ConceptQL Specification

ConceptQL (pronounced concept-Q-L) is a high-level language that allows researchers to unambiguously define their research algorithms.

Motivation for ConceptQL

Outcomes Insights intends to build a vast library of research algorithms and apply those algorithms to large databases of claims data. Early into building the library, we realized we had to overcome two major issues:

1. Methods sections of research papers commonly use natural language to specify the criteria used to build cohorts from a claims database.
    - Algorithms defined in natural language are often imprecise, open to multiple interpretations, and generally difficult to reproduce.
    - Researchers could benefit from a language that removes the ambiguity of natural language while increasing the reproducibility of their research algorithms.
2. Querying against claims databases is often difficult.
    - Hand-coding algorithms to extract cohorts from datasets is time-consuming, error-prone, and opaque.
    - Researchers could benefit from a language that allows algorithms to be defined at a high-level and then gets translated into the appropriate queries against a database.

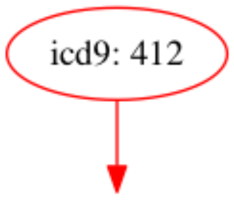
We developed ConceptQL to address these two issues.

We are writing a tool that can read research algorithms defined in ConceptQL. The tool can create a diagram for the algorithm which makes it easy to visualize and understand. The tool can also translate the algorithm into a SQL query which runs against data structured in OMOP's Common Data Model (CDM). The purpose of the CDM is to standardize the format and content of observational data, so standardized applications, tools and methods can be applied to them.

For instance, using ConceptQL we can take a statement that looks like this:

```
:icd9: '412'
```

And generate a diagram that looks like this:



```
---
:icd9: '412'
```

And generate SQL that looks like this:

```
SELECT *
FROM cdm_data.condition_occurrence AS co
JOIN vocabulary.source_to_concept_map AS scm ON (c.condition_concept_id = sc
m.target_concept_id)
WHERE scm.source_code IN ('412')
AND scm.source_vocabulary_id = 2
AND scm.source_code = co.condition_source_value
```

As stated above, one of the goals of ConcegtQL is to make it easy to assemble fairly complex queries without having to roll up our sleeves and write raw SQL. To accommodate this complexity, ConceptQL itself has some complexities of its own. That said, we believe ConceptQL will help researchers define, hone, and share their research algorithms.

ConceptQL Overview

What ConceptQL Looks Like

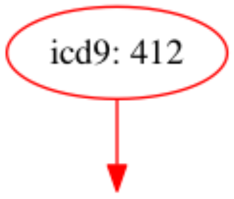
I find seeing examples to be the quickest way to get a sense of a language. Here is a trivial example to whet your appetite. The example is in YAML, but could just as easily be in JSON or any other markup language capable of representing nested sets of heterogeneous arrays and hashes. In fact, the ConceptQL "language" is a just set of nested hashes and arrays representing search criteria and some set operations and temporal operations to glue those criteria together.

```
# Example 1: A simple example in YAML
# This is just a simple hash with a key of :icd9 and a value of 412
# This example will search the condition_occurrence table for all conditions
 that match the ICD-9 concept of 412.
---
:icd9: '412'
```

ConceptQL Diagrams

Reading ConceptQL in YAML or JSON seems hard to me. I prefer to explore ConceptQL using directed graphs. For instance, the diagram for the simple example listed in YAML above is:

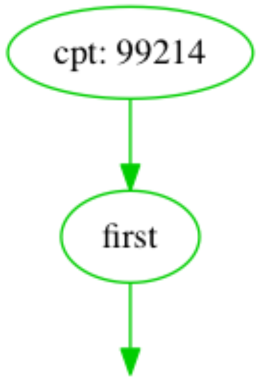
All Conditions Matching MI



```
---
:icd9: '412'
```

Each oval depicts a “node”, or rather, a ConceptQL expression. An arrow between a pair of nodes indicates that the results from the node on the tail of the arrow pass on to the node at the head of the arrow. A simple example should help here:

First Office Visit Per Patient



```
---
:first:
  :cpt: '99214'
```

The diagram above reads “get all procedures that match the CPT 99214 (Office Visit) and then filter them down to the first occurrence for each person”. The diagram is much more terse than that and to accurately read the diagram, you need a lot of implicit knowledge about how each node operates. Fortunately, this document will (hopefully) impart that knowledge to you.

Please note that all of my diagrams end with an arrow pointing at nothing. You’ll see why soon.

Think of Results as a Stream

I draw my ConceptQL diagrams with leaf nodes at the top and the “trunk” nodes at the bottom. I like to think of the results of a ConceptQL statement as a flowing stream of data. The leaf nodes, or nodes that gather results out of the database, act like tributaries. The results flow downwards and either join with other results, or filter out other results until the streams emerge at the bottom of the diagram. Think of each arrow as a stream of results, flowing down through one node to the next.

The trailing arrow in the diagrams serves as a reminder that ConceptQL yields a stream of results.

Streams have Types

You might have noticed that the nodes and edges in the diagrams often have a color. That color represents what "type" of stream the node or edge represents. There are many types in ConceptQL, and you'll notice they are **strongly** correlated with the tables found in CDM v4.0:

- condition_occurrence
    - red
- death
    - brown
- drug_cost
    - TBD
- drug_exposure
    - purple
- observation
    - TBD
- payer_plan_period
    - TBD
- person
    - blue
- procedure_cost
    - gold
- procedure_occurrence
    - green
- visit_occurrence
    - orange

Each stream has a point of origin (essentially, the table from which we pulled the results for a stream). Based on that origin, each stream will have a particular type. The stream carries this type information as it moves through each node. When certain nodes, particularly set and temporal operation nodes, need to perform filtering, they can use this type information to determine how to best filter a stream. There will be much more discussion about types woven throughout this document. For now, it is sufficient to know that each stream has a type.

You'll also notice that the trailing arrow(s) at the end of the diagrams indicate which types of streams are ultimately passed on at the end of a ConceptQL statement.

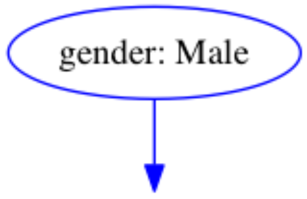
What *are* Streams Really?

Though I think that a “stream” is a helpful abstraction when thinking in ConceptQL, on a few occasions we need to know what’s going on under the hood.

Every table in the CDM structure has a surrogate key column (an ID column). When we execute a ConceptQL statement, the “streams” that are generated by the statement are just sets of these IDs for rows that matched the ConceptQL criteria. So each stream is just a set of IDs that point back to some rows in one of the CDM tables. When a stream has a “type” it is really just that the stream contains IDs associated with its table of origin.

So when we execute this ConceptQL statement, the resulting “stream” is all the person IDs for all male patients in the database:

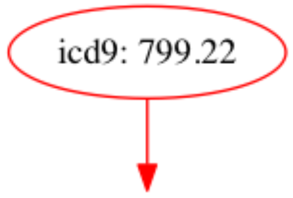
All Male Patients



```
---
:gender: Male
```

When we execute this ConceptQL statement, the resulting “stream” is all condition_occurrence IDs that match ICD–9 799.22:

All Condition Occurrences that match ICD–9 799.22



```
---
:icd9: '799.22'
```

Generally, I find it helpful to just think of those queries generating a “stream of people” or a “stream of conditions” and not worry about the table of origin or the fact that they are just IDs.

When a ConceptQL statement is executed, it yields a final set of streams that are just all the IDs that passed through all the criteria. What is done with that set of IDs is up to the user who assembled the ConceptQL statement. If a user gathers all 799.22 Conditions, they will end up with a set of condition_occurrence_ids. They could take those IDs and do all sorts of things like:

- Gather the first and last date of occurrence per person
- Count the number of occurrences per person
- Count number of persons with the condition
- Count the total number of occurrences for the entire population

This kind of aggregation and analysis is beyond the scope of ConceptQL. ConceptQL will get you the IDs of the rows you’re interested in, its up to other parts of the calling system to determine what you do with them.

Criterion Nodes

Criterion nodes are the parts of a ConceptQL query that search for specific values within the CDM data, e.g. searching the condition_occurrence table for a diagnosis of an old myocardial infarction (ICD–9 412) is a criterion. Criterion nodes are always leaf nodes.

There are *many* criterion nodes. A list of currently implemented nodes is available in Appendix A.

All Other Nodes

Virtually all other nodes add, remove, filter, or otherwise alter streams of results. They are discussed in this section.

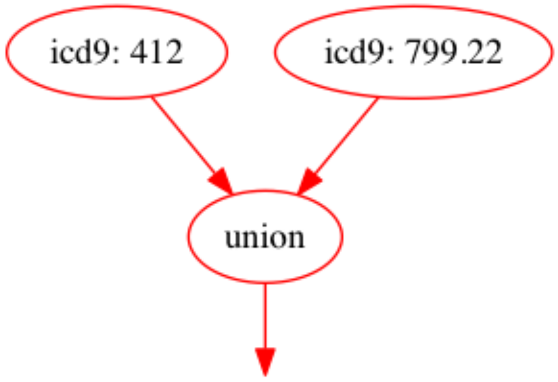
Set Operation Nodes

Because streams represent sets of results, its makes sense to include a nodes that operate on sets

Union

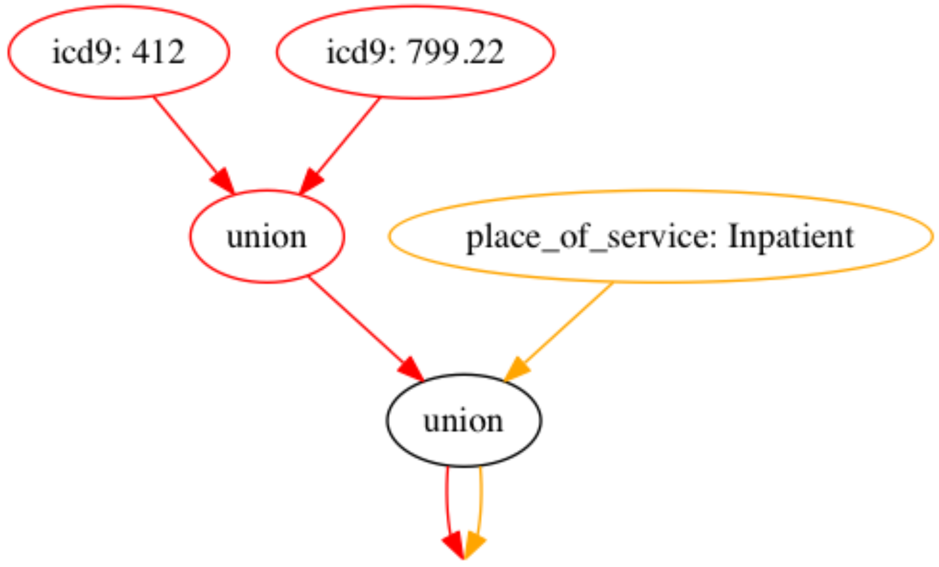
- Takes any number of child nodes and aggregates their streams
    - Unions together streams with identical types
        - Think of streams with the same type flowing together into a single stream
        - We're really just gathering the union of all IDs for identically-typed streams
    - Streams with the different types flow along together concurrently without interacting
        - It does not make sense to union, say, condition_occurrence_ids with visit_occurrence_ids, so streams with different types won't mingle together, but will continue to flow downstream in parallel

Two streams of the same type (condition_occurrence) joined into a single stream



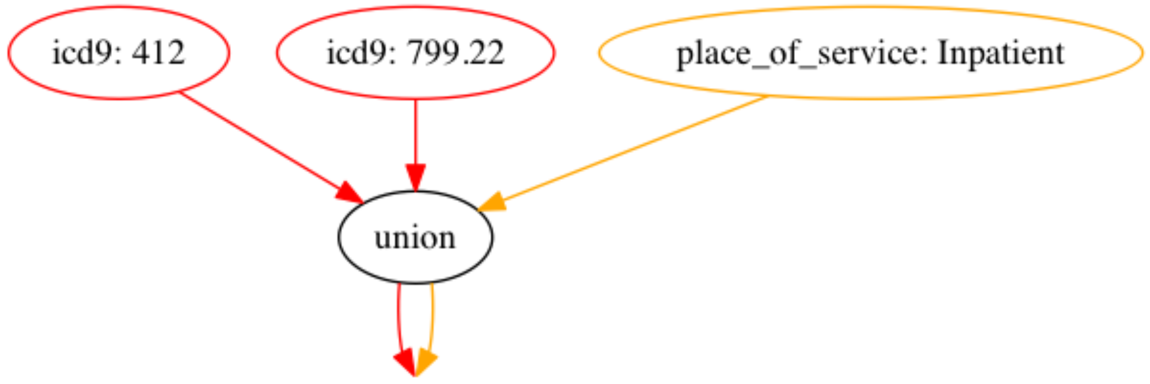
```
---
:union:
- :icd9: '412'
- :icd9: '799.22'
```

Two streams of the same type (condition_occurrence) joined into a single stream, then a different stream (visit_occurrence) flows concurrently



```
---
:union:
- :union:
  - :icd9: '412'
  - :icd9: '799.22'
- :place_of_service: Inpatient
```

Two streams of the same type (condition_occurrence) joined into a single stream, along with a different stream (visit_occurrence) flows concurrently (same as above example)

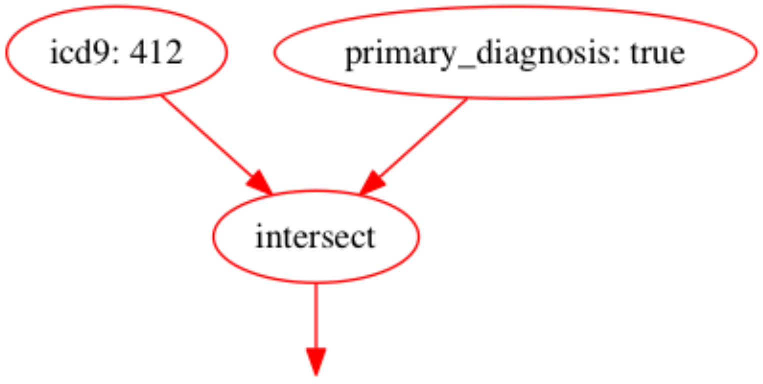


```
---
:union:
- :icd9: '412'
- :icd9: '799.22'
- :place_of_service: Inpatient
```

Intersect

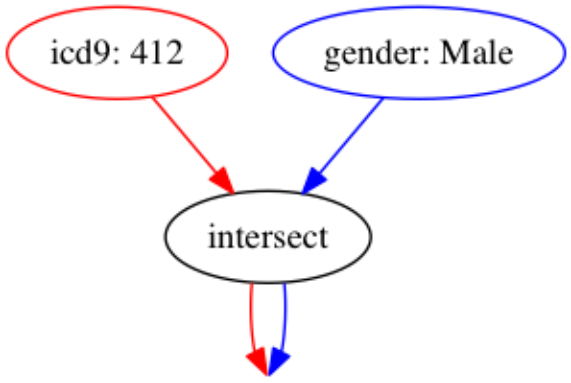
1. Group incoming streams by type
2. For each group of same-type streams
    1. Intersect all streams, yielding a single stream that contains only those IDs common to those streams
3. A single stream for each incoming type is sent downstream
    1. If only a single stream of a type is upstream, that stream is essentially unaltered as it is passed downstream

Yields a single stream of all Conditions where MI was Primary Diagnosis. This involves two Condition streams and so results are intersected



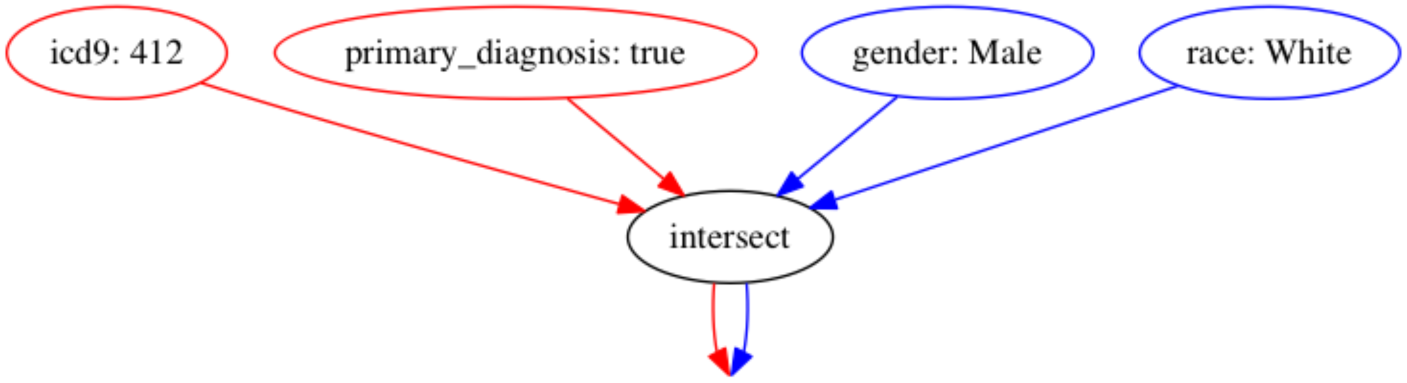
```
---
:intersect:
- :icd9: '412'
- :primary_diagnosis: true
```

Yields two streams: a stream of all MI Conditions and a stream of all Male patients. This is essentially the same behavior as Union in this case



```
---
:intersect:
- :icd9: '412'
- :gender: Male
```

Yields two streams: a stream of all Conditions where MI was Primary Diagnosis and a stream of all White, Male patients.

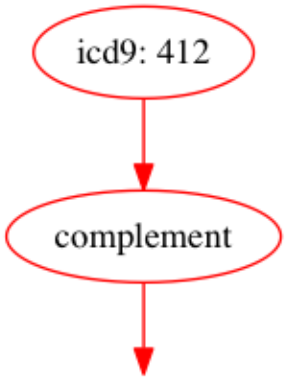


```
---
:intersect:
- :icd9: '412'
- :primary_diagnosis: true
- :gender: Male
- :race: White
```

Complement

This node will take the complement of each set of IDs in the incoming streams.

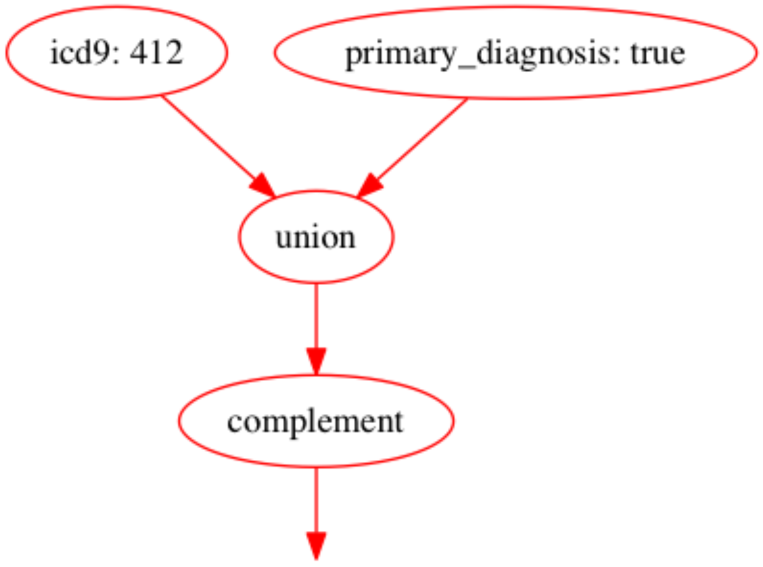
All non-MI Conditions



```
---
:complement:
  :icd9: '412'
```

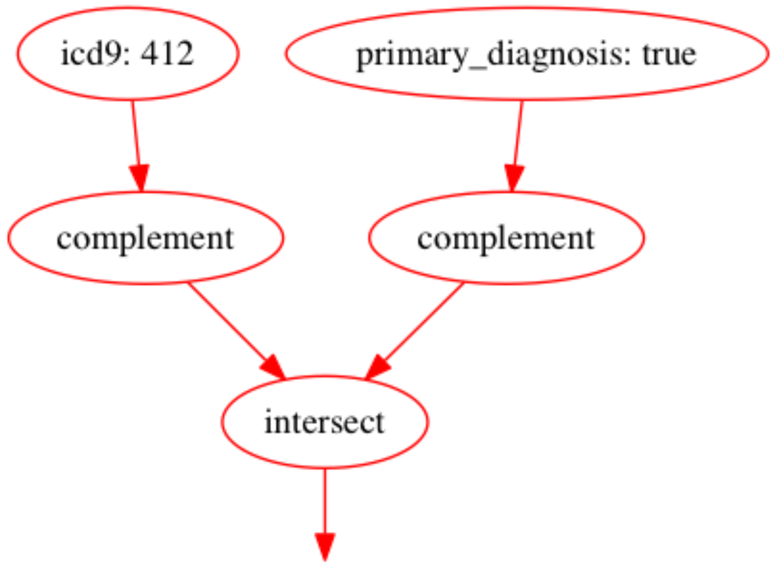
If you’re familiar with set operations, the complement of a union is the intersect of the complements of the items unioned. So in our world, these next two examples are identical:

All Conditions where the Condition isn’t an MI as the Primary Diagnosis



```
---
:complement:
  :union:
  - :icd9: '412'
  - :primary_diagnosis: true
```

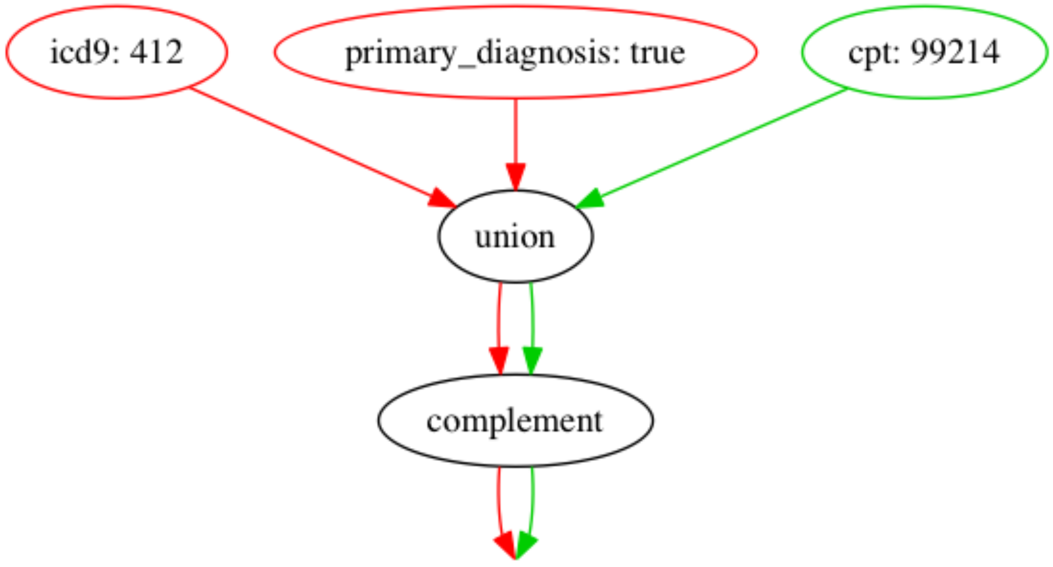
All Conditions where the Condition isn’t an MI as the Primary Diagnosis (same as above)



```
---
:intersect:
- :complement:
    :icd9: '412'
- :complement:
    :primary_diagnosis: true
```

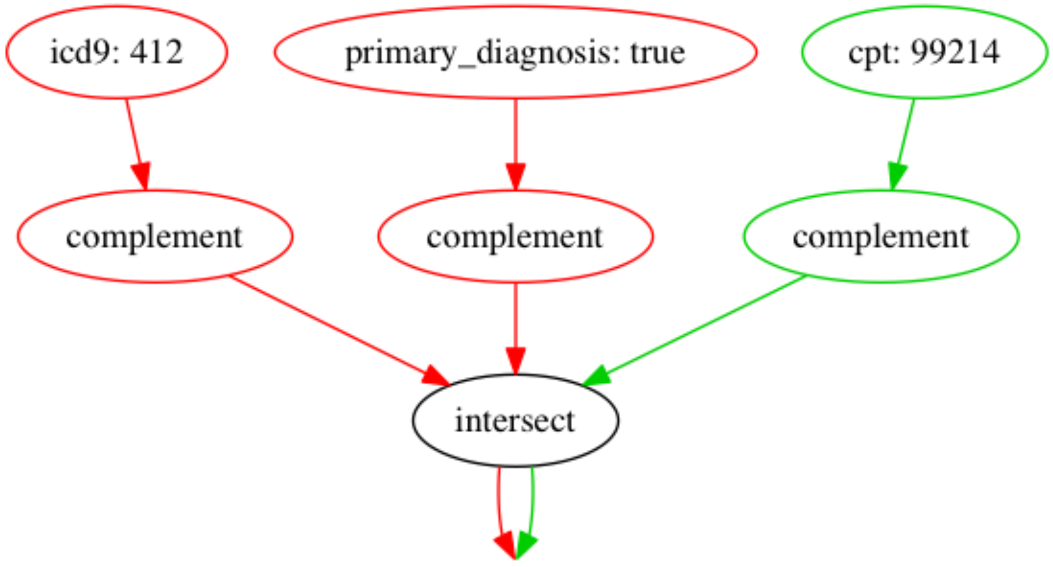
But please be aware that this behavior of complement only affects streams of the same type. If more than one stream is involved, you need to evaluate the effects of complement on a stream-by-stream basis:

Yields two streams: a stream of all Conditions where the conditions isn’t an MI and Primary Diagnosis and a stream of all non-office visit Procedures



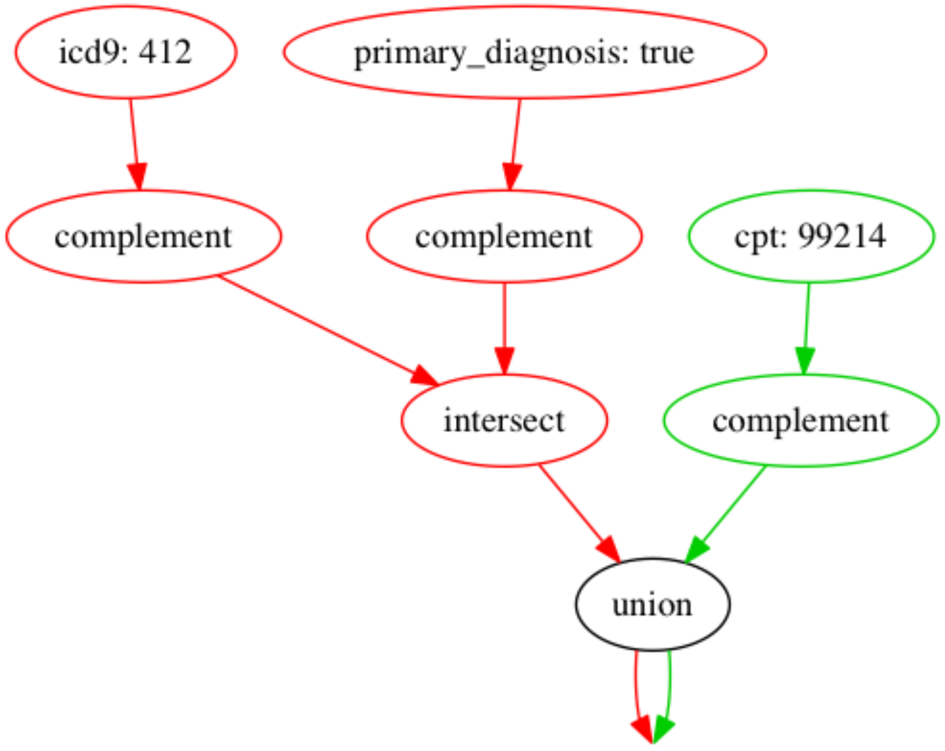
```
---
:complement:
  :union:
  - :icd9: '412'
  - :primary_diagnosis: true
  - :cpt: '99214'
```

Yields two streams: a stream of all Conditions where the conditions isn’t an MI and Primary Diagnosis and a stream of all non-office visit Procedures (same as above)



```
---
:intersect:
- :complement:
    :icd9: '412'
- :complement:
    :primary_diagnosis: true
- :complement:
    :cpt: '99214'
```

Yields two streams: a stream of all Conditions where the conditions isn't an MI and Primary Diagnosis and a stream of all non-office visit Procedures (same as above)

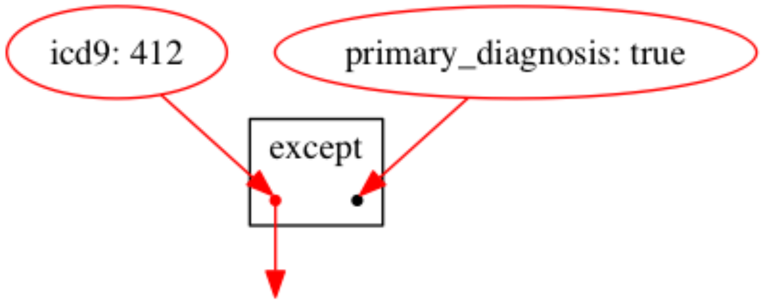


```
---
:union:
- :intersect:
  - :complement:
      :icd9: '412'
  - :complement:
      :primary_diagnosis: true
- :complement:
    :cpt: '99214'
```

Except

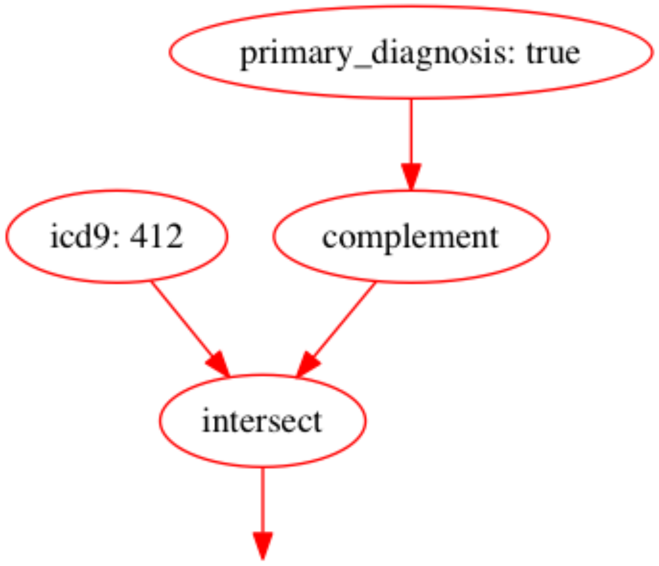
This node takes two sets of incoming streams, a left-hand stream and a right-hand stream. The node matches like-type streams between the left-hand and right-hand streams. The node removes any results in the left-hand stream if they appear in the right-hand stream. The node passes only results for the left-hand stream downstream. The node discards all results in the right-hand stream. For example:

All Conditions that are MI unless they are primary diagnoses



```
---
:except:
  :left:
    :icd9: '412'
  :right:
    :primary_diagnosis: true
```

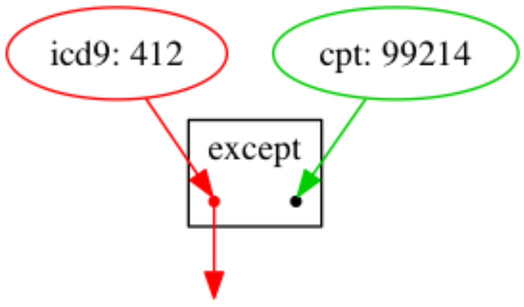
All Conditions that are MI unless they are primary diagnoses (same as above)



```
---
:intersect:
- :icd9: '412'
- :complement:
    :primary_diagnosis: true
```

If the left-hand stream has no types that match the right-hand stream, the left-hand stream passes through unaffected:

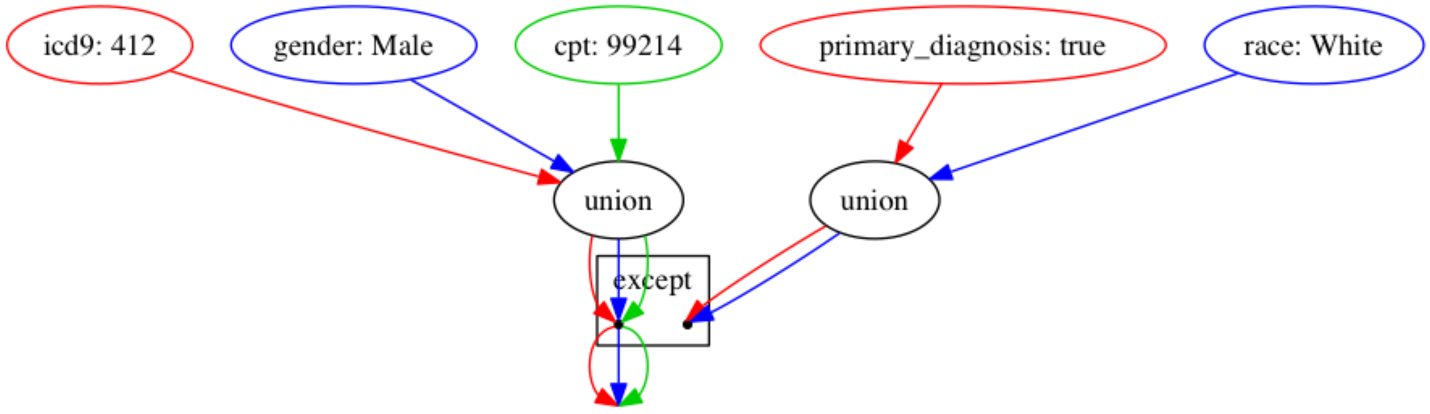
All Conditions that are MI



```
---
:except:
  :left:
    :icd9: '412'
  :right:
    :cpt: '99214'
```

And just to show how multiple streams behave:

Passes three streams downstream: a stream of Conditions that are MI but not primary diagnosis, a stream of People that are Male but not White, and a stream of Procedures that are office visits (this stream is completely unaffected by the right hand stream)



```
---
:except:
  :left:
    :union:
    - :icd9: '412'
    - :gender: Male
    - :cpt: '99214'
  :right:
    :union:
    - :primary_diagnosis: true
    - :race: White
```

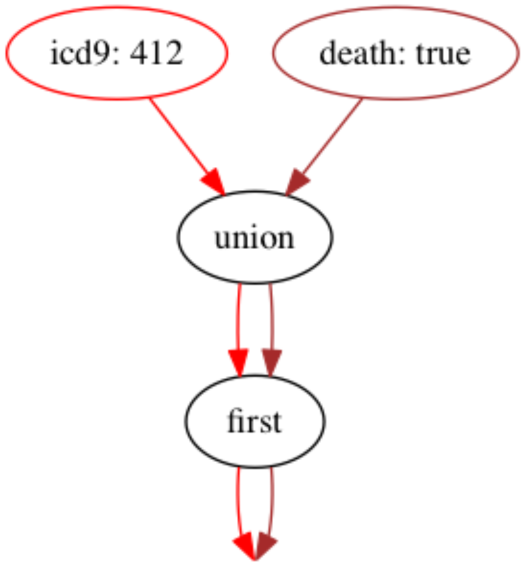
Discussion about Set Operation Nodes

Union Nodes

Q. Why should we allow two different types of streams to continue downstream concurrently?

- This feature lets us do interesting things, like find the first occurrence of either an MI or Death as in the example below
    - Throw in a few more criteria and you could find the first occurrence of all censor events for each patient

First occurrence of either MI or Death for each patient

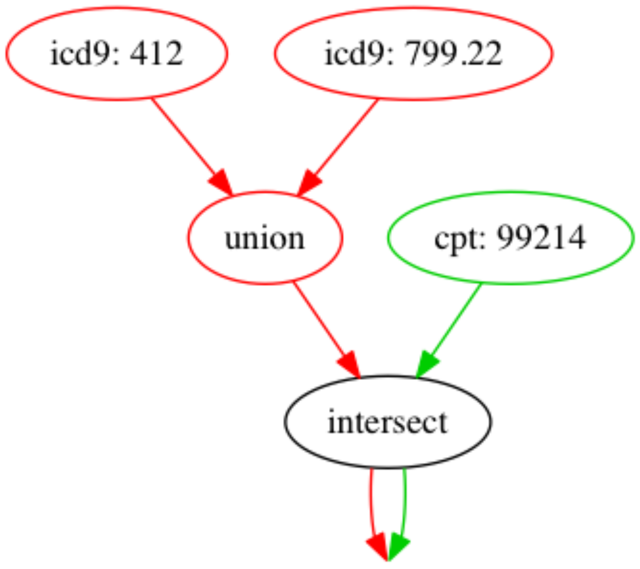


```
---
:first:
  :union:
  - :icd9: '412'
  - :death: true
```

Q. Why aren’t all streams passed forward unaltered? Why union like-typed streams?

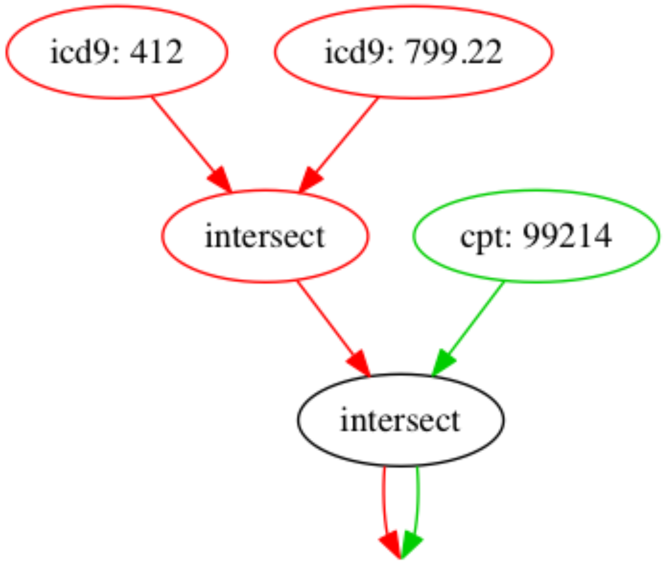
- The way Intersect works, if we passed like-typed streams forward without unioning them, Intersect would end up intersecting the two un-unioned like-type streams and that’s not what we intended
- Essentially, these two diagrams would be identical:

Two streams: a stream of all Conditions matching either 412 or 799.22 and a stream of Procedures matching 99214



```
---
:intersect:
- :union:
  - :icd9: '412'
  - :icd9: '799.22'
- :cpt: '99214'
```

Two streams: a stream of all Conditions matching either 412 AND 799.22 (an empty stream, a condition cannot be both 412 and 799.22 at the same time) and a stream of Procedures matching 99214



```
---
:intersect:
- :intersect:
  - :icd9: '412'
  - :icd9: '799.22'
- :cpt: '99214'
```

Time-oriented Nodes

All results in a stream carry a start_date and end_date with them. All temporal comparisons of streams use these two date columns. Each result in a stream derives its start and end date from its corresponding row in its table of origin.

For instance, a visit_occurrence result derives its start_date from visit_start_date and its end_date from visit_end_date.

If a result comes from a table that only has a single date value, the result derives both its start_date and end_date from that single date, e.g. an observation result derives both its start_date and end_date from its corresponding row's observation_date.

The person stream is a special case. Person results use the person's date of birth as the start_date and end_date. This may sound strange, but we will explain below why this makes sense.

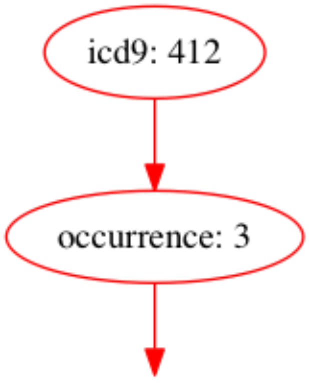
Relative Temporal Nodes

When looking at a set of results for a person, perhaps we want to select just the chronologically first or last result. Or maybe we want to select the 2nd result or 2nd to last result. Relative temporal nodes provide this type of filtering. Relative temporal nodes use a result's start_date to do chronological ordering.

occurrence

- Takes a two arguments: the stream to select from and an integer argument
- For the integer argument
    - Positive numbers mean 1st, 2nd, 3rd occurrence in chronological order
        - e.g. 1 => first
        - e.g. 4 => fourth
    - Negative numbers mean 1st, 2nd, 3rd occurrence in reverse chronological order
        - e.g. –1 => last
        - e.g. –4 => fourth from last
    - 0 is undefined?

For each patient, select the Condition that represents the third occurrence of an MI

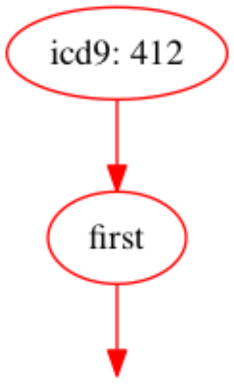


```
---
:occurrence:
- :icd9: '412'
- 3
```

first

- Node that is shorthand for writing “occurrence: 1”

For each patient, select the Condition that represents the first occurrence of an MI

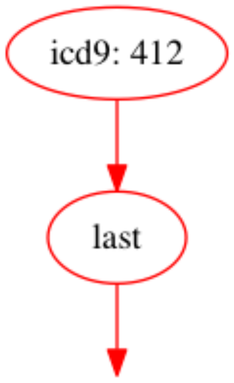


```
---
:first:
  :icd9: '412'
```

last

- Node that is just shorthand for writing “occurrence: –1”

For each patient, select the Condition that represents the last occurrence of an MI



```
---
:last:
  :icd9: '412'
```

Date Literals

For situations where we need to represent pre-defined date ranges, we can use “date literal” nodes.

date_range

- Takes a hash with two elements: { start: <date-format>, end: <date-format> }
- Creates an inclusive, continuous range of dates defined by a start and end date

day

- Takes a single argument: <date-format>
- Represents a single day
- Shorthand for creating a date range that starts and ends on the same date
- *Not yet implemented*

What is <date-format>?

Dates follow these formats:

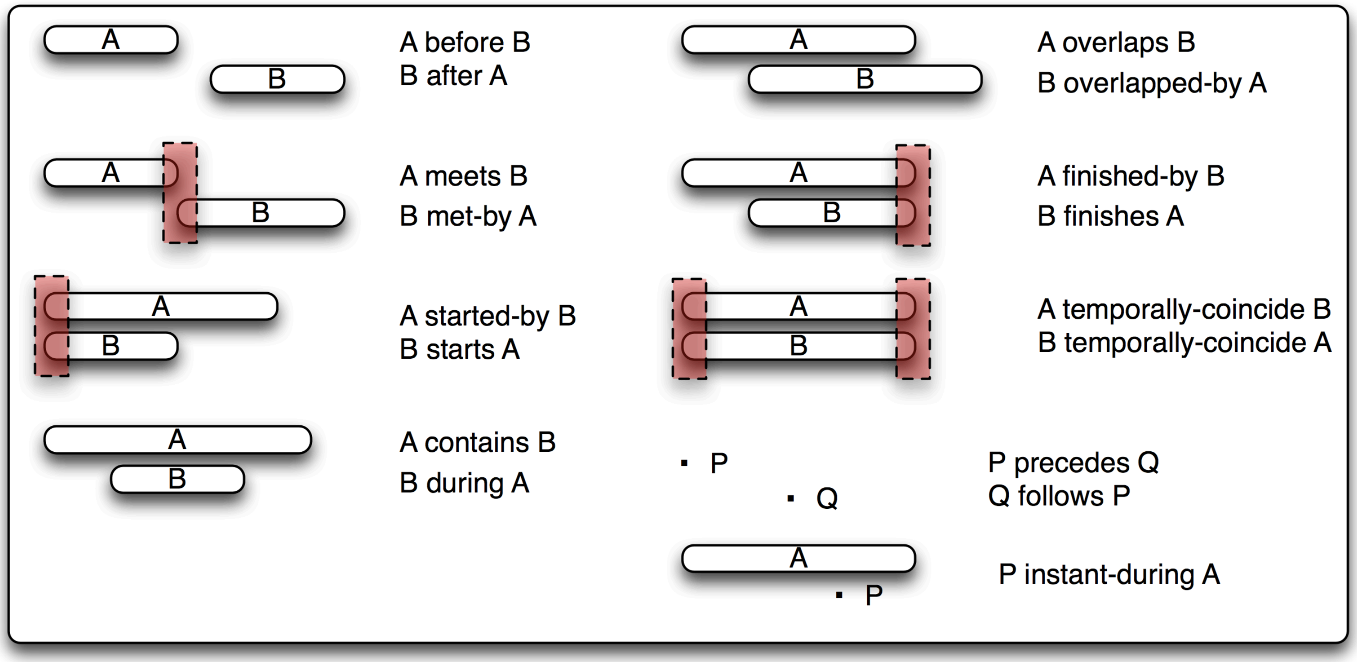
- “YYYY-MM-DD”
    - Four-digit year, two-digit month with leading 0s, two-digit day with leading 0s
- “START”
    - Represents the first date of information available from the data source
- “END”
    - Represents the last date of information available from the data source.

Temporal Comparison Nodes

As described above, each result carries a start and end date, defining its own date range. It is through these date ranges that we are able to do temporal filtering of streams via temporal nodes.

Temporal nodes work by comparing a left-hand stream (L) against a right-hand stream (R). R can be either a set of streams or a pre-defined date range. Each temporal node has a comparison operator which defines how it compares dates between L and R. A temporal node passes results only from L downstream. A temporal node discards all results in the R stream after it makes all comparisons.

The available set of temporal nodes comes from the work of Allen's Interval Algebra[1]. Interval Algebra defines 13 distinct temporal relationships, as shown in this handy chart borrowed from this website:



Our implementation of this algebra is originally going to be as strict as listed here, meaning that:

- Before/After
  - There must be a minimum 1-day gap between date ranges
- Meets/Met-by
  - Only if the first date range starts/ends a day before the next date range ends/starts
- Started-by/Starts
  - The start dates of the two ranges must be equal and the end dates must not be
- Finished-by/Finishes
  - The end dates of the two ranges must be equal and the start dates must not be
- Contains/During
  - The start/end dates of the two ranges must be different from each other
- Overlaps/Overlapped-by
  - The start date of one range and the end date of the other range must be outside the overlapping range
- Temporally coincides

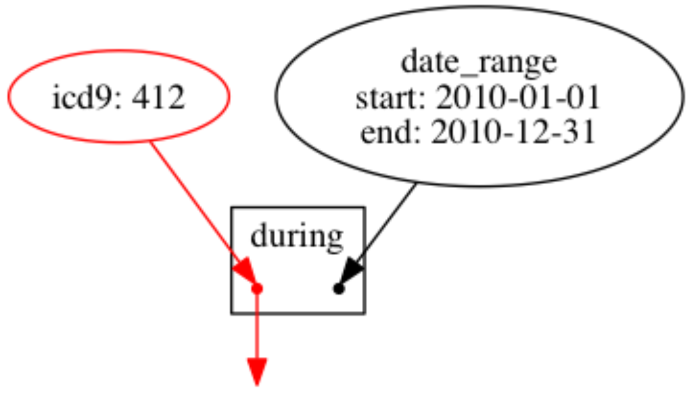
- Start dates must be equal, end dates must be equal

Ryan’s Sidebar on These Definitions:

> *These strict definitions may not be particularly handy or even intuitive. It seems like contains, starts, finishes, and coincides are all examples of overlapping ranges. Starts/finishes seem to be examples of one range containing another. Meets/met-by seem to be special cases of before/after. But these definitions, if used in their strict sense, are all mutually exclusive.*
>
> *We may want to adopt a less strict set of definitions, though their meaning may not be as easily defined as the one provided by Allen’s Interval Algebra*

When comparing results in L against a date range, results in L continue downstream only if they pass the comparison.

All MIs for the year 2010

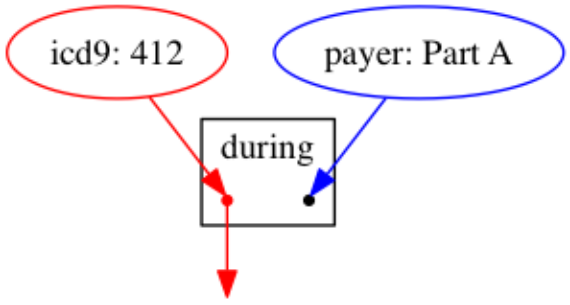


```
---
:during:
  :left:
    :icd9: '412'
  :right:
    :date_range:
      :start: '2010-01-01'
      :end: '2010-12-31'
```

When comparing results in L against a set of results in R, the temporal node compares results in stream L against results in stream R on a person-by-person basis.

- If a person has results in L or R stream, but not in both, none of their results continue downstream
- On a per person basis, the temporal node joins all results in the L stream to all results in the R stream
    - Any results in the L stream that meet the temporal comparison against any results in the R stream continue downstream

All MIs While Patients had Part A Medicare



```
---
:during:
  :left:
    :icd9: '412'
  :right:
    :payer: Part A
```

Edge behaviors

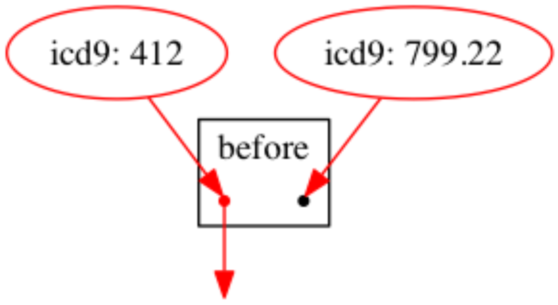
For 11 of the 13 temporal nodes, comparison of results is straight-forward. However, the before/after nodes have a slight twist.

Imagine events 1–1–2–1–2–1. In my mind, three 1’s come before a 2 and two 1’s come after a 2. Accordingly:

- When comparing L **before** R, the temporal node compares L against the **LAST** occurrence of R per person
- When comparing L **after** R, the temporal node compares L against the **FIRST** occurrence of R per person

If we're looking for events in L that occur before events in R, then any event in L that occurs before the last event in R technically meet the comparison of "before". The reverse is true for after: all events in L that occur after the first event in R technically occur after R.

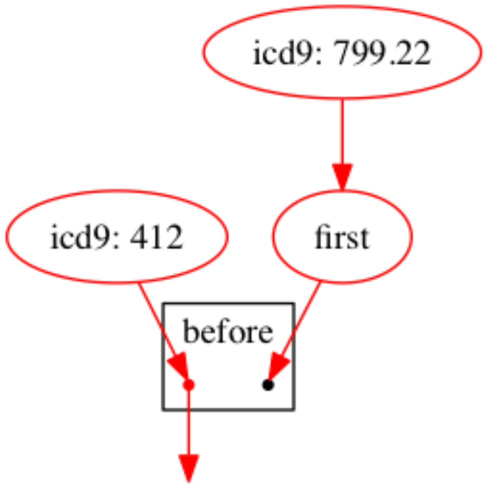
All MIs that occurred before a patient's **last** case of irritability (799.22)



```
---
:before:
  :left:
    :icd9: '412'
  :right:
    :icd9: '799.22'
```

If this is not the behavior you desire, use one of the sequence nodes to select which event in R should be the one used to do comparison

All MIs that occurred before a patient's **first** case of irritability (799.22)



```
---
:before:
  :left:
    :icd9: '412'
  :right:
    :first:
      :icd9: '799.22'
```

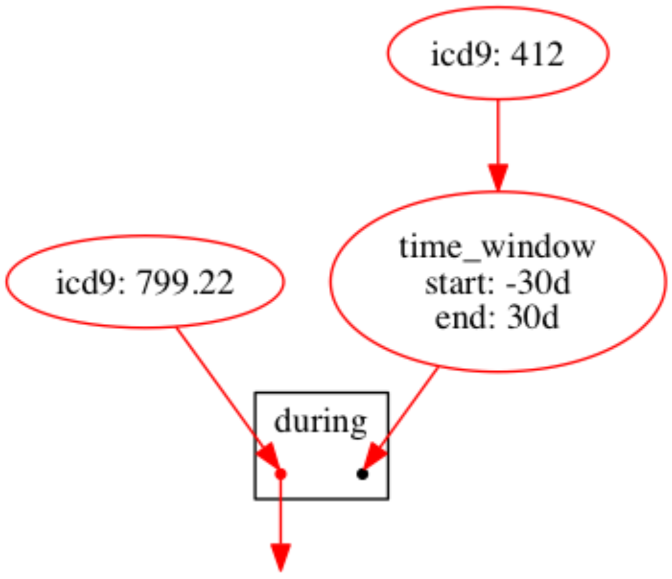
Time Windows

There are situations when the date columns associated with a result should have their values shifted forward or backward in time to make a comparison with another set of dates.

time_window

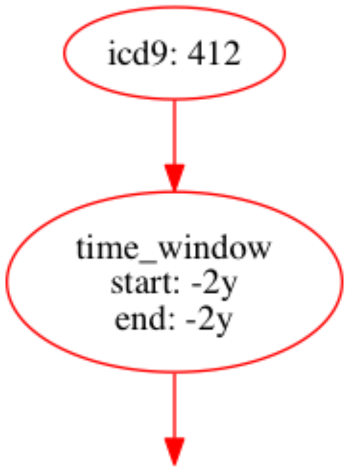
- Takes 2 arguments
    - First argument is the stream on which to operate
    - Second argument is a hash with two keys: [:start, :end] each with a value in the following format: “(-?\d+[dmy])+”
        - Both start and end must be defined, even if you are only adjusting one of the dates
    - Some examples
        - 30d => 30 days
        - 20 => 20 days
        - d => 1 day
        - 1y => 1 year
        - –1m => –1 month
        - 10d3m => 3 months and 10 days
        - –2y10m–3d => –2 years, +10 months, –3 days
    - The start or end value can also be ”, ‘0’, or nil
        - This will leave the date unaffected
    - The start or end value can also be the string ‘start’ or ‘end’
        - ‘start’ represents the start_date for each result
        - ‘end’ represents the end_date for each result
        - See the example below

All Diagnoses of Irritability (ICD–9 799.22) within 30 days of an MI



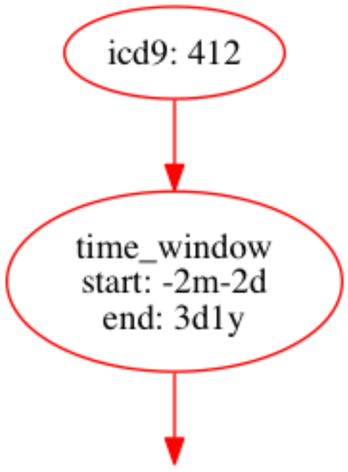
```
---
:during:
  :left:
    :icd9: '799.22'
  :right:
    :time_window:
    - :icd9: '412'
    - :start: "-30d"
      :end: 30d
```

Shift the window for all MIs back by 2 years



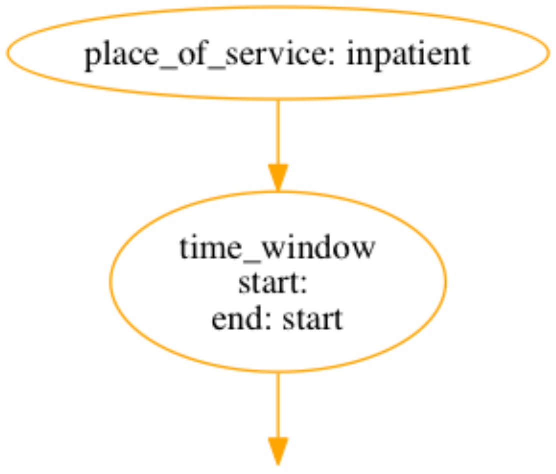
```
---
:time_window:
- :icd9: '412'
- :start: "-2y"
  :end: "-2y"
```

Expand the dates for all MIs to a window ranging from 2 months and 2 days prior to 1 year and 3 days after the MI



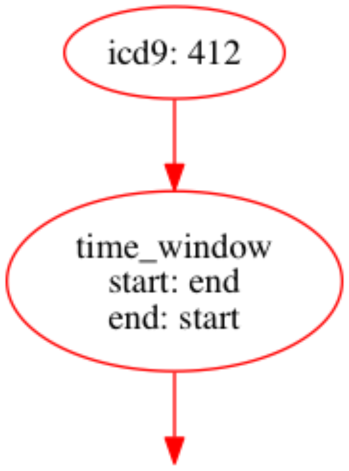
```
---
:time_window:
- :icd9: '412'
- :start: "-2m-2d"
  :end: 3d1y
```

Collapse all hospital visits' date ranges down to just the date of admission by leaving start_date unaffected and setting end_date to start_date



```
---
:time_window:
- :place_of_service: inpatient
- :start: ''
  :end: start
```

Nonsensical, but allowed: swap the start_date and end_date for a range

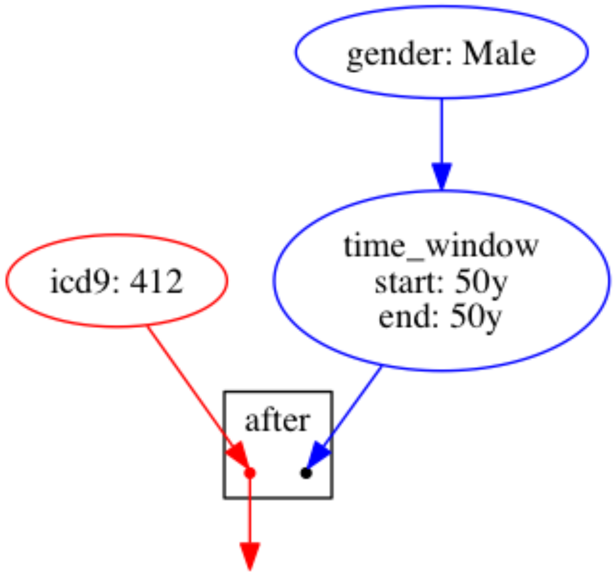


```
---
:time_window:
- :icd9: '412'
- :start: end
  :end: start
```

Temporal Nodes and Person Streams

Person streams carry a patient’s date of birth in their date columns. This makes them almost useless when they are part of the L stream of a temporal node. But person streams are useful as the R stream. By `time_window`ing the patient’s date of birth, we can filter based on the patient’s age like so:

All MIs that occurred after a male patient’s 50th birthday



```
---
:after:
  :left:
    :icd9: '412'
  :right:
    :time_window:
    - :gender: Male
    - :start: 50y
      :end: 50y
```

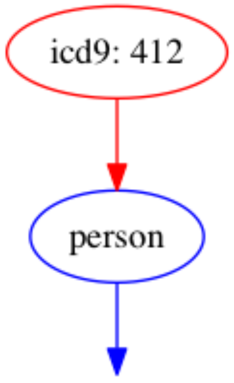
Type Conversion

There are situations where it is appropriate to convert the type of a stream of results into a different type. In programmer parlance, we say “typecasting” or “casting”, which is the terminology we’ll use here. A good analogy and mnemonic for casting is to think of taking a piece of metal, say a candle holder, melting it down, and recasting it into, say, a lamp. We’ll do something similar with streams. We’ll take, for example, a visit_occurrence stream and recast it into a stream of person.

Casting to person

- Useful if we’re just checking for the presence of a condition for a person
- E.g. We want to know *if* a person has an old MI, not when an MI or how many MIs occurred

All People Who Had an MI



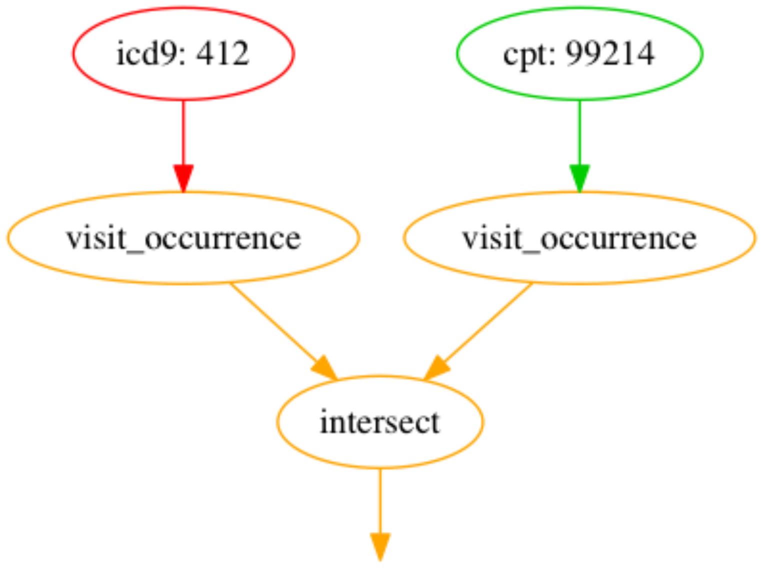
```
---
:person:
  :icd9: '412'
```

Casting to a visit_occurrence

- It is common to look for a set of conditions that coincide with a set of procedures
- Gathering conditions yields a condition stream, gathering procedures yields a procedure stream
    - It is not possible to compare those two streams directly using AND
    - It is possible to compare the streams temporally, but CDM provides a visit_occurrence table to explicitly tie a set of conditions to a set of procedures
- Casting both streams to visit_occurrence streams allows us to gather all

visit_occurrences for which a set of conditions/procedures occurred in the same visit

All Visits Where a Patient Had an MI During and Office Visit

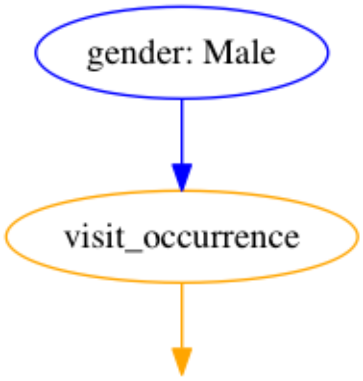


```
---
:intersect:
- :visit_occurrence:
    :icd9: '412'
- :visit_occurrence:
    :cpt: '99214'
```

Many tables have a foreign key (FK) reference to the visit_occurrence table. If we cast a result to a visit_occurrence, and its table of origin has a visit_occurrence_id FK column, the result becomes a visit_occurrence result corresponding to the row pointed to by visit_occurrence_id. If the row's visit_occurrence_id is NULL, the result is discarded from the stream.

If the result's table of origin has no visit_occurrence_id column, we will instead replace the result with ALL visit_occurrences for the person assigned to the result. This allows us to convert between a person stream and visit_occurrence stream and back. E.g. we can get all male patients, then ask for their visit_occurrences later downstream.

All Visits for All Male Patients



```
---
:visit_occurrence:
  :gender: Male
```

Casting Loses All Original Information

After a result undergoes casting, it loses its original information. E.g. casting a visit_occurrence to a person loses the visit_occurrence information and resets the start_date and end_date columns to the person's date of birth. As a side note, this is actually handy if a stream's dates have been altered by a time_window node and you want the original dates later on. Just cast the stream to its same type and it will regain its original dates.

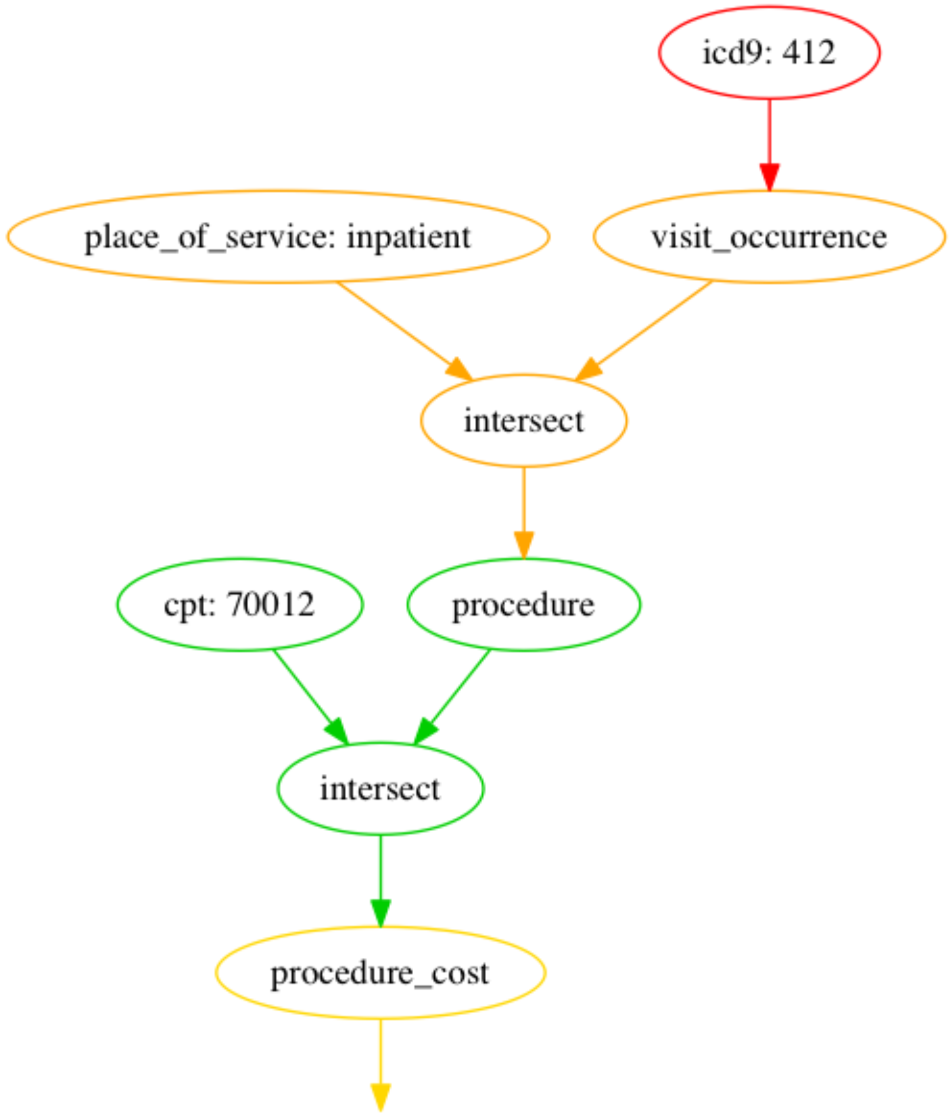
Cast all the Things!

Although casting to visit_occurrence and person are the most common types of casting, we can cast to and from any of the types in the ConceptQL system.

The general rule will be that if the source type has a defined relationship with the target type, we'll cast using that relationship, e.g. casting visit_occurrences to procedures will turn all visit_occurrence results into the set of procedure results that point at those original visit_occurrences. But if there is no direct relationship, we'll do a generous casting, e.g. casting observations to procedures will return all procedures for all persons in the observation stream.

INSERT HANDY TABLE SHOWING CONVERSION MATRIX HERE

Cost of 70012 while Hospitalized for MI

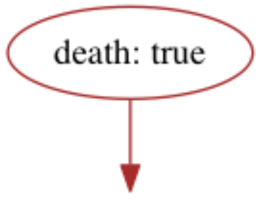


```
---
:procedure_cost:
  :intersect:
  - :cpt: '70012'
  - :procedure:
      :intersect:
      - :place_of_service: inpatient
      - :visit_occurrence:
          :icd9: '412'
```

Casting as a way to fetch all rows

The casting node doubles as a way to fetch all rows for a single type. Provide the casting node with an argument of `true` (instead of an upstream node) to get all rows as results:

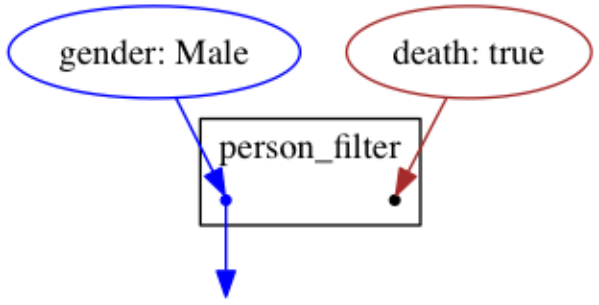
All death results in the database



```yaml
---
:death: true
```

This comes in handy for situations like these:

All Male patients who died



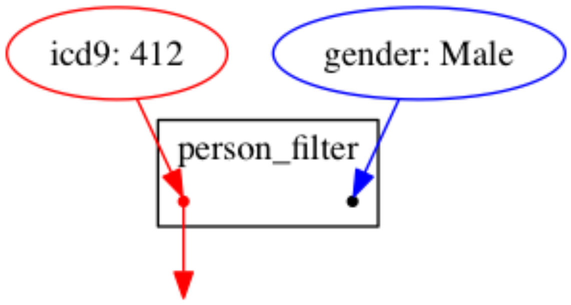
```yaml
---
:person_filter:
  :left:
    :gender: Male
  :right:
    :death: true
```

Filtering by People

Often we want to filter out a set of results by people. For instance, say we wanted to find all MIs for all males. We'd use the person_filter node for that. Like the Except node, it takes a left-hand stream and a right-hand stream.

Unlike the `except` node, the person_filter node will use all types of all streams in the right-hand side to filter out results in all types of all streams on the left hand side.

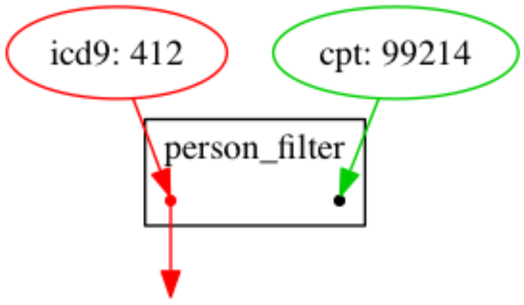
All MI Conditions for people who are male



```
---
:person_filter:
  :left:
    :icd9: '412'
  :right:
    :gender: Male
```

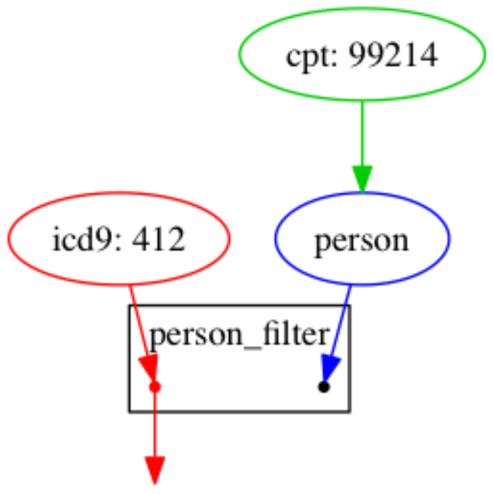
But we can get crazier. The right-hand side doesn't have to be a person stream. If a non-person stream is used in the right-hand side, the person_filter will cast all right-hand streams to person first and use the union of those streams:

All MI Conditions for people who had an office visit at some point in the data



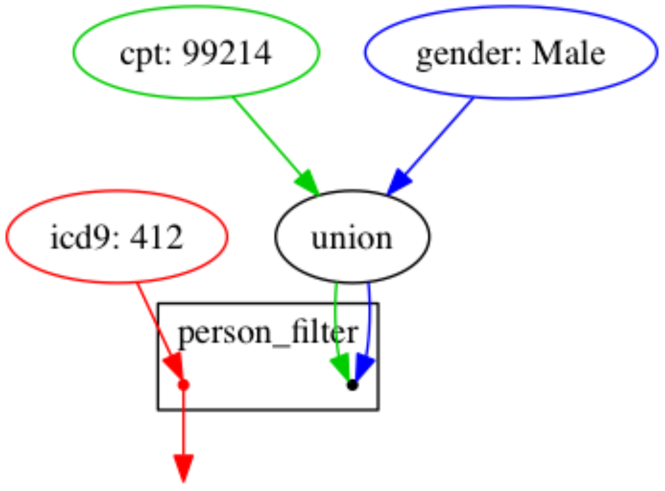
```
---
:person_filter:
  :left:
    :icd9: '412'
  :right:
    :cpt: '99214'
```

All MI Conditions for people who had an office visit at some point in the data (an explicit representation of what’s happening in the diagram above)



```
---
:person_filter:
  :left:
    :icd9: '412'
  :right:
    :person:
      :cpt: '99214'
```

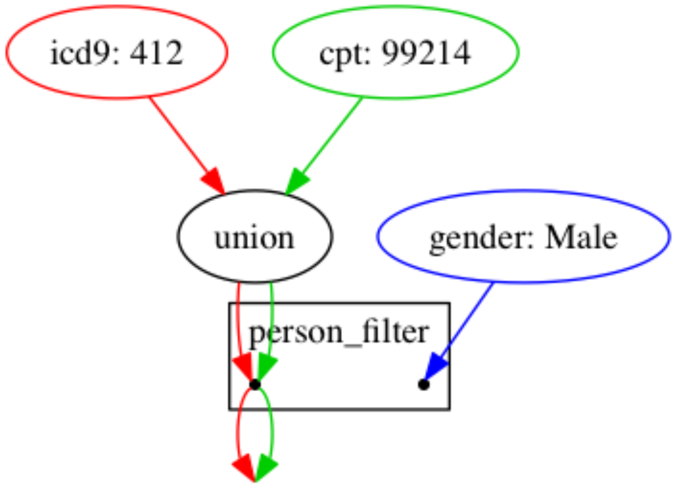
All MI Conditions for people who are Male OR had an office visit at some point in the data



```
---
:person_filter:
  :left:
    :icd9: '412'
  :right:
    :union:
    - :cpt: '99214'
    - :gender: Male
```

And don't forget the left-hand side can have multiple types of streams:

Yields two streams: a stream of all MI Conditions for people who are Male and a stream of all office visit Procedures for people who are Male



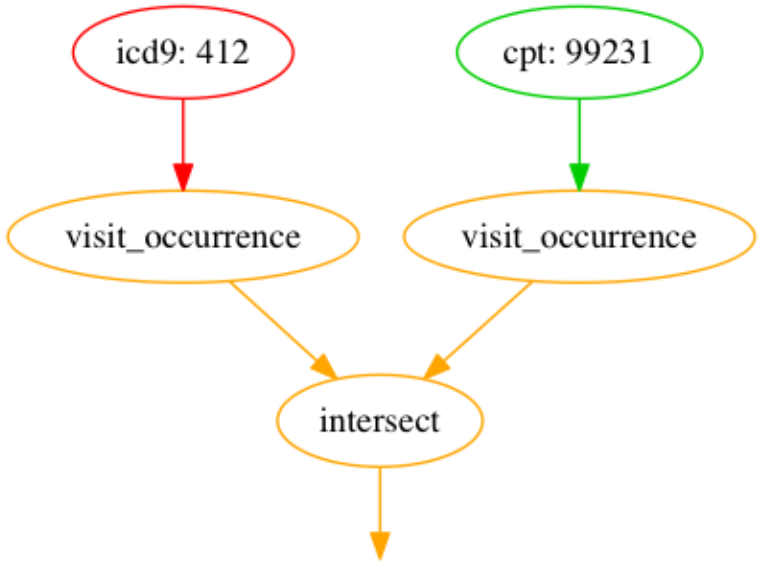
```
---
:person_filter:
  :left:
    :union:
    - :icd9: '412'
    - :cpt: '99214'
  :right:
    :gender: Male
```

Concepts within Concepts

One of the main motivations behind keeping ConceptQL so flexible is to allow users to build ConceptQL statements from other ConceptQL statements. This section loosely describes how this feature will work. Its actual execution and implementation will differ from what is presented here.

Say a ConceptQL statement gathers all visit_occurrences where a patient had an MI and a Hospital encounter (CPT 99231):

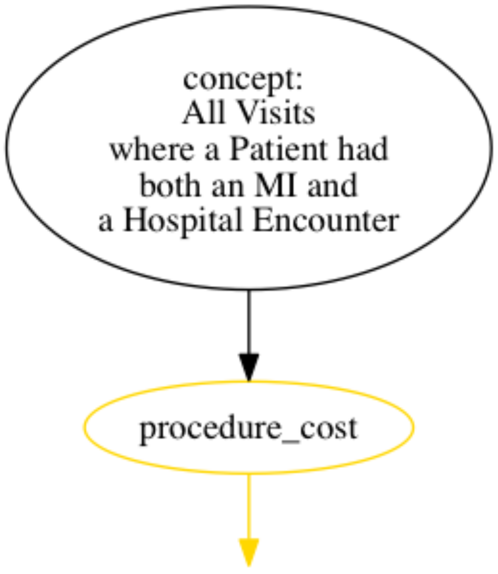
All Visits where a Patient had both an MI and a Hospital Encounter



```
---
:intersect:
- :visit_occurrence:
    :icd9: '412'
- :visit_occurrence:
    :cpt: '99231'
```

If we wanted to gather all costs for all procedures for those visits, we could use the “concept” node to represent the concept defined above in a new concept:

All Procedure Costs for All Visits as defined above

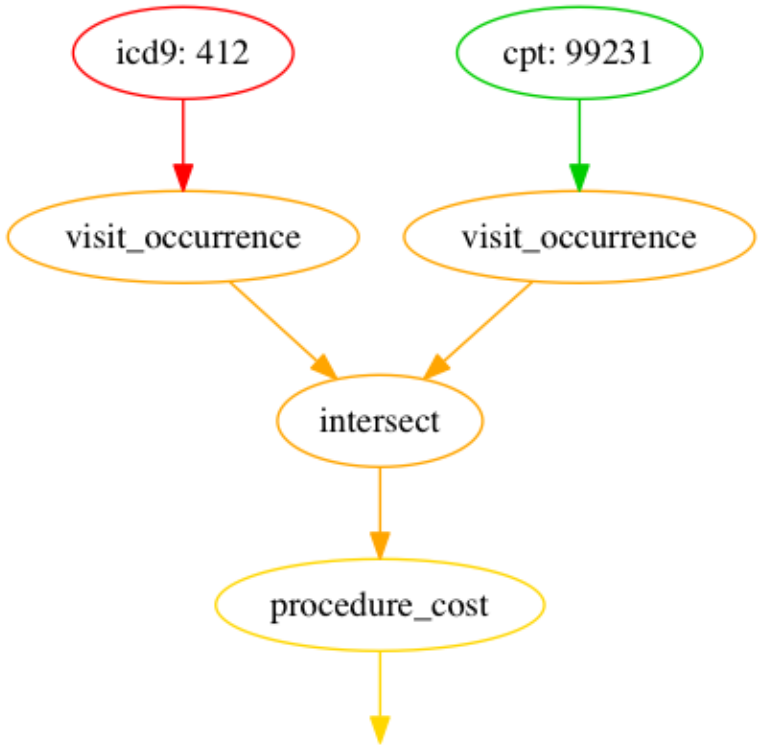


```
---
:procedure_cost:
  :concept: |2-

    All Visits
    where a Patient had
    both an MI and
    a Hospital Encounter
```

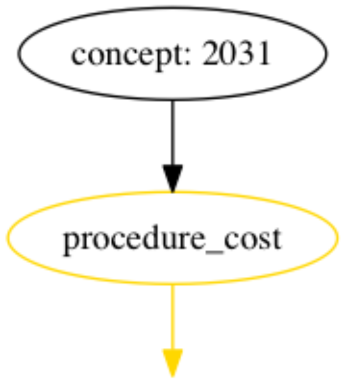
The color and edge coming from the concept node are black to denote that we don't know what types or streams are coming from the concept. In reality, any program that uses ConceptQL can ask the concept represented by the concept node for the concept's types. The result of nesting one concept within another is exactly the same had we taken concept node and replaced it with the ConceptQL statement for the concept it represents.

Procedure Costs for All Visits where a Patient had both an MI and a Hospital Encounter (same as above)



```
---
:procedure_cost:
  :intersect:
  - :visit_occurrence:
      :icd9: '412'
  - :visit_occurrence:
      :cpt: '99231'
```

In the actual implementation of the concept node, each ConceptQL statement will have a unique identifier which the concept node will use. So, assuming that the ID 2031 represents the concept we want to gather all procedure costs for, our example should really read:



```
---
:procedure_cost:
  :concept: 2031
```

Appendix A - Criterion Nodes

Node Name	Stream Type	Arguments	Returns
cpt	procedure_occurrence	1 or more CPT codes	All results whose source_value match any of the CPT codes
icd9	condition_occurrence	1 or more ICD–9 codes	All results whose source_value match any of the ICD–9 codes
icd9_procedure	procedure_occurrence	1 or more ICD–9 procedure codes	All results whose source_value match any of the ICD–9 procedure codes
icd10	condition_occurrence	1 or more ICD–10	All results whose source_value match any of the ICD–10 codes
hcpcs	procedure_occurrence	1 or more HCPCS	All results whose source_value

		codes	match any of the HCPCS codes
gender	person	1 or more gender concept_ids	All results whose gender_concept_id match any of the concept_ids
loinc	observation	1 or more LOINC codes	All results whose source_value match any of the LOINC codes
place_of_service_code	visit_occurrence	1 or more place of service codes	All results whose place of service matches any of the codes
race	person	1 or more race concept_ids	All results whose race_concept_id match any of the concept_ids
rxnorm	drug_exposure	1 or more RxNorm IDs	All results whose drug_concept_id match any of the RxNorm IDs
snomed	condition_occurrence	1 or more SNOMED codes	All results whose source_value match any of the SNOMED codes

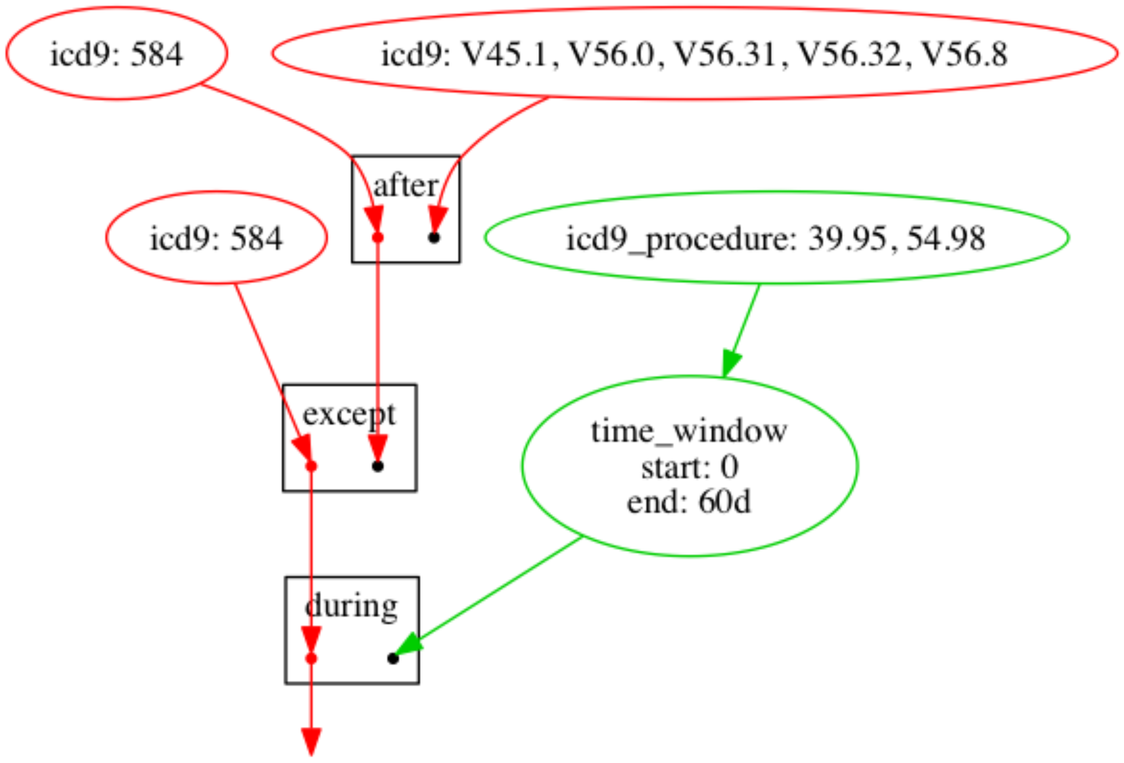
Appendix B - Concept Showcase

Here I take some concepts from OMOP's Health Outcomes of Interest and turn them into ConceptQL statements to give more examples. I truncated some of the sets of codes to help ensure the diagrams didn't get too large.

Acute Kidney Injury - Narrow Definition and diagnositc procedure

- ICD–9 of 584
- AND
- ICD–9 procedure codes of 39.95 or 54.98 within 60 days after diagnosis

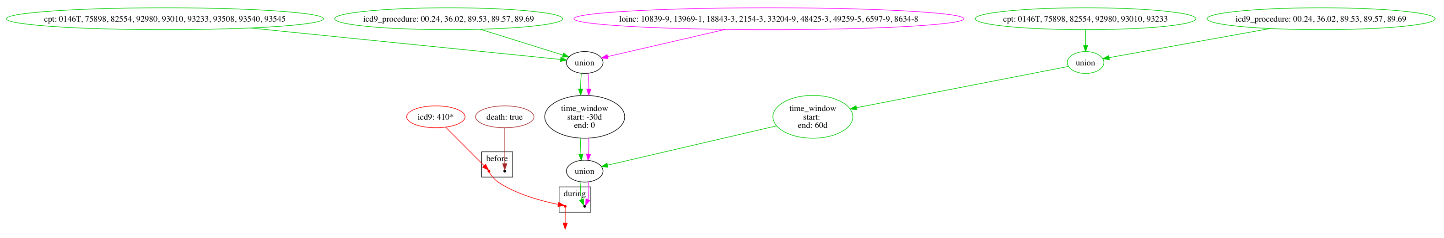
- AND NOT
- A diagnostic code of chronic dialysis any time before initial diagnosis
  - V45.1, V56.0, V56.31, V56.32, V56.8


icd9: 584
icd9: V45.1, V56.0, V56.31, V56.32, V56.8
after
icd9: 584
icd9_procedure: 39.95, 54.98
except
time_window
start: 0
end: 60d
during

```
---
:during:
  :left:
    :except:
      :left:
        :icd9: '584'
      :right:
        :after:
          :left:
            :icd9: '584'
          :right:
            :icd9:
            - V45.1
            - V56.0
            - V56.31
            - V56.32
            - V56.8
  :right:
    :time_window:
    - :icd9_procedure:
      - '39.95'
      - '54.98'
    - :start: '0'
      :end: 60d
```

Mortality after Myocardial Infarction #3

- Person Died
- And Occurrence of 410* prior to death
- And either
    - MI diagnosis within 30 days prior to 410
    - MI therapy within 60 days after 410

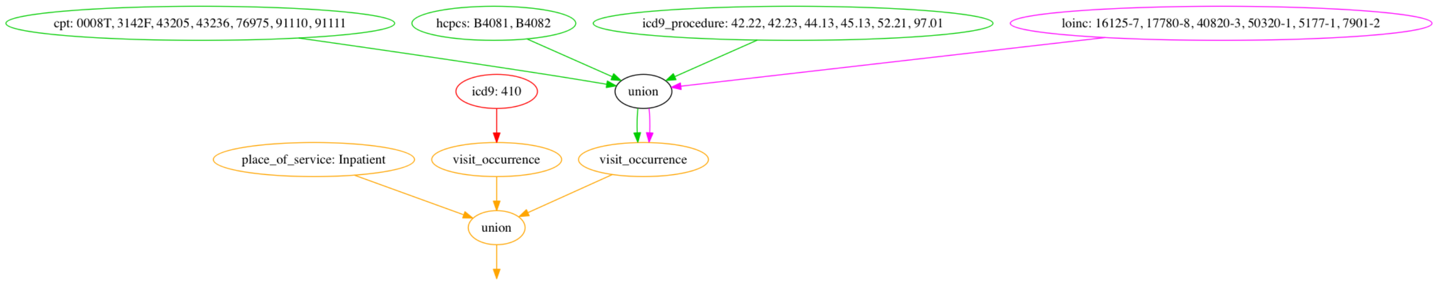

cpt: 0146T, 75898, 82554, 92980, 93010, 93233, 93508, 93540, 93545
icd9_procedure: 00.24, 36.02, 89.53, 89.57, 89.69
loinc: 10839-9, 13969-1, 18843-3, 2154-3, 33204-9, 48425-3, 49259-5, 6597-9, 8634-8
cpt: 0146T, 75898, 82554, 92980, 93010, 93233
icd9_procedure: 00.24, 36.02, 89.53, 89.57, 89.69
union
union
icd9: 410*
death: true
time_window
start: -30d
end: 0
time_window
start:
end: 60d
before
union
during

```yaml
---
:during:
  :left:
    :before:
      :left:
        :icd9: 410*
      :right:
        :death: true
  :right:
    :union:
    - :time_window:
      - :union:
        - :cpt:
          - 0146T
          - '75898'
          - '82554'
          - '92980'
          - '93010'
          - '93233'
          - '93508'
          - '93540'
          - '93545'
        - :icd9_procedure:
          - '00.24'
          - '36.02'
          - '89.53'
          - '89.57'
          - '89.69'
        - :loinc:
          - 10839-9
          - 13969-1
          - 18843-3
          - 2154-3
          - 33204-9
          - 48425-3
          - 49259-5
          - 6597-9
          - 8634-8
      - :start: "-30d"
        :end: '0'
    - :time_window:
      - :union:
        - :cpt:
          - 0146T
          - '75898'
          - '82554'
          - '92980'
          - '93010'
          - '93233'
        - :icd9_procedure:
          - '00.24'
          - '36.02'
          - '89.53'
          - '89.57'
          - '89.69'
      - :start: ''
        :end: 60d
```

GI Ulcer Hospitalization 2 (5000001002)

- Occurrence of GI Ulcer diagnostic code
- Hospitalization at time of diagnostic code
- At least one diagnostic procedure during same hospitalization

We use the fact that conditions, observations, and procedures all can be tied to a visit_occurrence to find situations where the appropriate conditions, diagnostic procedures, and place of service all occur in the same visit_occurrence


cpt: 0008T, 3142F, 43205, 43236, 76975, 91110, 91111
hcpcs: B4081, B4082
icd9_procedure: 42.22, 42.23, 44.13, 45.13, 52.21, 97.01
loinc: 16125-7, 17780-8, 40820-3, 50320-1, 5177-1, 7901-2
icd9: 410
union
place_of_service: Inpatient
visit_occurrence
visit_occurrence
union

```
---
:union:
- :place_of_service:
  - Inpatient
- :visit_occurrence:
    :icd9: '410'
- :visit_occurrence:
    :union:
    - :cpt:
      - 0008T
      - 3142F
      - '43205'
      - '43236'
      - '76975'
      - '91110'
      - '91111'
    - :hcpcs:
      - B4081
      - B4082
    - :icd9_procedure:
      - '42.22'
      - '42.23'
      - '44.13'
      - '45.13'
      - '52.21'
      - '97.01'
    - :loinc:
      - 16125-7
      - 17780-8
      - 40820-3
      - 50320-1
      - 5177-1
      - 7901-2
```

Appendix C - Under Development

ConceptQL is not yet fully specified. These are modifications/enhancements that are under consideration. These ideas are most likely not completely refined and might actually represent changes that would fundamentally break ConceptQL.

Todo List

1. Handle costs
    - How do we aggregate?
2. How do we count?
3. How do we handle missing values in streams?
    - For instance, missing DoB on patient?
4. What does it mean to pass a date range as an L stream?
    - I'm thinking we pass through no results
    - Turns out that, as implemented, a date_range is really a person_stream where the start and end dates represent the range (instead of the date of birth) so we're probably OK
5. How do we want to look up standard vocab concepts?
    - I think Marc's approach is a bit heavy-handed

Slots and Variables

Some statements maybe very useful and it would be handy to reuse the bulk of the statement, but perhaps vary just a few things about it. ConceptQL supports the idea of using variables to represent sub-expressions. The variable node is used as a place holder to say "some criteria set belongs here". That variable can be defined in another part of the criteria set and will be used in all places the variable node appears.

If a variable node is used, but not defined, the concept is still valid, but will fail to run until a definition for all missing variables is provided.

I don't have a good feel for:

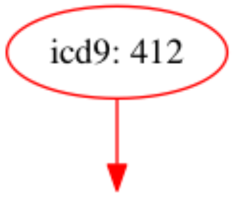
- How to represent a variable node in a diagram
- Whether we should have users name the variables, or auto-assign a name?
    - We risk name collisions if a concept includes a sub-concept with the same variable name
    - Probably need to name space all variables
- How to prompt users to enter values for variables in a concept
    - If we have name-spaced variables and sub-concepts needing values, how do we show this in a coherent manner to a user?
- We'll need to do a pass through a concept to find all variables and prompt a user, then do another pass through the concept before attempting to execute it to ensure all variables have values
    - Do we throw an exception if not?
    - Do we require calling programs to invoke a check on the concept before generating the query?

Value Nodes

So far, we can't recreate the Charlson comorbidity index using ConceptQL. If we added a "value" node, we could.

By default each result row will carry a value column, set to 1. Some examples:

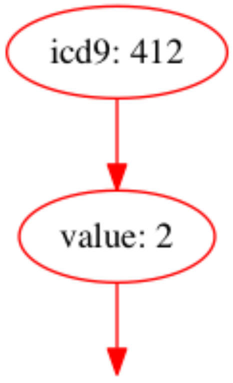
All MIs, defaulting value to 1



```
---
:icd9: '412'
```

Passing streams through a value node changes the number stored in the value column:

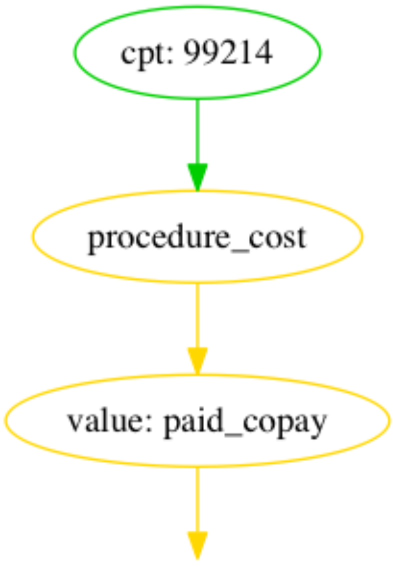
All MIs, changing value to 2



```
---
:value:
- :icd9: '412'
- 2
```

Value can also take a column name instead of a number. It will derive the results row's value from the value stored in the column specified.

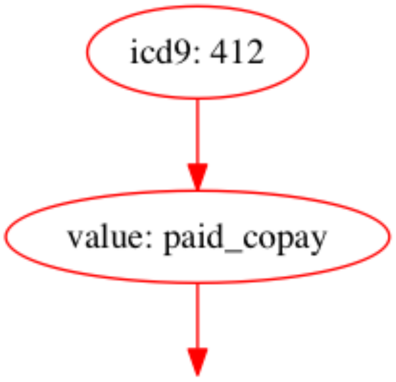
All copays for 99214s



```
---
:value:
- :procedure_cost:
    :cpt: '99214'
- :paid_copay
```

If something nonsensical happens, like the column specified isn’t present in the table pointed to by a result row, the value in the result row will be unaffected:

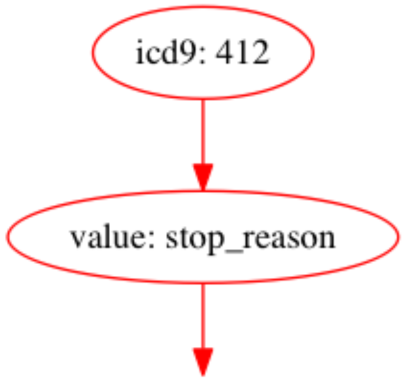
Still all MIs with value defaulted to 1. condition_occurrence table doesn’t have a “paid_copay” column



```
---
:value:
- :icd9: '412'
- :paid_copay
```

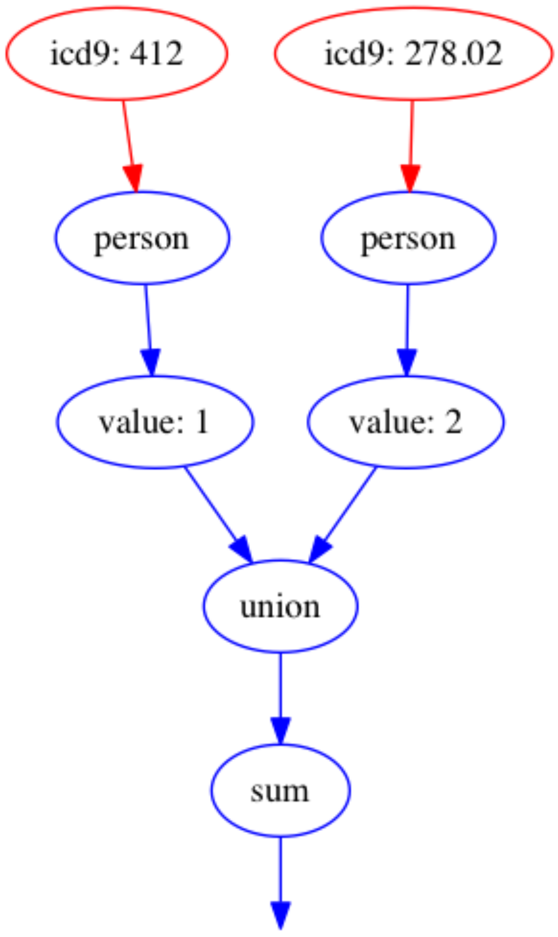
Or if the column specified exists, but refers to a non-numerical column, we'll set the value to 0

All MIs, with value set to 0 since the column specified by value node is a non-numerical column



```
---
:value:
- :icd9: '412'
- :stop_reason
```

With a value node defined, we could introduce a sum node that will sum by patient. This allows us to implement the Charlson comorbidity algorithm:



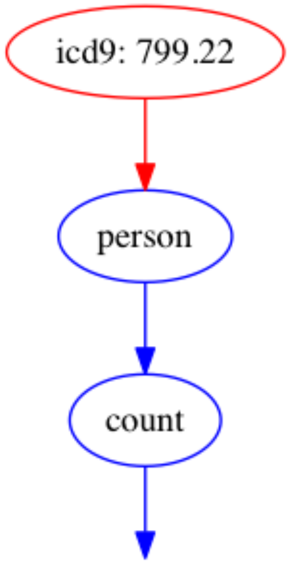
```
---
:sum:
- :union:
  - :value:
    - :person:
        :icd9: '412'
    - 1
  - :value:
    - :person:
        :icd9: '278.02'
    - 2
```

Counting

It might be helpful to count the number of occurrences of a result row in a stream. A simple “count” node could group identical rows and store the number of occurrences in the value column.

I need examples of algorithms that could benefit from this node. I’m concerned that we’ll want to roll up occurrences by person most of the time and that would require us to first cast streams to person before passing the person stream to count.

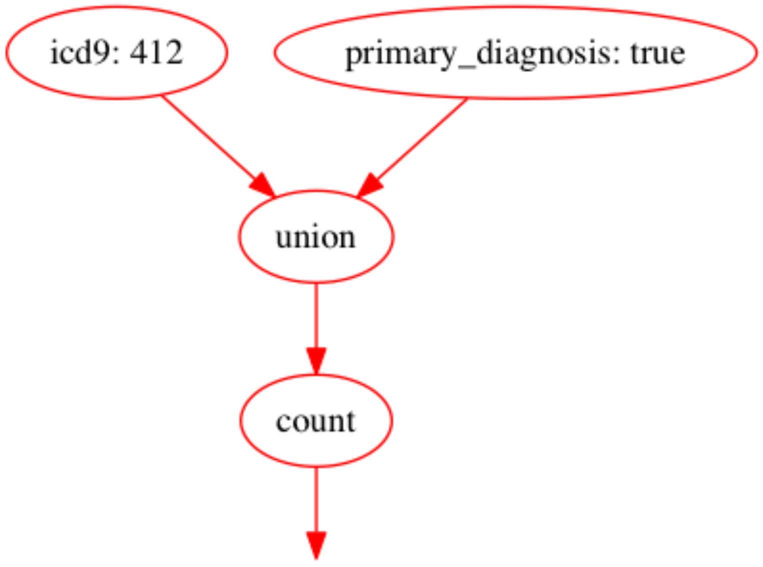
Count the number of times each person was irritable



```
---
:count:
  :person:
    :icd9: '799.22'
```

We could do dumb things like count the number of times a row shows up in a union:

All rows with a value of 2 would be rows that were both MI and Primary



```
---
:count:
  :union:
  - :icd9: '412'
  - :primary_diagnosis: true
```

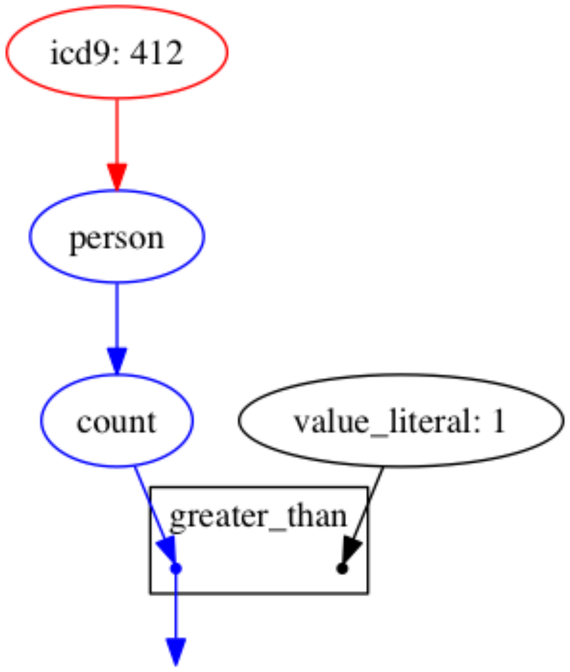
Value Comparison

Acts like any other binary node. L and R streams, joined by person. Any L that pass comparison go downstream. R is thrown out. Comparison based on result row’s value column.

- Less than
- Less than or equal
- Equal
- Greater than or equal
- Greater than
- Not equal
- Between

value_literal

People with more than 1 MI

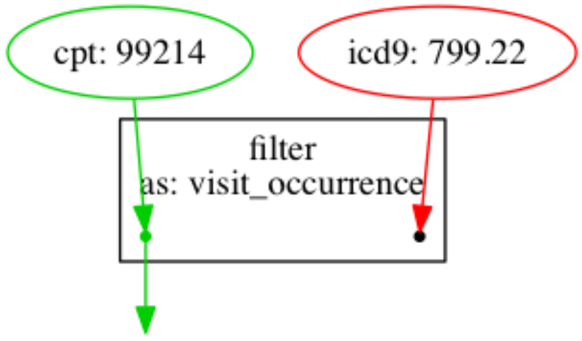


```
---
:greater_than:
  :left:
    :count:
      :person:
        :icd9: '412'
  :right:
    :value_literal: 1
```

Filter Node

Inspired by person_filter, why not just have a “filter” node that filters L by R. Takes L, R, and an “as” option. As option temporarily casts the L and R streams to the type specified by :as and then does person by person comparison, only keeping rows that occur on both sides. Handy for keeping procedures that coincide with conditions without fully casting the streams:

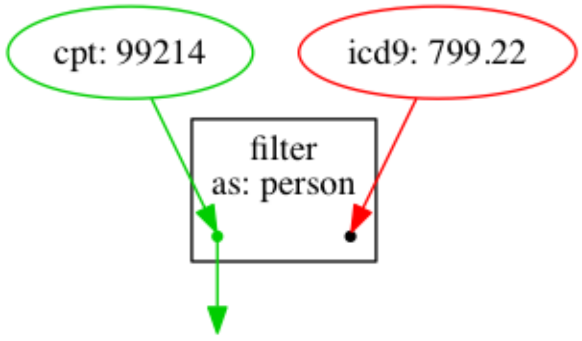
All 99214’s where person was irritable during a visit



```
---
:filter:
  :left:
    :cpt: '99214'
  :right:
    :icd9: '799.22'
  :as: visit_occurrence
```

person_filter then becomes a special case of general filter:

All 99214’s where person was irritable at some point in the data



```
---
:filter:
  :left:
    :cpt: '99214'
  :right:
    :icd9: '799.22'
  :as: person
```

Filter node is the opposite of Except. It only includes L if R matches.

AS option for Except

Just like Filter has an :as option, add one to Except node. This would simplify some of the algorithms I've developed.

---

1. J. Allen. Maintaining knowledge about temporal intervals. Communications of the ACM (1983) vol. 26 (11) pp. 832–843 ↩