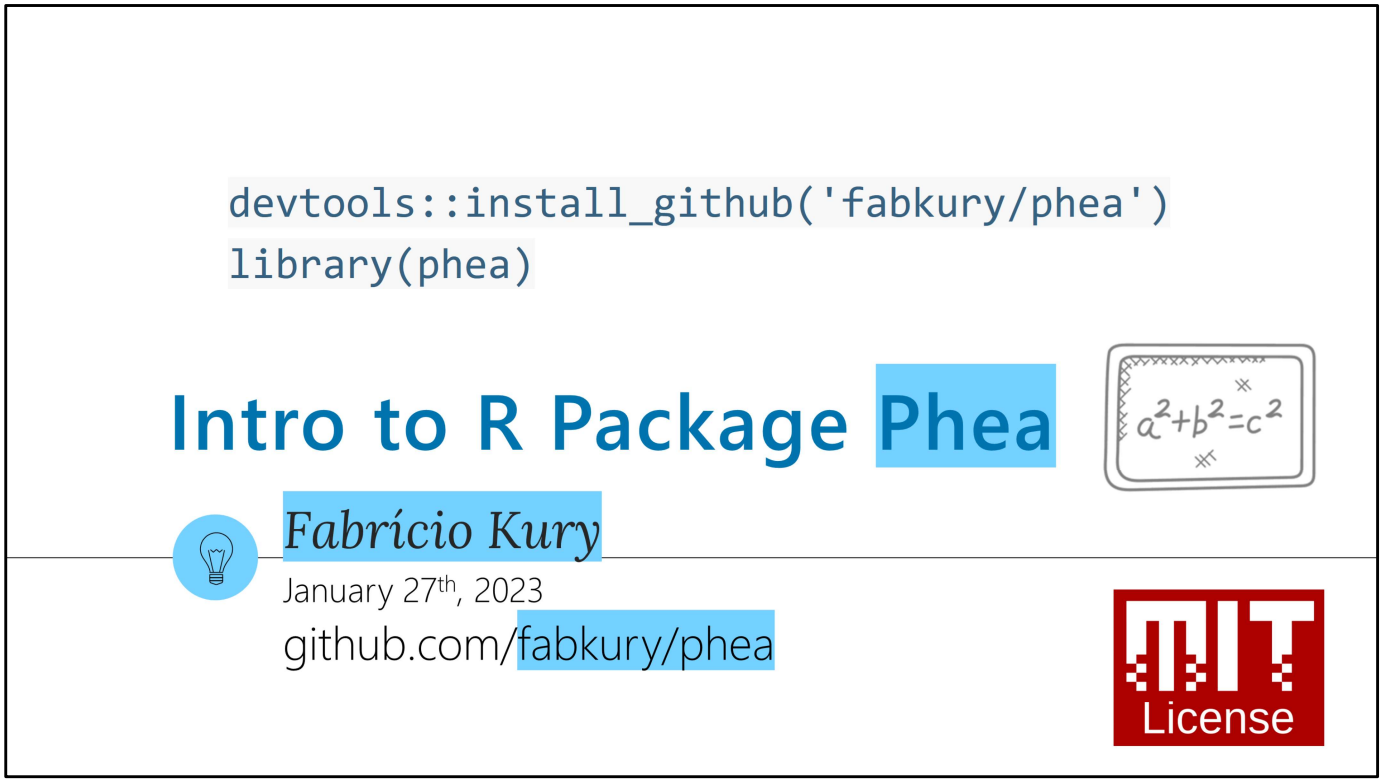


Good morning, everyone!

Thank you, Gowtham and the Phenotyping Working Group, for hosting me!

And thanks to you (small pause) for listening to me this morning.

My name is Fabrício. I have been doing electronic phenotyping for a good number of years, and I made

This R package offers functionalities that are not available in the OHDSI toolset. It lets you do things that are not possible in Atlas, or WebAPI, or for example.

I am very glad to get to talk about my package today, right in time for Phenotype Phebruary. Thanks for the opportunity.

You can see I'm using this real-time transcription feature, so my voice appears as text on the screen. I hope this is welcomed. I can have an accent sometimes.

And this presentation will be a bit fast. I have 21 slides in 30 minutes.

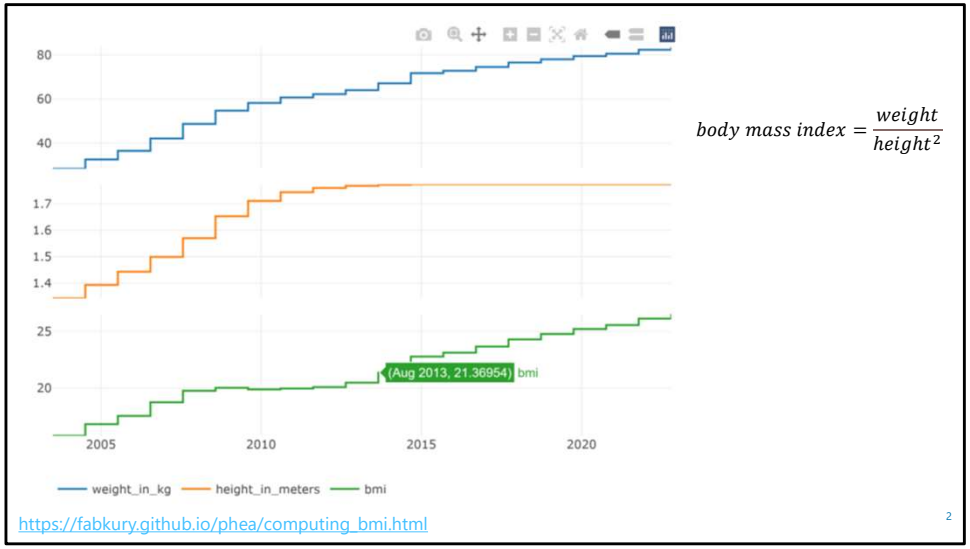
So, (long pause) may I introduce my R package, (small pause) called Phea.

The name "Phea" losely stands for "Phenotyping Algebra.“

This idea of "algebra" comes from the thought of expressing and manipulating phenotypes using formulas. In Phea, we compute formulas

using patient data.

Phea makes it very easy to calculate numbers, such as “body mass index = weight / height^2.”



Phea can calculate formulas for all patients, at all points in time. What this means is that you can, for example, check if the result of your formula ever met some kind of criteria for any patient, at any point in time. For example, “give me the patients who ever had BMI value under 18.”

This plot is an example of a phenotype made with Phea. This plot is the timeline of a single patient. What we are seeing is a timeline of all data points, all records, that went into the formula.

Our formula, in this case, is “body mass index = weight/height2".

Therefore, our input records are, body weight measurements, at the top in kilograms, (pause) body height measurements, in the middle, in meters. (pause) And at the bottom we see the result of the formula, the BMI value.

It is not always easy to see this, but notice that every time there is a body weight or a body height data point, we also have a BMI data point. The BMI is calculated again.

The phenotype calculation is performed at every point in time where there is a record of any of its input variables.

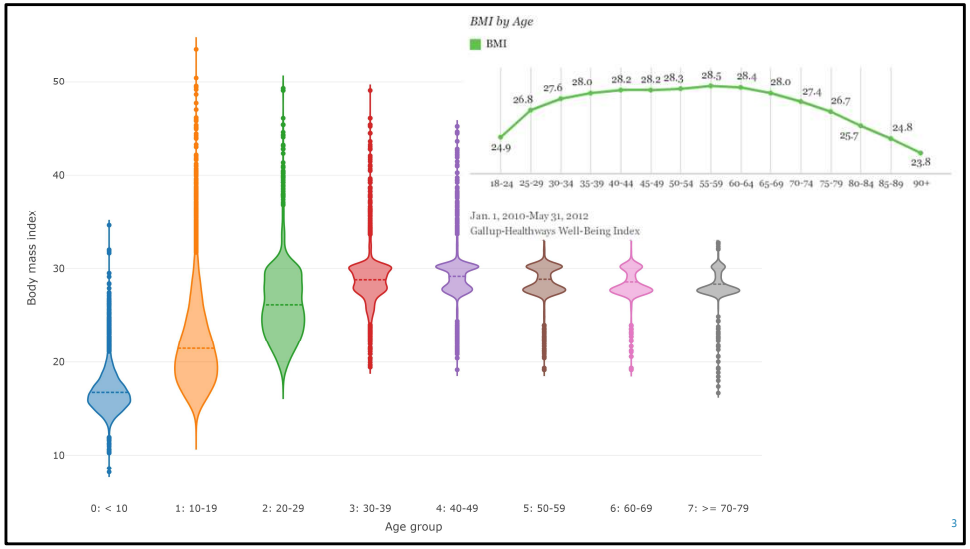
Let me say this again, because this may sound simple, but is foundational to how Phea works.

The phenotype calculation is performed at every point in time (small pause) where there is a record of any of its input variables.

Here, in this case, this is OMOP CDM data, so when we say one record, we mean one row, from the MEASUREMENT table, with the concept IDs of body weight or body height.

But they could be anything. Just like they can be a row from a SQL table, they can be a row from a SQL query. Any SQL query.

Phea can be promptly used on OMOP CDM, but is not restricted to it. Today I will use examples in OMOP CDM, but Phea can be used on other data models. I will explain.



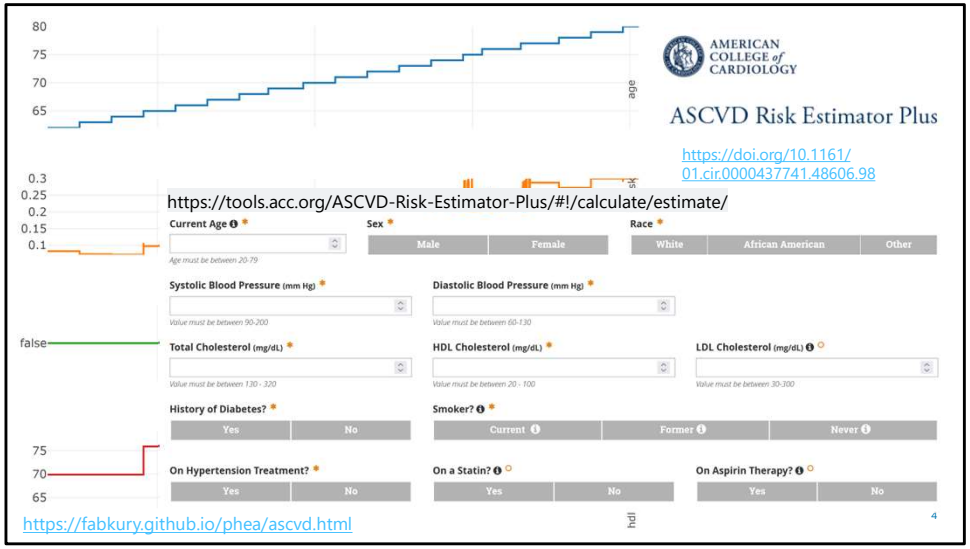
This is a look at the result of the phenotype across all patients. This is a Synthea-generated dataset of about 20 thousand patients.

The violin plots are showing the distributions of BMI stratified by age group.

-- advance slide: Gallup chart appears –

Average BMI peaks around the people's late thirties, then it kind of stays there, and begins to lower around people's late sixties.

To me it doesn't look too different from this chart by Gallup. My dataset is emulating the state of New York, while the Gallup data is nationwide, but this is just a very quick comparison against some external data.



BMI is very simple. Now we look at more complicated formula.

This is the ASCVD Risk Estimator Plus. I will just call it “ASCVD score.”

Published by the American College of Cardiology, it predicts the 10-year risk of *atherosclerotic cardiovascular events*.

At the top is patient age in years. We see the patient becoming one year older every year, which is good, because it makes sense.

In the middle, in orange, we see **estimated_risk**, which is actually here the final result of the phenothpe. The ASCVD score itself.

This particular patient started with a risk around 10%, zero point one, at age of 65. Over the years, by age 80, the risk had increased to around 30%.

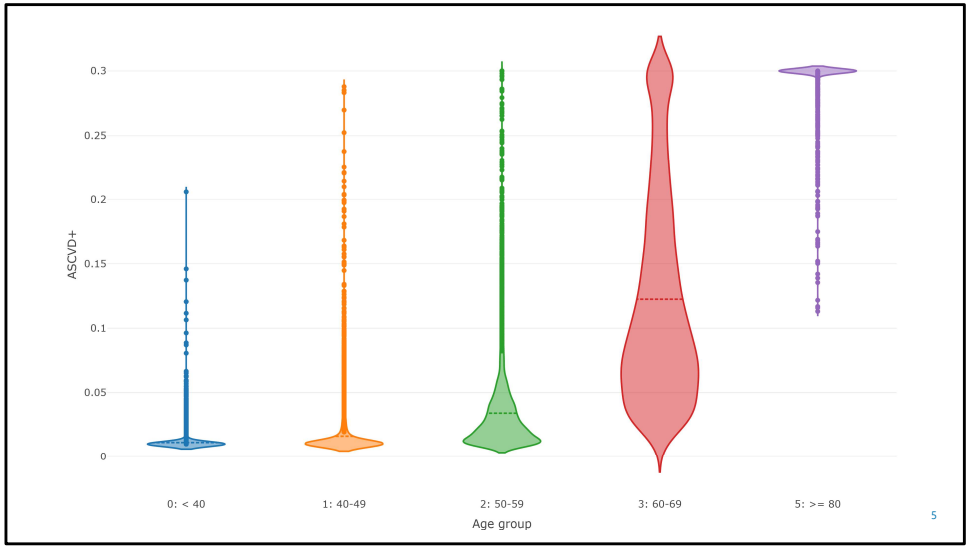
(points with the mouse) These fizzy lines here, they were caused by fluctuations in the patient's blood pressure. Something happened to the patient, maybe a medical event, and during this period there were many blood pressure measurements. Blood pressure was going up and down. The score depends directly on the value of blood pressure, so, mathematically, the score went up and down too.

The green chart has only one value during the entire available history in the data for this patient. The patient never had diabetes. Notice that this is a Boolean value. Binary yes or no.

-- advance: variables appear –

The American College of Cardiology provides an official online tool to calculate the ASCVD score for you. You put in the data for a patient, and it tells you the score. This is a screenshot.

So by looking at the tool, you can see all the variables that you need from the patient data, to calculate the ASCVD. Age, cholesterol levels, smoking status, and other things.



From the distributions of the ASCVD score, across all patients, we can see that in each age group you can find some outliers, people with very low or very high ASCVD score.

But there is a clear trend of worse scores, higher risk, in older patients. Especially after age 60.

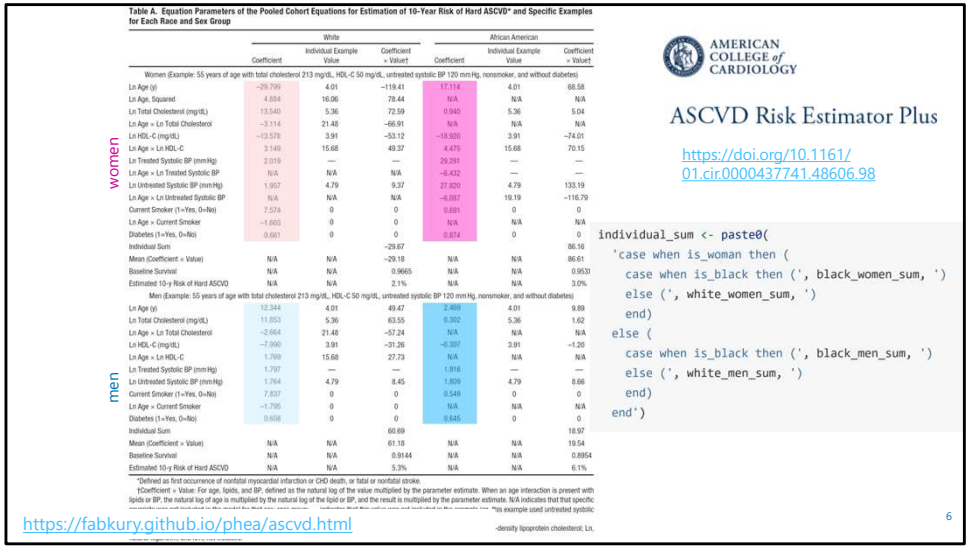


Table A. Equation Parameters of the Pooled Cohort Equations for Estimation of 10-Year Risk of Hard ASCVD* and Specific Examples for Each Race and Sex Group

	White			African American		
	Coefficient	Individual Example Value	Coefficient × Value†	Coefficient	Individual Example Value	Coefficient × Value†
Women (Example: 55 years of age with total cholesterol 213 mg/dL, HDL-C 50 mg/dL, untreated systolic BP 120 mm Hg, nonsmoker, and without diabetes)						
Ln Age (y)	−29.799	4.01	−119.41	17.114	4.01	68.58
Ln Age, Squared	4.884	16.06	78.44	N/A	N/A	N/A
Ln Total Cholesterol (mg/dL)	13.540	5.36	72.59	0.940	5.36	5.04
Ln Age × Ln Total Cholesterol	−3.114	21.48	−66.91	N/A	N/A	N/A
Ln HDL-C (mg/dL)	−13.578	3.91	−53.12	−18.920	3.91	−74.01
Ln Age × Ln HDL-C	3.149	15.68	49.37	4.475	15.68	70.15
Ln Treated Systolic BP (mm Hg)	2.019	—	—	29.291	—	—
Ln Age × Ln Treated Systolic BP	N/A	N/A	N/A	−6.432	—	—
Ln Untreated Systolic BP (mm Hg)	1.957	4.79	9.37	27.820	4.79	133.19
Ln Age × Ln Untreated Systolic BP	N/A	N/A	N/A	−6.087	19.19	−116.79
Current Smoker (1=Yes, 0=No)	7.574	0	0	0.691	0	0
Ln Age × Current Smoker	−1.665	0	0	N/A	N/A	N/A
Diabetes (1=Yes, 0=No)	0.661	0	0	0.874	0	0
Individual Sum			−29.67			86.16
Mean (Coefficient × Value)	N/A	N/A	−29.18	N/A	N/A	86.61
Baseline Survival	N/A	N/A	0.9665	N/A	N/A	0.9533
Estimated 10-y Risk of Hard ASCVD	N/A	N/A	2.1%	N/A	N/A	3.0%
Men (Example: 55 years of age with total cholesterol 213 mg/dL, HDL-C 50 mg/dL, untreated systolic BP 120 mm Hg, nonsmoker, and without diabetes)						
Ln Age (y)	12.344	4.01	49.47	2.469	4.01	9.89
Ln Total Cholesterol (mg/dL)	11.853	5.36	63.55	0.302	5.36	1.62
Ln Age × Ln Total Cholesterol	−2.664	21.48	−57.24	N/A	N/A	N/A
Ln HDL-C (mg/dL)	−7.990	3.91	−31.26	−0.307	3.91	−1.20
Ln Age × Ln HDL-C	1.769	15.68	27.73	N/A	N/A	N/A
Ln Treated Systolic BP (mm Hg)	1.797	—	—	1.916	—	—
Ln Untreated Systolic BP (mm Hg)	1.764	4.79	8.45	1.809	4.79	8.66
Current Smoker (1=Yes, 0=No)	7.837	0	0	0.549	0	0
Ln Age × Current Smoker	−1.795	0	0	N/A	N/A	N/A
Diabetes (1=Yes, 0=No)	0.658	0	0	0.645	0	0
Individual Sum			60.69			18.97
Mean (Coefficient × Value)	N/A	N/A	61.18	N/A	N/A	19.54
Baseline Survival	N/A	N/A	0.9144	N/A	N/A	0.8954
Estimated 10-y Risk of Hard ASCVD	N/A	N/A	5.3%	N/A	N/A	6.1%

*Defined as first occurrence of nonfatal myocardial infarction or CHD death, or fatal or nonfatal stroke.

†Coefficient × Value: For age, lipids, and BP, defined as the natural log of the value multiplied by the parameter estimate. When an age interaction is present with lipids or BP, the natural log of age is multiplied by the natural log of the lipid or BP, and the result is multiplied by the parameter estimate. N/A indicates that that specific

‡This example used untreated systolic

-density lipoprotein cholesterol; Ln,

women

men

AMERICAN COLLEGE of CARDIOLOGY

ASCVD Risk Estimator Plus

https://doi.org/10.1161/01.cir.0000437741.48606.98

```
individual_sum <- paste0(
  'case when is_woman then (
    case when is_black then (', black_women_sum, ')
    else (', white_women_sum, ')
    end)
  else (
    case when is_black then (', black_men_sum, ')
    else (', white_men_sum, ')
    end)
  end')
```

https://fabkury.github.io/phea/ascvd.html

6

As a mathematical formula, the ASCVD is not very easy to calculate.

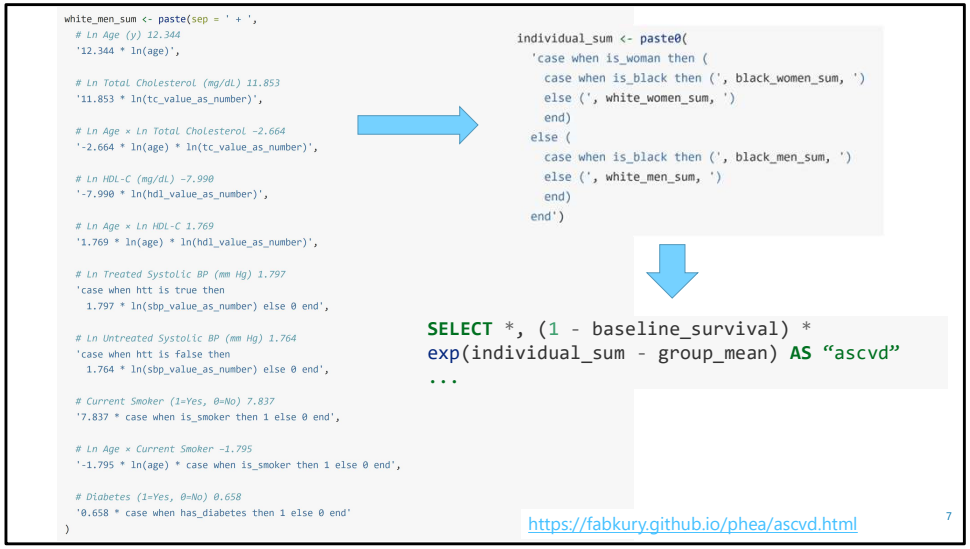
It is a generalized linear regression model. The coefficients of the model are on this table, from the original paper of the ASCVD.

The calculation includes squares, natural logarithms, and natural exponentiation.

Moreover, the formula has 4 variants. Which variant you use depends on which of the four demographic buckets the patient falls in. The buckets for the ASCVD are just woman yes or no, and African American yes or no.

What I want to hightlight here is that the formulas in Phea are *not* just mathematical. They are SQL code.

SQL includes many maths functions, but also things that are not maths, such as CASE … WHEN statements. In this phenotype, I used a CASE … WHEN statement to pick the right formula variant.



This is the formula for the demographic bucket of white men.

In Phea, you provide your formula in text. A character string in R.

If you know R, you understand that this function paste() here, it is just concatenating the pieces of text. **white_men_sum** is one line of text, that reads this, plus this, plus this, plus this, etc. That line is the formula for white men. Those constants, they came from the table in the previous slide.

In the formula, when I write **sbp_value_as_number**, Phea understands that I mean column **value_as_number** from the component that I called **sbp**, systolic blood pressure. Phea will make that variable, that column, exist for me.

What Phea actually, actually does is, it gathers the records for you. Phea assembles the table, one column per variable, and one row per point in time. It also lets you filter those rows in a few different ways, but I will not go into the filters now.

-- advance slide: more parts of the query appear –

The thing is, once you have that table, with the variables you need, all lined up according to their timestamps, calculating a formula is easy. It's a matter of SELECT'ing your formula, like here on the screen.

Remember that the values can be Boolean, so you can also write decision rules as Boolean expressions in SQL.

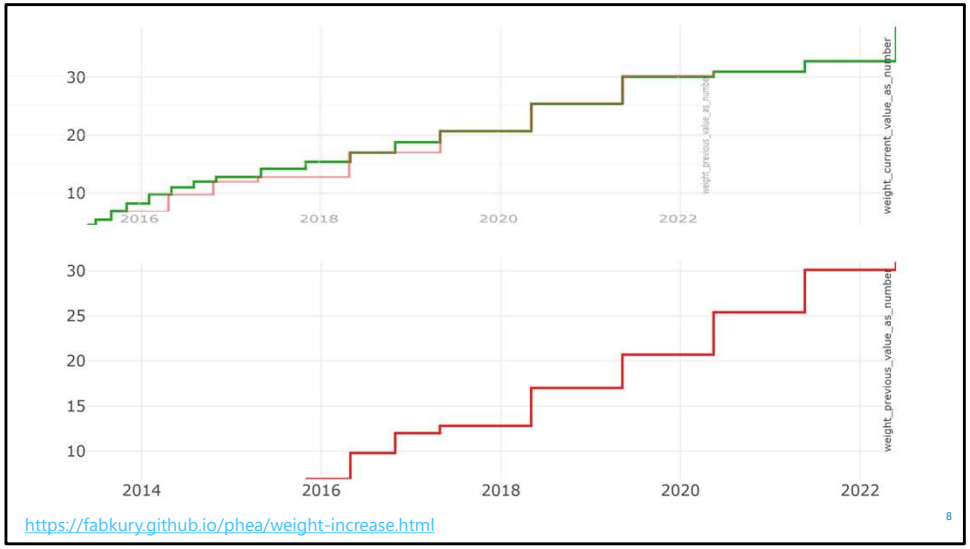
Once you have that table ready to use, ready to calculate a formula, maybe you will even feel that, you know what, actually you don't need to calculate any formula. You would rather just download the whole table, and use it yourself.

That's fine, because in Phea, the formula is actually optional. A phenotype can contain any number of formulas, and you can also create a phenotype with no formula.

The thing that Phea does for you is, it assembles the table, with the variables you asked for. The formula that you give to Phea, it will be simply "copy-pasted" into the SQL query. No modification.

This also means that, whatever functionality that your SQL server supports, in a SELECT statement, you can use it. In this example, I used CASE ... WHEN statements to pick which of the 4 formula variants to use in the ASCVD phenotype.

That spared me from having to split the records myself and run through Phea four times, which would produce the same results.



When you're assembling the table, with the variables lined up according to time, the usual case is this.

At each point in time, you want to know the most recently available value, of each variable that you need for your formula.

-- advance slide: chart appears --

That is the conceptual equivalent of being an observer, there in the moment when the events happened. You know only what has already happened, up until the present time.

To give an example, this is a timeline of a patient's body weight, in kilograms.

A different case would be, if your formula was, divide the patient's body weight from today, by the body weight from two years ago, and check if the ratio is bigger than 1.5. In other words, tell me if the patient's body weight has

increased by more than 50% over the past two years.

Phea can do that for you by letting you apply masks, filters, to time.

If you mask out the past 2 full years, the most recently available record becomes something from *at least more* than 2 full years ago.

-- advance slide: second chart appears --

This chart is coming from the same data as the above, but lagging behind by *between* two to three years.

The horizontal axis, time, is aligned between the charts.

At each point in time, that is, each vertical line, you have the most recently available body weight, and the most recently available body weight that is at least 2 years old.

These two charts come from the exact same data, but are not exact offsets of each other.

-- advance slide: third chart appears --

All we are doing is, every time there is a body weight record, you stand there, and just look back, and tell me what is the first record that you find, that is at least 2 years old *for you*. You pick that other record as well. That other record, and today's record, are what go into the formula as of today.

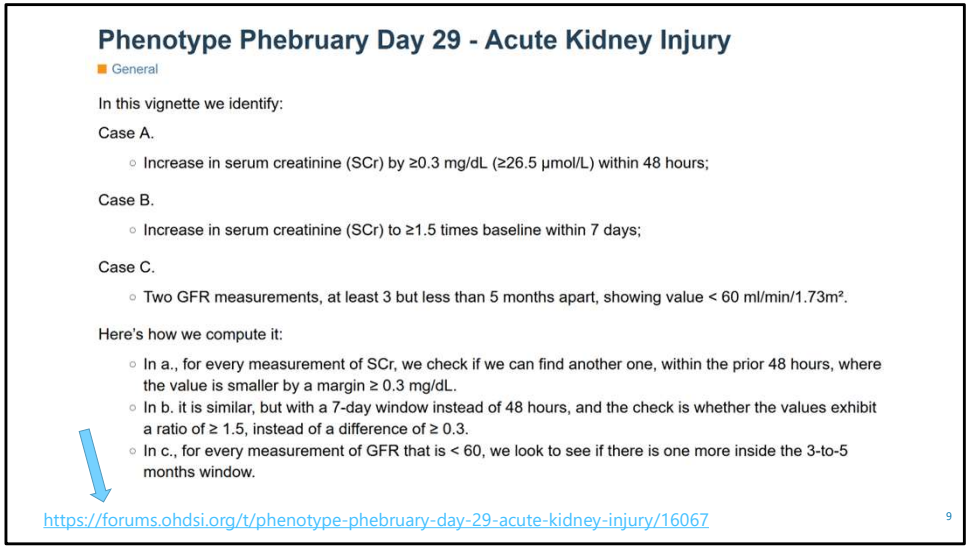
If the other, older record doesn't exist, the value is NULL.

Here I copy-pasted the chart, and aligned both axes the best I could in PowerPoint. You can see they always touch each other. But sometimes it looks like a record got left out. Like here.

Because this record did get left out. In the future, by the time a new body weight record appeared, this record here was

already no longer the most recently available one that was at least 2 years old.

By the time the next new body weight measurement appeared in the future, the most recent one that's at least 2 years old was already this other one.



This is how I computed the lab presentation of acute kidney injury, following a post on the forum.

These definitions of acute kidney injury were brought by Marcela, in her work for day 29 of Phenotype Phebruary last year.

Her review also pointed out that “studies have recommended NOT to use diagnoses codes to identify patients with AKI, based on the poor sensitivity. Ideally the phenotype should include lab values.”

Which is a problem for Atlas, because Atlas can’t compare a value between two rows.

In the case of a lab value in OMOP CDM, that value would be column value_as_number. You can’t compare the value of column value_as_number between two rows, using Atlas.

Atlas will let you compare value_as_number against a constant, but not against the value of another row.

But SQL can allow you to do that. And Phea can help you write the query.

The approach that Phea takes to write the query, in a nutshell, is based on window functions.

I'll just say that, window functions allow you to define a frame for each row on a table. The frame is the set of rows from the table that will be allowed into the computation. The frame can be defined using offsets from the current row. That's how you can look around, backward or forward in time, according to need of your phenotype.

This is how the ASCVD query looks. Part of it.

There are a few layers of things going on here. More than one layer of window functions. There is also preparation that happens before the window functions, and post-processing afterwards.

The ASCVD in this example, I deliberately used compatibility mode in Phea.

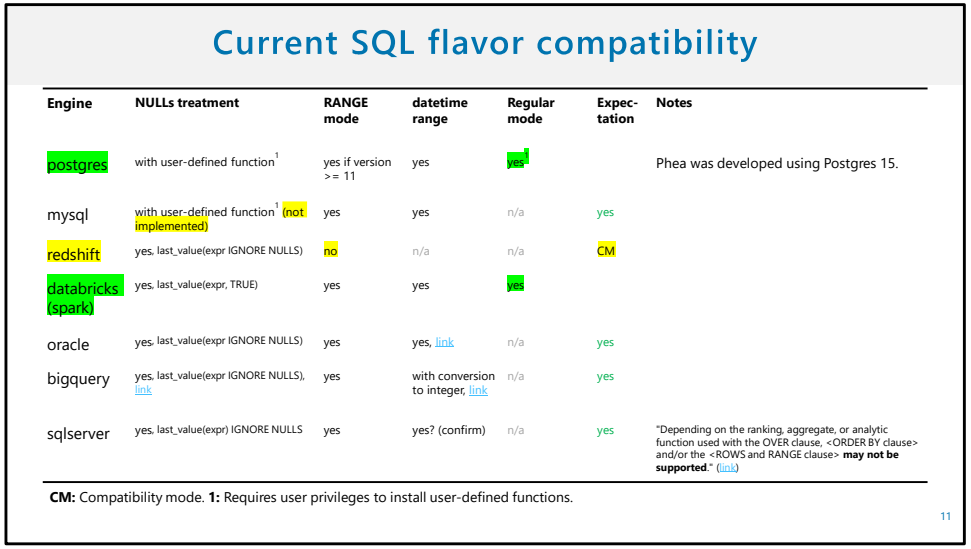
Phea offers two modes for its SQL generation engine. *Regular* mode, and *compatibility* mode.

The compatibility mode you can turn on if your SQL server doesn't support all the features that Phea needs. In compatibility mode, some features from Phea are disabled.

The features that do remain, you still get the same final result, but the query is different. Phea builds the query in a different way. That different way works on more SQL flavors than the query from regular mode. But the query from

regular mode is more computationally performant.

I expect compatibility mode to work in any of the major SQL flavors of today.



Current SQL flavor compatibility

Engine	NULLs treatment	RANGE mode	datetime range	Regular mode	Expec-tation	Notes
postgres	with user-defined function[1]	yes if version >= 11	yes	yes[1]		Phea was developed using Postgres 15.
mysql	with user-defined function[1] (not implemented)	yes	yes	n/a	yes	
redshift	yes, last_value(expr IGNORE NULLS)	no	n/a	n/a	CM	
databricks (spark)	yes, last_value(expr, TRUE)	yes	yes	yes		
oracle	yes, last_value(expr IGNORE NULLS)	yes	yes, link	n/a	yes	
bigquery	yes, last_value(expr IGNORE NULLS), link	yes	with conversion to integer, link	n/a	yes	
sqlserver	yes, last_value(expr) IGNORE NULLS	yes	yes? (confirm)	n/a	yes	"Depending on the ranking, aggregate, or analytic function used with the OVER clause, <ORDER BY clause> and/or the <ROWS and RANGE clause> **may not be supported**." (link)

CM: Compatibility mode. **1:** Requires user privileges to install user-defined functions.

11

So far I have had the opportunity to test Phea only on Postgres and Databricks. I am very happy to say that regular mode worked on both.

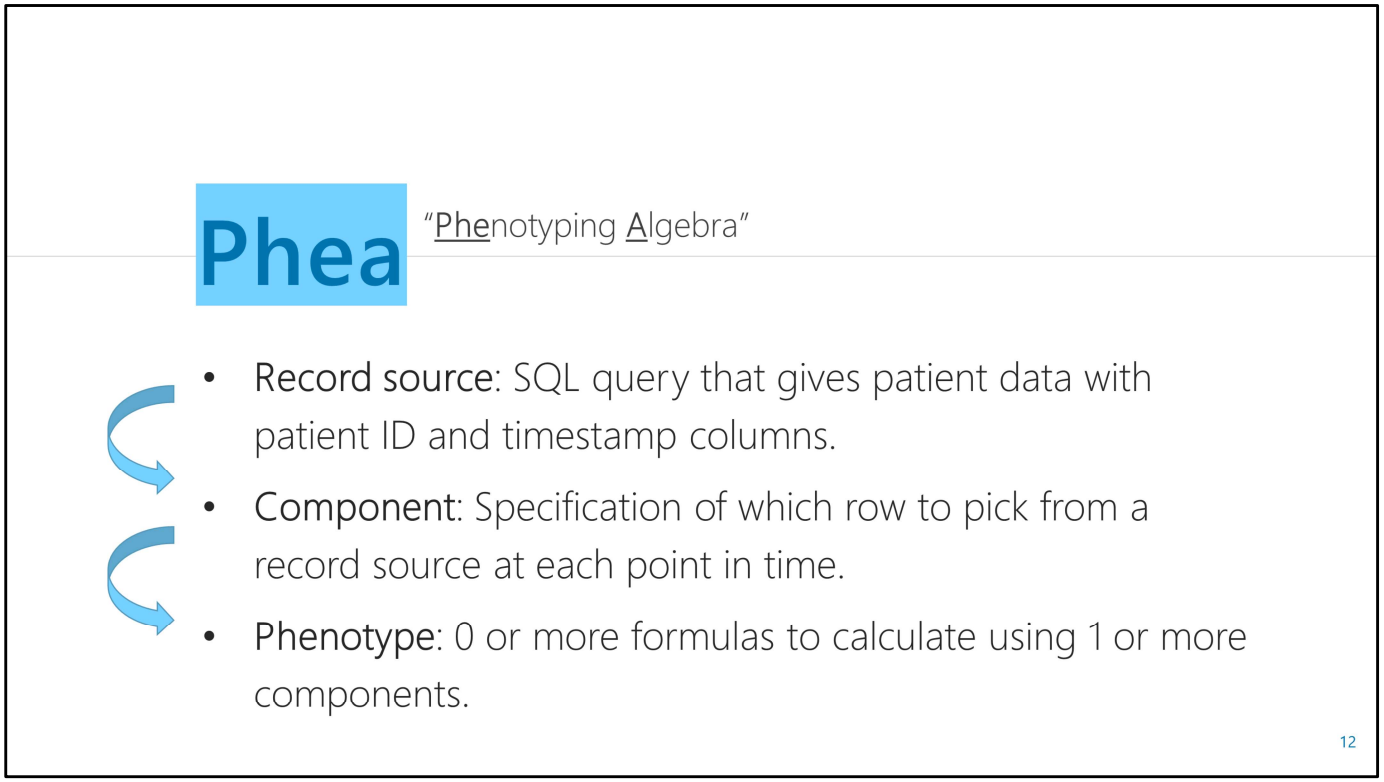
However, on Postgres, to get regular mode, your SQL credentials need access privileges to create user-defined aggregate functions. This sometimes can be a bit difficult to setup, depending on the situation.

On Databricks, those user-defined functions are not necessary. Therefore, the SQL user only needs *read* access to the data.

From reading the online documentation of Redshift, I can see that Redshit will only work in compatibility mode. That will be so until AWS implements the RANGE mode for window functions, in Redshift.

The other SQL flavors in this table, Oracle, BigQuery, and Microsoft SQL Server, I also could not test, but I expect regular mode to work, based on their online documentation pages.

MySQL is the same case as Postgres, but I haven’t implemented the UDF yet, so you also only get compatibility mode.



Ok, great.

So now I wanted to get a little bit more technical. I want explain how to use Phea. I want to show the R code that you need to write.

As I've said, Phea is only a SQL query builder. So, at the end of the day, what Phea is doing is allowing you to write those SQL queries, in another way. A simpler way.

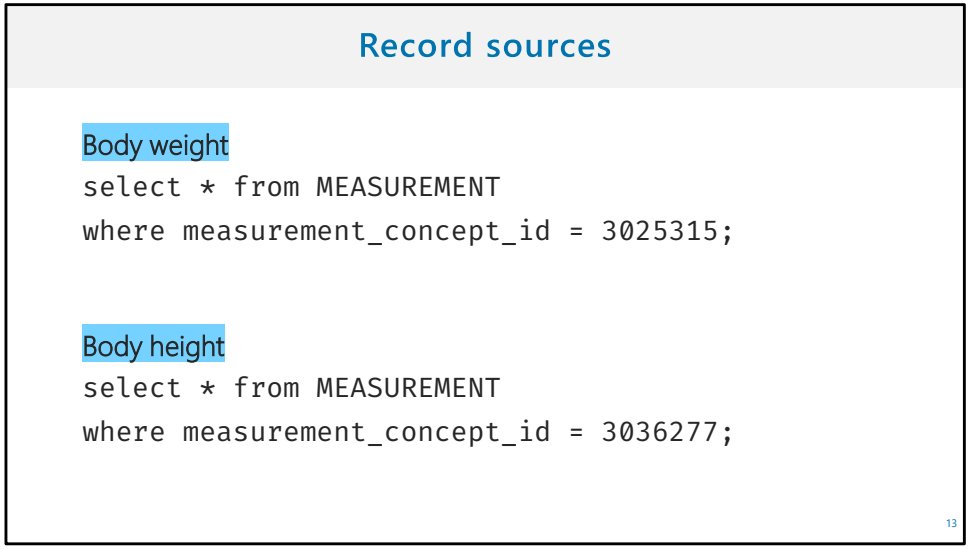
these three concepts. Record source, component, phenotype.

(reads the slide)

Notice the zero or more formulas. The formulas are optional. I will explain that.

(pause)

These are the basics of Phea.



In the BMI example, the data came from these two queries:

-- advance: queries appear --

These two concept IDs indeed are SNOMED codes for body weight and body height.

-- advance: title appears –

The query for a record source can be of any complexity. These two queries here are simple, but they could be anything. They could have any number of JOINs, UNIONs, subqueries, or other things.

Moreover in this example, these two record sources have the same columns, because actually, they even come from the same table; but that is not required. Record sources can be arbitrarily different from each other.

(pause) Phea is agnostic about the data model. The only thing that Phea requires, is that there must be two mandatory columns: **patient ID** column, and **timestamp** column.

The patient ID and the timestamp columns can have different names in different record sources, but they must exist.

Record sources

Body weight

```
weight_record_source <- make_record_source(
  records = sql0("select * from measurement where measurement_concept_id = 3025315"),
  ts = 'measurement_datetime',
  pid = 'person_id')
```

Body height

```
height_record_source <- make_record_source(
  records = sql0("select * from measurement where measurement_concept_id = 3036277"),
  ts = 'measurement_datetime',
  pid = 'person_id')
```

We take a SQL query, and pass it to the function make_record_source(), to create a record source.

To create a record source, you must specify the names of the patient ID (pid) and timestamp (ts) columns.

-- advance: code appears --

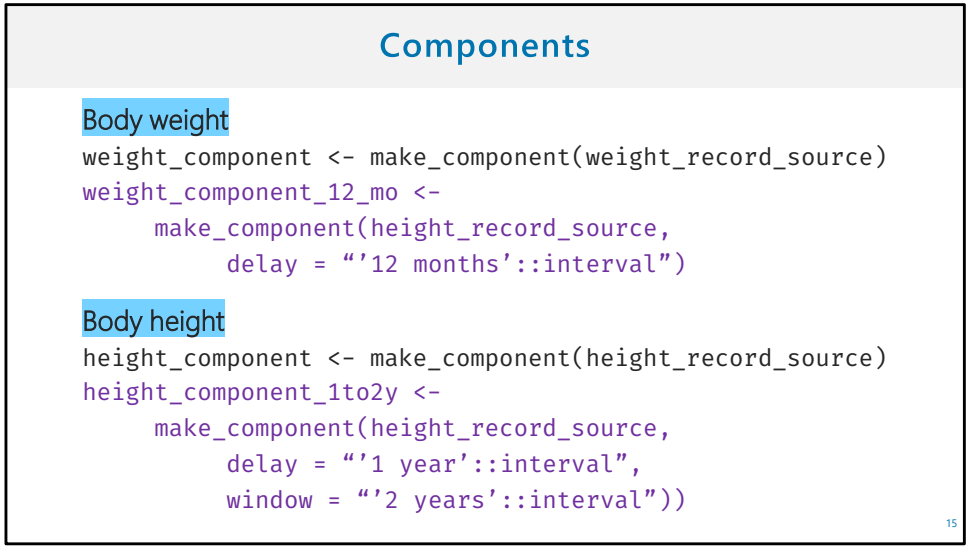
As we know, in table MEASUREMENT those would be *person_id*, and

measurement_datetime.

I am using this function **sql0()** here. It also comes from Phea. I can't go into it right now, but we are really just passing a SQL query to the **records** argument of function **make_record_source()**.

Once we have a record source, we can create a component.

-- next slide: blank --



A **component** is a specification of what records to look at, from a record source, at each point in time.

You need to provide a record source (small pause) to create a component. That's the only required argument.

When creating a component, the default case is that the record you want to pick is, (pause) the most recently available record at the given point in time.

In this default case, we don't need extra arguments to create a component. The default arguments take over.

A different case would be if, instead of the most recently available record, you wanted to pick an older one. For example, the patient's weight from one year ago. To do that, you would specify the optional **delay** argument.

-- advance slide –

You would do **delay** equals to, then write out 12 months, in text, using SQL language.

The delay argument hides records that are too recent. 12 months of delay means, give me the most recently available weight measurement that is at least 12 months old.

-- advance slide –

Another optional argument, the **window** argument, hides records that are too

old. For example, If you give delay = 1 year, and window = 2 years, you're saying, give me the most recent record that is at least 1 full year old, as long as it is not more than 2 full years old.

If such a record does not exist, the value is NULL.

The window argument is very useful if you want the data points to expire.

For example, imagine if one of the components in your phenotype is the patent's level of troponin in the blood, in a phenotype about myorcardial infarction. The troponin level rises within hours, and its gone within days. It's a number that expires very quickly.

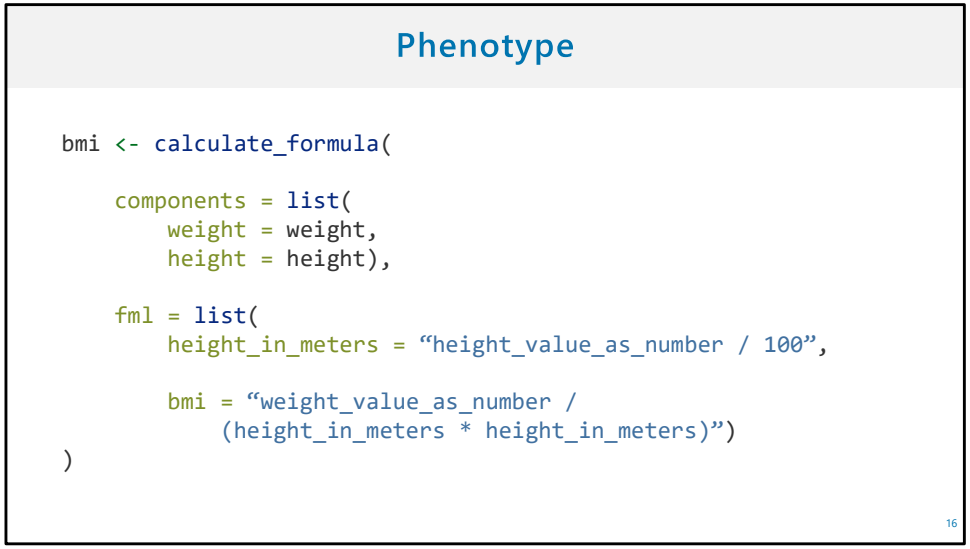
That's in contrast to, say, a diabetes diagnosis. You don't an expiration date for the diabetes variable. Once a patient is diagnosed with diabetes, it is permanent.

There are other options for creating components.

For example, you can ask for the *average* of a number during a time period, or

the *maximum* value, or a few other things. I mean the analytical functions in SQL, such as avg(), max(), min(), ntile(), nth_value(), dense_rank(), and others.

But for now, let‘s just get back to the BMI phenotype.



So what's happening here?

We pass our two components to **calculate_formula()** inside a **list()**. Notice we give them names inside that list.

The names you use in this list() are the names that become available in the formula.

Then, we have actually two formulas. **fml** is short for *formula*.

The first formula is converting height from centimeters to meters. Just divide by 100.

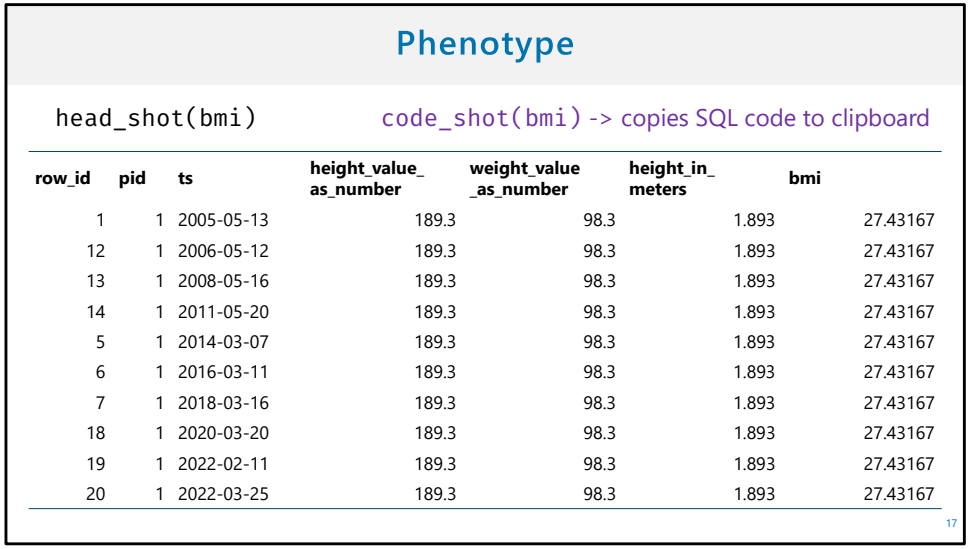
The second formula, the **bmi** formula, is using the result of the first.

The second formula accesses the result of the first simply by using the name we gave to the first formula.

The components themselves, they're accessed using the syntax component name, underscore, column name.

So when we write height_value_as_number, Phea understands we mean column value_as_number, from the record source of the height component.

Lastly here, this R object that we just created, **bmi**, this is what we call a **lazy table**, or **remote** data frame. If you are not familiar with this, it just means, this object is **a SQL query.**



Phenotype

`head_shot(bmi)` `code_shot(bmi)` -> copies SQL code to clipboard

row_id	pid	ts	height_value_as_number	weight_value_as_number	height_in_meters	bmi
1	1	2005-05-13	189.3	98.3	1.893	27.43167
12	1	2006-05-12	189.3	98.3	1.893	27.43167
13	1	2008-05-16	189.3	98.3	1.893	27.43167
14	1	2011-05-20	189.3	98.3	1.893	27.43167
5	1	2014-03-07	189.3	98.3	1.893	27.43167
6	1	2016-03-11	189.3	98.3	1.893	27.43167
7	1	2018-03-16	189.3	98.3	1.893	27.43167
18	1	2020-03-20	189.3	98.3	1.893	27.43167
19	1	2022-02-11	189.3	98.3	1.893	27.43167
20	1	2022-03-25	189.3	98.3	1.893	27.43167

17

The results of the query are not inside the **bmi** object.

Whenever you ask R to print the **bmi** object, R will then query the server, using a LIMIT clause to get just the first few rows, and print those rows on the screen.

The function **head_shot()** comes from Phea. It works like the function **head()** in base R, but it works with lazy tables. The function **head()** shows the first few rows of a data frame. **head_shot()** does the same

thing, but with remote data frames.

If you want to get just the code of the phenotype, without running the query, that can be done in a few ways. One of them is function **code_shot()** from Phea.

-- advance: code_shot appears –

code_shot() copies the code of the query to the clipboard, so you can paste it somewhere else.

(pause)

The phenotype that we created is a SQL query, just like the SQL queries that are the input data to this phenotype.

You can repeat this process if you want. You can use a phenotype to create a record source, then a component, then use it inside other phenotypes. You can use phenotype results as variables inside other phenotypes, seamlessly, no special treatment. They are all SQL queries just the same.

(pause)

This **bmi** phenotype is finished.

We see that the first ten rows happen to come from the same patient. Patient ID, **pid**, equal to 1.

It also happens that this patient's weight or height never changed, and so the BMI never changed. But they could have.

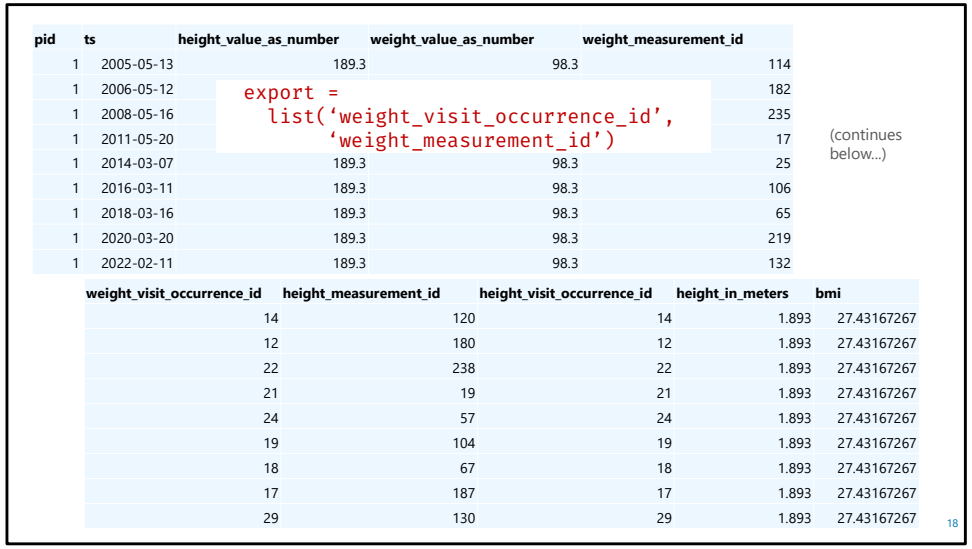
Remember, at every point in time, where there is input data, the phenotype calculation is performed again. That is what you are seeing here.

If you want, you can export additional columns from the record source. Columns that you didn't use in the formula. Columns that you just want to have as well, for something else.

For example, let's say you don't trust these results you're looking at. Maybe

you could be thinking, wait a minute Fabrício, these numbers look weird. It looks like some kind of bug is happening somewhere.

Well, you could ask **calculate_formula()** to incude the column **measurement_id** from the rows that were used. That would allow you to go on, and query back the original MEASUREMENT table, and verify what numbers are there in the data.



pid	ts	height_value_as_number	weight_value_as_number	weight_measurement_id
1	2005-05-13	189.3	98.3	114
1	2006-05-12			182
1	2008-05-16			235
1	2011-05-20			17
1	2014-03-07	189.3	98.3	25
1	2016-03-11	189.3	98.3	106
1	2018-03-16	189.3	98.3	65
1	2020-03-20	189.3	98.3	219
1	2022-02-11	189.3	98.3	132

```
export =
  list('weight_visit_occurrence_id',
       'weight_measurement_id')
```

(continues below...)

weight_visit_occurrence_id	height_measurement_id	height_visit_occurrence_id	height_in_meters	bmi
14	120	14	1.893	27.43167267
12	180	12	1.893	27.43167267
22	238	22	1.893	27.43167267
21	19	21	1.893	27.43167267
24	57	24	1.893	27.43167267
19	104	19	1.893	27.43167267
18	67	18	1.893	27.43167267
17	187	17	1.893	27.43167267
29	130	29	1.893	27.43167267

18

You could also ask for the **provider_id** of the rows. And the **visit_occurence_id**. Just to cite examples.

We can see that the weight and height measurements came from the same **visit ID**s.

But they have unique **measurement_id**s, showing that they indeed did come from a different row each time.

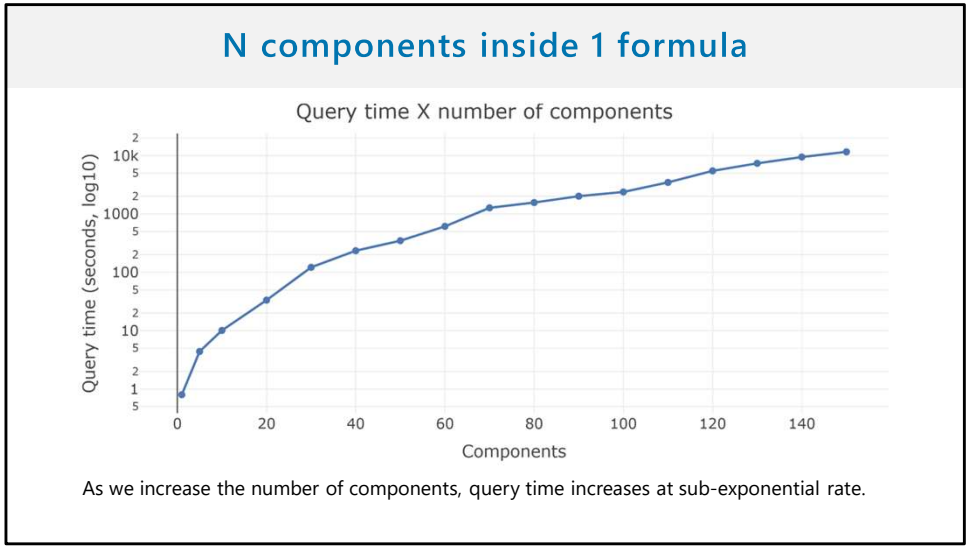
The function **calculate_formula()** offers you the **export** argument.

-- advance: “export =“ appears --

You specify the names of the columns you want, and they will be included in the results.

You specify those names using the syntax component name, underscore, column name.

Any colum that was used in a formula, Phea exports it automatically by default. You only need to use **export** for colums that you didn’t use in any formula.



In Phea, one phenotype can have any number of formulas.

And each formula can use any number of components.

I have tested Phea with up to one hundred and fifty components inside a single formula. Each component came from a different SQL query. It ran without a problem. The server was Postgres 15, running on my laptop.

I was increasing the number of components, and measuring query time. The query with 150 components took 3 hours to process approximately 20 thousand patients.

I stopped at 150 not because of any problem, but just because I thought 150 was enough.

As for the phenotype-inside-phenotype layering, the formula result-inside-formula thing, I stress-tested that

too.

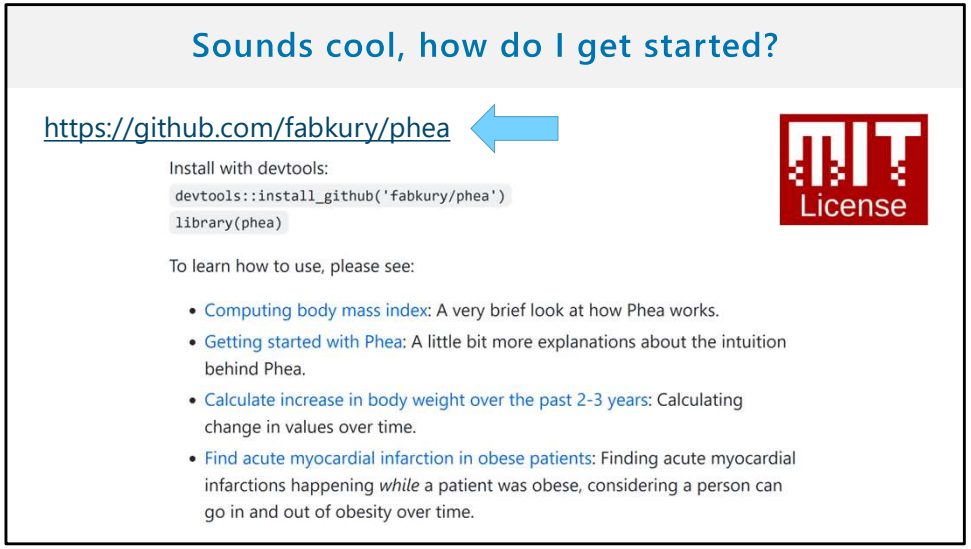
I started increasing the number of times, the number of formula-inside-formula layers, to see how long until I get an error.

I was able to do it over one hundred times consecutively, without error.

Just before I could reach 110 times, the server was unable to process the query due to an error of “stack depth.”

My point here is to show that Phea can take on big jobs.

Even if your phenotype is really, very complex, my experiments show that the server breaks before Phea itself does.



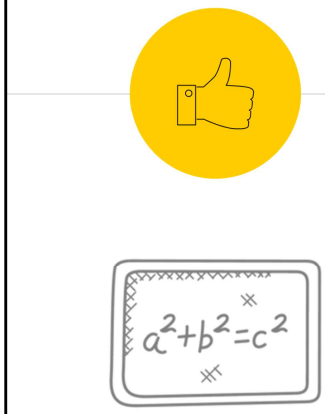
If this sounds interesting to you, and you would like to check it out, please visit on GitHub.

As I said, Phea is released under an MIT License, so I hope you can use it, even if at work inside an institution. The MIT license allows commercial use.

I have written some vignettes to teach some of the features. I posted them on the forum, but they're also listed on GitHub.

I admit, however, that not all features are in the vignettes.

For some features, you will need to open R, use the interrogation mark command, and read the help documentation.



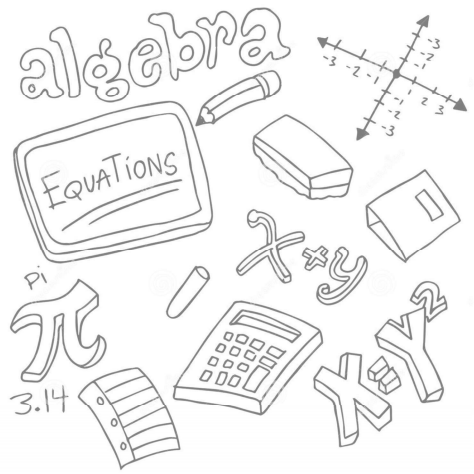
Thanks!

Any questions ?

Fabrício Kury

January 27th, 2023

- fab@kury.dev
- github.com/fabkury/phea



...*or*, you can reach out to me, and I can try to help you if you want to use Phea ☺

Thanks very much for listening to my talk today.

I am more than happy to take any questions.