `n` in the list `l`. This function will return `null` if the access is out of bounds. Indices start with 0.

[16] https://en.wikipedia.org/wiki/Whitespace_character
[17] https://en.wikipedia.org/wiki/Unicode_equivalence

length(*list*)

Returns the length of the list.

Can also be applied to a string or a byte array.

slice(*l*, *start*, *end*)

Returns the slice of list between the index `start` (inclusive) and `end` (exclusive). Negative numbers may be used, which is interpreted as counting from the end of the list. E.g. `slice([1, 2, 3, 4], 1, 3) == [2, 3]`, `slice([1, 2, 3, 4], 1, -1) == [2, 3]`.

concat(*x*, ...)

Concatenates lists. The binary case is equivalent to *x* ++ *y*.

Can also be applied to strings.

prepend(*l*, *x*)

Prepends `x` to `l`.

append(*l*, *x*)

Appends `x` to `l`.

reverse(*l*)

Reverses the list.

sorted(*l*)

Sorts the list and returns the sorted copy.

chunks(*l*, *n*)

Splits the list `l` into chunks of `n`, e.g. `chunks([1, 2, 3, 4, 5], 2) == [[1, 2], [3, 4], [5]]`.

chunks_exact(*l*, *n*)

Splits the list `l` into chunks of `n`, discarding any trailing elements, e.g. `chunks([1, 2, 3, 4, 5], 2) == [[1, 2], [3, 4]]`.

windows(*l*, *n*)

Splits the list `l` into overlapping windows of length `n`. e.g. `windows([1, 2, 3, 4, 5], 3) == [[1, 2, 3], [2, 3, 4], [3, 4, 5]]`.

union(*x*, *y*, ...)

Computes the set-theoretic union of all the list arguments.

intersection(*x*, *y*, ...)

Computes the set-theoretic intersection of all the list arguments.

difference(*x*, *y*, ...)

Computes the set-theoretic difference of the first argument with respect to the rest.

7.6 Binary functions

length(*bytes*)

Returns the length of the byte array.

Can also be applied to a list or a string.

bit_and(*x*, *y*)

Calculate the bitwise and. The two bytes must have the same lengths.

bit_or(*x*, *y*)

Calculate the bitwise or. The two bytes must have the same lengths.

bit_not(*x*)

Calculate the bitwise not.

bit_xor(*x*, *y*)

Calculate the bitwise xor. The two bytes must have the same lengths.

pack_bits([...])

packs a list of booleans into a byte array; if the list is not divisible by 8, it is padded with `false`.

unpack_bits(*x*)

Unpacks a byte array into a list of booleans.

encode_base64(*b*)

Encodes the byte array `b` into the Base64[18]-encoded string.

Note: `encode_base64` is automatically applied when output to JSON since JSON cannot represent bytes natively.

---

decode_base64(*str*)

Tries to decode the `str` as a Base64[19]-encoded byte array.

7.7 Type checking and conversions

to_string(*x*)

Convert `x` to a string: the argument is unchanged if it is already a string, otherwise its JSON string representation will be returned.

to_float(*x*)

Tries to convert `x` to a float. Conversion from numbers always succeeds. Conversion from strings has the following special cases in addition to the usual string representation:

- `INF` is converted to infinity;
- `NEG_INF` is converted to negative infinity;
- `NAN` is converted to NAN (but don't compare NAN by equality, use `is_nan` instead);
- `PI` is converted to pi (3.14159...);
- `E` is converted to the base of natural logarithms, or Euler's constant (2.71828...).

to_uuid(*x*)

Tries to convert `x` to a UUID. The input must either be a hyphenated UUID string representation or already a UUID for it to succeed.

uuid_timestamp(*x*)

Extracts the timestamp from a UUID version 1, as seconds since the UNIX epoch. If the UUID is not of version 1, `null` is returned. If `x` is not a UUID, an error is raised.

is_null(*x*)

Checks for `null`.

[18] https://en.wikipedia.org/wiki/Base64
[19] https://en.wikipedia.org/wiki/Base64

is_int(*x*)
: Checks for integers.

is_float(*x*)
: Checks for floats.

is_finite(*x*)
: Returns `true` if `x` is an integer or a finite float.

is_infinite(*x*)
: Returns `true` if `x` is infinity or negative infinity.

is_nan(*x*)
: Returns `true` if `x` is the special float `NAN`. Returns `false` when the argument is not of number type.

is_num(*x*)
: Checks for numbers.

is_bytes(*x*)
: Checks for bytes.

is_list(*x*)
: Checks for lists.

is_string(*x*)
: Checks for strings.

is_uuid(*x*)
: Checks for UUIDs.

7.8 Random functions

rand_float()
: Generates a float in the interval [0, 1], sampled uniformly.

rand_bernoulli(*p*)
: Generates a boolean with probability `p` of being `true`.

rand_int(*lower*, *upper*)
: Generates an integer within the given bounds, both bounds are inclusive.

rand_choose(*list*)
: Randomly chooses an element from `list` and returns it. If the list is empty, it returns `null`.

rand_uuid_v1()
: Generate a random UUID, version 1 (random bits plus timestamp).

rand_uuid_v4()
: Generate a random UUID, version 4 (completely random bits).

7.9 Regex functions

regex_matches(*x*, *reg*)

Tests if `x` matches the regular expression `reg`.

regex_replace(*x*, *reg*, *y*)

Replaces the first occurrence of the pattern `reg` in `x` with `y`.

regex_replace_all(*x*, *reg*, *y*)

Replaces all occurrences of the pattern `reg` in `x` with `y`.

regex_extract(*x*, *reg*)

Extracts all occurrences of the pattern `reg` in `x` and returns them in a list.

regex_extract_first(*x*, *reg*)

Extracts the first occurrence of the pattern `reg` in `x` and returns it. If none is found, returns `null`.

7.9.1 Regex syntax

Matching one character:

```
.             any character except new line
\d            digit (\p{Nd})
\D            not digit
\pN           One-letter name Unicode character class
\p{Greek}     Unicode character class (general category or script)
\PN           Negated one-letter name Unicode character class
\P{Greek}     negated Unicode character class (general category or script)
```

Character classes:

```
[xyz]         A character class matching either x, y or z (union).
[^xyz]        A character class matching any character except x, y and z.
[a-z]         A character class matching any character in range a-z.
[[:alpha:]]   ASCII character class ([A-Za-z])
[[:^alpha:]]  Negated ASCII character class ([^A-Za-z])
[x[^xyz]]     Nested/grouping character class (matching any character except y and z)
[a-y&&xyz]    Intersection (matching x or y)
[0-9&&[^4]]   Subtraction using intersection and negation (matching 0-9 except 4)
[0-9--4]      Direct subtraction (matching 0-9 except 4)
[a-g~~b-h]    Symmetric difference (matching `a` and `h` only)
[\[\]]        Escaping in character classes (matching [ or ])
```

Composites:

```
xy    concatenation (x followed by y)
x|y   alternation (x or y, prefer x)
```

Repetitions:

```
x*        zero or more of x (greedy)
x+        one or more of x (greedy)
x?        zero or one of x (greedy)
```

(continues on next page)

(continued from previous page)

```
x*?       zero or more of x (ungreedy/lazy)
x+?       one or more of x (ungreedy/lazy)
x??       zero or one of x (ungreedy/lazy)
x{n,m}    at least n x and at most m x (greedy)
x{n,}     at least n x (greedy)
x{n}      exactly n x
x{n,m}?   at least n x and at most m x (ungreedy/lazy)
x{n,}?    at least n x (ungreedy/lazy)
x{n}?     exactly n x
```

Empty matches:

```
^     the beginning of the text
$     the end of the text
\A    only the beginning of the text
\z    only the end of the text
\b    a Unicode word boundary (\w on one side and \W, \A, or \z on the other)
\B    not a Unicode word boundary
```

7.10 Timestamp functions

now()

Returns the current timestamp as seconds since the UNIX epoch.

format_timestamp(*ts*, *tz?*)

Interpret `ts` as seconds since the epoch and format as a string according to RFC3339[20].

If a second string argument is provided, it is interpreted as a timezone[21] and used to format the timestamp.

parse_timestamp(*str*)

Parse `str` into seconds since the epoch according to RFC3339.

[20] https://www.rfc-editor.org/rfc/rfc3339
[21] https://en.wikipedia.org/wiki/Tz_database

CHAPTER

EIGHT

AGGREGATIONS

Aggregations in Cozo can be thought of as a function that acts on a string of values and produces a single value (the aggregate). Due to Datalog semantics, the stream is never empty.

There are two kinds of aggregations in Cozo, *ordinary aggregations* and *meet aggregations*. They are implemented differently in Cozo, with meet aggregations generally faster and more powerful (e.g. only meet aggregations can be recursive).

The power of meet aggregations derive from the additional properties they satisfy by forming a semilattice[22]:

idempotency
: the aggregate of a single value `a` is `a` itself,

commutativity
: the aggregate of `a` then `b` is equal to the aggregate of `b` then `a`,

associativity
: it is immaterial where we put the parentheses in an aggregate application.

Meet aggregations can be used as ordinary ones, but the reverse is impossible.

8.1 Meet aggregations

`min`(*x*)
: Aggregate the minimum value of all `x`.

`max`(*x*)
: Aggregate the maximum value of all `x`.

`and`(*var*)
: Aggregate the logical conjunction of the variable passed in.

`or`(*var*)
: Aggregate the logical disjunction of the variable passed in.

`union`(*var*)
: Aggregate the unions of `var`, which must be a list.

`intersection`(*var*)
: Aggregate the intersections of `var`, which must be a list.

[22] https://en.wikipedia.org/wiki/Semilattice

choice(*var*)

Non-deterministically chooses one of the values of `var` as the aggregate. It simply chooses the first value it meets (the order that it meets values should be considered non-deterministic).

choice_last(*var*)

Non-deterministically chooses one of the values of `var` as the aggregate. It simply chooses the last value it meets.

min_cost([*data*, *cost*])

The argument should be a list of two elements and this aggregation chooses the list of the minimum `cost`.

shortest(*var*)

`var` must be a list. Returns the shortest list among all values. Ties will be broken non-deterministically.

coalesce(*var*)

Returns the first non-null value it meets. The order is non-deterministic.

bit_and(*var*)

`var` must be bytes. Returns the bitwise 'and' of the values.

bit_or(*var*)

`var` must be bytes. Returns the bitwise 'or' of the values.

8.2 Ordinary aggregations

count(*var*)

Count how many values are generated for `var` (using bag instead of set semantics).

count_unique(*var*)

Count how many unique values there are for `var`.

collect(*var*)

Collect all values for `var` into a list.

unique(*var*)

Collect `var` into a list, keeping each unique value only once.

group_count(*var*)

Count the occurrence of unique values of `var`, putting the result into a list of lists, e.g. when applied to `'a'`, `'b'`, `'c'`, `'c'`, `'a'`, `'c'`, the results is `[['a', 2], ['b', 1], ['c', 3]]`.

bit_xor(*var*)

`var` must be bytes. Returns the bitwise 'xor' of the values.

latest_by([*data*, *time*])

The argument should be a list of two elements and this aggregation returns the `data` of the maximum `cost`. This is very similar to `min_cost`, the differences being that maximum instead of minimum is used, only the data itself is returned, and the aggregation is deliberately note a meet aggregation. Intended to be used in timestamped audit trails.

choice_rand(*var*)

Non-deterministically chooses one of the values of `var` as the aggregate. Each value the aggregation encounters has the same probability of being chosen. This version of `choice` is not a meet aggregation since it is impossible to satisfy the uniform sampling requirement while maintaining no state, which is an implementation restriction unlikely to be lifted.

8.2.1 Statistical aggregations

mean(*x*)

The mean value of `x`.

sum(*x*)

The sum of `x`.

product(*x*)

The product of `x`.

variance(*x*)

The sample variance of `x`.

std_dev(*x*)

The sample standard deviation of `x`.

CHAPTER

NINE

UTILITIES AND ALGORITHMS

Fixed rules in CozoScript apply utilities and/or algorithms. The purpose of the native, built-in utilities and algorithms in Cozo is to enable easy computation of results that would require either queries too awkward to express in pure Datalog, or time or space requirements that are unreasonable if implemented in the interpreted queries framework.

9.1 Utilities

Constant(*data: [...]*)

Returns a constant relation containing the data passed in. The constant rule `?[] <- ...` is actually syntax sugar for `?[] <~ Constant(data: ...)`.

Parameters

data – A list of lists, representing the rows of the returned relation.

ReorderSort(*rel[...], out: [...], sort_by: [...], descending: false, break_ties: false, skip: 0, take: 0*)

Sort and then extract new columns of the passed in relation `rel`.

Parameters

- **out** (*required*) – A list of expressions which will be used to produce the output relation. Any bindings in the expressions will be bound to the named positions in `rel`.
- **sort_by** – A list of expressions which will be used to produce the sort keys. Any bindings in the expressions will be bound to the named positions in `rel`.
- **descending** – Whether the sorting process should be done in descending order. Defaults to `false`.
- **break_ties** – Whether ties should be broken, e.g. whether the first two rows with *identical sort keys* should be given ordering numbers `1` and `2` instead of `1` and `1`. Defaults to false.
- **skip** – How many rows to skip before producing rows. Defaults to zero.
- **take** – How many rows at most to produce. Zero means no limit. Defaults to zero.

Returns

The returned relation, in addition to the rows specified in the parameter `out`, will have the ordering prepended. The ordering starts at `1`.

Tip: This algorithm serves a similar purpose to the global `:order`, `:limit` and `:offset` options, but can be applied to intermediate results. Prefer the global options if it is applied to the final output.

`CsvReader`(*url: ..., types: [...], delimiter: ',', prepend_index: false, has_headers: true*)

Read a CSV file from disk or an HTTP GET request and convert the result to a relation.

Parameters

- **`url`** (*required*) – URL for the CSV file. For local file, use `file://<PATH_TO_FILE>`.
- **`types`** (*required*) – A list of strings interpreted as types for the columns of the output relation. If any type is specified as nullable and conversion to the specified type fails, `null` will be the result. This is more lenient than other functions since CSVs tend to contain lots of bad values.
- **`delimiter`** – The delimiter to use when parsing the CSV file.
- **`prepend_index`** – If `true`, row index will be prepended to the columns.
- **`has_headers`** – Whether the CSV file has headers. The reader will not interpret the header in any way but will instead simply ignore it.

`JsonReader`(*url: ..., fields: [...], json_lines: true, null_if_absent: false, prepend_index: false*)

Read a JSON file for disk or an HTTP GET request and convert the result to a relation.

Parameters

- **`url`** (*required*) – URL for the JSON file. For local file, use `file://<PATH_TO_FILE>`.
- **`fields`** (*required*) – A list of field names, for extracting fields from JSON arrays into the relation.
- **`json_lines`** – If `true`, parse the file as lines of JSON objects, each line containing a single object; if false, parse the file as a JSON array containing many objects.
- **`null_if_absent`** – If a `true` and a requested field is absent, will output `null` in its place. If `false` and the requested field is absent, will throw an error.
- **`prepend_index`** – If `true`, row index will be prepended to the columns.

9.2 Connectedness algorithms

`ConnectedComponents`(*edges[from, to]*)

Computes the connected components[23] of a graph with the provided edges.

Returns

Pairs containing the node index, and its component index.

`StronglyConnectedComponent`(*edges[from, to]*)

Computes the strongly connected components[24] of a graph with the provided edges.

Returns

Pairs containing the node index, and its component index.

`SCC`(...)

See *`Algo.StronglyConnectedComponent`*.

`MinimumSpanningForestKruskal`(*edges[from, to, weight?]*)

Runs Kruskal's algorithm[25] on the provided edges to compute a minimum spanning forest[26]. Negative weights are fine.

[23] https://en.wikipedia.org/wiki/Connected_component_(graph_theory)
[24] https://en.wikipedia.org/wiki/Strongly_connected_component

Returns

Triples containing the from-node, the to-node, and the cost from the tree root to the to-node. Which nodes are chosen to be the roots are non-deterministic. Multiple roots imply the graph is disconnected.

`MinimumSpanningTreePrim`(*edges[from, to, weight?], starting?[idx]*)

Runs Prim's algorithm[27] on the provided edges to compute a minimum spanning tree[28]. `starting` should be a relation producing exactly one node index as the starting node. Only the connected component of the starting node is returned. If `starting` is omitted, which component is returned is arbitrary.

Returns

Triples containing the from-node, the to-node, and the cost from the tree root to the to-node.

`TopSort`(*edges[from, to]*)

Performs topological sorting[29] on the graph with the provided edges. The graph is required to be connected in the first place.

Returns

Pairs containing the sort order and the node index.

9.3 Pathfinding algorithms

`ShortestPathDijkstra`(*edges[from, to, weight?], starting[idx], goals[idx], undirected: false, keep_ties: false*)

Runs Dijkstra's algorithm[30] to determine the shortest paths between the `starting` nodes and the `goals`. Weights, if given, must be non-negative.

Parameters

- **`undirected`** – Whether the graph should be interpreted as undirected. Defaults to `false`.
- **`keep_ties`** – Whether to return all paths with the same lowest cost. Defaults to `false`, in which any one path of the lowest cost could be returned.

Returns

4-tuples containing the starting node, the goal, the lowest cost, and a path with the lowest cost.

`KShortestPathYen`(*edges[from, to, weight?], starting[idx], goals[idx], k: expr, undirected: false*)

Runs Yen's algorithm[31] (backed by Dijkstra's algorithm) to find the k-shortest paths between nodes in `starting` and nodes in `goals`.

Parameters

- **`k`** (*`required`*) – How many routes to return for each start-goal pair.
- **`undirected`** – Whether the graph should be interpreted as undirected. Defaults to `false`.

Returns

4-tuples containing the starting node, the goal, the cost, and a path with the cost.

[25] https://en.wikipedia.org/wiki/Kruskal%27s_algorithm
[26] https://en.wikipedia.org/wiki/Minimum_spanning_tree
[27] https://en.wikipedia.org/wiki/Prim%27s_algorithm
[28] https://en.wikipedia.org/wiki/Minimum_spanning_tree
[29] https://en.wikipedia.org/wiki/Topological_sorting
[30] https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
[31] https://en.wikipedia.org/wiki/Yen%27s_algorithm

BreadthFirstSearch(*edges[from, to], nodes[idx, ...], starting?[idx], condition: expr, limit: 1*)

Runs breadth first search on the directed graph with the given edges and nodes, starting at the nodes in `starting`. If `starting` is not given, it will default to all of `nodes`, which may be quite a lot to calculate.

Parameters

- **condition** (*required*) – The stopping condition, will be evaluated with the bindings given to `nodes`. Should evaluate to a boolean, with `true` indicating an acceptable answer was found.
- **limit** – How many answers to produce for each starting nodes. Defaults to 1.

Returns

Triples containing the starting node, the answer node, and the found path connecting them.

BFS(...)

See *Algo.BreadthFirstSearch*.

DepthFirstSearch(*edges[from, to], nodes[idx, ...], starting?[idx], condition: expr, limit: 1*)

Runs depth first search on the directed graph with the given edges and nodes, starting at the nodes in `starting`. If `starting` is not given, it will default to all of `nodes`, which may be quite a lot to calculate.

Parameters

- **condition** (*required*) – The stopping condition, will be evaluated with the bindings given to `nodes`. Should evaluate to a boolean, with `true` indicating an acceptable answer was found.
- **limit** – How many answers to produce for each starting nodes. Defaults to 1.

Returns

Triples containing the starting node, the answer node, and the found path connecting them.

Tip: You probably don't want to use depth first search for path finding unless you have a really niche use case.

DFS(...)

See *Algo.DepthFirstSearch*.

ShortestPathAStar(*edges[from, to, weight], nodes[idx, ...], starting[idx], goals[idx], heuristic: expr*)

Computes the shortest path from every node in `starting` to every node in `goals` by the A* algorithm[32].

`edges` are interpreted as directed, weighted edges with non-negative weights.

Parameters

heuristic (*required*) – The search heuristic expression. It will be evaluated with the bindings from `goals` and `nodes`. It should return a number which is a lower bound of the true shortest distance from a node to the goal node. If the estimate is not a valid lower-bound, i.e. it over-estimates, the results returned may not be correct.

Returns

4-tuples containing the starting node index, the goal node index, the lowest cost, and a path with the lowest cost.

Tip: The performance of A* star algorithm heavily depends on how good your heuristic function is. Passing in `0` as the estimate is always valid, but then you really should be using Dijkstra's algorithm.

[32] https://en.wikipedia.org/wiki/A*_search_algorithm

Good heuristics usually come about from a metric in the ambient space in which your data live, e.g. spherical distance on the surface of a sphere, or Manhattan distance on a grid. `Func.Math.haversine_deg_input` could be helpful for the spherical case. Note that you must use the correct units for the distance.

Providing a heuristic that is not guaranteed to be a lower-bound *might* be acceptable if you are fine with inaccuracies. The errors in the answers are bound by the sum of the margins of your over-estimates.

9.4 Community detection algorithms

`ClusteringCoefficients`(*edges[from, to, weight?]*)

Computes the clustering coefficients[33] of the graph with the provided edges.

Returns

4-tuples containing the node index, the clustering coefficient, the number of triangles attached to the node, and the total degree of the node.

`CommunityDetectionLouvain`(*edges[from, to, weight?], undirected: false, max_iter: 10, delta: 0.0001, keep_depth?: depth*)

Runs the Louvain algorithm[34] on the graph with the provided edges, optionally non-negatively weighted.

Parameters

- **`undirected`** – Whether the graph should be interpreted as undirected. Defaults to `false`.
- **`max_iter`** – The maximum number of iterations to run within each epoch of the algorithm. Defaults to 10.
- **`delta`** – How much the modularity[35] has to change before a step in the algorithm is considered to be an improvement.
- **`keep_depth`** – How many levels in the hierarchy of communities to keep in the final result. If omitted, all levels are kept.

Returns

Pairs containing the label for a community, and a node index belonging to the community. Each label is a list of integers with maximum length constrained by the parameter `keep_depth`. This list represents the hierarchy of sub-communities containing the list.

`LabelPropagation`(*edges[from, to, weight?], undirected: false, max_iter: 10*)

Runs the label propagation algorithm[36] on the graph with the provided edges, optionally weighted.

Parameters

- **`undirected`** – Whether the graph should be interpreted as undirected. Defaults to `false`.
- **`max_iter`** – The maximum number of iterations to run. Defaults to 10.

Returns

Pairs containing the integer label for a community, and a node index belonging to the community.

[33] https://en.wikipedia.org/wiki/Clustering_coefficient
[34] https://en.wikipedia.org/wiki/Louvain_method
[35] https://en.wikipedia.org/wiki/Modularity_(networks)
[36] https://en.wikipedia.org/wiki/Label_propagation_algorithm

9.5 Centrality measures

`DegreeCentrality`(*edges[from, to]*)

Computes the degree centrality of the nodes in the graph with the given edges. The computation is trivial, so this should be your first thing to try when exploring new data.

Returns

4-tuples containing the node index, the total degree (how many edges involve this node), the out-degree (how many edges point away from this node), and the in-degree (how many edges point to this node).

`PageRank`(*edges[from, to, weight?], undirected: false, theta: 0.8, epsilon: 0.05, iterations: 20*)

Computes the PageRank[37] from the given graph with the provided edges, optionally weighted.

Parameters

- **`undirected`** – Whether the graph should be interpreted as undirected. Defaults to `false`.
- **`theta`** – A number between 0 and 1 indicating how much weight in the PageRank matrix is due to the explicit edges. A number of 1 indicates no random restarts. Defaults to 0.8.
- **`epsilon`** – Minimum PageRank change in any node for an iteration to be considered an improvement. Defaults to 0.05.
- **`iterations`** – How many iterations to run. Fewer iterations are run if convergence is reached. Defaults to 20.

Returns

Pairs containing the node label and its PageRank. For a graph with uniform edges, the PageRank of every node is 1. The L2-norm[38] of the results is forced to be invariant, i.e. in the results those nodes with a PageRank greater than 1 is "more central" than the average node in a certain sense.

`ClosenessCentrality`(*edges[from, to, weight?], undirected: false*)

Computes the closeness centrality[39] of the graph. The input relation represent edges connecting node indices which are optionally weighted.

Parameters

`undirected` – Whether the edges should be interpreted as undirected. Defaults to `false`.

Returns

Node index together with its centrality.

`BetweennessCentrality`(*edges[from, to, weight?], undirected: false*)

Computes the betweenness centrality[40] of the graph. The input relation represent edges connecting node indices which are optionally weighted.

Parameters

`undirected` – Whether the edges should be interpreted as undirected. Defaults to `false`.

Returns

Node index together with its centrality.

Warning: `BetweennessCentrality` is very expensive to compute for medium to large graphs. Plan resources accordingly.

[37] https://en.wikipedia.org/wiki/PageRank
[38] https://en.wikipedia.org/wiki/Norm_(mathematics)
[39] https://en.wikipedia.org/wiki/Closeness_centrality
[40] https://en.wikipedia.org/wiki/Betweenness_centrality

9.6 Miscellaneous

RandomWalk(*edges[from, to, ...], nodes[idx, ...], starting[idx], steps: 10, weight?: expr, iterations: 1*)

Performs random walk on the graph with the provided edges and nodes, starting at the nodes in `starting`.

Parameters

- **steps** (*required*) – How many steps to walk for each node in `starting`. Produced paths may be shorter if dead ends are reached.
- **weight** – An expression evaluated against bindings of `nodes` and bindings of `edges`, at a time when the walk is at a node and choosing between multiple edges to follow. It should evaluate to a non-negative number indicating the weight of the given choice of edge to follow. If omitted, which edge to follow is chosen uniformly.
- **iterations** – How many times walking is repeated for each starting node.

Returns

Triples containing a numerical index for the walk, the starting node, and the path followed.

INDEX

A

abs() (*in module Func.Math*), 24
acos() (*in module Func.Math*), 25
acosh() (*in module Func.Math*), 25
add() (*in module Func.Math*), 24
and() (*in module Aggr.Meet*), 33
and() (*in module Func.Bool*), 24
append() (*in module Func.List*), 28
asin() (*in module Func.Math*), 25
asinh() (*in module Func.Math*), 25
assert() (*in module Func.Bool*), 24
atan() (*in module Func.Math*), 25
atan2() (*in module Func.Math*), 25
atanh() (*in module Func.Math*), 25

B

BetweennessCentrality() (*in module Algo*), 42
BFS() (*in module Algo*), 40
bit_and() (*in module Aggr.Meet*), 34
bit_and() (*in module Func.Bin*), 28
bit_not() (*in module Func.Bin*), 29
bit_or() (*in module Aggr.Meet*), 34
bit_or() (*in module Func.Bin*), 28
bit_xor() (*in module Aggr.Ord*), 34
bit_xor() (*in module Func.Bin*), 29
BreadthFirstSearch() (*in module Algo*), 39

C

ceil() (*in module Func.Math*), 24
chars() (*in module Func.String*), 27
choice() (*in module Aggr.Meet*), 33
choice_last() (*in module Aggr.Meet*), 34
choice_rand() (*in module Aggr.Ord*), 34
chunks() (*in module Func.List*), 28
chunks_exact() (*in module Func.List*), 28
ClosenessCentrality() (*in module Algo*), 42
ClusteringCoefficients() (*in module Algo*), 41
coalesce() (*in module Aggr.Meet*), 34
collect() (*in module Aggr.Ord*), 34
CommunityDetectionLouvain() (*in module Algo*), 41
concat() (*in module Func.List*), 28
concat() (*in module Func.String*), 26
ConnectedComponents() (*in module Algo*), 38
Constant() (*in module Algo*), 37
cos() (*in module Func.Math*), 25
cosh() (*in module Func.Math*), 25
count() (*in module Aggr.Ord*), 34
count_unique() (*in module Aggr.Ord*), 34
CsvReader() (*in module Algo*), 37

D

decode_base64() (*in module Func.Bin*), 29
deg_to_rad() (*in module Func.Math*), 25
DegreeCentrality() (*in module Algo*), 42
DepthFirstSearch() (*in module Algo*), 40
DFS() (*in module Algo*), 40
difference() (*in module Func.List*), 28
div() (*in module Func.Math*), 24

E

encode_base64() (*in module Func.Bin*), 29
ends_with() (*in module Func.String*), 27
eq() (*in module Func.EqCmp*), 23
exp() (*in module Func.Math*), 24
exp2() (*in module Func.Math*), 25

F

first() (*in module Func.List*), 27
floor() (*in module Func.Math*), 24
format_timestamp() (*in module Func.Regex*), 32
from_substrings() (*in module Func.String*), 27

G

ge() (*in module Func.EqCmp*), 23
get() (*in module Func.List*), 27
group_count() (*in module Aggr.Ord*), 34
gt() (*in module Func.EqCmp*), 23

H

haversine() (*in module Func.Math*), 26
haversine_deg_input() (*in module Func.Math*), 26

I

intersection() (*in module Aggr.Meet*), 33

intersection() (*in module Func.List*), 28
is_bytes() (*in module Func.Typing*), 30
is_finite() (*in module Func.Typing*), 30
is_float() (*in module Func.Typing*), 30
is_in() (*in module Func.List*), 27
is_infinite() (*in module Func.Typing*), 30
is_int() (*in module Func.Typing*), 29
is_list() (*in module Func.Typing*), 30
is_nan() (*in module Func.Typing*), 30
is_null() (*in module Func.Typing*), 29
is_num() (*in module Func.Typing*), 30
is_string() (*in module Func.Typing*), 30
is_uuid() (*in module Func.Typing*), 30

J

JsonReader() (*in module Algo*), 38

K

KShortestPathYen() (*in module Algo*), 39

L

LabelPropagation() (*in module Algo*), 41
last() (*in module Func.List*), 27
latest_by() (*in module Aggr.Ord*), 34
le() (*in module Func.EqCmp*), 23
length() (*in module Func.Bin*), 28
length() (*in module Func.List*), 27
length() (*in module Func.String*), 26
list() (*in module Func.List*), 27
ln() (*in module Func.Math*), 25
log10() (*in module Func.Math*), 25
log2() (*in module Func.Math*), 25
lowercase() (*in module Func.String*), 26
lt() (*in module Func.EqCmp*), 23

M

max() (*in module Aggr.Meet*), 33
max() (*in module Func.EqCmp*), 23
maybe_get() (*in module Func.List*), 27
mean() (*in module Aggr.Ord*), 35
min() (*in module Aggr.Meet*), 33
min() (*in module Func.EqCmp*), 23
min_cost() (*in module Aggr.Meet*), 34
MinimumSpanningForestKruskal() (*in module Algo*), 38
MinimumSpanningTreePrim() (*in module Algo*), 39
minus() (*in module Func.Math*), 24
mod() (*in module Func.Math*), 24
mul() (*in module Func.Math*), 24

N

negate() (*in module Func.Bool*), 24
neq() (*in module Func.EqCmp*), 23
now() (*in module Func.Regex*), 32

O

or() (*in module Aggr.Meet*), 33
or() (*in module Func.Bool*), 24

P

pack_bits() (*in module Func.Bin*), 29
PageRank() (*in module Algo*), 42
parse_timestamp() (*in module Func.Regex*), 32
pow() (*in module Func.Math*), 24
prepend() (*in module Func.List*), 28
product() (*in module Aggr.Ord*), 35

R

rad_to_deg() (*in module Func.Math*), 25
rand_bernoulli() (*in module Func.Rand*), 30
rand_choose() (*in module Func.Rand*), 30
rand_float() (*in module Func.Rand*), 30
rand_int() (*in module Func.Rand*), 30
rand_uuid_v1() (*in module Func.Rand*), 30
rand_uuid_v4() (*in module Func.Rand*), 30
RandomWalk() (*in module Algo*), 43
regex_extract() (*in module Func.Regex*), 31
regex_extract_first() (*in module Func.Regex*), 31
regex_matches() (*in module Func.Regex*), 31
regex_replace() (*in module Func.Regex*), 31
regex_replace_all() (*in module Func.Regex*), 31
ReorderSort() (*in module Algo*), 37
reverse() (*in module Func.List*), 28
round() (*in module Func.Math*), 24

S

SCC() (*in module Algo*), 38
shortest() (*in module Aggr.Meet*), 34
ShortestPathAStar() (*in module Algo*), 40
ShortestPathDijkstra() (*in module Algo*), 39
signum() (*in module Func.Math*), 24
sin() (*in module Func.Math*), 25
sinh() (*in module Func.Math*), 25
slice() (*in module Func.List*), 28
sorted() (*in module Func.List*), 28
starts_with() (*in module Func.String*), 27
std_dev() (*in module Aggr.Ord*), 35
str_includes() (*in module Func.String*), 26
StronglyConnectedComponent() (*in module Algo*), 38
sub() (*in module Func.Math*), 24
sum() (*in module Aggr.Ord*), 35

T

tan() (*in module Func.Math*), 25
tanh() (*in module Func.Math*), 25
to_float() (*in module Func.Typing*), 29

to_string() (*in module Func.Typing*), 29
to_uuid() (*in module Func.Typing*), 29
TopSort() (*in module Algo*), 39
trim() (*in module Func.String*), 26
trim_end() (*in module Func.String*), 26
trim_start() (*in module Func.String*), 26

U

unicode_normalize() (*in module Func.String*), 27
union() (*in module Aggr.Meet*), 33
union() (*in module Func.List*), 28
unique() (*in module Aggr.Ord*), 34
unpack_bits() (*in module Func.Bin*), 29
uppercase() (*in module Func.String*), 26
uuid_timestamp() (*in module Func.Typing*), 29

V

variance() (*in module Aggr.Ord*), 35

W

windows() (*in module Func.List*), 28