The CozoScript Manual

Release 0.3.0

Ziyang Hu

Dec 20, 2022

CONTENTS

CHAPTER

ONE

WELCOME

Welcome to the CozoScript Manual. The latest version of this manual can be read at https://cozodb.github.io/current/manual. Alternatively, you can download a PDF version for offline viewing at https://cozodb.github.io/current/manual.pdf.

This manual touches upon all features currently accessible in the Cozo database via CozoScript, though the coverage of some topics may be sketchy at this stage. In addition, the last chapter describes some functionalities beyond CozoScript.

This manual assumes that you already know the basics of the Cozo database, at the level of the Tutorial[1].

For installation instructions, refer to the project homepage[2].

[1] https://github.com/cozodb/cozo-docs/blob/main/tutorial/tutorial.ipynb
[2] https://github.com/cozodb/cozo

CHAPTER

TWO

QUERIES

CozoScript, a Datalog[3] dialect, is the query language of the Cozo database.

A CozoScript query consists of one or many named rules. Each named rule represents a *relation*, i.e. collection of data divided into rows and columns. The rule named `?` is the *entry* to the query, and the relation it represents is the result of the query. Each named rule has a rule head, which corresponds to the columns of the relation, and a rule body, which specifies the content of the relation, or how the content should be computed.

Relations in Cozo (stored or otherwise) abide by the *set semantics*. Thus even if a rule computes a row multiple times, the resulting relation only contains a single copy.

There are two types of named rules in CozoScript:

- *Inline rules*, distinguished by using `:=` to connect the head and the body. The logic used to compute the resulting relation is defined *inline*.
- *Fixed rules*, distinguished by using `<~` to connect the head and the body. The logic used to compute the resulting relation is *fixed* according to which algorithm or utility is requested.

The *constant rules* which use `<-` to connect the head and the body are syntax sugar. For example:

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

is identical to:

```
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.

[3] 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 (because this is its first appearance), the unification will always succeed. For subsequent atoms, the variable becomes *bound*: it take on the value of whatever it was unified with in the named relation. When a bound variable is unified again, for example `b` in `rule_b[b, d, a, e]`, this unification will only succeed when the unified value is the same as the current value. Thus, repeated use of the same variable in named rules corresponds to inner joins in relational algebra.

Atoms representing applications of *stored relations* are written as:

```
*stored_relation[bind1, bind2]
```

with the asterisk before the name. Written in this way using square brackets, as many bindings as the arity of the stored relation must be given.

You can also bind columns by name:

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

In this form, any number of columns may be omitted. If the name you want to give the binding is the same as the name of the column, you can write instead `*stored_relation{col1}`, which is the same as `*stored_relation{col1: col1}`.

Expressions are also atoms, such as:

```
a > b + 1
```

`a` and `b` must be bound somewhere else in the rule. Expression atoms must evaluate to booleans, and act as *filters*. Only rows where the expression atom evaluates to `true` are kept.

Unification atoms unify explicitly:

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

Whatever appears on the left-hand side must be a single variable and is unified with the result of the right-hand side.

Note: This is different from the equality operator ==, where the left-hand side is a completely bound expression. When the left-hand side is a single *bound* variable, the equality and the unification operators are equivalent.

Unification atoms can also unify with multiple values in a list:

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

If the right-hand side does not evaluate to a list, an error is raised.

2.1.2 Head

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

Each variable in the head must be bound in the body (the *safety rule*). 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 formed by the *disjunction* of the multiple definitions (a *union* of rows).

You may also use the explicit disjunction operator `or` in a single rule definition:

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

There is also an `and` operator, semantically identical to the comma `,` but has higher operator precedence than `or` (the comma has the lowest precedence).

2.1.4 Negation

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

```
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 (another *safety rule*). The unbound bindings in negated rules remain unbound: negation cannot introduce new bindings to be used in the head.

Negated expressions act as negative filters, which is semantically equivalent to putting `!` in front of the expression. Explict unification cannot be negated unless the left-hand side is bound, in which case it is treated as an expression atom and then negated.

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 including itself.

Recursion cannot occur in negated positions (*safety rule*): `r[a] := not r[a]` is not allowed.

Warning: As CozoScript allows explicit unification, queries that produce infinite relations may be accepted by the compiler. One of the simplest examples is:

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

It is not even in principle possible for Cozo to rule out all infinite queries without wrongly rejecting valid ones. If you accidentally submitted one, refer to the system ops chapter for how to terminate queries. Alternatively, you can give a timeout for the query when you submit.

2.1.6 Aggregation

In CozoScript, aggregations are specified for inline rules by applying *aggregation operators* to variables in the rule head:

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

here we have used the familiar `count` operator. Any variables in the head without aggregation operators are treated as *grouping variables*, and aggregation is applied using them as keys. If you do not specify any grouping variables, then the resulting relation contains at most one row.

Aggregation operators are applied to the rows computed by the body of the rule using bag 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 is empty.

If a rule has several definitions, they must have identical aggregations applied in the same positions.

Cozo allows aggregations for self-recursion for a limited subset of aggregation operators, the so-called *semi-lattice aggregations*:

```
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]
```

Here self-recursion of `shortest_distance` contains the `min` aggregation.

For a rule-head to be considered semi-lattice-aggregate, the aggregations must come at the end of the rule head. In the above example, if you write the head as `shortest_distance[min(distance), destination]`, the query engine will complain about unsafe recursion through aggregation, since written this way `min` is considered an ordinary aggregation.

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. For example:

```
?[] <~ 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 for their input relations. 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 algorithm-specific ways, for example in the DFS algorithm bindings.

In the example 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. Thus, the bindings in the rule head can be omitted, but if they are provided, you must abide by the arity.

2.3 Query options

Each query can have options associated with it:

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

:limit 10
:offset 20
```

In the example, `:limit` and `:offset` are query options with familiar meanings. All query options start with a single colon `:`. Queries options can appear before or after rules, or even sandwiched between rules.

Several query options deal with transactions for the database. Those will be discussed in the chapter on stored relations and transactions. The rest of the query options are explained in the following.

`: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. Useful for deliberately interleaving concurrent queries to test complex logic.

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

Sort the output relation. If `:limit` or `:offset` are specified, they are applied after `:sort`. 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 order, e.g. `:sort -count(employee), dept_name` sorts by employee count in reverse order first, then break ties with department name in ascending alphabetical order.

> **Warning:** Aggregations must be done in inline rules, not in output sorting. In the above example, the entry rule head must contain `count(employee)`, `employee` alone is not acceptable.

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

Alias for `:sort`.

`:assert none`

The query returns nothing if the output relation is empty, otherwise execution aborts with an error. Useful for transactions and triggers.

`:assert some`

The query returns nothing if the output relation contains at least one row, otherwise, execution aborts with an error. Useful for transactions and triggers. You should consider adding `:limit 1` to the query to ensure early termination if you do not need to check all return tuples.

CHAPTER

THREE

STORED RELATIONS AND TRANSACTIONS

In Cozo, data are stored in *stored relations* on disk.

3.1 Stored relations

To query stored relations, use the `*relation[...]` or `*relation{...}` atoms in inline or fixed rules, as explained in the last chapter. To manipulate stored relations, use one of the following query options:

`:create <NAME> <SPEC>`

Create a stored relation with the given name and spec. No stored relation with the same name can exist beforehand. If a query is specified, data from the resulting relation is put into the newly created stored relation. This is the only stored relation-related query option in which a query may be omitted.

`:replace <NAME> <SPEC>`

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`. If there are any triggers associated, they will be preserved. Note that this may lead to errors if `:replace` leads to schema change.

`:put <NAME> <SPEC>`

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

`:ensure <NAME> <SPEC>`

Ensure that rows specified by the output relation and spec exist in the database, and that no other process has written to these rows when the enclosing transaction commits. Useful for ensuring read-write consistency.

`:rm <NAME> <SPEC>`

Remove rows from the named stored relation. Only keys should be specified in <SPEC>. Removing a non-existent key is not an error and does nothing.

`:ensure_not <NAME> <SPEC>`

Ensure 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 when the enclosing transaction commits. 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, but the semantics is a bit different. We first describe the format and semantics for `:create` and `:replace`.

A spec, or a specification for columns, is 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. The order of columns matters. Rows are stored in lexicographically sorted order in trees according to their keys.

In the above example, we explicitly specified the types for all columns. In case of type mismatch, the system will first try to coerce the values given, and if that fails, the query is aborted with an error. You can omit types for columns, in which case their types default to `Any?`, i.e. all values are acceptable. For example, the above query with all types omitted is:

```
?[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: String = b
}
```

You *must* use explicit correspondence if the entry head contains aggregation, since names such as `count(c)` are not valid column names. The `address` field above shows how to specify both a type and a correspondence.

Instead of specifying bindings, you can specify an expression that generates default values by using `default`:

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

:create dept_info {
    company_name = a,
    department_name = b,
    =>
    head_count default 0,
```

(continues on next page)

(continued from previous page)

```
    address default ''
}
```

The expression is evaluated anew for each row, so if you specified a 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, or if the type of the column is nullable, in which case the value defaults to `null`. For these operations, 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.

The `:assert (some|none)`, `:ensure` and `:ensure_not` query options allow you to express complicated constraints that must be satisfied for your transaction to commit.

This example uses three queries to put and remove rows atomically (either all succeed or all fail), and ensure that at the end of the transaction an untouched row exists:

```
{
    ?[a, b] <- [[1, 'one'], [3, 'three']]
    :put rel {a => b}
}
{
    ?[a] <- [[2]]
    :rm rel {a}
}
{
    ?[a, b] <- [[4, 'four']]
    :ensure rel {a => b}
}
```

When a transaction starts, a snapshot is used, so that only already committed data, or data written within the same transaction, are visible to queries. At the end of the transaction, changes are only committed if there are no conflicts and no errors are raised. If any mutation activate triggers, those triggers execute in the same transaction.

3.3 Triggers and indices

Cozo does not have traditional indices on stored relations. Instead, you define regular stored relations that are used as indices. 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 leads to lots of repetitions and is error-prone, Cozo supports *triggers* to do it automatically for you.

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

```
::set_triggers <REL_NAME>

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

<QUERY> can be any valid query.

The `on put` triggers will run when new data is inserted or upserted, which can be activated by `:put`, `:create` and `:replace` query options. The implicitly defined rules `_new[]` and `_old[]` can be used in the triggers, and contain the added rows and the replaced rows respectively.

The `on rm` triggers will run when data is deleted, which can be activated by a `:rm` query option. The implicitly defined rules `_new[]` and `_old[]` can be used in the triggers, and contain the keys of the rows for deleted rows (even if no row with the key actually exist) and the rows actually deleted (with both keys and non-keys).

The `on replace` triggers will be activated by a `:replace` query option. They are run before any `on put` triggers.

All triggers for a relation must be specified together, in the same `::set_triggers` system op. If used again, all the triggers associated with the stored relation are replaced. To remove all triggers from a stored relation, use `::set_triggers` <REL_NAME> followed by nothing.

As an example of using triggers to maintain an index, suppose we have the following relation:

```
:create rel {a => b}
```

We often want to query `*rel[a, b]` with `b` bound but `a` unbound. This will cause a full scan, which can be expensive. So we need an index:

```
:create rel.rev {b, a}
```

In the general case, we cannot assume a functional dependency `b => a`, so in the index both fields appear as keys.

To manage the index automatically:

```
::set_triggers rel

on put {
    ?[a, b] := _new[a, b]

    :put rel.rev{ b, a }
}
on rm {
    ?[a, b] := _old[a, b]

```

(continues on next page)

(continued from previous page)

```
    :rm rel.rev{ b, a }
}
```

With the index set up, you can use `*rel.rev{..}` in place of `*rel{..}` in your queries.

Indices in Cozo are manual, but extremely flexible, since you need not conform to any predetermined patterns in your use of `_old[]` and `_new[]`. For simple queries, the need to explicitly elect to use an index can seem cumbersome, but for complex ones, the deterministic evaluation entailed can be a huge blessing.

Triggers can be creatively used for other purposes as well.

Warning: Loops in your triggers can cause non-termination. 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, you can decipher the result.

4.2 Ops for stored relations

`::relations`

List all stored relations in the database

`::columns <REL_NAME>`

List all columns for the stored relation `<REL_NAME>`.

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

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

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

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

`::show_triggers <REL_NAME>`

Display triggers associated with the stored relation `<REL_NAME>`.

`::set_triggers <REL_NAME> ...`

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

`::access_level <ACCESS_LEVEL> <REL_NAME> (, <REL_NAME>)*`

Sets the access level of `<REL_NAME>` to the given level. The levels are:

- `normal` allows everything,
- `protected` disallows `::remove` and `:replace`,
- `read_only` additionally disallows any mutations and setting triggers,
- `hidden` additionally disallows any data access (metadata access via `::relations`, etc., are still allowed).

The access level functionality is to protect data from mistakes of the programmer, not from attacks by malicious parties.

4.3 Monitor and kill

`::running`

Display 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`

Instructs Cozo to run a compaction job. Compaction makes the database smaller on disk and faster for read queries.

CHAPTER

FIVE

TYPES

5.1 Runtime types

Values in Cozo have the following *runtime 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:

- `false < true`;
- `-1 == -1.0 < 0 == 0.0 < 0.5 == 0.5 < 1 == 1.0`;
- Lists are ordered lexicographically by their elements;
- Bytes are compared lexicographically;
- Strings are compared 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: `1 == 1.0` evaluates to `true`, but `1` and `1.0` are distinct values, meaning that a relation can contain both as keys according to set semantics. This is especially confusing when using JavaScript, which converts all numbers to float, and python, which does not show a difference between the two when printing. Using floating point numbers in keys is not recommended if the rows are accessed by these keys (instead of accessed by iteration).

5.2 Literals

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

Besides the usual decimal notation for signed integers, you can prefix a number with `0x` or `-0x` for hexadecimal representation, with `0o` or `-0o` for octal, or with `0b` or `-0b` for binary. Floating point numbers include the decimal dot (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 using double quotes `""`, with the same escape rules. You can also use single quotes `''` in which case the roles of double quotes and single quotes are switched. There is also a "raw string" notation:

```
___"I'm a raw string"___
```

A raw string starts with 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. By varying the number of underscores, you can represent any string without quoting.

There is no literal representation for `Bytes` or `Uuid`. Use the appropriate functions to create them. If you are inserting data into a stored relation with a column specified to contain bytes or UUIDs, auto-coercion will kick in and use `decode_base64` and `to_uuid` for conversion.

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

Databases often consider how queries are executed an implementation detail hidden behind an abstraction barrier that users need not care about, so that databases can utilize query optimizers to choose the best query execution plan regardless of how the query was originally written. This abstraction barrier is leaky, however, since bad query execution plans invariably occur, and users need to "reach behind the curtain" to fix performance problems, which is a difficult and tiring task. The problem becomes more severe the more joins a query contains, and graph queries tend to contain a large number of joins.

So in Cozo we take the pragmatic approach and make query execution deterministic and easy to tell from how the query was written. The flip side is that we demand the user to know what is the best way to store their data, which is in general less demanding than coercing the query optimizer. Then, armed with knowledge of this chapter, writing efficient queries is easy.

6.1 Disjunctive normal form

Evaluation starts by canonicalizing inline rules into disjunction normal form[4], i.e., a disjunction of conjunctions, with any negation pushed to the innermost level. Each clause of the outmost disjunction is then 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.

6.2 Stratification

The next step in the processing is *stratification*. It 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.

Next, 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,
- when a fixed rule has another rule as an input relation.

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

The strongly connected components of the graph of rules 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 the strata 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.

6.3 Magic set rewrites

Within each stratum, the input rules are rewritten using the technique of *magic sets*. This rewriting ensures that the query execution does not waste time calculating results that are later 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 rewriting avoids this problem. You can see the result of the rewriting using `::explain`. The rewritten query is guaranteed to yield the same relation for `?`, and will in general yield fewer intermediate rows.

The rewrite currently only applies to inline rules without aggregations.

6.4 Semi-naïve evaluation

Now each stratum contains either a single fixed rule or a set of inline rules. The single fixed rules are executed by running their specific implementations. For the inline rules, each of them 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.

Note: 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.

6.5 Ordering of atoms

The compiler reorders the atoms in the body of the inline rules, and then the atoms are evaluated.

After conversion to disjunctive normal forms, 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 reordering make 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. 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. By applying filters as early as possible, we minimize the number of rows before joining them with the next relation.

When writing the body of rules, we 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.

6.6 Evaluating atoms

We now explain how a single atom which generates new bindings is processed.

For unifications, the right-hand side, 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).

Rules or stored relations are conceptually trees, with composite keys sorted lexicographically. The complexity of their applications in atoms is therefore determined by whether the bound variables and constants in the application bindings form a *key prefix*. 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 scan. On the other hand, if `a` is bound, then this is only a logarithmic-time existence 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.

Rows are generated in a streaming fashion, meaning that relation joins proceed as soon as one row is available, and do not wait until the whole relation is generated.

6.7 Early stopping

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. This only works when the entry rule is inline and you do not specify `:order`.

CHAPTER

SEVEN

TIPS FOR WRITING QUERIES

7.1 Dealing with nulls

Cozo is strict about types. A simple query such as:

```
?[a] := *rel[a, b], b > 0
```

will throw if some of the `b` is null: comparisons can only be made between values of the same type. There are various ways you can deal with it: if you decide that the condition should be `false` if values are not of the same type, you write:

```
?[a] := *rel[a, b], try(b > 0, false)
```

Alternatively, you may decide to consider any `null` values to be equivalent to some default values, in which case you write:

```
?[a] := *rel[a, b], (b ~ -1) > 0
```

here `~` is the `coalesce` operator. The parentheses are not necessary, but it reads better this way. We recommend using the `coalesce` operator over using `try`, since it is more explicit.

You can also be very explicit:

```
?[a] := *rel[a, b], if(is_null(b), false, b > 0)
```

but this is rather verbose. `cond` is also helpful in this case.

7.2 How to join relations

Suppose we have the following relation:

```
:create friend {fr, to}
```

Let's say we want to find Alice's friends' friends' friends' friends' friends. One way to write this is:

```
?[who] := *friends{fr: 'Alice', to: f1},
          *friends{fr: f1, to: f2},
          *friends{fr: f2, to: f3},
          *friends{fr: f3, to: f4},
          *friends{fr: f4, to: who}
```

Another way is:

```
f1[who] := *friends{fr: 'Alice', to: who}
f2[who] := f1[fr], *friends{fr, to: who}
f3[who] := f2[fr], *friends{fr, to: who}
f4[who] := f3[fr], *friends{fr, to: who}
?[who] := f4[fr], *friends{fr, to: who}
```

These two queries yield identical values. But on real networks, where loops abound, the second way of writing executes exponentially faster than the first. Why? Because of set semantics in relations, the second way of writing deduplicates at every turn, whereas the first way of writing builds up all paths to the final layer of friends. Depending on the size of your graph, your computer may not even have enough memory to hold all these paths!

The moral of the story is, always prefer to break your query into smaller rules. It usually reads better, and unlike in some other databases, it almost always executes faster in Cozo as well. But for this particular case, in which the query is largely recursive, prefer to make it a recursive relation:

```
f_n[who, min(layer)] := *friends{fr: 'Alice', to: who}, layer = 1
f_n[who, min(layer)] := f_n[fr, last_layer], *friends{fr, to: who}, layer = last_layer +␣
↪1, layer <= 5
?[who] := f_n[who, 5]
```

The condition `layer <= 5` is necessary to ensure termination.

Are there any situations where the first way of writing is acceptable? Yes:

```
?[who] := *friends{fr: 'Alice', to: f1},
          *friends{fr: f1, to: f2},
          *friends{fr: f2, to: f3},
          *friends{fr: f3, to: f4},
          *friends{fr: f4, to: who}
:limit 1
```

in this case, we stop at the first path, and this way of writing avoids the overhead of multiple rules and is perhaps very slightly faster.

Also, if you want to count the different paths, you must write:

```
?[count(who)] := *friends{fr: 'Alice', to: f1},
                 *friends{fr: f1, to: f2},
                 *friends{fr: f2, to: f3},
                 *friends{fr: f3, to: f4},
                 *friends{fr: f4, to: who}
```

The multiple-rules way of writing gives wrong results due to set semantics. Due to the presence of the aggregation `count`, this query only keeps a single path in memory at any instant, so it won't blow up your memory even on web-scale data.

CHAPTER

EIGHT

FUNCTIONS AND OPERATORS

Functions can be used to build expressions.

All functions except those having names starting with `rand_` are deterministic.

8.1 Non-functions

Functions must take in expressions as arguments, evaluate each argument in turn, and then evaluate its implementation to produce a value that can be used in an expression. We first describe constructs that look like, but are not functions.

These are language constucts that return Horn clauses instead of expressions:

- `var = expr` unifies `expr` with `var`. Different from `expr1 == expr2`.
- `not clause` negates a Horn clause `clause`. Different from `!expr` or `negate(expr)`.
- `clause1 or clause2` connects two Horn-clauses by disjunction. Different from `or(expr1, expr2)`.
- `clause1 and clause2` connects two Horn-clauses by conjunction. Different from `and(expr1, expr2)`.
- `clause1, clause2` connects two Horn-clauses by conjunction.

For the last three, `or` binds more tightly from `and`, which in turn binds more tightly than `,`: `and` and `,` are identical in every aspect except their binding powers.

These are constructs that return expressions:

- `try(a, b, ...)` evaluates each argument in turn, stops at the first expression that does not throw and return its value.
- `if(a, b, c)` evaluates `a`, and if the result is `true`, evaluate `b` and returns its value, otherwise evaluate `c` and returns its value. `a` must evaluate to a boolean.
- `if(a, b)` same as `if(a, b, null)`
- `cond(a1, b1, a2, b2, ...)` evaluates `a1`, if the results is `true`, returns the value of `b1`, otherwise continue with `a2` and `b2`. An even number of arguments must be given and the `a``s must evaluate to booleans. If all ``a``s are ``false`, `null` is returned. If you want a catch-all clause at the end, put `true` as the condition.

8.2 Operators representing functions

Some functions have equivalent operator forms, which are easier to type and perhaps more familiar. First the binary operators:

- `a && b` is the same as `and(a, b)`
- `a || b` is the same as `or(a, b)`
- `a ^ b` is the same as `pow(a, b)`
- `a ++ b` is the same as `concat(a, b)`
- `a + b` is the same as `add(a, b)`
- `a - b` is the same as `sub(a, b)`
- `a * b` is the same as `mul(a, b)`
- `a / b` is the same as `div(a, b)`
- `a % b` is the same as `mod(a, b)`
- `a >= b` is the same as `ge(a, b)`
- `a <= b` is the same as `le(a, b)`
- `a > b` is the same as `gt(a, b)`
- `a < b` is the same as `le(a, b)`
- `a == b` is the same as `eq(a, b)`
- `a != b` is the same as `neq(a, b)`
- `a ~ b` is the same as `coalesce(a, b)`

These operators have precedence as follows (the earlier rows binds more tightly, and within the same row operators have equal binding power):

- `~`
- `^`
- `*`, `/`
- `+`, `-`, `++`
- `==`, `!=`
- `%`
- `>=`, `<=`, `>`, `<`
- `&&`
- `||`

With the exception of `^`, all binary operators are left associative: `a / b / c` is the same as `(a / b) / c`. `^` is right associative: `a ^ b ^ c` is the same as `a ^ (b ^ c)`.

And the unary operators are:

- `-a` is the same as `minus(a)`
- `!a` is the same as `negate(a)`

Function applications using parentheses bind the tightest, followed by unary operators, then binary operators.

8.3 Equality and Comparisons

eq(*x*, *y*)

Equality comparison. The operator form is `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 runtime 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.

8.4 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.

8.5 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 function.

cos(*x*)

The cosine trigonometric function.

tan(*x*)

The tangent trigonometric function.

asin(*x*)

The inverse sine.

acos(*x*)

The inverse cosine.

atan(*x*)

The inverse tangent.

atan2(*x*, *y*)

The inverse tangent atan2[5] 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.

rad_to_deg(*x*)

Converts radians to degrees.

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

Computes with the haversine formula[6] 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 when you are dealing with maps, since most maps measure angles in degrees instead of 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.

[5] https://en.wikipedia.org/wiki/Atan2
[6] https://en.wikipedia.org/wiki/Haversine_formula

Note: 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.

8.6 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[7] from both ends of the string.

`trim_start`(*x*)

Removes whitespace[8] from the start of the string.

`trim_end`(*x*)

Removes whitespace[9] from the end of the string.

`starts_with`(*x*, *y*)

Tests if `x` starts with `y`.

Tip: `starts_with(var, str)` is preferred 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`.

[7] https://en.wikipedia.org/wiki/Whitespace_character
[8] https://en.wikipedia.org/wiki/Whitespace_character
[9] https://en.wikipedia.org/wiki/Whitespace_character

unicode_normalize(*str*, *norm*)

Converts `str` to the normalization[10] 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`.

8.7 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`. Raises 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`. Returns `null` if the access is out of bounds. Indices start with 0.

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`.

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

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.

8.8 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[11]-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[12]-encoded byte array.

8.9 Type checking and conversions

`coalesce`(*x*, ...)

Returns the first non-null value; *coalesce(x, y)* is equivalent to *x ~ y*.

`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...).

Converts *null* and *false* to *0.0*, *true* to *1.0*

`to_unity`(*x*)

Tries to convert `x` to `0` or `1`: `null`, `false`, `0`, `0.0`, `""`, `[]`, and the empty bytes are converted to `0`, and everything else is converted to `1`.

This is useful in conjunction with aggregation functions. For example, `?[x, count(x)] := rel[x, y], y > 3` with a filter in the body omit groups that are completely filtered out. Instead, use `?[x, sum(should_count)] := rel[x, y], should_count = to_unity(y > 3)`.

`to_bool`(*x*)

Tries to convert `x` to a boolean. The following are converted to `false`, and everything else is converted to `true`:

- `null`
- `false`
- `0`, `0.0`
- `""` (empty string)
- the empty byte array
- the nil UUID (all zeros)
- `[]` (the empty list)

[11] https://en.wikipedia.org/wiki/Base64
[12] https://en.wikipedia.org/wiki/Base64

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`.

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.

8.10 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). The resolution of the timestamp part is much coarser on WASM targets than the others.

rand_uuid_v4()

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

8.11 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`.

8.11.1 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)
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
```

8.12 Timestamp functions

now()

Returns the current timestamp as seconds since the UNIX epoch. The resolution is much coarser on WASM targets than the others.

format_timestamp(*ts*, *tz?*)

Interpret `ts` as seconds since the epoch and format as a string according to RFC3339[13].

If a second string argument is provided, it is interpreted as a timezone[14] and used to format the timestamp.

parse_timestamp(*str*)

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

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

CHAPTER

NINE

AGGREGATIONS

Aggregations in Cozo can be thought of as a function that acts on a stream of values and produces a single value (the aggregate).

There are two kinds of aggregations in Cozo, *ordinary aggregations* and *semi-lattice aggregations*. They are implemented differently in Cozo, with semi-lattice aggregations generally faster and more powerful (only the latter can be used recursively).

The power of semi-lattice aggregations derive from the additional properties they satisfy: a semilattice[15]:

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.

In auto-recursive semi-lattice aggregations, there are soundness constraints on what can be done on the bindings coming from the auto-recursive parts within the body of the rule. Usually you do not need to worry about this at all since the obvious ways of using this functionality are all sound, but as for non-termination due to fresh variables introduced by function applications, Cozo does not (and cannot) check for unsoundness in this case.

9.1 Semi-lattice 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.

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

intersection(*var*)

Aggregate the intersections of `var`, which must be a list.

choice(*var*)

Returns a non-null value. If all values are null, returns *null*. Which one is returned is deterministic but implementation-dependent and may change from version to version.

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.

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.

9.2 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 `time`. This is very similar to `min_cost`, the differences being that maximum instead of minimum is used, only `data` is returned, and the aggregation is deliberately not a semi-lattice aggregation.

Note: This aggregation is intended to be used in timestamped audit trails. As an example:

```
?[id, latest_by(status_ts)] := *data[id, status, ts], status_ts = [status, ts]
```

returns the latest `status` for each `id`.

---

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.

Note: This version of `choice` is not a semi-lattice aggregation since it is impossible to satisfy the uniform sampling requirement while maintaining no state, which is an implementation restriction unlikely to be lifted.

9.2.1 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

TEN

UTILITIES AND ALGORITHMS

Fixed rules in CozoScript apply utilities or algorithms.

The algorithms described here are only available if your distribution of Cozo is compiled with the `graph-algo` feature flag. Currently all prebuilt binaries are compiled with this flag on.

Some algorithms make use of multiple threads to greatly improve running time if the `rayon` feature flag is on. All prebuilt binaries except WASM have this flag on. As a result, and also because of other platform restrictions, algorithms on WASM in general run much slower than on other platforms.

10.1 Utilities

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

Returns a relation containing the data passed in. The constant rule `?[] <- ...` is 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.

This utility is not available on WASM targets. In addition, if the feature flag `requests` is off, only reading from local file is supported.

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.

This utility is not available on WASM targets. In addition, if the feature flag `requests` is off, only reading from local file is supported.

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.

10.2 Connectedness algorithms

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

Computes the connected components[16] of a graph with the provided edges.

Returns

Pairs containing the node index, and its component index.

[16] https://en.wikipedia.org/wiki/Connected_component_(graph_theory)

StronglyConnectedComponent(*edges[from, to]*)

Computes the strongly connected components[17] 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[18] on the provided edges to compute a minimum spanning forest[19]. Negative weights are fine.

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[20] on the provided edges to compute a minimum spanning tree[21]. `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[22] 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.

10.3 Pathfinding algorithms

ShortestPathBFS(*edges[from, to], starting[start_idx], goals[goal_idx]*)

Runs breadth-first search to determine the shortest path between the ``starting`` nodes and the ``goals`

Assumes the graph to be directed and all edges to be of unit weight.
Ties will be broken in an unspecified way.
If you need anything more complicated, use one of the other algorithms below.

Returns

Triples containing the starting node, the goal, and a shortest path.

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

Runs Dijkstra's algorithm[23] to determine the shortest paths between the `starting` nodes and the `goals`. Weights, if given, must be non-negative.

[17] https://en.wikipedia.org/wiki/Strongly_connected_component
[18] https://en.wikipedia.org/wiki/Kruskal%27s_algorithm
[19] https://en.wikipedia.org/wiki/Minimum_spanning_tree
[20] https://en.wikipedia.org/wiki/Prim%27s_algorithm
[21] https://en.wikipedia.org/wiki/Minimum_spanning_tree
[22] https://en.wikipedia.org/wiki/Topological_sorting

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[24] (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.

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.

DFS(...)

See *Algo.DepthFirstSearch*.

[23] https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
[24] https://en.wikipedia.org/wiki/Yen%27s_algorithm

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[25].

`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.

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.

10.4 Community detection algorithms

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

Computes the clustering coefficients[26] 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[27] 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[28] 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

[25] https://en.wikipedia.org/wiki/A*_search_algorithm
[26] https://en.wikipedia.org/wiki/Clustering_coefficient

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[29] 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.

10.5 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: 0.85, epsilon: 0.0001, iterations: 10*)

Computes the PageRank[30] from the given graph with the provided edges, optionally weighted.

This algorithm is implemented differently if the *rayon* is not enabled, in which case it runs much slower. This affects only the WASM platform.

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[31] 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[32] of the graph. The input relation represent edges connecting node indices which are optionally weighted.

[27] https://en.wikipedia.org/wiki/Louvain_method
[28] https://en.wikipedia.org/wiki/Modularity_(networks)
[29] https://en.wikipedia.org/wiki/Label_propagation_algorithm
[30] https://en.wikipedia.org/wiki/PageRank
[31] https://en.wikipedia.org/wiki/Norm_(mathematics)

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[33] 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 for medium to large graphs. If possible, collapse large graphs into supergraphs by running a community detection algorithm first.

10.6 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.

[32] https://en.wikipedia.org/wiki/Closeness_centrality
[33] https://en.wikipedia.org/wiki/Betweenness_centrality

CHAPTER

ELEVEN

BEYOND COZOSCRIPT

Most functionalities of the Cozo database are accessible via the CozoScript API. However, other functionalities either cannot conform to the "always return a relation" constraint, or are of such a nature as to make a separate API desirable. These are described here.

The calling convention and even names of the APIs may differ on different target languages, please refer to the respective language-specific documentation. Here we use the Python API as an example to describe what they do.

`export_relations`(*self*, *relations*)

Export the specified `relations`. It is guaranteed that the exported data form a consistent snapshot of what was stored in the database.

Parameters

`relations` – names of the relations in a list.

Returns

a dict with string keys for the names of relations, and values containing all the rows.

`import_relations`(*self*, *data*)

Import data into a database. The data are imported inside a transaction, so that either all imports are successful, or none are. If conflicts arise because of concurrent modification to the database, via either CosoScript queries or other imports, the transaction will fail.

The relations to import into must exist beforehand, and the data given must match the schema defined.

This API can be used to batch-put or remove data from several stored relations atomically. The `data` parameter can contain relation names such as `"rel_a"`, or relation names prefixed by a minus sign such as `"-rel_a"`. For the former case, every row given for the relation will be `put` into the database, i.e. upsert semantics. For the latter case, the corresponding rows are removed from the database, and you should only specify the key part of the rows. As for `rm` in CozoScript, it is not an error to remove non-existent rows.

> **Warning:** Triggers are not run for direct imports.

Parameters

`data` – should be given as a dict with string keys, in the same format as returned by *export_relations*. For example: `{"rel_a": {"headers": ["x", "y"], "rows": [[1, 2], [3, 4]]}, "rel_b": {"headers": ["z"], "rows": []}}`

`backup`(*self*, *path*)

Backup a database to the specified path. The exported data is guaranteed to form a consistent snapshot of what was stored in the database.

This backs up everything: you cannot choose what to back up. It is also much more efficient than exporting all stored relations via `export_relations`, and only a tiny fraction of the total data needs to reside in memory during backup.

This function is only available if the `storage-sqlite` feature flag was on when compiling. The flag is on for all pre-built binaries except the WASM binaries. The backup produced by this API can then be used as an independent SQLite-based Cozo database. If you want to store the backup for future use, you should compress it to save a lot of disk space.

Parameters

path – the path to write the backup into. For a remote database, this is a path on the remote machine.

`restore`(*self*, *path*)

Restore the database from a backup. Must be called on an empty database.

This restores everything: you cannot choose what to restore.

Parameters

path – the path to the backup. You cannot restore remote databases this way: use the executable directly.

`import_from_backup`(*self*, *path*, *relations*)

Import stored relations from a backup.

In terms of semantics, this is like `import_relations`, except that data comes from the backup file directly, and you can only `put`, not `rm`. It is also more memory-efficient than `import_relations`.

Warning: Triggers are not run for direct imports.

Parameters

- **path** – path to the backup file. For remote databases, this is a path on the remote machine.
- **relations** – a list containing the names of the relations to import. The relations must exist in the database.

INDEX

I

J

K

L

M

N

O

P

R

S

T

U

V

W