

FunSQL:
A library for compositional construction
of SQL queries

Find all patients born at or after 1950.

```
function find_patients(conn)
    sql = """
    SELECT person_id
    FROM patient
    WHERE year_of_birth >= 1950
    """
    DBInterface.execute(conn, sql)
end
```

- What is SQL? Data is often stored in relational databases, and to retrieve it, we write queries in SQL.
- Popular databases with SQL interface include MySQL, PostgreSQL, SQLite, Microsoft SQL Server, Redshift, BigQuery, and many others.
- Julia already has a number of libraries that let you interact with SQL databases.
- Why another tool?

Find all patients born between

1950

and

```
function find_patients(conn; start_year = nothing,  
                        end_year = nothing)  
  
    sql = ""  
    SELECT person_id  
    FROM patient  
    ""  
  
    conditions = String[]  
    if start_year != nothing  
        push!(conditions, "year_of_birth >= $start_year")  
    end  
    if end_year != nothing  
        push!(conditions, "year_of_birth <= $end_year")  
    end  
    if !isempty(conditions)  
        sql *= "\nWHERE " * join(conditions, " AND ")  
    end  
    DBInterface.execute(conn, sql)  
end
```

Find all patients born between

and

living in

with conditions

who visited a doctor

"location"

"condition_occurrence"

"visit_occurrence"

```
function find_patients(conn; start_year = nothing,  
                          end_year = nothing,  
                          state = nothing,  
                          conditions = []  
                          last_visit = nothing)
```

```
    sql = ""
```

```
    SELECT person_id
```

```
    FROM patient
```

```
    ""
```

```
    ???
```

```
    DBInterface.execute(conn, sql)
```

```
end
```

- Clearly, a more systematic approach is necessary.



- A small diversion to introduce the database schema which we will use in subsequent examples.
- OMOP Common Data Model is a popular open-source used in healthcare of observational research.
- As typical in healthcare, the schema is patient-centric. The *person* table stores information about patients including basic demographic information. Their address is stored in a separate table called *location*.
- Most of the patient data consists of clinical events: encounters with healthcare providers, recorded observations, diagnosed conditions, performed procedures, etc.

```
using FunSQL: SQLTable

const person =
  SQLTable(name = :person,
    columns = [:person_id, :year_of_birth, :location_id])

const location =
  SQLTable(name = :location,
    columns = [:location_id, :city, :state, :zip])

const condition_occurrence =
  SQLTable(name = :condition_occurrence,
    columns = [:condition_occurrence_id, :person_id,
      :condition_concept_id,
      :condition_start_date, :condition_end_date])

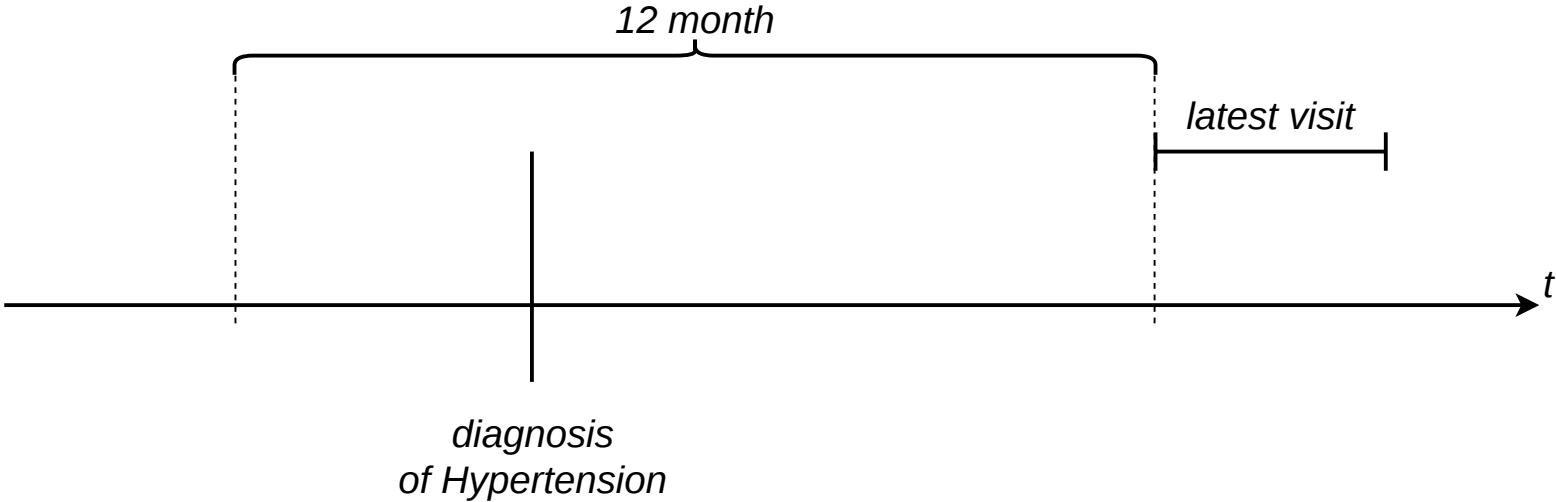
const visit_occurrence =
  SQLTable(name = :visit_occurrence,
    columns = [:visit_occurrence_id, :person_id,
      :visit_concept_id,
      :visit_start_date, :visit_end_date])
```

Find all patients born in 1970 or later and living in Illinois who have been diagnosed with Hypertension no more than 12 months before their latest visit to a healthcare provider.

- To demonstrate FunSQL, we will use it to construct one SQL query.
- We will state the full query in English, then construct it using FunSQL step by step.
- This will also give us an opportunity to discuss different capabilities of FunSQL.

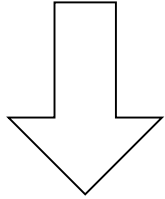
Find patients

- *born in 1970 or later,*
- *living in Illinois,*
- *such that their latest visit to the healthcare provider satisfies:*

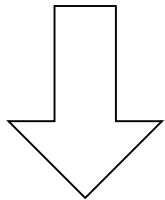


Find all patients born in 1970 or later.

FROM person p

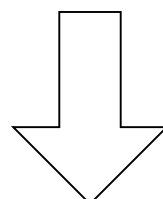


FROM person p
WHERE p.year_of_birth >= 1950

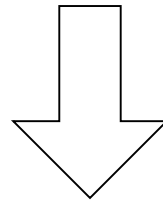


SELECT p.person_id
FROM person p
WHERE p.year_of_birth >= 1950

q = From(person)



q = From(person) |>
Where(Get.year_of_birth .>= 1950)



q = From(person) |>
Where(Get.year_of_birth .>= 1950) |>
Select(Get.person_id)

sql = render(q, dialect = :postgresql)

- A SQL query starts with a FROM clause, where you can choose the starting table.
- What follows it is a sequence of operations which let you shape the output.
- The final clause is always SELECT, which is written at the top, but is always performed last.
- In FunSQL, we replicate the structure of the query using appropriate constructors and the chain operator for composing them.

Bound References

`q.year_of_birth`

`q.person_id`

```
q1 = From(person)
q2 = q1 |> Where(q1.year_of_birth .>= 1950)
q3 = q2 |> Select(q2.person_id)
```

- Both bound and unbound references are supported.
- Unbound references make decomposition easier.

Unbound References

`Get.year_of_birth`

`Get.person_id`

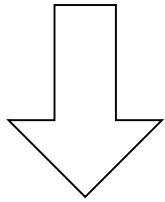
```
q = From(person) |>
  Where(Get.year_of_birth .>= 1950) |>
  Select(Get.person_id)
```

```
BornInOrLater(y) =
  Get.year_of_birth .>= y
```

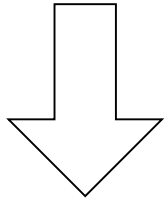
```
q = From(person) |>
  Where(BornInOrLater(1950)) |>
  Select(Get.person_id)
```

Show patients with their state of residence.

FROM person p

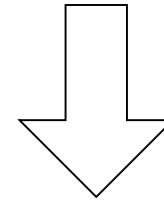


```
FROM person p
JOIN location l
  ON (p.location_id = l.location_id)
```

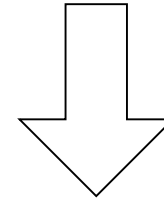


```
SELECT p.person_id, l.state
FROM person p
JOIN location l
  ON (p.location_id = l.location_id)
```

From(person)

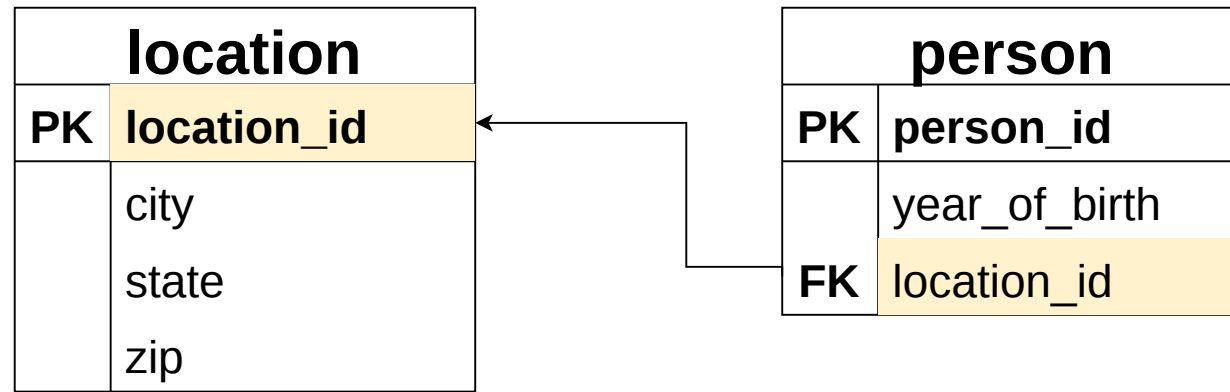


```
person |>
Join(:location => location,
      Get.location_id .==
      Get.location.location_id)
```



```
person |>
Join(:location => location,
      Get.location_id .==
      Get.location.location_id) |>
Select(Get.person_id,
        Get.location.state)
```

- The JOIN operator combines the output of two queries.
- Suppose we want to display patients with their state of residence. We need to combine the data from the table person, which store the patient record, and the table location, which stores information about addresses.



```
q1 = From(person)
q2 = From(location)
q3 = q1 |> Join(q2, q1.location_id .== q2.location_id)
```

```
From(person) |>
Join(From(location) |>
  As(:location),
  on = Get.location_id .== Get.location.location_id)
```

- Joining two tables poses a challenge for unbound references.
- For instance, both person and location table have a column called location_id.
- This isn't a problem if you use bound references.
- To solve this problem, we introduce hierarchical references: As() constructor to introduce a level of hierarchy, which can be then traversed using Get.
- To clarify: although this operation is similar to SQL AS operator, it does not really affect the generated SQL.

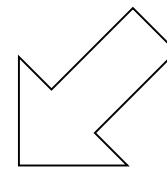
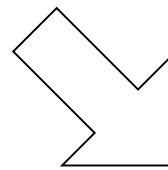
Find patients

- *born in 1970 or later,*
- *living in Illinois*

- Recall that we want to find patients with certain constraints on date of birth and location.
- Suppose we are already assembled appropriate constraints.

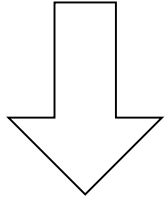
```
From(person) |>  
Where(Get.year_of_birth .>= 1970)
```

```
From(location) |>  
Where(Get.state .== "IL")
```

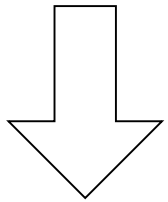


```
From(person) |>  
Where(Get.year_of_birth .>= 1970)  
Join(:location => From(location) |>  
                        Where(Get.state .== "IL"),  
      on = Get.location_id .==  
            Get.location.location_id)
```

FROM person p

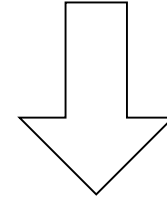


FROM person p
WHERE p.year_of_birth >= 1950

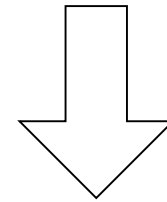


FROM person p
WHERE p.year_of_birth >= 1950
JOIN location l
ON (p.location_id = l.location_id)

From(person)



From(person) |>
Where(Get.year_of_birth .>= 1950)



From(person) |>
Where(Get.year_of_birth .>= 1950) |>
Join(:location => location,
Get.location_id .==
Get.location.location_id)