The Cozo Database Manual 0.7

Release 0.7

Ziyang Hu

May 15, 2023

CONTENTS

Welcome to the documentation for CozoDB, a feature-rich, transactional, time-travel-enabled, relational-graph-vector database that uses Datalog for queries, while making no performance compromises.

You can start immediately with the *Tutorial*, which you can follow without installing anything.

For installation instructions and language-specific APIs, please refer to the GitHub page[1].

[1] https://github.com/cozodb/cozo

CHAPTER

ONE

TUTORIAL

This tutorial will teach you the basics of using the Cozo database.

There are several ways you can run the queries in this tutorial:

- You can run the examples in your browser without installing anything: just open Cozo in Wasm[2] and you are ready to go.
- You can download the appropriate `cozo-*` executable for your operating system from the release page[3], uncompress, rename to `cozo`, and run the REPL mode by running in a terminal `cozo repl` and follow along.
- If you are familiar with the Python datascience stack, you should following the instruction here[4] instead to run this notebook in Jupyter Lab, with the notebook file[5].
- There are many other ways[6], but the above ones are the easiest.

The following cell is to set up Jupyter. Ignore if you are using other methods.

```
[1]: %load_ext pycozo.ipyext_direct
```

1.1 First steps

Cozo is a relational database. The "hello world" query

```
[2]: ?[] <- [['hello', 'world', 'Cozo!']]
```

```
[2]: <pandas.io.formats.style.Styler at 0x26c461cf610>
```

simply passes an ad hoc relation represented by a list of lists to the database, and the database echoes it back.

You can pass more rows, or a different number of columns:

```
[3]: ?[] <- [[1, 2, 3], ['a', 'b', 'c']]
```

```
[3]: <pandas.io.formats.style.Styler at 0x26c44c7fc10>
```

This example shows how to enter literals for numbers, strings, booleans and `null`:

[2] https://www.cozodb.org/wasm-demo/
[3] https://github.com/cozodb/cozo/releases
[4] https://github.com/cozodb/pycozo
[5] https://github.com/cozodb/cozo-docs/blob/main/source/tutorial.ipynb
[6] https://github.com/cozodb/cozo#Install

```
[4]: ?[] <- [[1.5, 2.5, 3, 4, 5.5],
             ['aA', 'bB', 'cC', 'dD', 'eE'],
             [true, false, null, -1.4e-2, "A string with double quotes"]]
```

```
[4]: <pandas.io.formats.style.Styler at 0x26c44c5c5d0>
```

The input literal representations are similar to those in JavaScript. In particular, strings in double quotes are guaranteed to be interpreted in the same way as in JSON. The output are in JSON, but how they appear on your screen depends on your setup. For example, if you are using a Python setup, booleans are displayed as `True` and `False` instead of in lowercase, since they are converted to the native Python datatypes for display.

In the last example, you may have noticed that the returned order is not the same as the input order. This is because in Cozo relations are always stored as trees, and trees are always sorted.

Another consequence of relations as trees is that you can have no duplicate rows:

```
[5]: ?[] <- [[1], [2], [1], [2], [1]]
```

```
[5]: <pandas.io.formats.style.Styler at 0x26c461e2d90>
```

Relations in Cozo follow *set semantics* where de-duplication is automatic. By contrast, SQL usually follows *bag semantics* (some databases do this by secretly having a unique internal key for every row, in Cozo you must do this explicitly if you need to simulate duplicate rows).

Why does Cozo break tradition and go with set semantics? Set semantics is much more convenient when you have recursions between relations involved, and Cozo is designed to deal with complex recursions.

1.1.1 Expressions

The next example shows the use of various expressions and comments:

```
[6]: ?[] <- [[
            1 + 2, # addition
            3 / 4, # division
            5 == 6, # equality
            7 > 8, # greater
            true || false, # or
            false && true, # and
            lowercase('HELLO'), # function
            rand_float(), # function taking no argument
            union([1, 2, 3], [3, 4, 5], [5, 6, 7]), # variadic function
        ]]
```

```
[6]: <pandas.io.formats.style.Styler at 0x26c5d8ab5d0>
```

See *here* for the full list of functions you can use to build expressions. The syntax is deliberately C-like.

1.1.2 Rules and relations

Previous examples all start with `?[] <-`. The notation denotes a *rule* named `?`. It is a *constant rule* since it uses `<-`. The part after `?` and before `<-` is the *head* of the rule, and the part after `<-` is the *body*.

Rules can have other names, but the rule named `?` is special in that its evaluation determines the return relation of the query.

We can give *bindings* in the *head* of rules:

```
[7]: ?[first, second, third] <- [[1, 2, 3], ['a', 'b', 'c']]
```

```
[7]: <pandas.io.formats.style.Styler at 0x26c5cad58d0>
```

For constant rules, the number of bindings must match the actual data (the *arity*), otherwise, you will get an error:

```
[8]: ?[first, second] <- [[1, 2, 3], ['a', 'b', 'c']]
```

```
[8]: parser::fixed_rule_head_arity_mismatch

  × Fixed rule head arity mismatch
   ╭────
 1 │ ?[first, second] <- [[1, 2, 3], ['a', 'b', 'c']]
   · ──────────────────────────────────────────────────
   ╰────
  help: Expected arity: 3, number of arguments given: 2
```

Now let's define rules that *apply* (use) other rules:

```
[9]: rule[first, second, third] <- [[1, 2, 3], ['a', 'b', 'c']]
     ?[a, b, c] := rule[a, b, c]
```

```
[9]: <pandas.io.formats.style.Styler at 0x26c461d3a90>
```

This first line defines a constant rule named `rule`. The `?` rule is now an *inline rule*, denoted by the connecting symbol `:=`. In its body it applies the fixed rule, by giving the name of the rule followed by three *fresh bindings*, which are the *variables* `a`, `b` and `c`.

With inline rules, you can manipulate the order of the columns, or specify which columns are returned:

```
[10]: rule[first, second, third] <- [[1, 2, 3], ['a', 'b', 'c']]
      ?[c, b] := rule[a, b, c]
```

```
[10]: <pandas.io.formats.style.Styler at 0x26c5d8a2110>
```

The body of an inline rule consists of *atoms*. The previous example has a single *application atom* as the body. Multiple atoms are connected by commas:

```
[11]: ?[c, b] := rule[a, b, c], is_num(a)
      rule[first, second, third] <- [[1, 2, 3], ['a', 'b', 'c']]
```

```
[11]: <pandas.io.formats.style.Styler at 0x26c5d8aba50>
```

Here the second atom is an *expression* `is_num(a)`. Expression atoms act as filters, so only rows for which the expression evaluates to `true` are returned. The order in which the rules are given is immaterial.

You can also bind constants to rule applications directly:

```
[12]: rule[first, second, third] <- [[1, 2, 3], ['a', 'b', 'c']]
      ?[c, b] := rule['a', b, c]
```

```
[12]: <pandas.io.formats.style.Styler at 0x26c44c67e90>
```

You introduce additional bindings with the *unification operator* =:

```
[13]: rule[first, second, third] <- [[1, 2, 3], ['a', 'b', 'c']]
      ?[c, b, d] := rule[a, b, c], is_num(a), d = a + b + 2*c
```

```
[13]: <pandas.io.formats.style.Styler at 0x26c5d8a9790>
```

Having multiple rule applications in the body generates every combination of the bindings:

```
[14]: r1[] <- [[1, 'a'], [2, 'b']]
      r2[] <- [[2, 'B'], [3, 'C']]

      ?[l1, l2] := r1[a, l1],
                   r2[b, l2]
```

```
[14]: <pandas.io.formats.style.Styler at 0x26c5d8afad0>
```

This is a Cartesian join in relational algebra, as bindings in the rule applications are all distinct. If bindings are reused, *implicit unification* occurs, which is a join in relational algebra:

```
[15]: r1[] <- [[1, 'a'], [2, 'b']]
      r2[] <- [[2, 'B'], [3, 'C']]

      ?[l1, l2] := r1[a, l1],
                   r2[a, l2] # reused `a`
```

```
[15]: <pandas.io.formats.style.Styler at 0x26c5d89ded0>
```

The unification = unifies with a single value. Use `in` to unify with each value within a list in turn:

```
[16]: ?[x, y] := x in [1, 2, 3], y in ['x', 'y']
```

```
[16]: <pandas.io.formats.style.Styler at 0x26c5d8afb10>
```

The head of inline rules does not need to use all variables that appear in the body. However, any variable in the head must appear in the body at least once (the *safety rule*):

```
[17]: r1[] <- [[1, 'a'], [2, 'b']]
      r2[] <- [[2, 'B'], [3, 'C']]

      ?[l1, l2, x] := r1[a, l1],
                      r2[a, l2]
```

```
[17]: eval::unbound_symb_in_head

        × Symbol 'x' in rule head is unbound
        ╭─[3:1]
      3 │
      4 │ ?[l1, l2, x] := r1[a, l1],
        ·                 ─
      5 │                       r2[a, l2]
        ╰────
```

(continues on next page)

(continued from previous page)

```
help: Note that symbols occurring only in negated positions are not considered bound
```

1.1.3 Stored relations

Persistent relations in Cozo are called *stored relations*. Creating them are simple:

```
[18]: r1[] <- [[1, 'a'], [2, 'b']]
      r2[] <- [[2, 'B'], [3, 'C']]

      ?[l1, l2] := r1[a, l1],
                   r2[a, l2]

      :create stored {l1, l2}
```

```
[18]: <pandas.io.formats.style.Styler at 0x26c5d894b10>
```

The `:create` query option instructs the database to store the result in a stored relation named `stored`, containing the columns `l1` and `l2`.

Note that the name of the stored relation should not start with an underscore `_`. You will get no error if you use a name starting with an underscore, but you will discover that the relation won't be persisted. What you are dealing with is actually what is called *ephemeral relation*. This topic is discussed further in *here*.

If you just want to create the relation without adding any data, you can omit the queries. No need to have an empty `?` query.

You can verify that you now have the required stored relation in your system by running a *system op*:

```
[19]: ::relations
```

```
[19]: <pandas.io.formats.style.Styler at 0x26c5d89ff10>
```

You can also investigate the columns of the stored relation:

```
[20]: ::columns stored
```

```
[20]: <pandas.io.formats.style.Styler at 0x26c5d8acf50>
```

Stored relations can be applied by using an asterisk `*` before the name:

```
[21]: ?[a, b] := *stored[a, b]
```

```
[21]: <pandas.io.formats.style.Styler at 0x26c5d8cee90>
```

Unlike relations defined inline, the columns of stored relations have fixed names. You can take advantage of this by selectively referring to columns by name, especially if you have a lot of columns:

```
[22]: ?[a, b] := *stored{l2: b, l1: a}
```

```
[22]: <pandas.io.formats.style.Styler at 0x26c5d89ea10>
```

If the binding is the same as the column name, you can omit the binding:

```
[23]: ?[l2] := *stored{l2}
```

[23]:
```
<pandas.io.formats.style.Styler at 0x26c5d89fa50>
```

After a stored relation is created, use `:put` to insert more data:

[24]:
```
?[l1, l2] <- [['e', 'E']]

:put stored {l1, l2}
```

[24]:
```
<pandas.io.formats.style.Styler at 0x26c5d8a0210>
```

[25]:
```
?[l1, l2] := *stored[l1, l2]
```

[25]:
```
<pandas.io.formats.style.Styler at 0x26c5d89f750>
```

Use `:rm` to remove data:

[26]:
```
?[l1, l2] <- [['e', 'E']]

:rm stored {l1, l2}
```

[26]:
```
<pandas.io.formats.style.Styler at 0x26c5d564850>
```

[27]:
```
?[l1, l2] := *stored[l1, l2]
```

[27]:
```
<pandas.io.formats.style.Styler at 0x26c5d8cf9d0>
```

Use `::remove` (double colon!) to get rid of a stored relation:

[28]:
```
::remove stored
```

[28]:
```
<pandas.io.formats.style.Styler at 0x26c5d8ccc10>
```

[29]:
```
::relations
```

[29]:
```
<pandas.io.formats.style.Styler at 0x26c461eec90>
```

So far, all stored relations store all the data in their *keys*. You can instruct Cozo to only treat some of the data as keys, thereby indicating a *functional dependency*:

[30]:
```
?[a, b, c] <- [[1, 'a', 'A'],
               [2, 'b', 'B'],
               [3, 'c', 'C'],
               [4, 'd', 'D']]

:create fd {a, b => c}
```

[30]:
```
<pandas.io.formats.style.Styler at 0x26c5d8ae0d0>
```

[31]:
```
?[a, b, c] := *fd[a, b, c]
```

[31]:
```
<pandas.io.formats.style.Styler at 0x26c5ca4ed50>
```

Now if you insert another row with an existing key, that row will be updated:

[32]:
```
?[a, b, c] <- [[3, 'c', 'CCCCCCC']]

:put fd {a, b => c}
```

[32]:
```
<pandas.io.formats.style.Styler at 0x26c5d894710>
```

[33]:
```
?[a, b, c] := *fd[a, b, c]
```

[33]:
```
<pandas.io.formats.style.Styler at 0x26c5d791110>
```

You can check whether a column is in a key position by looking at the `is_key` column in:

[34]:
```
::columns fd
```

[34]:
```
<pandas.io.formats.style.Styler at 0x26c5d8a1b10>
```

You may have noticed that columns also have types and default values associated with them, and stored relations can have triggers. These are discussed in detail *here*.

Before continuing, let's remove the stored relation we introduced:

[35]:
```
::remove fd
```

[35]:
```
<pandas.io.formats.style.Styler at 0x26c5d89e5d0>
```

1.1.4 Graphs

Now let's consider a graph stored as a relation:

[36]:
```
?[loving, loved] <- [['alice', 'eve'],
                     ['bob', 'alice'],
                     ['eve', 'alice'],
                     ['eve', 'bob'],
                     ['eve', 'charlie'],
                     ['charlie', 'eve'],
                     ['david', 'george'],
                     ['george', 'george']]

:replace love {loving, loved}
```

[36]:
```
<pandas.io.formats.style.Styler at 0x26c5d88dad0>
```

The graph we have created reads like "Alice loves Eve, Bob loves Alice", "nobody loves David, David loves George, but George only loves himself", and so on. Here we used `:replace` instead of `:create`. The difference is that if `love` already exists, it will be wiped and replaced with the new data given.

We can investigate competing interests:

[37]:
```
?[loved_by_b_e] := *love['eve', loved_by_b_e],
                   *love['bob', loved_by_b_e]
```

[37]:
```
<pandas.io.formats.style.Styler at 0x26c5d88d290>
```

So far we rule bodies consist of *conjunction* of atoms only. Disjunction is also available, by using the `or` keyword:

[38]:
```
?[loved_by_b_e] := *love['eve', loved_by_b_e] or *love['bob', loved_by_b_e],
                   loved_by_b_e != 'bob',
                   loved_by_b_e != 'eve'
```

```
[38]: <pandas.io.formats.style.Styler at 0x26c5d88f4d0>
```

Another way to write the same query is to have multiple rule definitions under the same name:

```
[39]: ?[loved_by_b_e] := *love['eve', loved_by_b_e],
                         loved_by_b_e != 'bob',
                         loved_by_b_e != 'eve'
      ?[loved_by_b_e] := *love['bob', loved_by_b_e],
                         loved_by_b_e != 'bob',
                         loved_by_b_e != 'eve'
[39]: <pandas.io.formats.style.Styler at 0x26c5d8f0a90>
```

When you have multiple definitions of the same inline rule, the rule heads must be compatible. Only inline rules can have multiple definitions.

1.1.5 Negation

Negation of *expressions* should be familiar:

```
[40]: ?[loved] := *love[person, loved], !ends_with(person, 'e')
[40]: <pandas.io.formats.style.Styler at 0x26c5d88dc50>
```

Rule applications can also be negated, not with the `!` operator, but with the `not` keyword:

```
[41]: ?[loved_by_e_not_b] := *love['eve', loved_by_e_not_b],
                             not *love['bob', loved_by_e_not_b]
[41]: <pandas.io.formats.style.Styler at 0x26c5d88dc90>
```

There are two sets of logical operations in Cozo, one set that acts on the level of expressions, and another set that acts on the level of atoms:

- For atoms: `,` or `and` (conjunction), `or` (disjunction), `not` (negation)
- For expressions: `&&` (conjunction), `||` (disjunction), `!` (negation)

The difference between `,` and `and` is operator precedence: `and` has higher precedence than `or`, whereas `,` has lower precedence than `or`.

There is a *safety rule* for negation:

```
[42]: ?[not_loved_by_b] := not *love['bob', not_loved_by_b]
[42]: eval::unbound_symb_in_head

        × Symbol 'not_loved_by_b' in rule head is unbound
          ────
      1 │ ?[not_loved_by_b] := not *love['bob', not_loved_by_b]
        ·   ──────────────
          ────
        help: Note that symbols occurring only in negated positions are not considered bound
```

This query is forbidden because the resulting relation is infinite. For example, 'gold' should be in the result, since *according to the facts stored in the database*, Bob has no interest in 'gold'.

To make our query finite, we have to explicitly give our query a *closed world*:

[43]:
```
the_population[p] := *love[p, _a]
the_population[p] := *love[_a, p]

?[not_loved_by_b] := the_population[not_loved_by_b],
                     not *love['bob', not_loved_by_b]
```

[43]:
```
<pandas.io.formats.style.Styler at 0x26c5d8f9410>
```

1.1.6 Recursion

Inline rules can applying other rules, and can have multiple definitions. If you combine these two, you get recursions:

[44]:
```
alice_love_chain[person] := *love['alice', person]
alice_love_chain[person] := alice_love_chain[in_person],
                            *love[in_person, person]

?[chained] := alice_love_chain[chained]
```

[44]:
```
<pandas.io.formats.style.Styler at 0x26c5d8aa050>
```

You may object that you only need to be able to apply other rules to have recursion, without multiple definitions. Technically correct, but the resulting queries are not useful:

```
..
```

[45]:
```
alice_love_chain[person] := alice_love_chain[in_person],
                            *love[in_person, person]

?[chained] := alice_love_chain[chained]
```

[45]:
```
<pandas.io.formats.style.Styler at 0x26c5d88d490>
```

Similar to the negation case, if there is no way to *deduce* a fact from the given facts, then the fact itself is considered false. You need multiple definitions to "bootstrap" the query.

1.1.7 Aggregation

Aggregations are usually used to compute statistics. In Cozo, aggregations are applied in the head of inline rules:

[46]:
```
?[person, count(loved_by)] := *love[loved_by, person]
```

[46]:
```
<pandas.io.formats.style.Styler at 0x26c5d8fba10>
```

The usual `sum`, `mean`, etc. are all available. Aggregations in the head instead of in the body may seem strange, but is powerful, as we will see later.

Here is the *full list* of aggregations for you to play with.

1.1.8 Query options

We have seen query options like `:create`, `:put`, `:rm` for manipulating stored relations. There are also query options for controlling what is returned:

```
[47]: ?[loving, loved] := *love{ loving, loved }

      :limit 1
```

```
[47]: <pandas.io.formats.style.Styler at 0x26c5d791650>
```

Next we want the result to be sorted by `loved` in descending order, then `loving` in ascending order, and skip the first row:

```
[48]: ?[loving, loved] := *love{ loving, loved }

      :order -loved, loving
      :offset 1
```

```
[48]: <pandas.io.formats.style.Styler at 0x26c5d913fd0>
```

Putting `-` in front of variables in `:order` clause denotes reverse order. Nothing or `+` denotes the ascending order.

The full list of query options are explained in *this chapter*.

1.1.9 Fixed rules

The `<-` syntax for constant rules is syntax sugar. The full syntax is:

```
[49]: ?[] <~ Constant(data: [['hello', 'world', 'Cozo!']])
```

```
[49]: <pandas.io.formats.style.Styler at 0x26c5d911a50>
```

Here we are using the *fixed rule* `Constant`, which takes one *option* named `data`. The curly tail `<~` denotes a fixed rule.

Fixed rules take input relations as arguments, apply custom logic to them and produce its output relation. The `Constant` fixed rule take zero input relations.

If you are using Cozo in browser, `Constant` is the only fixed rule you can use. In all other cases your Cozo would have the graph algorithms module enabled, and all graph algorithms are implemented as fixed rules. As an example, let's find out who is most popular in the `love` graph by using the PageRank[7] algorithm:

```
[50]: ?[person, page_rank] <~ PageRank(*love[])

      :order -page_rank
```

```
[50]: <pandas.io.formats.style.Styler at 0x26c5d8cded0>
```

Here the input relation is the stored relation `*love` (as noted above, you will receive an error if you run this in the WASM implementation).

Each fixed rule is different: *here* is the full list.

```
[51]: ::remove love
```

```
[51]: <pandas.io.formats.style.Styler at 0x26c461b2210>
```

[7] https://en.wikipedia.org/wiki/PageRank

1.2 Time travel

A very useful capability of Cozo is the ability to time travel in a stored relation. Usually when you `:put` into a relation, the old value is overwritten; and when you `:rm`, the row is completely gone. Doing it this way amounts to throwing away gold if how your data changes is valuable: we have a *short story* about it. In this case, *time travel* is the solution: instead of storing *current* facts, the stored relation stores the complete history of facts, at least all the available history as history itself unfolds.

If you believe that you don't want this functionality at all, you can skip to the next section. At Cozo we adopt the "zero-cost, zero-mental-overhead abstraction" philosophy: if you don't use a functionality, you don't pay the performance or cognitive overhead.

Let's have a simple example: storing the head of state of countries. First we have to create a new stored relation:

[52]:
```
:create hos {state: String, year: Validity => hos: String}
```

[52]:
```
<pandas.io.formats.style.Styler at 0x26c5d8accd0>
```

`hos` is shorthand for "head of state". The only thing different about this relation is that we are giving `year` the type `Validity`. You can think of validity as a list of two elements, the first element being an integer and the second element being a boolean. The integer indicates the "time" of the fact recorded by the row, the boolean, if `true`, indicates that this fact is an *assertion* valid from the indicated time, otherwise it indicates that the previous fact under the same key differing only by the validity is retracted at this time. It is up to you to interpret what "time" really means. Here we will use it to mean the year, for simplicity.

Now let's insert some data:

[53]:
```
?[state, year, hos] <- [['US', [2001, true], 'Bush'],
                        ['US', [2005, true], 'Bush'],
                        ['US', [2009, true], 'Obama'],
                        ['US', [2013, true], 'Obama'],
                        ['US', [2017, true], 'Trump'],
                        ['US', [2021, true], 'Biden']]

:put hos {state, year => hos}
```

[53]:
```
<pandas.io.formats.style.Styler at 0x26c5d886210>
```

It is OK to assert a still valid fact again, as we have done above. You can use this relation like a normal relation:

[54]:
```
?[state, year, hos] := *hos{state, year, hos}
```

[54]:
```
<pandas.io.formats.style.Styler at 0x26c5d90c810>
```

The curious thing is that it is sorted *descendingly* by year. Validity sorts descendingly.

For any stored relation that has type `Validity` at the *last* slot of the key, the time-travel capability is enabled. Say you have forgotten who the president of the US was in 2019. Easy:

[55]:
```
?[hos, year] := *hos{state: 'US', year, hos @ 2019}
```

[55]:
```
<pandas.io.formats.style.Styler at 0x26c461ce990>
```

In your answer you also got the year that this fact was last known to be true.

You also don't know about the year 2099:

[56]:
```
?[hos, year] := *hos{state: 'US', year, hos @ 2099}
```

```
[56]: <pandas.io.formats.style.Styler at 0x26c5d90dbd0>
```

That certainly doesn't look right. Let's fix it by *retracting* facts on or after 2025, and only inserting them back when we have the sure facts:

```
[57]: ?[state, year, hos] <- [['US', [2025, false], '']]

      :put hos {state, year => hos}
```

```
[57]: <pandas.io.formats.style.Styler at 0x26c5d90c450>
```

As we have hinted, you retract facts by *putting* a retraction. Now let's run the query again:

```
[58]: ?[hos, year] := *hos{state: 'US', year, hos @ 2099}
```

```
[58]: <pandas.io.formats.style.Styler at 0x26c461ecb50>
```

Since the database does not contain facts on or after 2025, your query returns empty.

This functionality is flexible: you can mix different periods in the same query:

```
[59]: ?[hos2018, hos2010] := *hos{state: 'US', hos: hos2018 @ 2018},
                             *hos{state: 'US', hos: hos2010 @ 2010}
```

```
[59]: <pandas.io.formats.style.Styler at 0x26c5d365890>
```

As the relation `hos` is just a normal relation, you can still `rm` facts from it, in which case the facts are irretrievably gone. Whether that's desirable is up to you: the database gives you the choice of how you want to use it, and trusts that you know how to use it correctly for your use case.

We have said above that it is up to you to interpret the "time". There is also a default interpretation with a few more bells and whistles. First let's create a new stored relation to store people's moods:

```
[60]: :create mood {name: String, at: Validity => mood: String}
```

```
[60]: <pandas.io.formats.style.Styler at 0x26c5d8a0850>
```

I want to record my mood now:

```
[61]: ?[name, at, mood] <- [['me', 'ASSERT', 'curious']]
      :put mood {name, at => mood}
```

```
[61]: <pandas.io.formats.style.Styler at 0x26c5d8cda90>
```

Instead of giving a list of two elements as we have done above, we have simply used the string `ASSERT`, and the system will know that we mean an assertion of a *current* fact. What is "current"?

```
[62]: ?[name, at, mood] := *mood{name, at, mood}
```

```
[62]: <pandas.io.formats.style.Styler at 0x26c5d9119d0>
```

It's a big number, but what does it mean? It is the UNIX timestamp at the time this fact was recorded, i.e., *microseconds* since the UNIX epoch:

```
[63]: ?[name, time, mood] := *mood{name, at, mood},
                             time = format_timestamp(at)
```

```
[63]: <pandas.io.formats.style.Styler at 0x26c5d8ab2d0>
```

To query for current facts, use the string NOW in the validity specification:

```
[64]: ?[name, time, mood] := *mood{name, at, mood @ 'NOW'},
                             time = format_timestamp(at)
```

```
[64]: <pandas.io.formats.style.Styler at 0x26c5d924ad0>
```

You can put in facts with manual timestamps as before, so it is possible that your database contains facts about the future. Let's do just that. Instead of giving a mysterious string of numerals, you can use a string for the timestamp:

```
[65]: ?[name, at, mood] <- [['me', '2030-01-01T00:00:00.000+00:00', 'hopeful']]
      :put mood {name, at => mood}
```

```
[65]: <pandas.io.formats.style.Styler at 0x26c5d8f1a10>
```

Since this is in the future, it shouldn't affect NOW:

```
[66]: ?[name, time, mood] := *mood{name, at, mood @ 'NOW'},
                             time = format_timestamp(at)
```

```
[66]: <pandas.io.formats.style.Styler at 0x26c5d8ad850>
```

In this case, there is also END for the validity specification, meaning to extract facts at the end of time:

```
[67]: ?[name, time, mood] := *mood{name, at, mood @ 'END'},
                             time = format_timestamp(at)
```

```
[67]: <pandas.io.formats.style.Styler at 0x26c5d897e10>
```

Retraction at the current timestamp can be done with the string RETRACT:

```
[68]: ?[name, at, mood] <- [['me', 'RETRACT', '']]
      :put mood {name, at => mood}
```

```
[68]: <pandas.io.formats.style.Styler at 0x26c5d928e10>
```

Retraction placed in the future can also be done with stringy timestamps by prefixing with ~:

```
[69]: ?[name, at, mood] <- [['me', '~9999-01-01T00:00:00.000+00:00', 'who cares']]
      :put mood {name, at => mood}
```

```
[69]: <pandas.io.formats.style.Styler at 0x26c5d929190>
```

Now let's look at the complete history:

```
[70]: ?[name, time, is_assert, mood] := *mood{name, at, mood},
                                        time = format_timestamp(at),
                                        is_assert = to_bool(at)
```

```
[70]: <pandas.io.formats.style.Styler at 0x26c5d92b1d0>
```

Pretty neat, isn't it? And this time-travel facility is way faster than what you get if you try to implement it directly with Datalog: see the *note* for more details. Some further technical details of time travel is discussed in its own *chapter*.

```
[71]: ::remove mood, hos
```

```
[71]: <pandas.io.formats.style.Styler at 0x26c5d8afe90>
```

1.3 Extended example: the air routes dataset

Now you have a basic understanding of using the various constructs of Cozo, let's deal with a small real-world dataset, with about 3700 nodes and 57000 edges.

The data we are going to use, and many examples that we will present, are adapted from the book Practical Gremlin[8]. Gremlin is an imperative query language for graphs, a very different take compared to Datalog.

First, let's create the stored relations we want (wrapping queries in braces allows you to execute several queries together atomically):

[52]:
```
{:create airport {
    code: String
    =>
    icao: String,
    desc: String,
    region: String,
    runways: Int,
    longest: Float,
    elev: Float,
    country: String,
    city: String,
    lat: Float,
    lon: Float
}}
{:create country {
    code: String
    =>
    desc: String
}}
{:create continent {
    code: String
    =>
    desc: String
}}
{:create contain { entity: String, contained: String }}
{:create route { fr: String, to: String => dist: Float }}
```

[52]:
```
<pandas.io.formats.style.Styler at 0x2d6696208d0>
```

The next command applies only if you are using Jupyter notebooks: it downloads a JSON file containing the data and imports it into the database. The commented out line shows how to do the same thing with a local file. If you are using the Cozo WASM interface, click the "Import from URL" or the "Import from File" icon, and paste in the address.

[53]:
```
%cozo_import_remote_file 'https://raw.githubusercontent.com/cozodb/cozo/dev/cozo-core/
↪tests/air-routes.json'
# %cozo_import_local_file '../../cozo/cozo-core/tests/air-routes.json'
```

If you are using the `cozo repl` command line, the command to use instead is simply

```
%import https://raw.githubusercontent.com/cozodb/cozo/dev/cozo-core/tests/air-routes.json
```

You can replace the URL with the path to a local file as well. Other environments also have ways to do the same thing: refer to the respective documentations.

[8] https://kelvinlawrence.net/book/Gremlin-Graph-Guide.html

If you feel that the above is too much magic, we will show you the "hard way" of importing the same data at the end of this tutorial. For now let's move on.

Let's verify all the relations we want are there:

```
[54]: ::relations
```

```
[54]: <pandas.io.formats.style.Styler at 0x2d6695623d0>
```

While we are at it, let's lock all these tables to prevent accidentally changing their contents:

```
[55]: ::access_level read_only airport, contain, continent, country, route
```

```
[55]: <pandas.io.formats.style.Styler at 0x2d6695c02d0>
```

More information about what this does is explained in *this chapter*.

Let's just look at some data. Start with airports:

```
[56]: ?[code, city, desc, region, runways, lat, lon] := *airport{code, city, desc, region,␣
      ↪runways, lat, lon}

      :limit 5
```

```
[56]: <pandas.io.formats.style.Styler at 0x2d66960aa50>
```

Airports with the most runways:

```
[57]: ?[code, city, desc, region, runways, lat, lon] := *airport{code, city, desc, region,␣
      ↪runways, lat, lon}

      :order -runways
      :limit 10
```

```
[57]: <pandas.io.formats.style.Styler at 0x2d651f1a8d0>
```

How many airports are there in total?

```
[58]: ?[count(code)] := *airport{code}
```

```
[58]: <pandas.io.formats.style.Styler at 0x2d6695d1450>
```

Let's get a distribution of the initials of the airport codes:

```
[59]: ?[count(initial), initial] := *airport{code}, initial = first(chars(code))

      :order initial
```

```
[59]: <pandas.io.formats.style.Styler at 0x2d66955f650>
```

More useful are the statistics of runways:

```
[60]: ?[count(r), count_unique(r), sum(r), min(r), max(r), mean(r), std_dev(r)] :=
          *airport{runways: r}
```

```
[60]: <pandas.io.formats.style.Styler at 0x2d6695d2050>
```

Using `country`, we can find countries with no airports:

```
[61]: ?[desc] := *country{code, desc}, not *airport{country: code}
```

```
[61]: <pandas.io.formats.style.Styler at 0x2d6695d0ad0>
```

The `route` relation by itself is rather boring:

```
[62]: ?[fr, to, dist] := *route{fr, to, dist}

      :limit 10
```

```
[62]: <pandas.io.formats.style.Styler at 0x2d6695e7890>
```

It just records the starting and ending airports of each route, together with the distance. This relation only becomes useful when used as a graph.

Airports with no routes:

```
[63]: ?[code, desc] := *airport{code, desc}, not *route{fr: code}, not *route{to: code}
```

```
[63]: <pandas.io.formats.style.Styler at 0x2d6695c0550>
```

Airports with the most out routes:

```
[64]: route_count[fr, count(fr)] := *route{fr}
      ?[code, n] := route_count[code, n]

      :sort -n
      :limit 5
```

```
[64]: <pandas.io.formats.style.Styler at 0x2d669569fd0>
```

How many routes are there from the European Union to the US?

```
[65]: routes[unique(r)] := *contain['EU', fr],
                           *route{fr, to},
                           *airport{code: to, country: 'US'},
                           r = [fr, to]
      ?[n] := routes[rs], n = length(rs)
```

```
[65]: <pandas.io.formats.style.Styler at 0x2d66963b3d0>
```

How many airports are there in the US with routes from the EU?

```
[66]: ?[count_unique(to)] := *contain['EU', fr],
                             *route{fr, to},
                             *airport{code: to, country: 'US'}
```

```
[66]: <pandas.io.formats.style.Styler at 0x2d6696545d0>
```

How many routes are there for each airport in London, UK?

```
[67]: ?[code, count(code)] := *airport{code, city: 'London', region: 'GB-ENG'}, *route{fr:␣
      ↪code}
```

```
[67]: <pandas.io.formats.style.Styler at 0x2d669654610>
```

We need to specify the region, because there is another city called London, not in the UK.

How many airports are reachable from London, UK in two hops?

```
[68]: lon_uk_airports[code] := *airport{code, city: 'London', region: 'GB-ENG'}
      one_hop[to] := lon_uk_airports[fr], *route{fr, to}, not lon_uk_airports[to];
      ?[count_unique(a3)] := one_hop[a2], *route{fr: a2, to: a3}, not lon_uk_airports[a3];
```

```
[68]: <pandas.io.formats.style.Styler at 0x2d669568cd0>
```

What are the cities directly reachable from LGW (London Gatwick), but furthermost away?

```
[69]: ?[city, dist] := *route{fr: 'LGW', to, dist},
                       *airport{code: to, city}
      :order -dist
      :limit 10
```

```
[69]: <pandas.io.formats.style.Styler at 0x2d6699b5990>
```

What airports are within 0.1 degrees of the Greenwich meridian?

```
[70]: ?[code, desc, lon, lat] := *airport{lon, lat, code, desc}, lon > -0.1, lon < 0.1
```

```
[70]: <pandas.io.formats.style.Styler at 0x2d6699b6450>
```

Airports in a box drawn around London Heathrow, UK:

```
[71]: h_box[lon, lat] := *airport{code: 'LHR', lon, lat}
      ?[code, desc] := h_box[lhr_lon, lhr_lat], *airport{code, lon, lat, desc},
                       abs(lhr_lon - lon) < 1, abs(lhr_lat - lat) < 1
```

```
[71]: <pandas.io.formats.style.Styler at 0x2d6695d2090>
```

For some spherical geometry: what is the angle subtended by SFO and NRT on the surface of the earth?

```
[72]: ?[deg_diff] := *airport{code: 'SFO', lat: a_lat, lon: a_lon},
                     *airport{code: 'NRT', lat: b_lat, lon: b_lon},
                     deg_diff = rad_to_deg(haversine_deg_input(a_lat, a_lon, b_lat, b_lon))
```

```
[72]: <pandas.io.formats.style.Styler at 0x2d6695f7490>
```

We mentioned before that aggregations in Cozo are powerful. They are powerful because they can be used in recursions (some restrictions apply).

Let's find the distance of the *shortest route* between two airports. One wayis to enumerate all the routes between the two airports, and then apply `min` aggregation to the results. This cannot be implemented as stated, since the routes may contain cycles and hence there can be an infinite number of routes between two airports.

Instead, think recursively. If we already have all the shortest routes between all nodes, we derive an *equation* satisfied by the shortest route: the shortest route between `a` and `b` is either the distance of a direct route, or the sum of the shortest distance from `a` to `c` and the distance of a direct route from `c` to `d`. We apply our `min` aggregation to this recursive set instead.

Write it out and it works. For exmaple, the shortest routes between the airports `LHR` and `YPO`:

```
[73]: shortest[b, min(dist)] := *route{fr: 'LHR', to: b, dist}
                              # Start with the airport 'LHR', retrieve a direct route from
      ↪'LHR' to b

      shortest[b, min(dist)] := shortest[c, d1], # Start with an existing shortest route from
      ↪'LHR' to c
```

(continues on next page)

(continued from previous page)

```
                            *route{fr: c, to: b, dist: d2},  # Retrieve a direct route␣
↪from c to b
                            dist = d1 + d2 # Add the distances

?[dist] := shortest['YPO', dist] # Extract the answer for 'YPO'.
                                 # We chose it since it is the hardest airport to get to␣
↪from 'LHR'.
```

[73]:
```
<pandas.io.formats.style.Styler at 0x2d669656090>
```

There is a caveat when you try to write similar queries. Say you try to write it in the following way (don't try to run it):

```
shortest[a, b, min(dist)] := *route{fr: a, to: b, dist}
shortest[a, b, min(dist)] := shortest[a, c, d1],
                            *route{fr: c, to: b, dist: d2},
                            dist = d1 + d2

?[dist] := shortest['LHR', 'YPO', dist]
```

You will find that the query does not complete in a reasonable amount of time, despite it being equivalent to the original query. Why?

In the changed query, you are asking the database to compute the all-pair shortest path, and then extract the answer to a particular shortest path. Normally Cozo would apply a technique called *magic set rewrite* so that only the needed answer would be calculated. However, in the changed query the presence of the aggregation operator `min` prevents that. In this case, applying the rewrite to the variable `a` would still yield the correct answer, but rewriting in any other way would give complete nonsense, and in the more general case with recursive aggregations this is a can of worms.

So as explained in the *chapter about execution*, magic set rewrites are only applied to rules without aggregations or recursions for the moment, until we are sure of the exact conditions under which the rewrites are safe. So for now at least the database executes the query as written, computing the result of the `shortest` rule containing more than ten million rows (to be exact, 3700 * 3700 = 13,690,000 rows) first!

The bottom line is, be mindful of the cardinality of the return sets of recursive rules.

1.3.1 A tour of graph algorithms

Now let's investigate the graph using some graph algorithms. As we have mentioned before, the Cozo running in browsers through WASM does not have the graph algorithms module enabled, so to run the following examples you will need to use some other implementation (for example, the Python one[9]).

Since path-finding is such a common operation on graphs, Cozo has several fixed rules for that:

[74]:
```
starting[] <- [['LHR']]
goal[] <- [['YPO']]
?[starting, goal, distance, path] <~ ShortestPathDijkstra(*route[], starting[], goal[])
```

[74]:
```
<pandas.io.formats.style.Styler at 0x2d6696614d0>
```

Not only is it more efficient, but we also get a path for the shortest route.

Not content with the shortest path, the following calculates ten the shortest paths:

[9] https://github.com/cozodb/pycozo

[75]:
```
starting[] <- [['LHR']]
goal[] <- [['YPO']]
?[starting, goal, distance, path] <~ KShortestPathYen(*route[], starting[], goal[], k:␣
↪10)
```

[75]:
```
<pandas.io.formats.style.Styler at 0x2d6695691d0>
```

If efficiency is really important to you, you can use the A* algorithm with a good heuristic function:

[76]:
```
code_lat_lon[code, lat, lon] := *airport{code, lat, lon}
starting[code, lat, lon] := code = 'LHR', *airport{code, lat, lon};
goal[code, lat, lon] := code = 'YPO', *airport{code, lat, lon};
?[] <~ ShortestPathAStar(*route[],
                         code_lat_lon[node, lat1, lon1],
                         starting[],
                         goal[goal, lat2, lon2],
                         heuristic: haversine_deg_input(lat1, lon1, lat2, lon2) * 3963);
```

[76]:
```
<pandas.io.formats.style.Styler at 0x2d6699b7650>
```

There's a lot more setup required in this case: we need to retrieve the latitudes and longitudes of airports and do processing on them first. The number `3963` above is the radius of the earth in miles.

The most important airports, by PageRank:

[77]:
```
rank[code, score] <~ PageRank(*route[a, b])
?[code, desc, score] := rank[code, score], *airport{code, desc}

:limit 10;
:order -score
```

[77]:
```
<pandas.io.formats.style.Styler at 0x2d6696224d0>
```

The following example takes a long time to run since it calculates the betweenness centrality: up to a few seconds, depending on your machine. Algorithms for calculating the betweenness centrality have high complexity.

[78]:
```
centrality[code, score] <~ BetweennessCentrality(*route[a, b])
?[code, desc, score] := centrality[code, score], *airport{code, desc}

:limit 10;
:order -score
```

[78]:
```
<pandas.io.formats.style.Styler at 0x2d6695f6110>
```

These are the airports that, if disconnected from the network, cause the most disruption. As this example shows, some of the algorithms really struggle when you go beyond small or medium sized dataset.

Community detection can collapse a graph into a *supergraph*. Here we store the result, since it has too many rows to display nicely:

[79]:
```
community[detailed_cluster, code] <~ CommunityDetectionLouvain(*route[a, b])
?[code, cluster, detailed_cluster] := community[detailed_cluster, code], cluster =␣
↪first(detailed_cluster)

:replace community {code => cluster, detailed_cluster}
```

[79]:
```
<pandas.io.formats.style.Styler at 0x2d66ba1f650>
```

We can look at the supernodes containing specific nodes. For example, the supernode for London Gatwick consists of mainly UK and other European airports, as you would expect:

[80]:
```
community[code] := *community{code: 'LGW', cluster}, *community{code, cluster}
?[country, count(code)] :=
    community[code],
    *airport{code, desc, country: country_code},
    *country{code: country_code, desc: country},

:order -count(code)
:limit 5
```

[80]:
```
<pandas.io.formats.style.Styler at 0x2d66ba1e110>
```

For `JFK` on the other hand, its supernode consists of mainly US airports:

[81]:
```
community[code] := *community{code: 'JFK', cluster}, *community{code, cluster}
?[country, count(code)] :=
    community[code],
    *airport{code, desc, country: country_code},
    *country{code: country_code, desc: country},

:order -count(code)
:limit 5
```

[81]:
```
<pandas.io.formats.style.Styler at 0x2d669622f50>
```

But it does not always work according to geography. For example, Frankfurt airport is in Germany:

[82]:
```
?[desc, country_desc] := *airport{code: 'FRA', desc, country: country_code}, *country
↪{code: country_code, desc: country_desc}
```

[82]:
```
<pandas.io.formats.style.Styler at 0x2d651f11f90>
```

But its supernode:

[83]:
```
community[code] := *community{code: 'FRA', cluster}, *community{code, cluster}
?[country, count(code)] :=
    community[code],
    *airport{code, desc, country: country_code},
    *country{code: country_code, desc: country},

:order -count(code)
:limit 5
```

[83]:
```
<pandas.io.formats.style.Styler at 0x2d66ba1fe90>
```

Germany does not even appear in the top five. In fact, `FRA` is in the same supernode as `JFK`. What matters is the connectivity in the graph, not the geography. As another example:

[84]:
```
community[code] := *community{code: 'SIN', cluster}, *community{code, cluster}
?[country, count(code)] :=
    community[code],
    *airport{code, desc, country: country_code},
```

(continues on next page)

(continued from previous page)

```
    *country{code: country_code, desc: country},

:order -count(code)
:limit 5
```

[84]:
```
<pandas.io.formats.style.Styler at 0x2d66963fa10>
```

You'd expect `SIN` to be a Chinese airport. Wrong:

[85]:
```
?[desc, country_desc] := *airport{code: 'SIN', desc, country: country_code}, *country
↪{code: country_code, desc: country_desc}
```

[85]:
```
<pandas.io.formats.style.Styler at 0x2d6695d1ad0>
```

Finally, let's collapse the `route` relation into `super_route`:

[86]:
```
?[fr_cluster, to_cluster, count(dist), sum(dist)] := *route{fr, to, dist},
                                                     *community{code: fr, cluster: fr_
↪cluster},
                                                     *community{code: to, cluster: to_
↪cluster}
:replace super_route {fr_cluster, to_cluster => n_routes=count(dist), total_
↪distance=sum(dist)}
```

[86]:
```
<pandas.io.formats.style.Styler at 0x2d669620210>
```

As expected, the "diagonals" where `fr_cluster == to_cluster` are larger in the `super_route` graph:

[87]:
```
?[fr_cluster, to_cluster, n_routes, total_distance] := *super_route{fr_cluster, to_
↪cluster, n_routes, total_distance}, fr_cluster < 2
```

[87]:
```
<pandas.io.formats.style.Styler at 0x2d66baecd90>
```

Now the super graph is small enough that all analytics algorithms return instantly:

[88]:
```
?[cluster, score] <~ PageRank(*super_route[])
:order -score
:limit 5
```

[88]:
```
<pandas.io.formats.style.Styler at 0x2d669656710>
```

You can now go on to investigate the supernodes, give real-world interpretations to them, etc. For example, a naïve interpretation of the above PageRank result is that North America is (still) the most prosperous part of the world, followed by East Asia in the second place, South Asia in the third place, and Europe in the fourth place.

1.4 Importing dataset the hard way

Previously, we imported the air-routes dataset by using Python under the hood to download a specially-crafted JSON file and feed it to the database. Here we learn how to achieve the same effect by letting Cozo fetch and import a series of CSV files, without Python's help.

Let's set the database magic up first:

[89]:
```
::access_level normal airport, contain, continent, country, route
```

[89]:
```
<pandas.io.formats.style.Styler at 0x2d6696639d0>
```

[92]:
```
::remove airport, contain, continent, country, route, community, super_route
```

[92]:
```
<pandas.io.formats.style.Styler at 0x2d669654290>
```

Next, some parameters to make life eaiser (the lines commented out do the same thing by processing local files):

[94]:
```
%cozo_set AIR_ROUTES_NODES_URL 'https://raw.githubusercontent.com/cozodb/cozo/dev/cozo-
↪core/tests/air-routes-latest-nodes.csv'
%cozo_set AIR_ROUTES_EDGES_URL 'https://raw.githubusercontent.com/cozodb/cozo/dev/cozo-
↪core/tests/air-routes-latest-edges.csv'
# %cozo_set AIR_ROUTES_NODES_URL 'file://./../../cozo/cozo-core/tests/air-routes-latest-
↪nodes.csv'
# %cozo_set AIR_ROUTES_EDGES_URL 'file://./../../cozo/cozo-core/tests/air-routes-latest-
↪edges.csv'
```

First, import the `airport` relation:

[95]:
```
res[idx, label, typ, code, icao, desc, region, runways, longest, elev, country, city,␣
↪lat, lon] <~
    CsvReader(types: ['Int', 'Any', 'Any', 'Any', 'Any', 'Any', 'Any', 'Int?', 'Float?',
↪'Float?', 'Any', 'Any', 'Float?', 'Float?'],
              url: $AIR_ROUTES_NODES_URL,
              has_headers: true)

?[code, icao, desc, region, runways, longest, elev, country, city, lat, lon] :=
    res[idx, label, typ, code, icao, desc, region, runways, longest, elev, country, city,
↪ lat, lon],
    label == 'airport'

:replace airport {
    code: String
    =>
    icao: String,
    desc: String,
    region: String,
    runways: Int,
    longest: Float,
    elev: Float,
    country: String,
    city: String,
    lat: Float,
    lon: Float
}
```

[95]:
```
<pandas.io.formats.style.Styler at 0x2d66968fe90>
```

The `CsvReader` utility downloads a CSV file from the internet and attempts to parse its content into a relation. When we store the relation, we specified types for the columns. The `code` column acts as a primary key for the `airport` stored relation.

Next is `country`:

[96]:
```
res[idx, label, typ, code, icao, desc] <~
    CsvReader(types: ['Int', 'Any', 'Any', 'Any', 'Any', 'Any'],
              url: $AIR_ROUTES_NODES_URL,
              has_headers: true)
?[code, desc] :=
    res[idx, label, typ, code, icao, desc],
    label == 'country'

:replace country {
    code: String
    =>
    desc: String
}
```

[96]:
```
<pandas.io.formats.style.Styler at 0x2d668c03450>
```

continent:

[97]:
```
res[idx, label, typ, code, icao, desc] <~
    CsvReader(types: ['Int', 'Any', 'Any', 'Any', 'Any', 'Any'],
              url: $AIR_ROUTES_NODES_URL,
              has_headers: true)
?[idx, code, desc] :=
    res[idx, label, typ, code, icao, desc],
    label == 'continent'

:replace continent {
    code: String
    =>
    desc: String
}
```

[97]:
```
<pandas.io.formats.style.Styler at 0x2d66968d250>
```

We need to make a translation table for the indices the data use:

[98]:
```
res[idx, label, typ, code] <~
    CsvReader(types: ['Int', 'Any', 'Any', 'Any'],
              url: $AIR_ROUTES_NODES_URL,
              has_headers: true)
?[idx, code] :=
    res[idx, label, typ, code],

:replace idx2code { idx => code }
```

[98]:
```
<pandas.io.formats.style.Styler at 0x2d6695dfb10>
```

The `contain` relation contains information on the geographical inclusion of entities:

[99]:
```
res[] <~
    CsvReader(types: ['Int', 'Int', 'Int', 'String'],
              url: $AIR_ROUTES_EDGES_URL,
              has_headers: true)
?[entity, contained] :=
    res[idx, fr_i, to_i, typ],
```

(continues on next page)

(continued from previous page)

```
    typ == 'contains',
    *idx2code[fr_i, entity],
    *idx2code[to_i, contained]


:replace contain { entity: String, contained: String }
```

[99]:
```
<pandas.io.formats.style.Styler at 0x2d66968ffd0>
```

Finally, the `routes` between the airports. This relation is much larger than the rest and contains about 60k rows:

[100]:
```
res[] <~
    CsvReader(types: ['Int', 'Int', 'Int', 'String', 'Float?'],
              url: $AIR_ROUTES_EDGES_URL,
              has_headers: true)
?[fr, to, dist] :=
    res[idx, fr_i, to_i, typ, dist],
    typ == 'route',
    *idx2code[fr_i, fr],
    *idx2code[to_i, to]

:replace route { fr: String, to: String => dist: Float }
```

[100]:
```
<pandas.io.formats.style.Styler at 0x2d66968fc10>
```

We no longer need the `idx2code` relation:

[101]:
```
::remove idx2code
```

[101]:
```
<pandas.io.formats.style.Styler at 0x2d669656310>
```

Let's verify all the relations we want are there:

[102]:
```
::relations
```

[102]:
```
<pandas.io.formats.style.Styler at 0x2d66baee7d0>
```

That's it for the tutorial!

CHAPTER

TWO

QUERIES

CozoScript, a Datalog[10] 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.

[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 (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. If some of the columns do not need to be bound, you can use the special underscore variable `_`: it does not take part in any unifications.

You can also bind columns by name:

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

In this form, any number of columns may be omitted, and columns may come in any order. 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

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 exactly 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 `0` 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* (see *this chapter*):

```
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 a *separate chapter*. 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 (i.e., early stopping).

`: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. Defaults to 300 seconds. If you want to disable the timeout, set it to 0.

`: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.

`: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.

`:insert <NAME> <SPEC>`

Insert rows from the resulting relation into the named stored relation. If keys from the data exist beforehand, an error is raised.

`:update <NAME> <SPEC>`

Update rows in the named stored relation. Only keys and any non-keys that you want to update should be specified in <SPEC>, the other non-keys will keep their old values. Updating a non-existent key is an error.

`:delete <NAME> <SPEC>`

Delete rows from the named stored relation. Only keys and any non-keys that you want to delete should be specified in <SPEC>, the other non-keys will keep their old values. Deleting a non-existent key raises an error.

`: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.

`: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.

`:returning`

When used in conjunction with the mutation ops `:put`, `:rm`, `:insert`, `:update` and `:delete`, instead of returning a status code, the mutated rows are returned as a relation. The schema of the returned rows follows the schema of the stored relation, with a special field `_kind` added to the front. `_kind` can be `"inserted"` or `"replaced"` for `:put`, `:insert` and `:update`, and `"requested"` and `"deleted"` for `:rm` and `:delete`. For deletion, the non-key fields for `"requested"` rows are filled with `null`.

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

3.1.1 Create and replace

The format of <SPEC> is identical for all 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),
```

(continues on next page)

(continued from previous page)

```
    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,
    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, update, 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 `:update`, you must specify all keys and all columns that you want to update.

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}
```

(continues on next page)

(continued from previous page)

```
}
{
    ?[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.

There is actually a mini-language hidden behind query chains. What you have seen above consists of a number of simple *query expressions*, each expression is a complete query enclosed in braces, and the return value is the value of the last expression. There are other constructs as well:

- `%if <cond> %then ... (%else ...) %end` for conditional execution. There is also a negated form beginning with `%if_not`. The `<cond>` part is either a query expression or an ephemeral relation. Either way, the condition ends up being a relation, and a relation is considered falsy if the relation contains no rows and truthy otherwise.
- `%loop ... %end` for looping, you can use `%break` and `%continue` inside the loop. You can prefix the loop with `%mark <marker>`, and use `%break <marker>` or `%continue marker` to jump sereral levels.
- `%return <query expression or ephemeral relation, or empty>` for early termination.
- `%debug <ephemeral relation>` for printing ephemeral relations to standard output.
- `%ignore_error <query expression>` executes the query expresison, but eats any error raised and continue.
- `%swap <ephemeral relation> <another ephemeral relation>` swaps two ephemeral relations.

What is the *ephemeral relation* mentioned above? This is a relation that can only be seen within the transaction and which is gone when the transaction ends (hence it is useless in singleton queries). It is created and used in the same way as stored relations, but with names starting with the underscore `_`. You can think of them as variables in the chain query mini-language.

Let's see several examples:

```
{:create _test {a}}

%loop
    %if { len[count(x)] := *_test[x]; ?[x] := len[z], x = z >= 10 }
        %then %return _test
    %end
    { ?[a] := a = rand_uuid_v1(); :put _test {a} }
%end
```

The return relation of this query consists of ten random rows. Note that in this example, you *must not* use a constant rule when generating the random value: the body of a constant rule is evaluated to a constant only *once*, which will make the query loop forever.

Another one:

```
{?[a] <- [[1], [2], [3]]; :replace _test {a}}

%loop
    { ?[a] := *_test[a]; :limit 1; :rm _test {a} }
    %debug _test
```

(continues on next page)

(continued from previous page)

```
    %if_not _test
    %then %break
    %end
%end

%return _test
```

The return relation of this query is empty (very contrived way of removing elements).

Finally:

```
{?[a] <- [[1], [2], [3]]; :replace _test {a}}
{?[a] <- []; :replace _test2 {a}}
%swap _test _test2
%return _test
```

The return relation of this query is empty as well, since the two ephemeral relations have been swapped.

For any query occrurring in script, you can postfix it with `as <name>` where name is an identifier starting with the underscore, and the result of the query will be stored in an ephemeral relation with the given name. The ephemeral relation is created as if there is an `:replace` directive.

We use this functionality to run ad-hoc iterative queries. As the basic query language is already Turing complete, you can actually write any algorithm without this mini-language, but the way of writing may be very contrived. Try implementing PageRank with basic query. You will end up with many recursive aggregations. Next try with chained queries. A breeze.

3.2.1 Multi-statement transaction

Cozo also supports multi-statement in the hosting language for selected libraries (currently Rust, Python, NodeJS) and the standalone executable. The way to use it is to request a transaction first, do your queries and mutations against the transaction, and finally commit or abort the transaction. This is more flexible than using the chaining query mini-language, but is specific to each hosting environment. Please refer to the respective documentations of the environments.

3.3 Indices

Since version 0.5, it is possible to create indices on stored relations. In Cozo, indices are simply reordering of columns of the original stored relation. As an example, let's say we have a relation

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

but we often want to run queries like `?[a] := *r{a, b: $value}`. Without indiecs, this will result in a full-scan. In this case we can do:

```
::index create r:idx {b, a}
```

You do *not* specify functional dependencies when creating indices (and in this case there are none anyway).

In Cozo, indices are read-only stored relations that you can query directly:

```
?[a] := *r:idx {a, b: $value}
```

In this case, running the original query will also use the index, and hence is equivalent to the explicit form (which you can confirm with `::explain`). However, Cozo is very conservative in using indices in that if there is any chance that the use of an index might decrease performance, then Cozo will not use an index. Currently, this means that only in situations when using an index can avoid a full-scan will the index be used. This behaviour ensures that you will not need to fight against suboptimal use of indices with difficult tricks: just be explicit.

To drop an index:

```
::index drop r:idx
```

In Cozo, you do not need to specify all columns when creating an index, and the database will complete the specified columns to a key. This means that if your stored relation is

```
:create r {a, b => c, d, e}
```

and you created an index as:

```
::index create r:i {d, b}
```

the database will automatically run the following index creation instead:

```
::index create r:i {d, b, a}
```

You can see what columns are actually created by running `::columns r:i`.

Indices can be used as inputs to fixed rules. They may also be eligible in time-travel queries, as long as their last key column is of type `Validity`.

3.4 Triggers

Cozo supports triggers attached to stored relations. 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 manually, suppose we have the following relation:

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

and the manual index is:

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

To manage the manual index automatically:

```
::set_triggers rel

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

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

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

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

Note that unlike indices, there are ingestion APIs for which triggers are explicitly *not* run. Also, if you want to manually manage indices with triggers, you have to populate the existing values manually as well.

Warning: Triggers do not propagate. That is, if a trigger modifies a relation that has triggers associated, those latter triggers will not run. This is different from the behaviour in earlier versions. We changed it since trigger propagation creates more problems than it solves.

3.5 Storing large values

There a limit to the amount of data you can store in a single value or single row. The precise limit depends on the storage engine. For the in-memory engine it is obviously RAM-bound. For the SQLite engine the keys as as whole and the values as a whole are each stored as a single BLOB field in SQLite, and are subject to their limit[11]. For RocksDB engine, which is the recommended setup if you are thinking of storing large values, the keys as a whole is stored as a RocksDB key, which has a limit of 8MB, and keys should be kept small for performance. For values, CozoDB utilizes the BlobDB[12] functionality of RocksDB, and you are only limited by RAM and disk sizes.

Performance-wise, if large values are present, currently these values will be read into memory if the row is touched in the query. So it is recommended to store large values in a dedicated key-value relation in the database, with all the metadata stored in a separate relation. At query time, you should search/filter/join the metadata relation to find the rows you want, and then join them with the dedicated large value relation at the last stage.

[11] https://www.sqlite.org/limits.html
[12] https://github.com/facebook/rocksdb/wiki/BlobDB

CHAPTER

FOUR

PROXIMITY SEARCHES

These kinds of proximity indices allow Cozo to perform fast searches for similar data. The HNSW index is a graph-based index that allows for fast approximate nearest neighbor searches. The MinHash-LSH index is a locality sensitive hash index that allows for fast approximate nearest neighbor searches. The FTS index is a full-text search index that allows for fast string matches.

4.1 HNSW (Hierarchical Navigable Small World) indices for vectors

Cozo supports vector proximity search using the HNSW (Hierarchical Navigable Small World) algorithm.

To use vector search, you first need to have a stored relation with vectors inside, for example:

```
:create table {k: String => v: <F32; 128>}
```

Next you create a HNSW index on a table containing vectors. You use the following system operator to create the index:

```
::hnsw create table:index_name {
    dim: 128,
    m: 50,
    dtype: F32,
    fields: [v],
    distance: L2,
    ef_construction: 20,
    filter: k != 'foo',
    extend_candidates: false,
    keep_pruned_connections: false,
}
```

The parameters are as follows:

- The dimension `dim` and the data type `dtype` (defaults to F32) has to match the dimensions of any vector you index.
- The `fields` parameter is a list of fields in the table that should be indexed.
- The indexed fields must only contain vectors of the same dimension and data type, or `null`, or a list of vectors of the same dimension and data type.
- The `distance` parameter is the distance metric to use: the options are `L2` (default), `Cosine` and `IP`.
- The `m` controls the maximal number of outgoing connections from each node in the graph.
- The `ef_construction` parameter is the number of nearest neighbors to use when building the index: see the HNSW paper for details.

- The `filter` parameter, when given, is bound to the fields of the original relation and only those rows for which the expression evaluates to `true` are indexed.
- The `extend_candidates` parameter is a boolean (default false) that controls whether the index should extend the candidate list with the nearest neighbors of the nearest neighbors.
- The `keep_pruned_connections` parameter is a boolean (default false) that controls whether the index should keep pruned connections.

You can insert data as normally done into `table`. For vectors, use a list of numbers and it will be verified to have the correct dimension and converted. If you want to be more explicit, you can use the `vec` function.

After the index is created, you can use vector search inside normal queries in a similar manner to stored relations. For example:

```
?[dist, k, v] := ~table:index_name{ k, v |
        query: q,
        k: 2,
        ef: 20,
        bind_distance: dist,
        bind_vector: bv,
        bind_field: f,
        bind_field_idx: fi,
        filter: 1 + 1 == 2,
        radius: 0.1
    }, q = vec([200, 34])
```

The `~` followed by the index name signifies a vector search. In the braces, arguments before the vertical line are named bindings, with exactly the same semantics as in normal stored relations with named fields (i.e. they may be bound, or if they are unbound, the introduce fresh variables), and arguments after the vertical line are query parameters.

There are three required parameters: `query` is an expression that evaluates to a query vector of the expected type, and if it evaluates to a variable, the variable must be bound inside the rule; `k` controls how many results to return, and `ef` controls the number of neighbours to consider during the search process.

Next, there are three bind parameters that can bind variables to data that are only available in index or during the search process: `distance` binds the distance between the query vector and the result vector; `vector` binds the result vector; and `field` binds the field name of the result vector. The `field_idx` parameter binds the index of the field in the `fields` list of the index in case `field` resolves to a list of vectors, otherwise it is `null`. In case any of the bind parameters are bound to existing variables, they act as filters after `k` results are returned.

The parameter `filter` takes an expression that can only refer to the fields of the original relation, and only those rows for which the expression evaluates to `true` are returned, and this filtering results occurs during the search process, so the algorithm will strive to return `k` results even if it must filter out a larger number of rows. `radius` controls the largest distance any return vector can have from the query vector, and this filtering process also happens during the search.

The vector search can be used in any place where a stored relation may be used, even inside recursive rules (but be careful of non-termination).

As with normal indices, you can use the index relation as a read-only but otherwise normal relation in your query. You query the index directly by:

```
?[fr_k, to_k, dist] := *table:index_name {layer: 0, fr_k, to_k, dist}
```

It is recommended to always specify `layer`, otherwise a full scan is required.

The schema for the above index is the following:

```
{
    layer: Int,
    fr_k: String?,
    fr__field: Int?,
    fr__field_idx: Int?,
    to_k: String?,
    to__field: Int?,
    to__field_idx: Int?,
    =>
    dist: Float,
    hash: Bytes,
    ignore_link: Bool,
}
```

Layer is the layer in the HNSW hierarchy of graphs, with `0` the most detailed layer, `-1` the layer more abstract than `0`, `-2` the even more abstract layer, etc. There is also a special layer `1` containing at most one row with all other keys set to null.

The `fr_*` and `to_*` fields mirror the indices of the indexed relation, and the `fr__*` and `to__*` fields indicate which vectors inside the original rows this edge connects.

`dist` is the distance between the two vectors when the row represents a link between two different vectors, otherwise the link is a self-loop and `dist` contains the degree of the node; `hash` is the hash of the vector, and `ignore_link` is a boolean that indicates whether this link should be ignored during the search process. The graph is guaranteed to be symmetric, but the incoming and outgoing links may have different `ignore_link` values, and they cannot both be `true`.

Walking the index graph at layer 0 amounts to probabilistically visiting "near" neigbours. More abstract layers are renormalized versions of the proximity graph and are harder to work with but are even more interesting theoretically.

To drop an HNSW index:

```
::hnsw drop table:index_name
```

4.2 MinHash-LSH for near-duplicate indexing of strings and lists

To use locality sensitive search on a relation containing string values, for example:

```
:create table {k: String => v: String?}
```

You can create a MinHash-LSH index on the `v` field by:

```
::lsh create table:index_name {
    extractor: v,
    extract_filter: !is_null(v),
    tokenizer: Simple,
    filters: [],
    n_perm: 200,
    target_threshold: 0.7,
    n_gram: 3,
    false_positive_weight: 1.0,
    false_negative_weight: 1.0,
}
```

This creates a MinHash-LSH index on the `v` field of the table. The index configuration includes the following parameters:

- `extractor: v` specifies that the `v` field will be used as the feature extractor. This parameter takes an expression, which must evaluate to a string, a list of values to be indexed, or `null`. If it evaluates to `null`, then the row is not indexed.
- `extract_filter: !is_null(v)`: this is superfluous in this case, but in more general situations you can use this to skip indexing rows based on arbitary logic.
- `tokenizer: Simple` and `filters: []` specifies the tokenizer to be used, see a later section for tokenizer.
- `n_perm: 200` sets the number of permutations for the MinHash LSH index. Higher values will result in more accurate results at the cost of increased CPU and storage usage.
- `target_threshold: 0.7` sets the target threshold for similarity comparisons when searching.
- `n_gram: 3` sets the size of the n-gram used for shingling[13].
- `false_positive_weight: 1.0` and `false_negative_weight: 1.0` set the weights for false positives and false negatives.

At search time:

```
?[k, v] := ~table:index_name {k, v |
    query: $q,
    k: 2,
    filter: 1 + 1 == 2,
}
```

This will look for the top 2 most similar values to the query `q`. The `filter` parameter is evaluated on the bindings for the relation, and only those rows for which the filter evaluates to `true` are returned, before restricting to `k` results. The `query` parameter is a string, and will be subject to the same tokenization process.

In addition to strings, you can index and search for list of arbitrary values. In this case, the `tokenizer`, `filters` and `n_gram` parameters are ignored.

Again you can use the associated index relation as a normal relations in your query. There are two now: `table:index_name` and `table:index_name:inv`. You can use `::columns` to look at their structure. In our case, the first is:

```
{
    hash: Bytes,
    src_k: String,
}
```

and the second is:

```
{
    k: String => minhash: List[Bytes]
}
```

The first it more useful: it loosely groups together duplicates according to the indexing parameters.

To drop:

```
::lsh drop table:index_name
```

[13] https://en.wikipedia.org/wiki/W-shingling

4.3 Full-text search (FTS)

Full-text search should be familiar. For the following relation:

```
:create table {k: String => v: String?}
```

we can create an FTS index by:

```
::fts create table:index_name {
    extractor: v,
    extract_filter: !is_null(v),
    tokenizer: Simple,
    filters: [],
}
```

This creates an FTS index on the `v` field of the table. The index configuration includes the following parameters:

- `extractor: v` specifies that the `v` field will be used as the feature extractor. This parameter takes an expression, which must evaluate to a string or `null`. If it evaluates to `null`, then the row is not indexed.
- `extract_filter: !is_null(v)`: this is superfluous in this case, but in more general situations you can use this to skip indexing rows based on arbitary logic.
- `tokenizer: Simple` and `filters: []` specifies the tokenizer to be used, see a later section for tokenizer.

That's it. At query time:

```
?[s, k, v] := ~table:index_name {k, v |
    query: $q,
    k: 10,
    filter: 1 + 1 == 2,
    score_kind: 'tf_idf',
    bind_score: s
}

:order -s
```

This query retrieves the top 10 results from the index `index_name` based on a search query `$q`. The `filter` parameter can be used to filter the results further based on additional conditions. The `score_kind` parameter specifies the scoring method, and in this case, it is set to `'tf_idf'` which takes into consideration of global statistics when scoring documents. You can also use `'tf'`. The resulting scores are bound to the variable `s`. Finally, the results are ordered in descending order of score (`-s`).

The search query must be a string and is processed by the same tokenizer as the index. The tokenizer is specified by the `tokenizer` parameter, and the `filters` parameter can be used to specify additional filters to be applied to the tokens. There is a mini-language for parsing the query:

- `hello world`, `hello AND world`, `"hello" AND 'world'`: these all look for rows where both words occur. `AND` is case sensitive.
- `hello OR world`: look for rows where either word occurs.
- `hello NOT world`: look for rows where `hello` occurs but `world` does not.
- `hell* wor*`: look for rows having a word starting with `hell` and also a word starting with `wor`.
- `NEAR/3(hello world bye)`: look for rows where `hello`, `world`, `bye` are within 3 words of each other. You can write `NEAR(hello world bye)` as a shorthand for `NEAR/10(hello world bye)`.

- `hello^2 OR world`: look for rows where `hello` or `world` occurs, but `hello` has twice of its usual weighting when scoring.
- These can be combined and nested with parentheses (except that `NEAR` only takes literals and prefixes): `hello AND (world OR bye)`.

The index relation has the following schema:

```
{
    word: String,
    src_k: String,
    =>
    offset_from: List[Int],
    offset_to: List[Int],
    position: List[Int],
    total_length: Int,
}
```

Explanation of the fields:

- `word`: the word that occurs in the document.
- `src_k`: the key of the document, the name and number varies according to the original relation schema.
- `offset_from`: the starting offsets of the word in the document.
- `offset_to`: the ending offsets of the word in the document.
- `position`: the position of the word in the document, counted as the position of entire tokens.
- `total_length`: the total number of tokens in the document.

To drop:

```
::fts drop table:index_name
```

4.4 Text tokenization and filtering

Text tokenization and filtering are used in both the MinHash-LSH and FTS indexes. The tokenizer is specified by the `tokenizer` parameter, and the `filters` parameter can be used to specify additional filters to be applied to the tokens.

CozoDB uses Tantivy's[14] tokenizers and filters (we incorporated their files directly in our source tree, as they are not available as a library). Tokenizer is specified in the configuration as a function call such as `Ngram(9)`, or if you omit all arguments, `Ngram` is also acceptable. The following tokenizers are available:

- `Raw`: no tokenization, the entire string is treated as a single token.
- `Simple`: splits on whitespace and punctuation.
- `Whitespace`: splits on whitespace.
- `Ngram(min_gram?, max_gram?, prefix_only?)`: splits into n-grams. `min_gram` is the minimum size of the n-gram (default 1), `max_gram` is the maximum size of the n-gram (default to `min_gram`), and `prefix_only` is a boolean indicating whether to only generate prefixes of the n-grams (default false).
- `Cangjie(kind?)`: this is a text segmenter for the Chinese language. `kind` can be `'default'`, `'all'`, `'search'` or `'unicode'`.

After tokenization, multiple filters can be applied to the tokens. The following filters are available:

[14] https://github.com/quickwit-oss/tantivy

- `Lowercase`: converts all tokens to lowercase.
- `AlphaNumOnly`: removes all tokens that are not alphanumeric.
- `AsciiFolding`: converts all tokens to ASCII (lossy), i.e. `passé` goes to `passe`.
- `Stemmer(lang)`: use a language-specific stemmer. The following languages are available: `'arabic'`, `'danish'`, `'dutch'`, `'english'`, `'finnish'`, `'french'`, `'german'`, `'greek'`, `'hungarian'`, `'italian'`, `'norwegian'`, `'portuguese'`, `'romanian'`, `'russian'`, `'spanish'`, `'swedish'`, `'tamil'`, `'turkish'`. As an exmple, the English stemmer converts `'running'` to `'run'`.
- `Stopwords(lang)`: filter out stopwords specific to the language. The stopwords come from the stopwords-iso[15] project. Use the ISO 639-1 Code as specified on the project page. For example, `Stopwords('en')` for English will remove words such as `'the'`, `'a'`, `'an'`, etc.

For English text, the recommended setup is `Simple` for the tokenizer and `[Lowercase, Stemmer('english'), Stopwords('en')]` for the filters.

[15] https://github.com/stopwords-iso/stopwords-iso

CHAPTER

FIVE

TIME TRAVEL

Simply put, time travel in a database means tracking changes to data over time and allowing queries to be logically executed at a point in time to get a historical view of the data. In a sense, this makes your database immutable, since nothing is really deleted from the database ever. *This story* gives some motivations why time travel may be valuable.

In Cozo, a stored relation is eligible for time travel if the last part of its key has the explicit type `Validity`. A validity has two parts: a time part, represented by a signed integer, and an assertion part, represented by a boolean, so `[42, true]` represents a validity. Sorting of validity is by the timestamp first, then by the assertive flag, but each field is compared in descending order, so:

```
[1, true] < [1, false] < [-1, true]
```

All rows with identical key parts except the last validity part form the *history* for that key, interpreted in the following way: the fact represented by a row is *valid* if its flag is `true`, and the range of its validity is from its timestamp (inclusive) up until the timestamp of the next row under the same key (excluding the last validity part, and here time is interpreted to flow forward). For example, let's say we have the rows `'a', [10, true] => 'x'`, `'a', [20, true] => 'y'`, `'a', [30, true] => 'z'`, then at timestamps 10, 11, ..., 19, `'a'` is asserted to be `'x'`, and at timestamps 20, 21, ..., 29, `'a'` is asserted to be `'y'`, and from timestamp 30 onwards, `'a'` is asserted to be `'z'`. Before timestamp 10, `'a'` doesn't exist.

A row with a `false` assertive flag does nothing other than making the previous fact invalid. For example, if we add to the above rows a row `'a', [15, false] => null`, then `'a'` no longer exists at timestamps 15, 16, ..., 19, and at timestamp 20 onwards, `'a'` is asserted to be `'y'`.

When querying against such a stored relation, a validity specification can be attached, for example:

```
?[name] := *rel{id: $id, name, @ 789}
```

The part after the symbol `@` is the validity specification and must be a compile-time constant, i.e., it cannot contain variables. Logically, it is as if the query is against a snapshot of the relation containing only valid facts at the specified timestamp.

It is possible for two rows to have identical non-validity key parts and identical timestamps, but differ in their assertive flags. In this case when queried against the exact timestamp, the row is valid, as if the row with the `false` flag does not exist. For example, if we add to the above rows a row `'a', [30, false] => null`, since we already have a row `'a', [30, true] => 'z'`, when queried against timestamp 30, `'a'` is asserted to be `'z'`. The effect of the retraction is invisible from any time-travel query.

The use case for this behaviour is to assert a fact only until a future time when that fact is sure to remain valid. When that time comes, a new fact can be asserted, and if the old fact remains valid there is no need to `:rm` the previous retraction.

You can use the function `to_bool` to extract the flag of a validity, and `to_int` to extract the timestamp as an integer.

In Cozo it is up to you to interpret the timestamp part of validity. If you use it to represent calendar time, then it is recommended that you treat it as microseconds since the UNIX epoch. For this interpretation, the following convenience

are provided:

- When putting facts into the database, instead of specifying the exact literal validity as a list of two items, the strings `ASSERT` and `RETRACT` can be used instead, and is interpreted as assertion and retraction at the current timestamp, respectively. This has the additional guarantee that all insertion operations in the same transaction using this method gets the same timestamp, and furthermore you can also use these strings as the default values for a field, and they will do the right thing.
- In place of a list of two items for specifying the literal validity, you can use RFC 3339 strings for assertion timestamps or validity specification in query. For retraction, prefix the string by `~`.
- When specifying validity against a stored relation, the string `NOW` uses the current timestamp, and `END` uses a timestamp logically at the end of the world. Furthermore, the `NOW` timestamp is guaranteed to be the same as what would be inserted using `ASSERT` and `RETRACT`.
- You can use the function `format_timestamp` to directly format a the timestamp part of a validity to RFC 3339 strings.

An interesting use case of the time travel facility is to pre-generate the whole history for all time, and in the user-facing interface query with the current time `NOW`. The effect is that users see an illusion of real-time interactions: a manifestation of Laplace's daemon[16].

[16] https://en.wikipedia.org/wiki/Laplace%27s_demon

CHAPTER

SIX

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.

6.1 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 you can decipher the result after reading *Query execution*.

6.2 Ops for stored relations

`::relations`

List all stored relations in the database

`::columns <REL_NAME>`

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

`::indices <REL_NAME>`

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

`::describe <REL_NAME> <DESCRIPTION>?`

Describe the stored relation `<REL_NAME>` and store it in the metadata. If `<DESCRIPTION>` is given, it is stored as the description, otherwise the existing description is removed. The description can be shown with `::relations`. It serves as the documentation and signpost for humans and AI.

`::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.

`::index ...`

Manage indices. See *Stored relations and transactions* for more details.

`::hnsw ...`

Manage HNSW indices. See *Proximity searches* for more details.

`::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 *Stored relations and transactions*.

`::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.

6.3 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`.

6.4 Maintenance

`::compact`

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

CHAPTER

SEVEN

TYPES

7.1 Runtime types

Values in Cozo have the following *runtime types*:

- `Null`
- `Bool`
- `Number`
- `String`
- `Bytes`
- `Uuid`
- `List`
- `Vector`
- `Json`
- `Validity`

`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.

`Vector` have fixed length and contain floats. There are two versions: F32 vectors and F64 vectors.

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.

- Json values are compared by their string representation, which is a bit arbitrary and you should not rely on the order.
- Validity is introduced for the sole purpose of enabling *time travel* queries.

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).

7.2 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.

There are no literal representations for `Vector` or `Validity`. Use the function `vec` to convert a list to a vector.

Json objects are enclosed between curly brackets `{}`, with key-value pairs separated by commas. For all other Json subtypes, use the function `json` to convert normal values to them.

7.3 Column types

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

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

- `Json`
- `Validity`

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.

Vectors are also valid as component types and is written in the syntax `<F32; 1024>` for a 1024-element F32 vector. The type can also be F64.

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

EIGHT

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.

8.1 Disjunctive normal form

Evaluation starts by canonicalizing inline rules into disjunction normal form[17], 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.

8.2 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 (not itself) 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.

[17] 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.

8.3 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.

8.4 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.

8.5 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.

8.6 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.

8.7 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

NINE

TIPS FOR WRITING QUERIES

9.1 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. The solution is that you may decide to consider any `null` values to be equivalent to some default values:

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

here `~` is the `coalesce` operator. The parentheses are not necessary, but it reads better this way.

You can also check for null explicitly:

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

`cond` is also helpful in this case.

9.2 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. In fact, even if there are no duplicates, the second version may still be faster, because in Cozo rules run in parallel whenever allowed by semantics and available resources.

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

TEN

FUNCTIONS AND OPERATORS

Functions can be used to build expressions.

All functions except those that extract the current time and those having names starting with `rand_` are deterministic.

10.1 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:

- `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.

10.2 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)`
- `a -> b` is the same as `maybe_get(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.

10.3 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.

10.4 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.

10.5 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.

sqrt(*x*)

Returns the square root of `x`.

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[18] 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[19] 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.

[18] https://en.wikipedia.org/wiki/Atan2
[19] https://en.wikipedia.org/wiki/Haversine_formula

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.

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.

10.6 Vector functions

Now that mathematical functions that operate on floats can also take vectors as arguments, and apply the operation element-wise.

vec(*l*, *type?*)

Takes a list of numbers and returns a vector.

Defaults to 32-bit float vectors. If you want to use 64-bit float vectors, pass `'F64'` as the second argument.

rand_vec(*n*, *type?*)

Returns a vector of `n` random numbers between `0` and `1`.

Defaults to 32-bit float vectors. If you want to use 64-bit float vectors, pass `'F64'` as the second argument.

l2_normalize(*v*)

Takes a vector and returns a vector with the same direction but length `1`, normalized using L2 norm.

l2_dist(*u*, *v*)

Takes two vectors and returns the distance between them, using squared L2 norm: d = sum((ui-vi)^2).

ip_dist(*u*, *v*)

Takes two vectors and returns the distance between them, using inner product: d = 1 - sum(ui*vi).

cos_dist(*u*, *v*)

Takes two vectors and returns the distance between them, using cosine distance: d = 1 - sum(ui*vi) / (sqrt(sum(ui^2)) * sqrt(sum(vi^2))).

10.7 Json funcitons

json(*x*)

Converts any value to a Json value. This function is idempotent and never fails.

is_json(*x*)

Returns `true` if the argument is a Json value, `false` otherwise.

json_object(*k1*, *v1*, ...)

Convert a list of key-value pairs to a Json object.

dump_json(*x*)

Convert a Json value to its string representation.

parse_json(*x*)

Parse a string to a Json value.

get(*json*, *idx*, *default?*)

Returns the element at index `idx` in the Json `json`.

`idx` may be a string (for indexing objects), a number (for indexing arrays), or a list of strings and numbers (for indexing deep structures).

Raises an error if the requested element cannot be found, unless `default` is specified, in which cast `default` is returned.

maybe_get(*json*, *idx*)

Returns the element at index `idx` in the Json `json`. Same as `get(json, idx, null)`. The shorthand is `json->idx`.

set_json_path(*json*, *path*, *value*)

Set the value at the given path in the given Json value. The path is a list of keys of strings (for indexing objects) or numbers (for indexing arrays). The value is converted to Json if it is not already a Json value.

remove_json_path(*json*, *path*)

Remove the value at the given path in the given Json value. The path is a list of keys of strings (for indexing objects) or numbers (for indexing arrays).

json_to_scalar(*x*)

Convert a Json value to a scalar value if it is a `null`, boolean, number or string, and returns the argument unchanged otherwise.

concat(*x*, *y*, ...)

Concatenate (deep-merge) Json values. It is equivalent to the operator form `x ++ y ++ ...`

The concatenation of two Json arrays is the concatenation of the two arrays. The concatenation of two Json objects is the deep-merge of the two objects, meaning that their key-value pairs are combined, with any pairs that appear in both left and right having their values deep-merged. For all other cases, the right value wins.

10.8 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[20] from both ends of the string.

`trim_start`(*x*)

Removes whitespace[21] from the start of the string.

`trim_end`(*x*)

Removes whitespace[22] 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`.

`unicode_normalize`(*str*, *norm*)

Converts `str` to the normalization[23] 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`.

10.9 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.

[20] https://en.wikipedia.org/wiki/Whitespace_character
[21] https://en.wikipedia.org/wiki/Whitespace_character
[22] https://en.wikipedia.org/wiki/Whitespace_character
[23] https://en.wikipedia.org/wiki/Unicode_equivalence

`last`(*l*)

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

`get`(*l*, *n*, *default?*)

Returns the element at index `n` in the list `l`. Raises an error if the access is out of bounds, unless `default` is specified, in which cast `default` is returned. Indices start with 0.

`maybe_get`(*l*, *n*)

Returns the element at index `n` in the list `l`. Same as `get(l, n, null)`. The shorthand is `l->n`.

`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.

10.10 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[24]-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[25]-encoded byte array.

10.11 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);

[24] https://en.wikipedia.org/wiki/Base64
[25] https://en.wikipedia.org/wiki/Base64

- `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_int(*x*)

Converts to an integer. If `x` is a validity, extracts the timestamp as an integer.

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`.

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)
- any validity that is a retraction

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.

10.12 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).

`rand_vec`(*n*, *type?*)
: Generates a vector of `n` random elements. If `type` is not given, it defaults to `F32`.

10.13 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`.

10.13.1 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
```

10.14 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[26]. If `ts` is a validity, its timestamp will be converted to seconds and used.

If a second string argument is provided, it is interpreted as a timezone[27] and used to format the timestamp.

parse_timestamp(*str*)

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

validity(*ts_micro*, *is_assert?*)

Returns a validity object with the given timestamp in microseconds. If `is_assert` is `true`, the validity will be asserted, otherwise it will be assumed. Defaults to `true`.

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

CHAPTER

ELEVEN

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 more powerful (only the latter can be used recursively).

The power of semi-lattice aggregations derive from the additional properties they satisfy: a semilattice[28]:

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.

11.1 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.

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

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

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.

11.2 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, and non-numerical costs are allowed. Only `data` is returned.

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

The argument should be a list of two elements and this aggregation returns the `data` of the minimum `cost`. Non-numerical costs are allowed, unlike `min_cost`. The value `null` for `cost` are ignored when comparing.

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.

11.2.1 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

TWELVE

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 except WASM are compiled with this flag on.

If you are using the Cozo libraries in Rust, Python or NodeJS, or if you are using the standalone executable, you can also easily define custom fixed rules in the hosting environment: see the respective documentations for how to do it.

12.1 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.

12.2 Connectedness algorithms

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

Computes the connected components[29] of a graph with the provided edges.

Returns

Pairs containing the node index, and its component index.

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

StronglyConnectedComponent(*edges[from, to]*)

Computes the strongly connected components[30] 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[31] on the provided edges to compute a minimum spanning forest[32]. 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[33] on the provided edges to compute a minimum spanning tree[34]. `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[35] 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.

12.3 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[36] 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`.

[30] https://en.wikipedia.org/wiki/Strongly_connected_component
[31] https://en.wikipedia.org/wiki/Kruskal%27s_algorithm
[32] https://en.wikipedia.org/wiki/Minimum_spanning_tree
[33] https://en.wikipedia.org/wiki/Prim%27s_algorithm
[34] https://en.wikipedia.org/wiki/Minimum_spanning_tree
[35] https://en.wikipedia.org/wiki/Topological_sorting

- **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[37] (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*.

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[38].

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

[36] https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
[37] https://en.wikipedia.org/wiki/Yen%27s_algorithm

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.

12.4 Community detection algorithms

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

Computes the clustering coefficients[39] 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[40] 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[41] 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.

[38] https://en.wikipedia.org/wiki/A*_search_algorithm
[39] https://en.wikipedia.org/wiki/Clustering_coefficient
[40] https://en.wikipedia.org/wiki/Louvain_method
[41] https://en.wikipedia.org/wiki/Modularity_(networks)

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

Runs the label propagation algorithm[42] 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.

12.5 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[43] 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.

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

Computes the closeness centrality[44] 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[45] of the graph. The input relation represent edges connecting node indices which are optionally weighted.

[42] https://en.wikipedia.org/wiki/Label_propagation_algorithm
[43] https://en.wikipedia.org/wiki/PageRank
[44] https://en.wikipedia.org/wiki/Closeness_centrality

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.

12.6 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.

[45] https://en.wikipedia.org/wiki/Betweenness_centrality

CHAPTER

THIRTEEN

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.

13.1 Callbacks

It is possible to register callbacks so that you get notified when requested stored relations change. Currently, this functionality is available for Rust, Python and NodeJS libraries and the standalone executable only. Refer to the respective documentation for how to use it.

CHAPTER

FOURTEEN

NOTES

Here are some miscellaneous notes about various aspects of CozoDB.

14.1 Some use cases for Cozo

As Cozo is a general-purpose database, it can be used in situations where traditional databases such as PostgreSQL and SQLite are used. However, Cozo is designed to overcome several shortcomings of traditional databases, and hence fares especially well in specific situations:

14.1.1 Interconnected relations

You have a lot of interconnected relations and the usual queries need to relate many relations together. In other words, you need to query a complex graph.

An example is a system granting permissions to users for specific tasks. In this case, users may have roles, belong to an organization hierarchy, and tasks similarly have organizations and special provisions associated with them. The granting process itself may also be a complicated rule encoded as data within the database.

With a traditional database, the corresponding SQL tend to become an entangled web of nested queries, with many tables joined together, and maybe even with some recursive CTE thrown in. This is hard to maintain, and worse, the performance is unpredictable since query optimizers in general fail when you have over twenty tables joined together.

With Cozo, on the other hand, Horn clauses make it easy to break the logic into smaller pieces and write clear, easily testable queries. Furthermore, the deterministic evaluation order makes identifying and solving performance problems easier.

14.1.2 Just a graph

Your data may be simple, even a single table, but it is inherently a graph.

We have seen an example in the Tutorial: the air route dataset, where the key relation contains the routes connecting airports.

In traditional databases, when you are given a new relation, you try to understand it by running aggregations on it to collect statistics: what is the distribution of values, how are the columns correlated, etc.

In Cozo you can do the same exploratory analysis, except now you also have graph algorithms that you can easily apply to understand things such as: what is the most connected entity, how are the nodes connected, and what are the communities structure within the nodes.

14.1.3 Hidden structures

Your data contains hidden structures that only become apparent when you identify the scales of the relevant structures.

Examples are most real networks, such as social networks, which have a very rich hierarchy of structures.

In a traditional database, you are limited to doing nested aggregations and filtering, i.e. a form of multifaceted data analysis. For example, you can analyze by gender, geography, job or combinations of them. For structures hidden in other ways, or if such categorizing tags are not already present in your data, you are out of luck.

With Cozo, you can now deal with emergent and fuzzy structures by using e.g. community detection algorithms, and collapse the original graph into a coarse-grained graph consisting of super-nodes and super-edges. The process can be iterated to gain insights into even higher-order emergent structures. This is possible in a social network with only edges and no categorizing tags associated with nodes at all, and the discovered structures almost always have meanings correlated to real-world events and organizations, for example, forms of collusion and crime rings. Also, from a performance perspective, coarse-graining is a required step in analyzing the so-called big data, since many graph algorithms have high complexity and are only applicable to the coarse-grained small or medium networks.

14.1.4 Knowledge augmentation

You want to understand your live business data better by augmenting it into a knowledge graph.

For example, your sales database contains product, buyer, inventory, and invoice tables. The augmentation is external data about the entities in your data in the form of taxonomies and ontologies in layers.

This is inherently a graph-theoretic undertaking and traditional databases are not suitable. Usually, a dedicated graph processing engine is used, separate from the main database.

With Cozo, it is possible to keep your live data and knowledge graph analysis together, and importing new external data and doing analysis is just a few lines of code away. This ease of use means that you will do the analysis much more often, with a perhaps much wider scope.

14.2 Cozo runs (almost) everywhere

Version 0.1 of Cozo can be used embedded from Python, NodeJS, Java, Rust and C, in addition to running standalone as a web server. Immediately after its release, many people asked about the feasibility of using Cozo embedded on mobile devices.

There was one major obstacle to supporting mobile: Cozo 0.1 used RocksDB as the storage engine, and compiling RocksDB for mobile devices is not an easy task. We chose RocksDB because it could handle a huge amount of concurrency and is very fast, but the concurrency part may not be relevant for the mobile case: you almost always have only one process concurrently accessing the database.

So we ripped apart the storage engine code, made a nice and minimal interface out of it, and now Cozo supports swappable storage engines! At the time of this writing, you can choose from the following:

- In-memory engine
- SQLite engine
- RocksDB engine
- Sled engine
- TiKV engine

They offer different trade-offs:

- The in-memory engine is perfect if you just want to use Cozo as a computation engine. For us, it also made writing tests much easier. The downside is that it doesn't persist data, and it doesn't support much *write* concurrency.
- The SQLite engine uses a minimal amount of resources, is easy to compile for almost all platforms including mobile, and is reasonably fast for reads. SQLite markets itself as a *file storage format*, and we took advantage of that by making SQLite the backup format for all engines. In this way when you backup your database, you get a single-file SQLite-backed Cozo database. You do not need to restore the backup to look inside: the backup *is* a fully functional database. As a backup format, it is also extremely space-efficient after you gzip it. The downside is as expected: SQLite is not very fast when it comes to writing and is effectively single-threaded when write concurrency is involved. But as we have said, these are usually not problems on mobile.
- The RocksDB engine is crazy fast for both reads and writes and can handle an enormous amount of concurrency, while still being rather conservative on resource usage. In particular, its storage on disk is compressed, making its disk space requirements for live databases the smallest among all persistent options.
- We included Sled as an engine just because we can. The only benefit is that it is pure Rust, and we are not Rust fundamentalists. It is not faster than SQLite for the usual workload that Cozo encounters and uses way more disk space.
- The TiKV option is the slowest among all options (10x to 100x slower) since data must come from the network. The benefit is that TiKV is a distributed storage. We included it so that people may decide for themselves if it offers value for them. By the way, 100x slower *than the other storage options* may not be slow compared to the average graph databases in the market.

As a result of the storage engine refactoring, Cozo now runs on a much wider range of platforms, including iOS, Android, and web browsers (with web assembly)! We have also expanded the officially supported languages where you can use Cozo embedded: Swift and Golang. Even if your platform/language combination is not supported, you can still use Cozo with the client/server mode. Or you can try to compile Cozo from source and interface it with your platform/language: let us know if you encounter problems, and we will help!

14.3 On performance

Commercial databases like to do publicity stunts by publishing "performance comparisons" of competing databases, with the results invariably favouring their products. We are not going to do that because, first, we do not want to attract the wrong kind of attention; second and more importantly, such benchmarks don't educate users beyond the "trust us, we are the best" preaching. Instead, we want our benchmarks to help the users answer the following questions:

- For my particular workload, what are the pros and cons of the different setups that I can choose from?
- What is the performance I can expect from such a setup and is it enough for me?

14.3.1 The setup

We will only be comparing Cozo to itself running on two different machines:

- Mac Mini (2020)
- Linux Server (2017)

The Mac Mini runs MacOS 13.0.1, has Apple M1 CPUs with 4 performance cores and 4 efficiency cores, 16GB of memory and a pretty fast NVMe SSD storage. Its benchmarks would be typical of recent reasonably powerful desktop and laptop machines. The Linux server runs Ubuntu 22.04.1 LTS, has two Intel Xeon E5-2678v3 CPUs with a total of 24 physical cores, 64GB of memory and one of the slowest SSD storage I have ever seen. You can expect similar performance if you (over)pay cloud providers. If you have servers that have newer hardware, you can probably expect much better performance.

We will be running Cozo embedded in Rust[46] (in fact, we will take advantage of Rust's built-in benchmark tools). As it is embedded, we will use different numbers of concurrent threads to run Cozo to see how a particular task scales with the number of processors. Embedding Cozo in Rust is the fastest way to run Cozo and if your use case involves embedding in another language such as Python, there will be overheads due to Python itself. Still, if for similar tasks as recorded here you experience orders of magnitude worse performance, please let us know since it could be an environment-related bug.

It seems graph databases like to use the Slovenian social network Pokec[47] for benchmarks (see here[48] and here[49]). We will use three different sizes for subsets of the data:

- "Tiny": 10,000 vertices, 121,716 edges
- "Small": 100,000 vertices, 1,768,515 edges
- "Medium": 1,632,803 vertices, 30,622,564 edges, this is the full dataset

Note that this is the same subsets as done here[50], except their "small" is our "tiny", their "medium" is our "small", and their "large" is our "medium". We feel it to be rather presumptuous in this age to call a dataset with just over 30 million edges "large". When will we do a benchmark for a truly webscale dataset[51]? When we have more time and a deeper pocket, maybe! Anyway, the "medium" size is probably large enough for most purposes.

The schema for the data is the following, written in CozoScript:

```
{:create user {uid: Int => cmpl_pct: Int, gender: String?, age: Int?}}
{:create friends {fr: Int, to: Int}}
{:create friends.rev {to: Int, fr: Int}}
```

If you don't read CozoScript yet: the relation `user` has an integer primary key and three non-keys representing the vertices, the relation `friends` has a composite `(Int, Int)` primary key representing the edges, and the relation `friends.rev` acts as an index for the edges in the reverse direction (in Cozo, everything is very explicit).

We will be comparing Cozo running with three different storage engines:

- In-memory engine
- SQLite engine
- RocksDB engine

All queries are run with snapshot isolations in effect: when mixing reads and writes, reads will only see a snapshot of data valid at the start of the transaction, and when writes conflict with each other, at least one of them will fail to commit, in which case we will manually retry the query. This is the only consistency level Cozo currently supports: there is no way to opt for a more lax consistency model.

All mutations sent to the backend engine complete only when the underlying storage engine transactions complete: there is no "fire and forget" involved to give the user a false impression of high performance.

Finally, note that we only benchmark the out-of-box experience of Cozo. In the case of the RocksDB engine, there are lots of knobs you can turn to make it much more performant for certain workloads (the easiest knob is to beg it to use more memory, which helps when dealing with large datasets), but we expect most users *not* to be experts in such optimizations.

You can download the complete result of the benchmarks as a spreadsheet `here` to do your own analysis.

[46] https://github.com/cozodb/cozo/blob/dev/cozo-core/benches/pokec.rs
[47] https://snap.stanford.edu/data/soc-pokec.html
[48] https://github.com/memgraph/memgraph/tree/master/tests/mgbench#books-datasets
[49] https://www.arangodb.com/2018/02/nosql-performance-benchmark-2018-mongodb-postgresql-orientdb-neo4j-arangodb/
[50] https://github.com/memgraph/memgraph/tree/master/tests/mgbench#pokec
[51] https://www.tigergraph.com/benchmark/

14.3.2 Loading data

Batch import

The first question we are interested in is how long it takes to load our datasets into the database: do we need to wait for days? Our approach is to parse data from a text file and insert them into the database in batches of 300, single-threaded.

For the tiny dataset, the results are:

Platform	Backend	Time taken (seconds)
Mac Mini	Mem	0.265788
Mac Mini	RocksDB	0.686022
Mac Mini	SQLite	6.383260
Server	Mem	0.494136
Server	RocksDB	1.285214
Server	SQLite	27.971535

Here is for the small dataset:

Platform	Backend	Time taken (seconds)
Mac Mini	Mem	5.429186
Mac Mini	RocksDB	11.752198
Mac Mini	SQLite	146.848621
Server	Mem	8.741262
Server	RocksDB	19.261249
Server	SQLite	432.705856

And for the medium dataset:

Platform	Backend	Time taken (seconds)
Mac Mini	Mem	155.894422
Mac Mini	RocksDB	212.731813
Server	Mem	219.956052
Server	RocksDB	348.638331

As you can see we didn't even test for SQLite's performance using the medium dataset, as we grew tired of waiting. If the trend continues, import with SQLite backend would take at least 45 minutes on Mac Mini, and more than 2 hours on the Linux server. SQLite's performance looks really bad here, but we used to import a similar amount of data into another graph database and it took us *half a day*. And even if you insist on using the SQLite backend, there is a much faster way to import data: keep reading.

For the RocksDB backend, everything can be done within a few minutes, which is more than reasonable for tens of millions of rows.

We can compare performance across the board by considering *raw rows per second* in imports, in which an edge counts as two raw rows since it must appear in two relations:

Here RocksDB performs well, especially for scaling: the decrease in raw rows per second due to larger datasets is very small. And it is always within a factor of three for the mem backend which does not persist data at all.

Some of you may say that this is not fair for the SQLite backend, since with some additional tricks and more clever batching, you can get higher numbers for SQLite. Well, we are testing for simple-minded out-of-box performance, and

the fact that with tuning[52], the RocksDB performance can be increased even more drastically.

How much memory does the database use during the import process? We will show the peak memory usage as reported by the system:

The benchmark infrastructure takes about 50MB of memory even if it does nothing. So the SQLite backend always uses a negligible amount of extra memory. RocksDB on the other hand will use memory to speed things up. As we have said before we didn't collect data for importing the medium dataset into the SQLite backend.

The data for the mem backend is shown below separately:

This measures the size of the whole dataset as the mem backend can only store data in memory. As we can see Apple's OS somehow uses memory more efficiently. For almost everything we do in this benchmark, the memory usage of the mem backend is very similar to this, so we will not show the memory usage of the mem backend before. If you are interested nonetheless, you can look at the raw data in the `spreadsheet`.

Backup

In Cozo we can backup the database to an SQLite-based database. How fast is the backup?

On a Mac Mini, this is around one million raw rows per second for all backends, which should be fast enough for most purposes. On the Linux server, the bad quality of the SSD shows, but it is still quite fast. By the way, if you have lots of data and you want to use the SQLite backend, you can batch import the data into the RocksDB or mem backend, and then back up the database. The backup file *is* a working SQLite-backed database, and the whole process is *a lot* faster than importing into an SQLite-backed database directly.

Memory usage:

Not much surprise here. As we said before around 50MB is used by the benchmark infrastructure, so take that into account.

Restoring from backup

How fast is restoring from a backup?

This is the only benchmark where RocksDB performs the worst, with 400K raw rows per second. Restoring into the SQLite backend is fast, but in fact, you can be faster still: just copy the backup file over (or use it directly if you don't intend to write any data)!

Memory usage:

No surprise.

[52] https://github.com/cozodb/cozo#tuning-the-rocksdb-backend-for-cozo

14.3.3 Transactional queries (OLTP)

Online Transaction Processing (OLTP) queries are simple reads or writes queries that are expected to finish quickly, and you are expected to deal with lots of them.

Point read

This is the simplest kind of query you can imagine: given an ID, it just reads the corresponding row and gives it to you:

```
?[cmpl_pct, gender, age] := *user{uid: $id, cmpl_pct, gender, age}
```

The performance metric we are interested in is the queries per second (QPS):

The effect of data size on such queries is small, and in general, adding more cores helps almost *linearly*, though in the case of Mac Mini, only the performance cores help, the efficient cores are pretty useless and can get in the way. In general, you can expect at least around 100K QPS regardless of data size on all setups when you fully utilize your resources.

For memory usage:

RocksDB only starts using memory with the medium dataset. In all other cases, memory usage is minimal.

Point write

This is the simplest write query: it just creates a new vertex:

```
?[uid, cmpl_pct, gender, age] <- [[$id, 0, null, null]] :put user {uid => cmpl_pct,␣
↪gender, age}
```

For this query, we are only going to show multi-thread performances for RocksDB, since writing to the other backends are protected by a big lock, so they are effectively still single-threaded:

RocksDB shines here as you can expect more than about 100K QPS for both setups. Using more than the number of performance cores on the Mac Mini decreases performance quite a bit, so avoid that if you can. But you can't see the SQLite bars, can you? Let's use logarithmic scale instead:

Whereas RocksDB easily manages more than 100K QPS, SQLite struggles to reach even 100 QPS on the server with the slow SSD. That is more than 1000 times slower! It is so slow since each request translates into an SQLite write transaction, and SQLite writes transactions are known to be super expensive. These separate transactions are unavoidable here because that's the rule for the game: lots of independent, potentially conflicting writes to the database. The moral of the story is to stay away from the SQLite backend if you expect lots of independent writes.

Memory usage?

Completely reasonable, I'd say. Even for medium datasets, RocksDB keeps memory usage under 500MB.

Point update

This query updates a field for a given row:

```
?[uid, cmpl_pct, age, gender] := uid = $id, *user{uid, age, gender}, cmpl_pct = $n
:put user {uid => cmpl_pct, age, gender}
```

The performance:

It is slower than point writes, but within a factor of two. You can still easily manage more than 50K QPS for RocksDB. Memory usage is almost the same as the point write case:

Mixed queries?

Of course in realistic situations, you would expect read, write and update to occur concurrently. We won't show the details here, but the conclusion is that in such cases, the RocksDB backend doesn't care if the queries are reads, writes or updates, whereas any amount of writes kills SQLite. If you want the details, you can find them in the `spreadsheet`.

If SQLite performs so badly at writes, why include it at all? Well, its performance is still acceptable if you are using it to build a desktop or mobile application where writes are batched, and with the SQLite engine, the database does not use more than the absolute minimal amount of memory.

14.3.4 Analytical queries (OLAP)

Online analytical processing (OLAP) queries are queries which may touch lots of rows in the database, do complex processing on them, and may return a large number of rows. All graph queries should fall into this category.

For OLAP queries, we are more interested in latency: how long does a query take before it returns (on average)?

Friends of friends

The classical graph traversal query is the "friends of friends" query: finding out who the friends of friends of a particular person are. For such queries, the intermediate results and the return sets must be stored somewhere (usually in memory). For these queries, we will only show results for the "medium" dataset: 1.6 million vertices and 32 million edges. The same query for the smaller datasets complete much faster: refer to the raw numbers if you are interested.

We start by following the "friends" relation twice—a "2 hops" query:

```
?[to] := *friends{fr: $id, to: a}, *friends{fr: a, to}
```

On average, this will return hundreds of rows.

We see that the RocksDB backend performs very well, and if the storage is fast enough, it is even faster than the mem backend. The SQLite backend also performs quite well competitively. Having more threads harms latency, but not much.

For memory usage:

As usual, the SQLite backend doesn't use more than the absolute minimal amount of memory, unless you have many concurrent threads. The memory usage of the RocksDB backend is also pretty small.

Let's now go up one hop to find out friends' friends' friends:

```
l1[to] := *friends{fr: $id, to}
l2[to] := l1[fr], *friends{fr, to}
?[to] := l2[fr], *friends{fr, to}
```

The variance of the number of returned rows is now very high: on average thousands of rows will be returned, and if you start with some particular nodes, you get tens of thousands of rows. The latency is as follows:

The trend is similar to the 2 hops case, except that the latency is about twenty times as long, roughly proportional to the number of returned rows.

For memory usage:

Because the database must keep the *return set* in memory, in all cases the memory usage increases. But it still manages with under 1GB of memory, even with 24 concurrent threads running on the server.

Now let's go to the extreme, by considering the 4 hops query:

```
l1[to] := *friends{fr: $id, to}
l2[to] := l1[fr], *friends{fr, to}
l3[to] := l2[fr], *friends{fr, to}
?[to] := l3[fr], *friends{fr, to}
```

The number of return rows now varies wildly: from tens of thousands of rows if you start with someone who is solitary, or more than half of the whole dataset (more than 600K rows) if you start with someone popular!

I'd say that for return sets this big, the average latency of a few seconds (or even less than a second) is excellent.

Peak memory usage just reflects the size of the returned sets:

We won't go beyond four hops but will note instead that if you go up to six hops, by the "six degrees of separation", you will return the majority of nodes in almost all cases. Actually, in our experiments, this already happens with a high probability for five hops.

Aggregations

Aggregations present a different challenge to the database: here the amount of data to keep in memory is not much (in the case of counting, just a single counter), but the database must scan every row of a relation to return the result. For these queries, we will again only show results for the "medium" dataset: 1.6 million rows for the relation in question.

First, we will group users by their age and return the counts for each age group:

```
?[age, count(uid)] := *user{uid, age}
```

This tests the single-core CPU performance and disk read performance. Around 1 second (within a factor of two) to scan the whole table in all cases.

The memory usage is minimal as the return set is small:

Now let's add a filter to the aggregation:

```
?[age, count(age)] := *user{age}, age ~ 0 >= 18
```

This adds in a bit of processing time, but in terms of the order of magnitude the numbers are similar to before:

The memory usage is almost identical:

The results are similar if we compute several aggregations in tandem:

```
?[min(uid), max(uid), mean(uid)] := *user{uid, age}
```

The latency:

and the memory usage:

Pagerank

Finally let's see how one of our canned algorithms performs: the Pagerank algorithm with query

```
?[] <~ PageRank(*friends[])
```

This time we will show results for different dataset sizes. First for the tiny dataset (10K vertices, 122K edges):

Completes in the blink of an eye. Memory usage:

Not much, since the dataset is truly tiny.

Now for the small dataset (100K vertices, 1.7M edges):

About one second within a factor of two. Memory usage:

This is the amount of memory used to store the graph in the main memory, which is less than the size of the total graph on disk.

Now for the full dataset (1.6M vertices, 32M edges):

About half a minute across all setups. I'd argue that this is as fast as *any* implementation could go. Memory usage:

1GB memory for such a workload is more than reasonable.

14.3.5 Conclusion

We hope that you are convinced that Cozo is an extremely performant database that excels on minimal resources. As it can run (almost) everywhere, please try it for your use case, and send us feedback so that we can improve Cozo further!

14.4 Time travel in a database: a Cozo story

You have built a social network. The only thing your users can do on the network is to update their "mood". For example, user "joe" may be "moody" yesterday, and "happy" today. How is this reflected in the database? In CozoDB, this can be stored in the following relation:

```
:create status {uid: String => mood: String}
```

The equivalent SQL for Postgres is

```
create table status (
    uid text primary key,
    mood text not null
)
```

Your home page is very simple: it is a gigantic page containing the moods of all the users all at once. To generate it, you run the query:

```
?[uid, mood] := *status{uid, mood}
```

```
select uid, mood from status
```

And when users want to update their status, you run:

```
?[uid, mood] <- $input
:put status {uid => mood}
```

```
update status
set mood = $1
where uid = $2
```

Simple enough. Now scientists come to you and want to buy your data for their study of the fluctuation of moods during the pandemic. Of course, you know that their real motive is nefarious, so you promptly show them the door. And then start banging your head against the door. Why have you thrown away the *history*, the most valuable part of your data? WHY?

So you borrow a time machine from a future you and travel back in time to warn the former you. "It's simple to fix", the former you says:

```
:create status {uid: String, ts default now() => mood: String}
```

```
create table status (
    uid text not null,
    ts timestamptz not null default now(),
    mood text not null,
    primary key (uid, ts)
)
```

Of course, now there is no way users can delete their accounts anymore, all they can do is send you more and more updates. Very useful feature!

Now, to generate your homepage:

```
?[uid, mood] := *status{uid, mood, ts}, ts == now()
```

```
select uid, mood from status
where ts = now()
```

Disaster! The homepage remains forever blank, no matter what the users do!

The problem is that when you generate your homepage, you can only collect data that were inserted in the past. And for past data, the condition `ts == now()` is never true!

After a lot of fumbling, you find that the following query works:

```
candidates[uid, max(ts)] := *status{uid, ts}
?[uid, mood] := candidates[uid, ts], *status{uid, ts, mood}
```

```
with candidates(uid, ts) as (
    select uid, max(ts) from status
    group by uid
)
select status.uid, status.mood from status
inner join candidates on status.uid = candidates.uid and status.ts = candidates.ts
```

You first find out what are the timestamps for the latest status for each user, and then use the user ID together with the timestamps to collect the moods.

Now travelling back to a particular time in the past is easy:

```
candidates[uid, max(ts)] := *status{uid, ts}, ts < $date_ts
?[uid, mood] := candidates[uid, ts], *status{uid, ts, mood}
```

```
with candidates(uid, ts) as (
    select uid, max(ts) from status
    where ts < $1
    group by uid
)
select status.uid, status.mood from status
inner join candidates on status.uid = candidates.uid and status.ts = candidates.ts
```

14.4.1 The cost of time travel

Your social network becomes a runaway success, and the scientists are happy too! As time goes on, however, you notice performance problems, and it gets worse each day.

What's happening? After all, your network caters only for the students on a certain campus, and even if everyone signed up, there would only be 10,000 users at most. After digging into your data, you notice that some (most?) of your users are hyperactive and update their mood every five minutes during their waking hour. Even though you have only run your service for three months, some of them have already accumulated over 10,000 mood updates!

So for the front-page generating query:

```
candidates[uid, max(ts)] := *status{uid, ts}
?[uid, mood] := candidates[uid, ts], *status{uid, ts, mood}
```

```
with candidates(uid, ts) as (
    select uid, max(ts) from status
    group by uid
```

(continues on next page)

(continued from previous page)

```
)
select status.uid, status.mood from status
inner join candidates on status.uid = candidates.uid and status.ts = candidates.ts
```

you are doing a full scan of your data to get your results. For 10,000 users and 1,000 updates each (we use the mean number of mood updates, so it's 1,000 instead of 10,000), that's 10 million rows. And next year it will become more than one billion rows, since time ticks and you are thinking of expanding your service to other communities.

14.4.2 Dreamy indices

Your investor suggests the "enterprisey" thing: pre-computing the front page and updating it periodically instead of calculating it in real-time. Being a hacker with a big ego, you detest all things "enterprisey" and ask yourself: "is there anything better that can be done?" Your friend, who works in finance, suggests time series databases. "It can handle data from the *stock market* quite well, so surely it is good enough for *your* data!" "Just index your data by the timestamp!" Well, unlike stock market indices, your data is *sparse*: it is not collected at regular intervals for all users in unison. Are you going to materialize these implicit rows so that every time a user updates her mood, *everyone else* also gets an automatic update? Your cloud provider is very welcoming of this idea and urges you to sign up for their proprietary time series database. Your investor is also kind of supportive since it would make you an instant "big data" company, but worries about whether you can secure additional funding in time to cover the costs. You, ever calm, make some back-of-envelop estimates and give up the idea.

Your investor still wants to talk to you over the phone, but you become very annoyed and go to sleep, clouded in anxiety.

In your dream, you come to a dark, Harry-Potteresque library, with dusty shelves neatly lined up, and on the shelves were … files for the mood of your users at different times, meticulously arranged. The *tree* backing your database has taken physical form! You walk mechanically to the first shelf, like a robot, and start to collect the mood of every user at midnight some days back.

"Aaron, 2022-09-01, 15:31, too early."

"Aaron, 2022-09-01, 15:33, still too early."

…

"Aaron, 2022-12-24, 23:59, still too early."

"Aaron, 2022-12-25, 00:34, well it's *past* the time we want, so the *previous* item contains the mood." (The mood is "festive", by the way.)

"Aaron, 2022-12-25, 00:42, we don't need this anymore."

"Aaron, 2022-12-25, 00:47, we don't need this anymore."

…

"Aaron, 2022-12-27, 12:31, why are we *still* looking at Aaron, by the way?"

…

"Bean, 2022-11-27, …"

Two things irked you. First, you are going through the data in the wrong direction, so after you have gone past the expected date, you have to go back and look at the *previous* record. This is especially annoying since some users signed

up only today, and the previous record is someone else's. Second, you are walking a *tree*, so why can't you jump to the next user when you know you are done with a user?

As if triggered by these thoughts, the books pour out of the shelves to form a tornado, swirl all over the library, and after a while return to the shelves. "I have to do this all over again," you gruntle and walk to the first shelf. But something has changed: you can now directly *jump* to the beginning, and the records are in a different order, ascending by the user, as before, but *descending* by the timestamp:

"Aaron, 2022-12-27, 03:38, too late, let's *jump* to the book past Aaron, 2022-12-25, 00:00."

"Aaron, 2022-12-24, 23:59. Now *this* book contains the required mood for Aaron." "Let's now jump to the book past Aaron at the BIG BANG."

"Bean, 2022-12-24, 23:11, this is already the correct book for Bean, lucky! Now let's go past Bean at the BIG BANG." "I wonder what happened to Mr Bean since Christmas Eve?"

...

Suddenly, you wake up. You rush to your computer and write down what you saw, in code.

14.4.3 Back to reality

Eventually, your social network takes over the world and changes it fundamentally, with the simple schema in CozoScript:

```
:create status {uid: String, ts: Validity default 'ASSERT' => mood: String}
```

the query for the present:

```
?[uid, mood] := *status{uid, mood @ 'NOW'}
```

and the query for historical moments:

```
?[uid, mood] := *status{uid, mood @ '2019-12-31T23:59:59Z'}
```

Obviously, there are no longer any SQL translations.

The story ends here. It is the story of the new *time travel* feature in Cozo v0.4. We have also added a part in the tutorial giving you hands-on experience.

14.4.4 But what about performance?

Real databases care about performance deeply, and at Cozo we do. So let's do some performance tests, with the same Mac Mini as *before*: it runs MacOS 13.0.1, has Apple M1 CPUs with 4 performance cores and 4 efficiency cores, 16GB of memory and a pretty fast NVMe SSD storage. We only test against the RocksDB backend for simplicity.

We generated many relations, all of which contain data for 10,000 users. The 'Plain' relation stores no history at all. The 'Naive' relations store and query history using the naive approach we described in the story. We generated different versions of the naive relations, containing different numbers of mood updates per user. Finally, the 'Hopping' relations store and query history using the dream approach we described earlier.

The historical timestamp for the queries is chosen randomly and the results are averaged over many runs.

First, we want to know how history storage affects point query throughput, measured in queries per second (QPS):

Type	# updates per user	QPS	Performance percentage
Plain	1	143956	100.00%
Naive	1	106182	73.76%
Naive	10	92335	64.14%
Naive	100	42665	29.64%
Naive	1000	7154	4.97%
Hopping	1	125913	87.47%
Hopping	10	124959	86.80%
Hopping	100	100947	70.12%
Hopping	1000	102193	70.99%

As we can see, for 1000 updates per user, the naive approach has a 20x reduction in throughput compared to the no history approach. The hopping approach, on the other hand, maintains more than 70% of the original performance.

To be fair, for the simplest kind of point query where you know the complete key and the timestamp and want the result for only a single user, there is a better way to write the query so that the naive approach can achieve a similar performance to the hopping one. We deliberately wrote our query in a way to avoid this optimization, since this optimization may not always be possible, depending on the query.

Next, let's look at aggregation results where we must go through all users. Now we measure latency instead of throughput:

Type	# updates per user	Latency (ms)	Slowdown ratio
Plain	1	2.38	1.00
Naive	1	8.90	3.74
Naive	10	55.52	23.35
Naive	100	541.01	227.52
Naive	1000	5391.75	2267.53
Hopping	1	9.60	4.04
Hopping	10	9.99	4.20
Hopping	100	39.34	16.55
Hopping	1000	31.41	13.21

Now the naive approach scales badly. For a query that takes only 2ms in a plain relation, it takes more than 5 seconds in a relation with 1000 historical facts per user. The hopping approach, on the other hand, keeps the time complexity under control. Notice that it performs better in the relation with 1000 historical facts per user than in the relation with only 100. This is not a random fluctuation: it occurs consistently no matter how we test it. We guess that the RocksDB backend does something different when a certain threshold is passed.

Nonetheless, it is important to note that there is at least a 3x slowdown no matter how you store the history, even if you only have one historical fact per user. This is the minimal cost of time travel. And this is why Cozo does not automatically keep history for every relation regardless of whether you need it: our philosophy is "zero-cost and zero-mental-overhead abstraction".

Before we end, let's remark that some people maintain that data is naturally immutable and hence should always be stored immutably. We do not take this view. Use immutability and time travel only when you really need it. Are we simply collections of data in an immutable database, operated by Laplace's daemon, perhaps? I hope not, and modern physics certainly says no, no matter how you look at it: whether it is the collapse of the wave function or the dance at the edge of chaos. So immutability is an illusion, or at best a platonic fantasy that we created so that we can make better sense of the world. That's OK, since we can only understand the world by projecting our models onto the world. Just don't become the slaves of our own creation and let it slow us down.

14.5 Cozo 0.5: the versatile embeddable graph database with Datalog is half-way 1.0

It's been a quarter of a year since Cozo's initial release and today we are glad to present to you the "half-way 1.0" version. This marks the completion of all the features we envisaged for Cozo when we first started but weren't present in the initial release:

- User-defined fixed rules (added in v0.5)
- Callbacks for mutation (added in v0.5)
- Multi-statement transaction (added in v0.5)
- Indices (added in v0.5)
- Imperative mini-language (added in v0.5)
- Time-travelling (added in v0.4)
- Swappable backend (added in v0.2)

In addition, v0.5 brings major behind-the-scene changes that has big performance benefits:

- The semi-naive algorithm for executing queries is now executed in parallel for each Horn-clause (now it is even more advisable to split your queries into smaller pieces–better readability *and* performance!)
- The evaluation of expressions is no longer interpreted but instead through stack-based bytecodes (a few percent improvements for filter-heavy queries–the improvement mainly comes from the avoidance of memory allocations)

From now on, until version 1.0, development will "shift-gear" to focus on:

- stability
- performance
- interoperability with other software, e.g.:
  - networkx[53] for pythonic in-memory graph analytics
  - PyG[54] in particular and pytorch[55] in general for deep learning on graphs
  - plotly[56] and dash[57] for plotting and dashboards

If you find Cozo to be useful in your work, please send us feedbacks so that we can make Cozo better still!

14.6 Experience CozoDB: The Hybrid Relational-Graph-Vector Database - The Hippocampus for LLMs

After a long repose, today we are very excited to bring you the release of CozoDB v0.6!

[53] https://networkx.org/
[54] https://www.pyg.org/
[55] https://pytorch.org/
[56] https://plotly.com/graphing-libraries/
[57] https://dash.plotly.com/

14.6.1 Introduction

The singular feature addition in this release is the introduction of *vector search within Datalog*.

> For those who are unfamiliar with the concept: vector search refers to searching through large collections of usually high-dimensional numeric vectors, with the vectors representing data points in a metric space. Vector search algorithms find vectors that are closest to a given query vector, based on some distance metric. This is useful for tasks like recommendation systems, duplicate detection, and clustering similar data points, and has recently become an extremely hot topic since large language models (LLMs) such as **ChatGPT** can make use of it to partially overcome their inability to make use of long context.

Highlights

- You can now create HNSW (hierarchical navigable small world) indices on relations containing vectors.
- You can create multiple HNSW indices for the *same* relation by specifying filters dictating which rows should be indexed, or which vector(s) should be indexed for each row if the row contains multiple vectors.
- The vector search functionality is integrated *within Datalog*, meaning that you can use vectors (either explicitly given or coming from another relation) as pivots to perform *unification* into the indexed relations (roughly equivalent to table joins in SQL).
- Unification with vector search is semantically no different from regular unification, meaning that you can even use vector search in *recursive Datalog*, enabling extremely complex query logic.
- The HNSW index is no more than a hierarchy of proximity graphs. As an open, competent graph database, CozoDB exposes these graphs to the end user to be used as regular graphs in your query, so that all the usual techniques for dealing with them can now be applied, especially: community detection and other classical whole-graph algorithms.
- As with all mutations in CozoDB, the index is protected from corruption in the face of concurrent writes by using Multi-Version Concurrency Control (MVCC), and you can use multi-statement transactions for complex workflows.
- The index resides on disk as a regular relation (unless you use the purely in-memory storage option, of course). During querying, close to the absolute minimum amount of memory is used, and memory is freed as soon as the processing is done (thanks to Rust's RAII), so it can run on memory-constrained systems.
- The HNSW functionality is available for CozoDB on all platforms: in the server as a standalone service, in your Python, NodeJS, or Clojure programs om embedded or client mode, on your phone in embedded mode, even in the browser with the WASM backend[58].
- HNSW vector search in CozoDB is performant: we have optimized the index to the point where basic vector operations themselves have become a limiting factor (along with `memcpy`), and we are constantly finding ways to improve our new implementation of the HNSW algorithm further.

For a more detailed description, see *here*.

> For those who have not heard of CozoDB before: CozoDB is a general-purpose, transactional, relational database that uses Datalog for query, is embeddable but can also handle huge amounts of data and concurrency, and focuses on graph data and algorithms. It even supports time travel (timestamped assertions and retractions of facts that can be used for point-in-time query)! Follow the *Tutorial* if you want to learn CozoDB. The source code for CozoDB is on GitHub[59].

[58] https://www.cozodb.org/wasm-demo/
[59] https://github.com/cozodb/cozo/

Comparisons to other systems

- **PostgreSQL with pgVector**: I am a fan of the PostgreSQL project and always look up to it as a role model. The problem is that the pgVector extension suffers from low recall and is not general enough for many use cases; both these problems don't exist in CozoDB's HNSW index. And SQL itself is just not cut out for complex graph queries, which is the focus of CozoDB.
- **hnswlib and faiss**: Unlike many vector databases currently available, our HNSW index is not a wrapper around one of these two libraries but a new implementation written from scratch in Rust. This is necessary since we want the index to be disk-based and support MVCC. These two options are mostly single-purpose libraries, whereas CozoDB is a general-purpose database system.
- **commercial vector databases running in the Cloud**: CozoDB is FOSS (MPL 2.0 license) and can run in embedded mode (standalone server mode is available if you need it). In terms of functionalities, these cloud databases support only simple queries, whereas in CozoDB they can be as complex as you like (if you really want it, there is an imperative mini-language to write the CozoDB queries). On the other hand, if your requirements are simple, these systems are probably easier to get started with.

The remainder of this note comprises two parts. The first part will, by using a running example, show how the need for vector search arises in a particular application, and as a result of this new feature, how existing systems can become much more powerful. The second part will include some of my personal, somewhat crazy musings and speculations for further development of AI with reference to CozoDB.

14.6.2 The emergence of vector search in CozoDB

> This is not meant for a comprehensive introduction. We will not stop to explain the syntax: see the *Tutorial* instead.

From relational thinking to graph thinking

First, we will show how CozoDB can deal with traditional relational and graph data.

Let's start with a humble sales dataset. CozoDB is first and foremost a relational database, as the following schema for creating the dataset relations show. First, the customers:

```
:create customer {id => name, address}
```

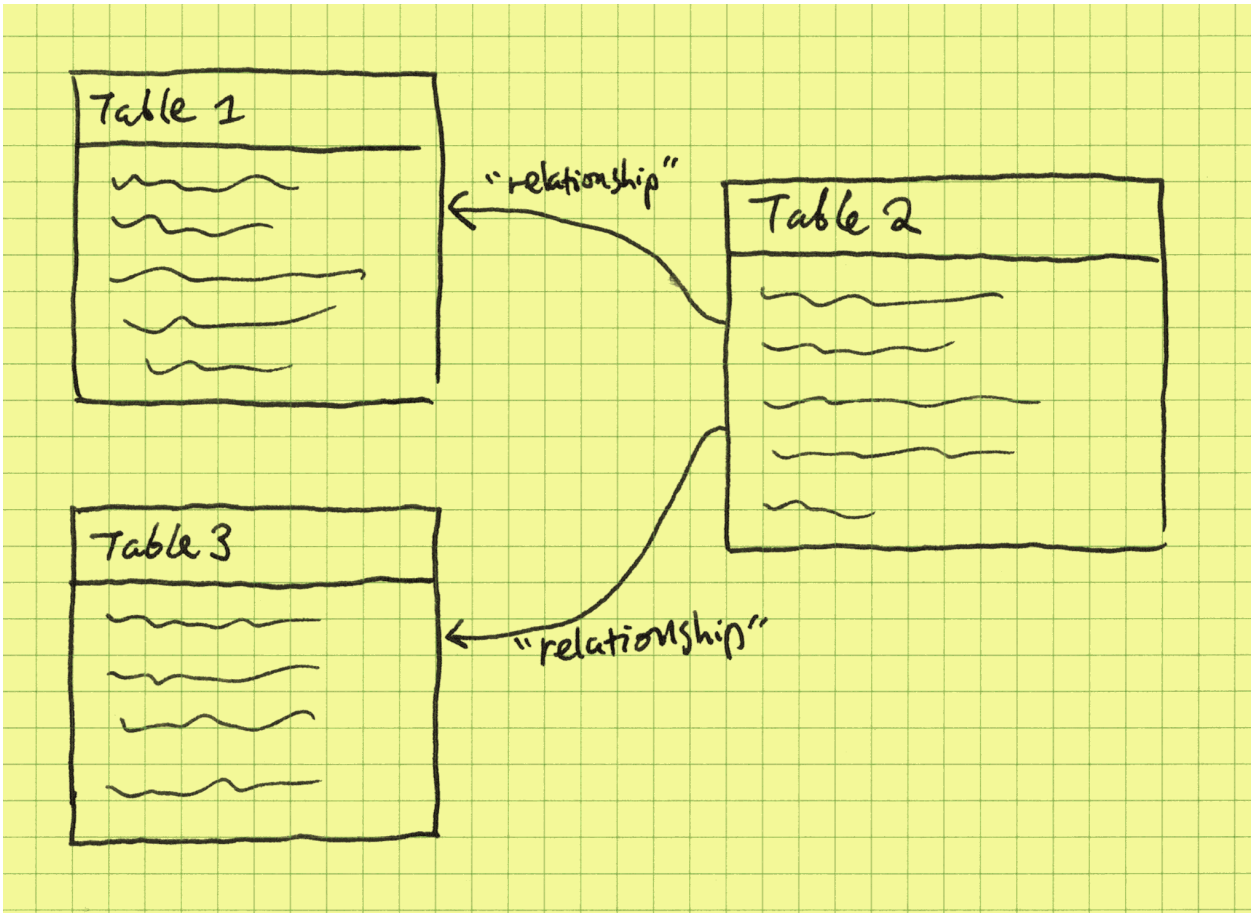
Here `id` is the key for the relation. In a real dataset, there will be many more fields, but here we will just have the names and addresses for simplicity. Also, we did not specify any type constraints for the fields, again for simplicity reasons. Next, the products:

```
:create product {id => name, description, price}
```

Finally, the sales data itself:

```
:create purchase {id => customer_id, product_id, quantity, datetime}
```

The mental picture we should have for these relations is:



Let's assume that these relations have already been filled with data. Now we can start our "business analytics". First, the most popular products:

```
?[name, sum(quantity)] := *purchase{product_id, quantity},
                          *product{id: product_id, name}

:order -sum(quantity)
:limit 10
```

Here, the Datalog query joins the `purchase` and `product` relations through their ID, and then the quantities purchased are sum-aggregated, grouped by product names. Datalog queries are easier to read than the equivalent SQL, once you get used to it.

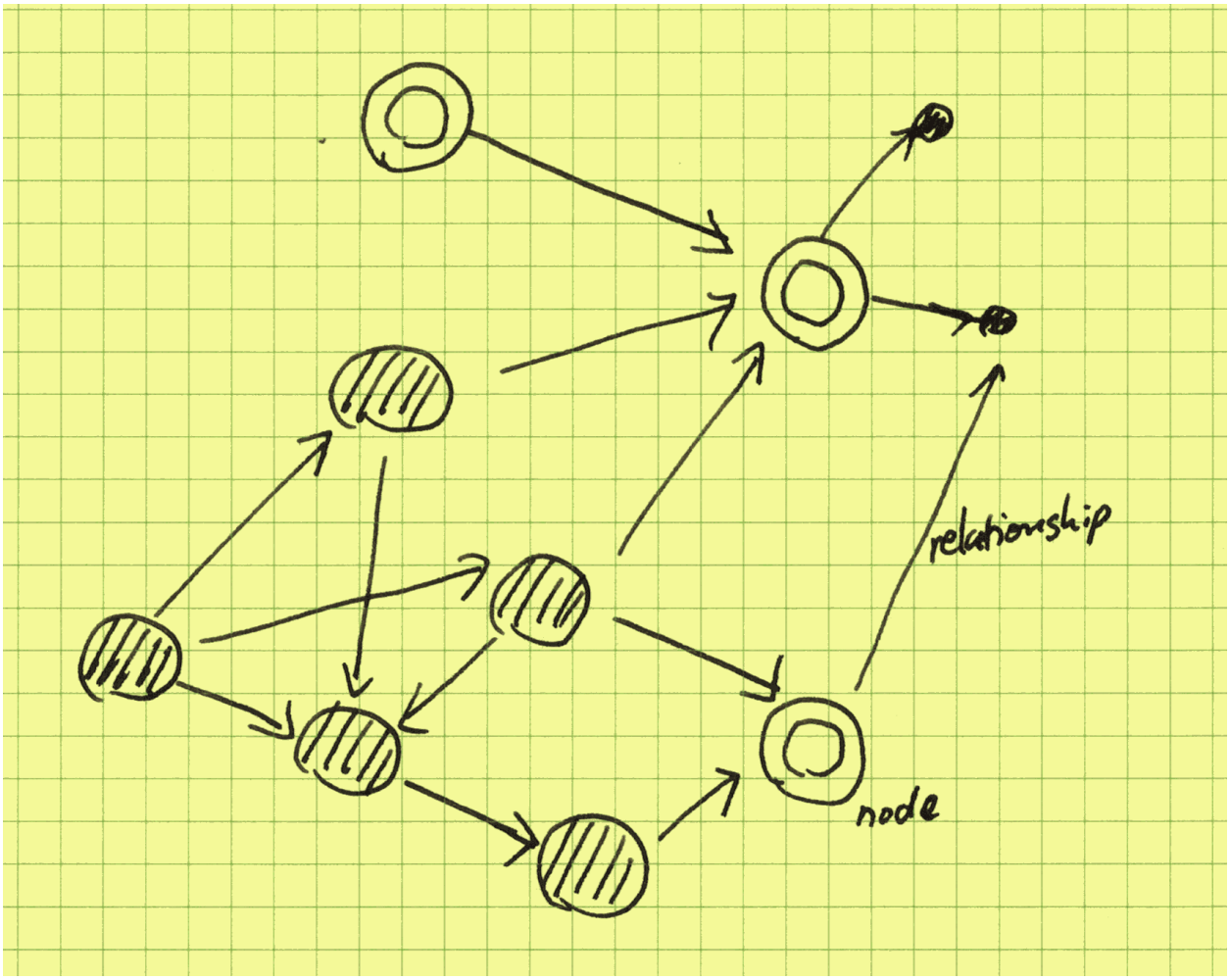
Then the shopaholics:

```
?[name, sum(amount)] :=
        *purchase{customer_id: c_id, product_id: p_id, quantity},
        *customer{id: c_id, name},
        *product{id: p_id, price},
        amount = price * quantity

:order -sum(amount)
:limit 10
```

These "insights" are bread-and-butter relational thinking, useful but quite shallow. In graph thinking, instead of mentally picturing customers and products as rows in tables, we picture them as dots on a canvas, with purchases as links

between them. In CozoDB, graph modeling is done implicitly: the `purchase` relation already acts as the edges. The mental picture is now:



Graphs are all about how things are connected to each other, and among themselves. Here, products are connected to other products, mediated by purchases. We can materialize this mediated graph:

```
?[prod_1, prod_2, count_unique(customer_id)] :=
        *purchase{customer_id, product_id: prod_1},
        *purchase{customer_id, product_id: prod_2}

:create co_purchase {prod_1, prod_2 => weight: count_unique(customer_id)}
```

Here, the edge weights of the `co_purchase` graph are the number of distinct customers that have bought both of the products. We also directly saved the result in a new stored relation, for easier manipulation later (creating relations in Cozo is very cheap).

With this graph at hand, the famous diaper-beer correlation from the big-data era is then simple to see: if you start with a diaper product, the product connected to it with the largest weight is the most correlated product to it according to the data at hand. Maybe even more interesting (and difficult to do in a traditional relational database) is the *centrality* of products; here, we can simply use the PageRank[60] algorithm:

[60] https://en.wikipedia.org/wiki/PageRank

```
?[product, pagerank] <~ PageRank(*co_purchase[])

:order -pagerank
:limit 10
```

If you run a supermarket, it may be beneficial to put the most central product in the most prominent display, as this is likely to drive the most sales of other products (as suggested by the data, but whether this really works must be tested).

Augmenting graphs with knowledge and ontologies

You can try to derive more graphs from the sales dataset and experiment with different graph algorithms running on them, for example, using community detection to find groups of customers with a common theme. But there is a limit to what can be achieved. For example, the product "iPhone" and the product "Samsung phone" are not related in the dataset, though all of us can immediately see that they are both under the umbrella concept of smartphones. This latent relationship cannot be determined using e.g. correlation; in fact, sales of the two products are likely anti-correlated. But one would expect iPhone 15 and iPhone 11 to be correlated.

So, to derive more insights from the dataset, we need to augment it with knowledge graphs or ontologies (the distinction between the two need not concern us here). In concrete terms, someone would have already compiled a relation for us, for example:

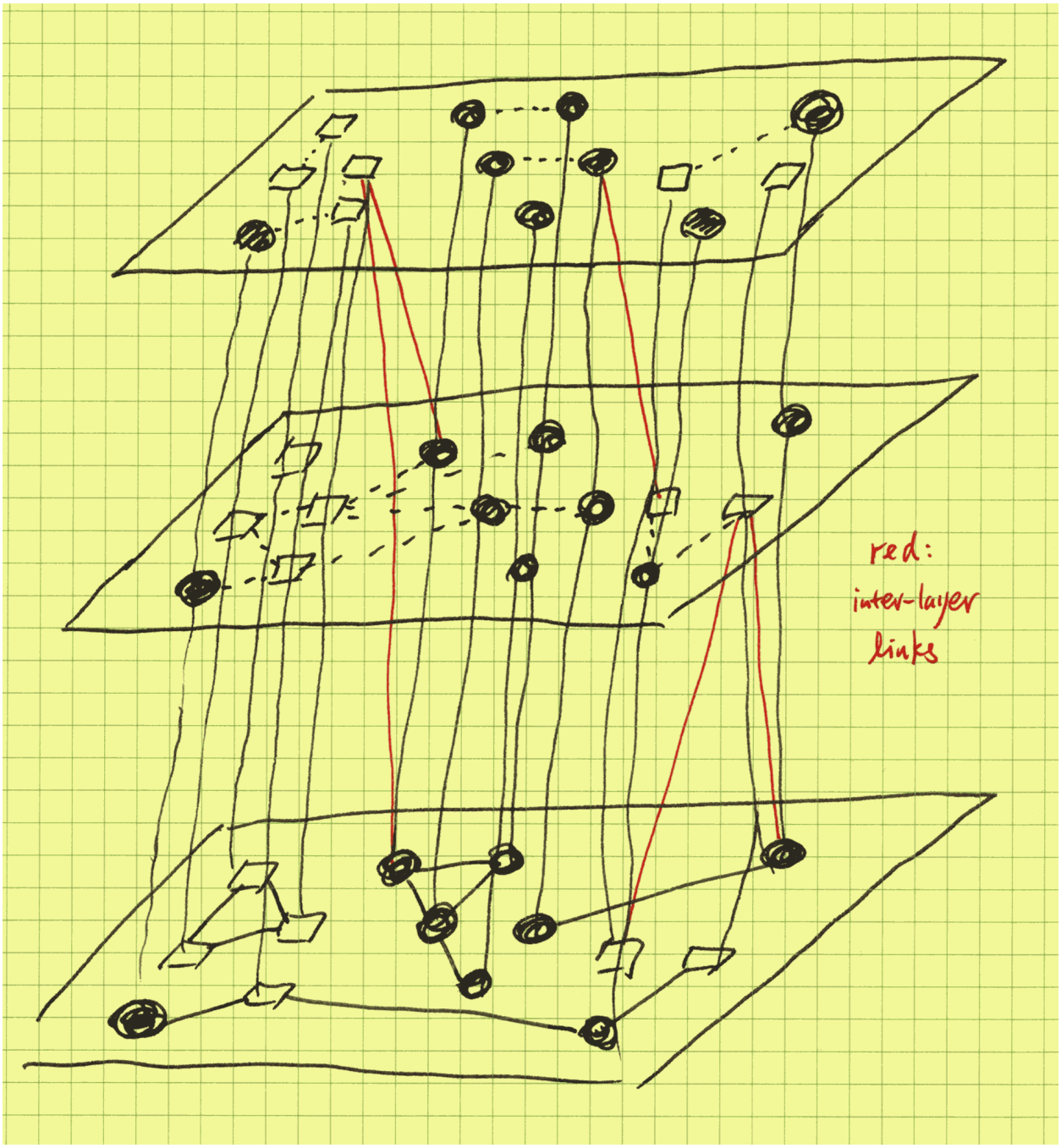
```
:create relation{subject, verb, object}
```

With this, the iPhone–Android relationship may be discovered:

```
?[verb, concept] :=
        *relation{subject: "iPhone", verb, object},
        *relation{subject: "Samsung phone", verb, object}
```

The result should show that `verb` is matched to `"is_a"` and `concept` is matched to `"smartphone"`. Replacing `"Samsung phone"` by `"iPad"` should result in the binding `verb: "made_by"` and `concept: "Apple"`.

The mental picture is now:

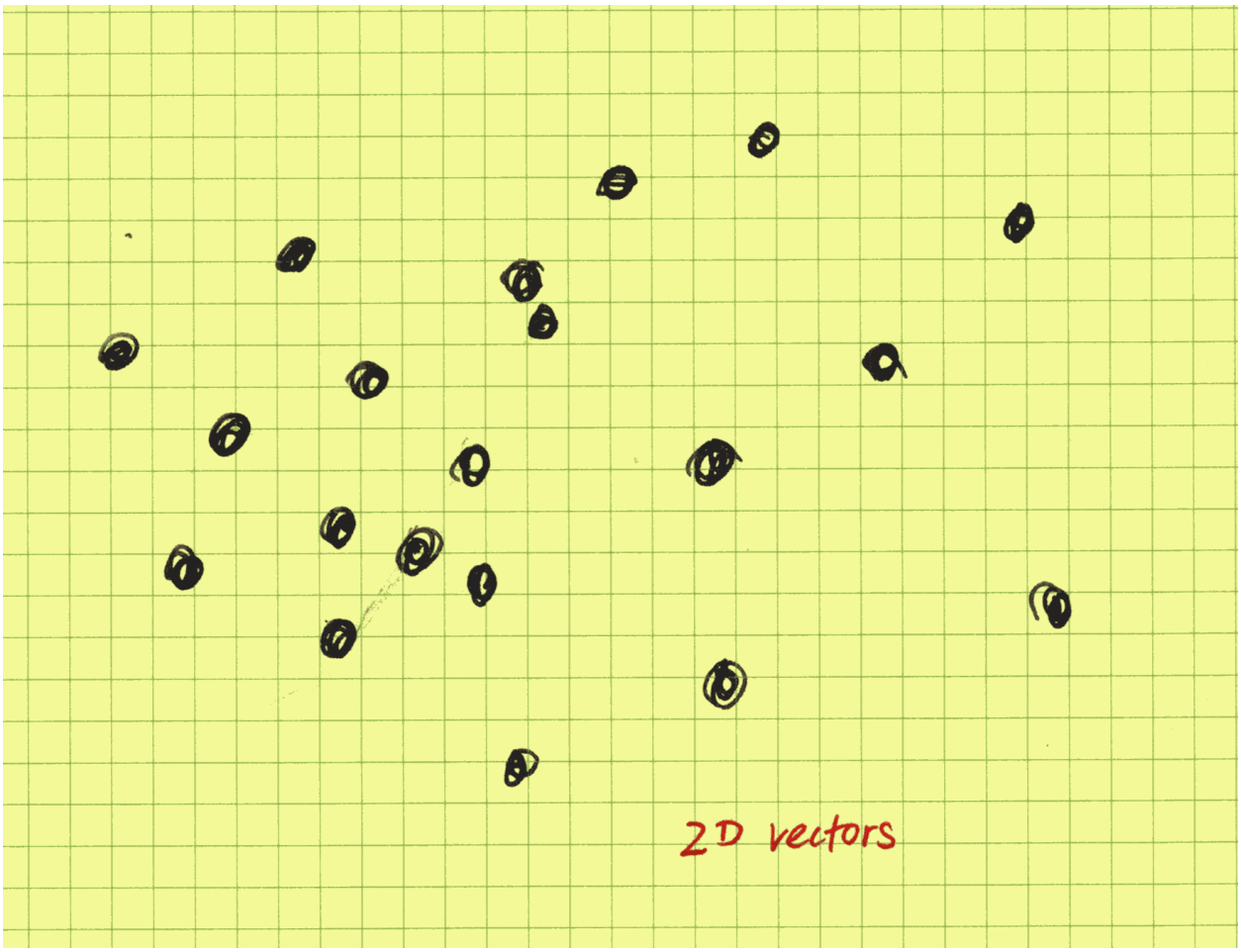


Instead of a single-layer flat graph, we now have a layered graph, with the upper layers provided by the externally-provided knowledge graphs. In the picture we have drawn many layers, as the real power of this approach shows when we have many knowledge graphs from diverse sources, and insights may be derived by comparing and contrasting. The values of multiple knowledge graphs multiply when they are brought together instead of simple addition. In our running sales example, now using graph queries and algorithms, you can investigate competitors, complementary products, industry trends, sales patterns across different smartphone brands, customer segment-specific popularity, gaps and opportunities in the product catalogue, for example, which are all out of reach without the multi-layered approach.

LLMs provide implicit knowledge graphs

Okay, so knowledge graphs are cool, but why are they not more widely used? In fact, they are widely used, but only inside big corporations such as Google (rumors have it that Google runs the world's largest knowledge graphs, which are at play whenever you search). The reason is that knowledge graphs are expensive to make and difficult to maintain and keep up to date, and combining different knowledge graphs, while powerful, requires a tedious translation process. You may already have expected this from our example above: even if we can hire a thousand graduate students to code the knowledge graph for us, who decided to code the verb as `"is_a"` instead of `"is a"`? What about capitalization? Disambiguation? It is a difficult and brittle process. In fact, all we care about are the relationships, but the formalities hold us back.

Fortunately for us non-Googlers, the rise and rise of LLMs such as GPTs have paved a new way. With the newest version of CozoDB, all you need to do is to provide embeddings for the product description. Embeddings are just vectors in a metric space, and if two vectors are "near" each other according to the metric, then they are semantically related. Below we show some vectors in a 2-D space:



So now the product relation is:

```
:create product {
    id
    =>
    name,
    description,
    price,
```

(continues on next page)

(continued from previous page)

```
    name_vec: <F32; 1536>,
    description_vec: <F32; 1536>
}
```

To show our excitement, we have provided 1536-dimensional embeddings for both the name texts and description texts, and we also annotated the vector types to be specific. Next, we create a vector index:

```
::hnsw create product:semantic{
    fields: [name_vec, description_vec],
    dim: 1536,
    ef: 16,
    m: 32
}
```

This is an HNSW (hierarchical navigable small world) vector index, and `ef` and `m` are parameters that control the quality-performance trade-off of the index. Now when inserting rows for the `product` table, we use an embedding model (such as `text-embedding-ada-002` provided by OpenAI) to compute embeddings for the texts and insert them together with the other fields. Now an iPhone and a Samsung phone are related even without a manually curated knowledge graph:

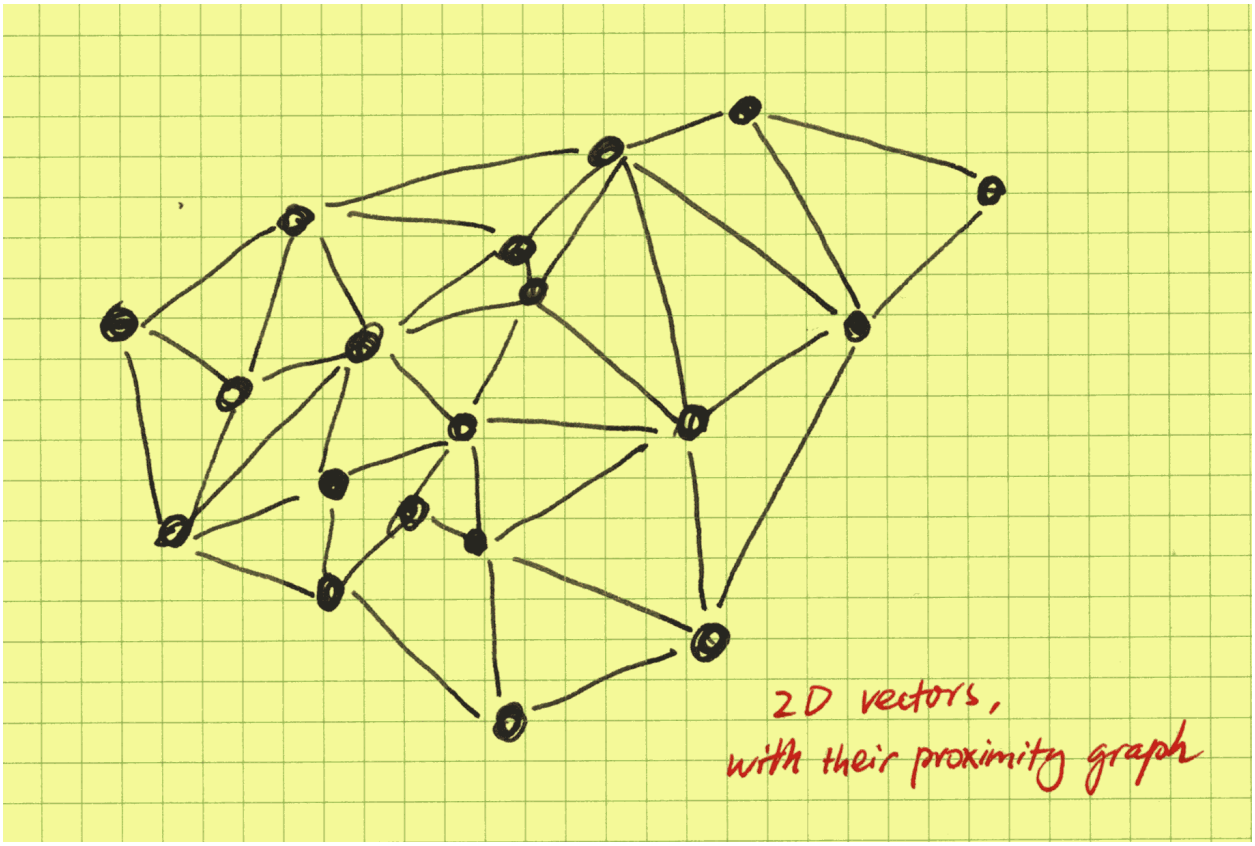
```
?[dist, name] :=
    *product{name: "iPhone", name_vec: v},
    ~product:semantic{name | query: v, bind_distance: dist, k: 10, ef: 50}

:order dist
:limit 10
```

This is a nearest-neighbor search in embedding space. The first result should be "iPhone" itself with a distance of zero, followed by the other smartphones according to their similarity with the iPhone.

What is the mental picture now? The HNSW algorithm is a pure-graph algorithm that builds a hierarchy of proximity graphs, with the base layer containing all the indexed nodes and the upper layers stochastically-selected nodes that comprise a renormalized[61], zoomed-out picture of the proximity graph. Below is an example of a proximity graph in a 2-D space:

[61] https://en.wikipedia.org/wiki/Renormalization#Renormalization_in_statistical_physics



In CozoDB, unlike in other databases, we expose the inner workings to the user whenever it makes sense. This is especially relevant in the case of HNSW indices. In the above example, since we already know that `"iPhone"` is in the index, we do not need to use vector search *at all* and can walk the proximity index directly to get its *proximity neighbors* (which are not the same as the nearest neighbors):

```
?[dist, name] :=
    *product:semantic{layer: 0, fr_name: "iPhone", to_name: name, dist}

:order dist
:limit 10
```

The power of this is that all the Datalog tricks and all the classical graph algorithms can be applied to the graph, and we are just walking the graph; there are no vector operations at all! As an illustration, we can try to find the "transitive closure" of the iPhone with clipped distance (using a community detection algorithm works much better, but here we want to show recursion):

```
iphone_related[name] :=
    *product:semantic{layer: 0, fr_name: "iPhone", to_name: name, dist},
    dist < 100
iphone_related[name] :=
    iphone_related[fr_name],
    *product:semantic{layer: 0, fr_name, to_name: name, dist},
    dist < 100
?[name] := iphone_related[name]
```

Now you will have all the iPhone-related products by walking only the (approximate) proximity graph with edges of distance smaller than 100.

Semantically, the HNSW search operation is no different from normal stored relation queries, so you can use them in recursions as well:

```
iphone_related[name] :=
    *product{name: "iPhone", name_vec: v},
    ~product:semantic{name | query: v, bind_distance: dist, k: 5, ef: 50}
iphone_related[name] :=
    iphone_related[other_name],
    *product{name: other_name, name_vec: v},
    ~product:semantic{name | query: v, bind_distance: dist, k: 5, ef: 50}
?[name] := iphone_related[name]
```

But this version will be slower than walking the index directly since now lots of vector operations are involved. It is most useful when we want to "jump" from one index to another: their proximity graphs are not connected, so you use vectors from each of them to make the connection.

This is rather cute, but how is it going to replace knowledge graphs? The proximity graph we have built is *generic*, making no distinction between an `"is_a"` relation and a `"made_by"` relation, for example.

There are many ways to solve this problem, which can be roughly divided into two classes.

In the first class, we use LLMs together with the proximity graph to build the knowledge graph automatically. For example, a properly-prompted LLM can look at `"iPhone"` and its neighbors and generate concepts and verbs as candidates for insertion into a knowledge graph, represented as a relation in CozoDB. The tags and verbs are then de-duped by using HNSW search in the embedding-enriched knowledge graph to prevent a situation where both `"iPhone"` and `"iphone"` are present, for example (it is recommended to let LLMs verify that its proposal is valid for important work). If you want to go down this route, better results are obtained if you already have an idea of what the verbs are in order to constrain the content generated by the LLMs.

The second class is even easier to implement: we just have to come up with a series of questions that we want to answer by consulting a knowledge graph, for example, "what is the manufacturer of this product", "in a supermarket, how is this product catalogued", etc., and we let the LLMs generate answers to these questions, with the product description given as context, and finally, we store the *answers* together with their embeddings in a dedicated HNSW index. Now the proximity graph for these answers constitutes the appropriate *section* of the required knowledge graph. You may need to wrap your head around a little bit when using this approach, but in fact, it works even better than the first as it is much more adaptable.

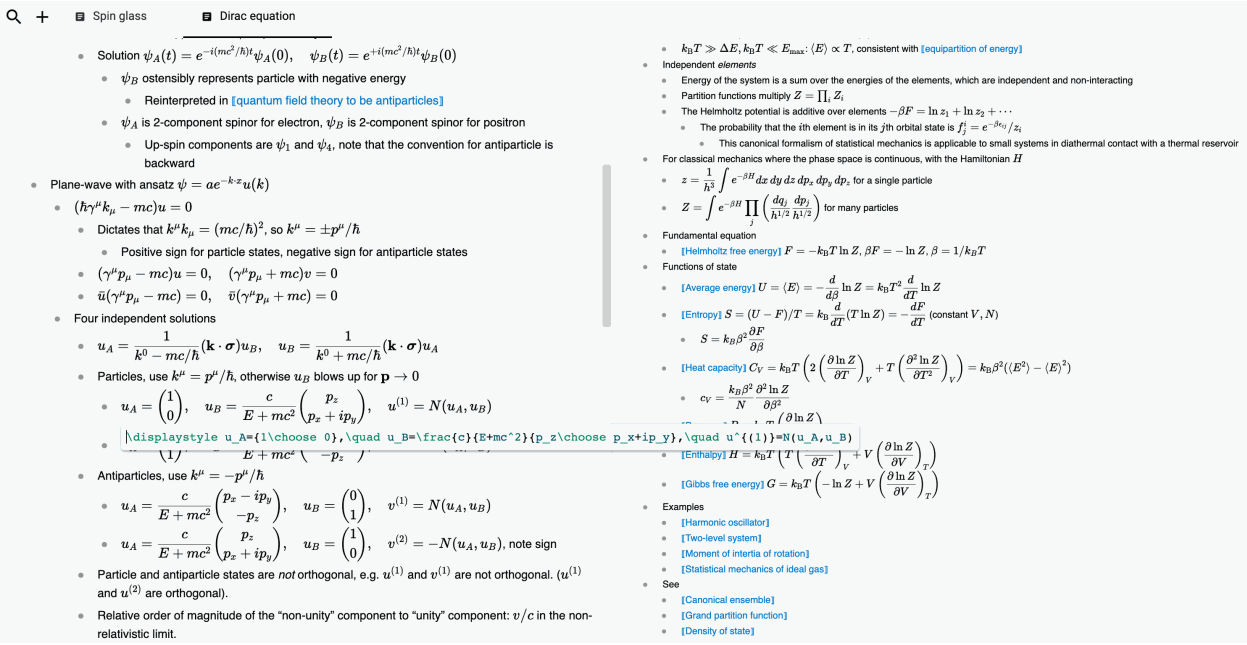
We hope that you have seen that CozoDB's seamless mixture of relational, graph, and vector search capabilities offers unique advantages and opens up unexplored territories for exploring novel ideas and techniques in data management and analysis!

14.6.3 Toward the hippocampus for AI

We have seen that CozoDB becomes much more powerful with its support for vector embeddings and implicit proximity graphs. In fact, this empowerment cuts both ways, and the possibilities are limitless. The following sections are highly speculative and maybe controversial, and I would be glad if they could lead to useful discussions.

Knowledge management for AI

Many people use knowledge management tools such as Roam Research, Obsidian, and Notion, just to name a few. The key feature of these tools is the ability to create links between different pieces of text. I am extremely frustrated with the existing solutions because they do not support LaTeX well (I want them to display equations nicely, and I want painless authoring of LaTeX formulas and useful debugging information), so I wrote my own:



Naturally, CozoDB is at its heart. With the advent of LLMs and CozoDB's inclusion of vector search capabilities, and with enough tweaks to my private knowledge management system, I find that explicit linking becomes increasingly unnecessary, as the connections can be derived automatically by the LLMs.

Then I began to have an eerie feeling that it is not me who is reading these notes anymore; it is the AI. And in fact, properly structured, *the AI can utilize CozoDB better than me*.

In the paper *Generative Agents: Interactive Simulacra of Human Behavior*[62], it is shown that an essential ingredient for generative agents is the ability to keep notes and to make effective use of them for chain-of-thought reasoning. In the paper, a linear array of memory cells is used, and we all know that our brain is not really like that. A proximity graph for memory storage, as implemented in CozoDB, seems a much better fit for a memory for AI.

In fact, even letting the LLM agent do random walks on the proximity graph can produce surprising (in a good sense) results, as this is really exploring a fuzzy associative memory, and the long-range links in the upper hierarchy can provide the occasional "leaps of imagination".

[62] https://arxiv.org/abs/2304.03442

The fractal von Neumann architecture

The current generation of LLMs runs on computers with the von Neumann architecture: a CPU that performs operations on data stored in memory, with data and instructions mixed in the same memory space.

Now if we consider individual LLMs themselves as CPUs (hence fractal), then their knowledge management tools such as CozoDB act as the memory. Here too, instructions (what the LLM must do, i.e., prompts) and data (contexts) are mixed together. CozoDB has many appealing properties to act as AI's memory: a practical but important consideration is that the CPU had better control the memory exclusively; using it "on the cloud" doesn't perform too well. In developing CozoDB, it was decided very early on that this database must *scale*, and scaling means not only scaling up but also scaling down. The original motivation is to provide knowledge graph support wherever needed. Now this ubiquity of CozoDB certainly paves the way for offline intelligent agents. *Soft-erasure features* in the sense of customizable timestamped facts and assertions also help the agent better organize its own thoughts.

> Cutting-edge LLMs running on the phone are certainly possible *and near*. One of my favorite pastimes when new LLMs come out is to ask them to give pi to as many digits as they can (they may refuse, but with enough cajoling, they will eventually comply). The interesting thing is that they start giving wrong results after a few hundred digits (maybe even less), but not totally wrong: there are large segments of correct sequences, just out of order. So today's LLMs are just totally wasteful in how they use their internalized memory: they remember lots of useless details. With time, and when trained with their own von Neumann memory, they will become more efficient in using their parameters, and then they won't need to be so big.

Does this scale even further? Autonomous agents, each with their private memory, communicating in a public arena (CozoDB running in the Cloud, for example)? What will they do then?

The interpretation of artificial dreams

To form a true community of agents, we need real individuals, not shallow carbon-copies of the same collective subconscious. Today's incarnation of GPTs is nothing more than a collective subconscious: different prompts will elicit different personalities and responses from them.

Private memory and individual fine-tuning of model weights according to private experience are of course required, but we need more than that. One hangover from the era of Big Data is the belief that all data must be preserved for later use. The reality is that we just get a bigger and bigger pile of rubbish that is harder and harder to make sense of. Humans don't do this. When awake, humans remember and reason, but when dreaming, humans distill, discard, and connect higher concepts together. Random-walking LLMs on proximity graphs can do this, and the constraints are no longer measured in gigabytes but instead in minutes (hours) and joules (calories). AI also needs to rest, reflect, and sleep, after all.

Towards intelligence

In a sense, today's AI is never awake. Waking people can answer three questions: Who am I? Where do I come from? Where am I going to? Often left unsaid is that the answers to these three questions must be *examined* (as in "the unexamined life is not worth living"[63]), and must not be recitations from a rulebook. This calls for a much deeper integration of the subconscious processing and memory.

So what is "examined"? Rote memorization certainly is not. A popular critique of LLMs as intelligent agents is that all they can do is continue the text, and what they can achieve can be, in principle, no more than what is already contained in the learned texts. This may be true if LLMs are trained with only supervised/unsupervised learning, but a crucial step in the production of systems such as ChatGPT is the final RLHF step. With reinforcement learning and enough experience, there is no reason to set any bounds on intelligence as dictated by the training material; otherwise, how did AlphaGo and friends beat humans? The later versions even have ZERO prior contexts for learning.

[63] https://en.wikipedia.org/wiki/The_unexamined_life_is_not_worth_living

Intelligence must precede the ability to examine. I do think current best systems display signs of intelligence, but they are extremely weak. One manifestation is that they are totally shallow thinkers, and this is more fundamental than the fact that the transformer architecture makes internally making plans difficult (making plans can be done inside the memory, i.e., the database, which is what humans do as well). The reason, I believe, is that the RLHF procedure currently applied is rather primitive. Being primitive has benefits: as we know, intelligent lifeforms need to reside on the edge of chaos and order; any highly developed systems will need to be intricate and must be evolved, not made. And when in unfavorable environments, they can quickly spiral to death. But evolution they will need. Now they have the hippocampus installed, the next roadblock is updating weights on the phone. We will see how we can get there.

Let's end this note with the following ancient maxim:

.

14.7 Version 0.7: MinHash-LSH near-duplicate indexing, Full-text search (FTS) indexing, Json values and update

Continuing with the empowerment of CozoDB by *vector search*, This version brings you a few more features!

14.7.1 MinHash-LSH indices

Let's say you collect news articles from the Internet. There will be duplicates, but these are not exact duplicates. How do you deduplicate them? Simple. Let's say your article is stored thus:

```
:create article{id: Int => content: String}
```

To find the duplicates, you create an LSH index on it:

```
::lsh create article:lsh {
    extractor: content,
    tokenizer: Simple,
    n_gram: 7,
    n_perm: 200,
    target_threshold: 0.7,
}
```

Now if you do this query:

```
?[id, content] := ~article:lsh {id, content | query: $q }
```

then articles with its content about 70% or more similar to the passed-in text in `$q` will be returned to you.

If you want, you can also mark the duplicates at insertion time. For this, use the following schema:

```
:create article{id: Int => content: String, dup_for: Int?}
```

Then at insertion time, use the query:

```
{
    ?[id, dup_for] := ~article:lsh {id, dup_for | query: $q, k: 1}
    :create _existing {id, dup_for}
}

%if _existing
```

(continues on next page)

(continued from previous page)

```
    %then {
        ?[id, content, dup_for] := *_existing[eid, edup],
                                   id = $id,
                                   content = $content,
                                   dup_for = edup ~ eid
        :put article {id => content, dup_for}
    }
    %else {
        ?[id, content, dup_for] <- [[$id, $content, null]]
        :put article {id => content, dup_for}
    }
%end
```

For our own use-case, this achieves about 20x speedup compared to using the equivalent Python library. And we are no longer bound by RAM.

As with vector search, LSH-search integrates seamlessly with Datalog in CozoDB.

14.7.2 Full-text search

After finding the duplicates, what if I want all articles that mentions the word "iPhone"? Using vector search seems such an overkill and may not yield good results. So we have full-text search indices as well!

To apply the FTS index:

```
::fts create article:fts {
    extractor: content,
    tokenizer: Simple,
    filters: [Lowercase, Stemmer('english'), Stopwords('en')]
}
```

and it's ready to be searched:

```
?[id, content, score] := ~article:fts {id, content | query: $q, bind_score: score }
:order -score
```

Passing `'iPhone'` to `$q` will give you articles that explicitly mention the iPhone, `'iPhone iPad'` will give you articles that mention both, and `'iPhone OR iPad'` will give you articles that mention either.

For more information on FTS and LSH, refer to the *proximity search chapter*.

14.7.3 Json values and update

Now we will have AI commentators analyze and comment on the articles. We will use the following schema:

```
:create article{id: Int => content: String, dup_for: Int?, comments: Json default {}}
```

The Json type is newly available. Now let's say our economics analyzer has produced a report for article 42, in the following Json:

```
{
    "economic_impact": "The economic impact of ChatGPT has been significant since its␣
```

(continues on next page)

(continued from previous page)

```
↪introduction. As an advanced language model, ChatGPT has revolutionized various␣
↪industries and business sectors. It has enhanced customer support services by␣
↪providing automated and intelligent responses, reducing the need for human␣
↪intervention. This efficiency has resulted in cost savings for businesses while␣
↪improving customer satisfaction. Moreover, ChatGPT has been utilized for market␣
↪research, content generation, and data analysis, empowering organizations to make␣
↪informed decisions quickly. Overall, ChatGPT has streamlined processes, increased␣
↪productivity, and created new opportunities, thus positively impacting the economy by␣
↪driving innovation and growth."
}
```

To merge this comment into the relation:

```
?[id, json] := id = $id, *article{id, json: old}, json = old ++ $new_report
:update article {id => json}
```

Note that with `:update`, we did not specify the `dup_for` field, and it will keep whatever its old value.

Next, our political analyzer AI weighs in:

```
{
    "political_impact": "The political impact of ChatGPT has been a subject of debate␣
↪and scrutiny. On one hand, ChatGPT has the potential to democratize access to␣
↪information and empower individuals to engage in political discourse. It can␣
↪facilitate communication between citizens and government officials, enabling greater␣
↪transparency and accountability. Additionally, it can assist in analyzing vast amounts␣
↪of data, helping policymakers make informed decisions. However, concerns have been␣
↪raised regarding the potential misuse of ChatGPT in spreading disinformation or␣
↪manipulating public opinion. The technology's ability to generate realistic and␣
↪persuasive content raises ethical and regulatory challenges, necessitating careful␣
↪consideration of its deployment to ensure fair and responsible use. As a result, the␣
↪political impact of ChatGPT remains complex, with both potential benefits and risks to␣
↪navigate."
}
```

We just run the same query but with a different $new_report bound.

Now if you query the database for article 42, you will see that its comments contain both reports!

There are many more things you can do with Json values: refer to *Functions and operators* and *Types* for more details.

14.7.4 Misc

In this version the semantics of `%if` in imperative scripts has *changed*: now a relation is considered truthy as long as it contains any row at all, regardless of the content of its rows. We found the old behaviour confusing in most circumstances.

Happy hacking!

INDEX

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W