# Introduction to Databases and SQL

Christoph Draxler
draxler@phonetik.uni-muenchen.de

August 22, 2018

# Characteristic Properties of Databases

- structured data
- separation of data and application
- data integrity
  - consistency
  - protection
  - security
- temporal permanence
- user-specific views

# Structured Data

A database has a well-defined and visible structure

- described in a schema
- in a formal notation
- human and machine readable

Users must know the structure to access data

# Separation of Data and Application

A database generally serves more than one purpose

- re-use of data
- evolving user needs
- technology changes

Not all uses can be foreseen at the time of database creation.

# Data Integrity

A database system provides different types of integrity

**consistency** data is free of inconsistencies

**protection** data can only be accessed with proper access privileges

**security** data is protected from physical damage

These requirements contradict each other – data integrity is always a compromise.

# Data Modelling

Process of representing the real world in a data base.

| | |
|---|---|
| real world | things, people, abstract terms, ... |
| logical view | descriptions of things and their relationships |
| physical view | files, data formats, and storage devices |

In this course, we will mainly be concerned with the *logical view*.

# Data Model

Language to describe data, e. g.

- hierarchical
- network
- relational
- object-oriented
- deductive
- graph

Think of a data model as a *filter* through which you view the world

# Data Definition and Manipulation

Data Definition or *schema*

- data description in a given data model

Data manipulation

- operations on data in a given data model, e. g. create, read or retrieve, update and delete

Two distinct languages: DDL (data definition language) and DML (data manipulation language)

# Relational Data Model

The relational data model

- was proposed in 1970 by [Cod70]
- only one data structure, the relational table with *atomic* attributes

Many commercial and freeware relational database systems are available.

- SQL has become the de facto standard relational database language, both for data definition and manipulation

Any modern database textbook will treat relational databases extensively, e. g. [EN99]

# SQL Overview

SQL

- is based on relational algebra and relational calculus
- borrows a lot from standard English
- is an ISO-Standard
- is continuously being extended and updated

Application programs access SQL databases via a programming interface (API)

# Introduction to SQL

Data definition language and data manipulation language are separate sublanguages

DDL modify the database *structure*

DML modify and search the database *content*

We will start with some SQL queries...

# SQL Queries

Queries retrieve matching records from a database. A query consists of clauses in a given order:

| | |
|---|---|
| SELECT | list of columns to retrieve |
| FROM | list of the tables to use |
| WHERE | conditions joined by logical operators |
| | ... |
| ORDER BY | sort the result |

All clauses except `SELECT` are optional.

# Syntax of SQL Queries

The following *simplified and incomplete* grammar defines the syntax of SQL queries as they are being used in the course:

```
SELECT [DISTINCT] * | expression [ [ AS ] output_name ] [, ...]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ ORDER BY expression [ ASC | DESC ] [, ...] ]
    [ LIMIT count ]
```

where `from_item` is:

```
table_name [ alias ]
[ JOIN from_item ON join_condition ]
```

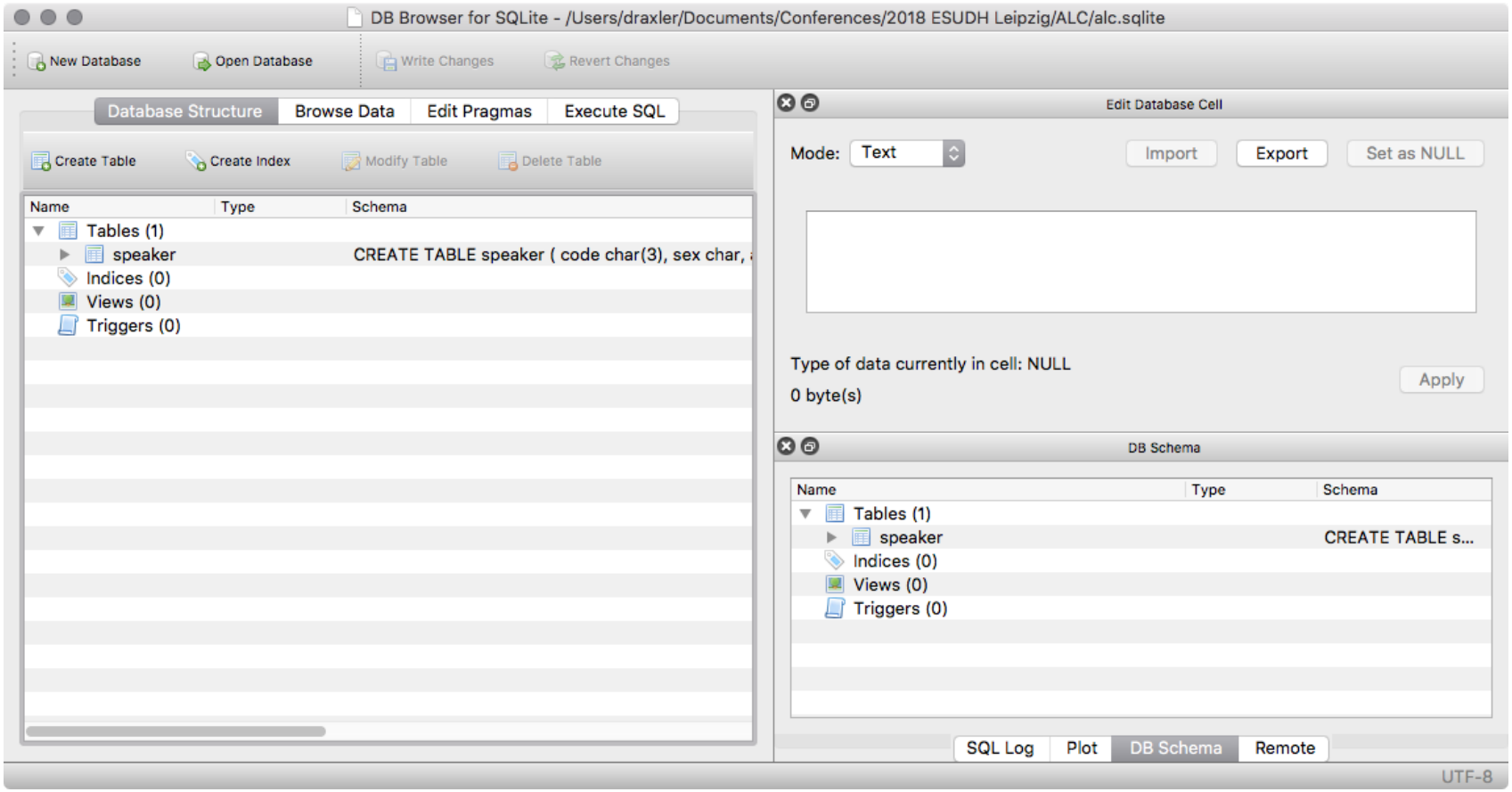
and expression is

```
[[ alias | table_name ] . ] column_name | function ( arguments )
```

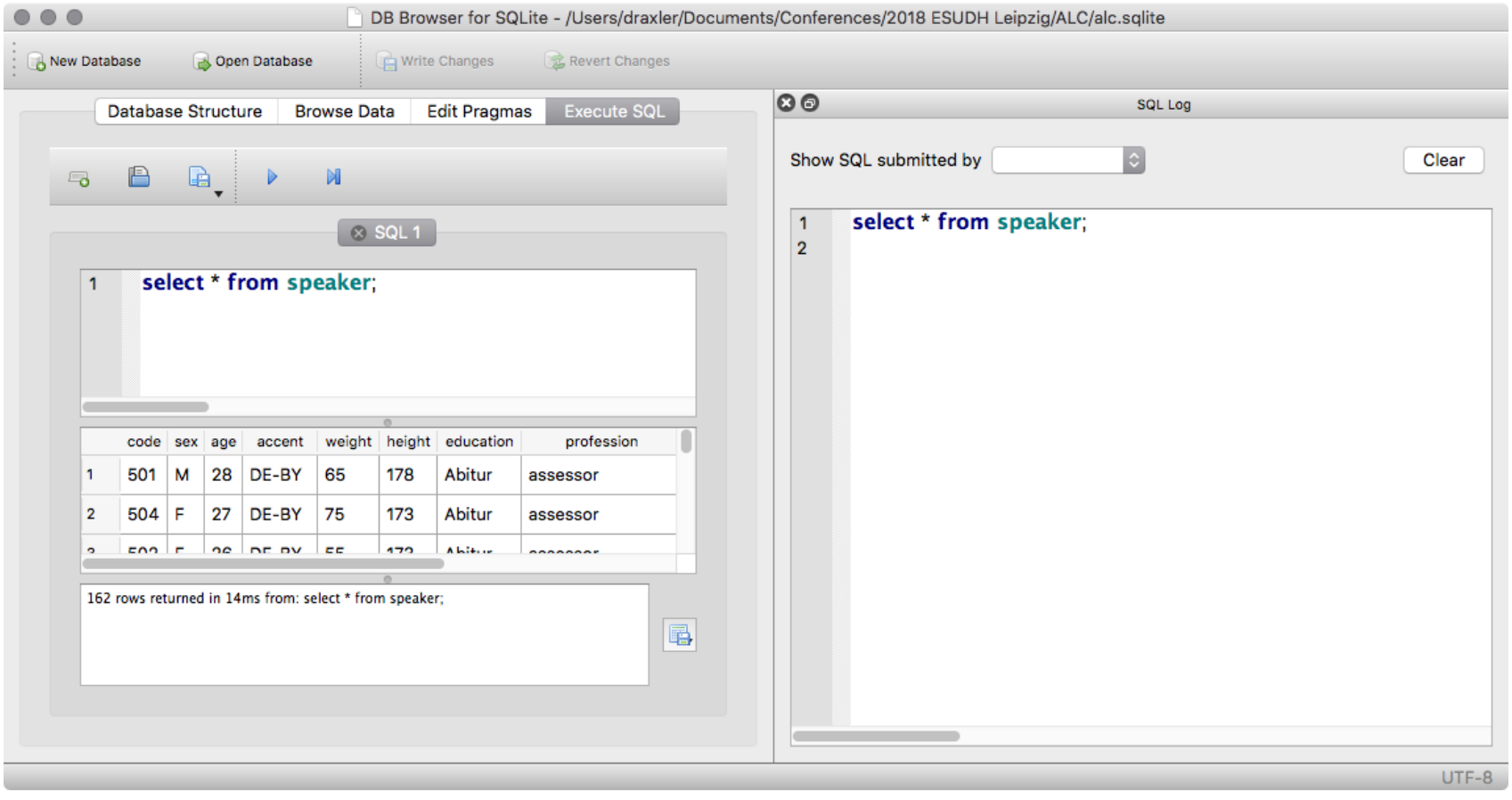
Check whether the queries in this text match this grammar!

# A First SQL Query

Open the ALC database either in a terminal or in DB Browser for SQLite.



# Enter SQL Queries




Press the little blue triangle to execute the query.

# Query with Conditions

'Retrieve all male speakers'

```
select *
from speaker
where sex = 'M'
```

# Query with Conditions

'Retrieve all male speakers from Bavaria'

```
select *
from speaker
where sex = 'M' and accent = 'DE-BY'
```

# Query with Conditions

'Retrieve all male speakers from Bavaria aged between 20 and 29'

```
select *
from speaker
where sex = 'M' and accent = 'DE-BY'
and age between 20 and 29
```

# Query with Conditions: Sort Records

'Retrieve all male speakers from Bavaria aged between 20 and 29 and order them by code'

```
select *
from speaker
where sex = 'M' and accent = 'DE-BY'
and age between 20 and 29
order by code
```

# Query with Conditions: Negation

'Retrieve all male speakers *not* from Bavaria and aged between 20 and 29 and order them by code'

```
select *
from speaker
where sex = 'M' and accent != 'DE-BY'
and age between 20 and 29
order by code
```

# Query with Conditions: Multiple Values

'Retrieve all speakers from Bavaria and Saxony and order them by code'

```
select *
from speaker
where sex = 'M'
and accent in ('DE-BY', 'DE-SN')
order by code
```

# Query with Conditions: Remove Duplicates

'Retrieve the different speaker accents'

```
select distinct accents
from speaker
order by accent
```

# Query with Alias Names

'Retrieve speaker code, age and accent for female speakers from Saxony'

```
select spk.code, spk.age, spk.accent
from speaker spk
where spk.sex = 'F' and spk.accent = 'DE-SN'
```

Alias names may be used as shortcuts or for disambiguation (see 24, 42)

# Queries with JOIN

The left and right queries produce the same result

```
select fedstate.label, city.*
from geolocation state
join geolocation fedstate
   on state.id = fedstate.reference
join geolocation city
   on fedstate.id = city.reference
where state.label = 'Deutschland'
order by city.latitude desc
```

```
select fedstate.label, city.*
from geolocation state,
   geolocation fedstate,
   geolocation city
where state.label = 'Deutschland'
and state.id = fedstate.reference
and fedstate.id = city.reference
order by city.latitude desc
```

I find the left one easier to understand because it separates the JOIN-comparisons from the content comparisons.

# Useful operators

`x between n and m` is `x` between `n` and `m` – this works for numerical and string data

`x like 'abc%'` does `x` begin with 'abc' – works for string data only

`x in (a,b,...,n)` is `x` in the set `(a, b, ..., n)` – inside the parentheses, there can be any set expression, i. e. also a query

`distinct` removes duplicates from the result set

# Queries vs. Data Manipulation

Queries do not change the database. Other commands do change data in the tables, e. g.

- INSERT records into a database table
- UPDATE records in a table
- DELETE records from a table

In general, these commands are combined with a `WHERE`-clause to restrict the effects.

# These commands are DANGEROUS!

`INSERT`, `UPDATE`, and `DELETE` generally have to be limited by a `WHERE`-clause.

```
update speaker set accent = 'DE-' || accent
```

will update *ALL* accent values in the table by adding a 'DE-' prefix.

# Transactions

take a database from a consistent state via some steps to a new consistent state - or back to the original state.

- INSERT, UPDATE and DELETE are *dangerous* because they modifiy the contents of the database

Embedding these commands in a transaction allows undoing these commands in case of an error

- `begin` starts a transaction
- `commit` ends the transaction successfully, the changes are accepted
- `rollback` returns to the database state before the begin of the transaction

# INSERT Command

```
insert into speaker (code, age, sex)
values ('X11', 25, 'm');
```

ads a 25-year old male speaker with the code X11 into the table `speaker`.

- if there are no attribute names after the table name, then there must be values for all attributes
- if attributes may not become null, then there have to be values for them

# UPDATE and DELETE Commands

```
update speaker set age = 35, sex = 'M' where code = 'X11';
```

changes the age of the speaker with code 'X11' to 35 and the value of the attribute `sex` to upper case.

```
delete from speaker where code = 'X11';
```

deletes all speakers from the table `speaker` with a code of 'X11'.

# Virtual Tables

Quite often, specific information must be extracted from several tables. This is cumbersome and error-prone

- virtual tables give (complex) queries a new name
- behave almost like real relational tables
- but are only computed when needed
- and they allow a very fine-grained specification of the visibilty of attributes

They can be used to monitor access to data and to dramatically reduce the query length

# Virtual Tables in SQL

In SQL a virtual table is called a *View*

- the definition of a view consists of two parts
- `CREATE VIEW` view name `AS` SELECT-query

In a query, a view can appear wherever a table can be written

# Aggregate Functions

Aggregate functions compute statistical values over the database:

- e. g. `count`, `sum`, `average`
- $f : R \to \mathbb{R}$
- are really useful in everday life
- go beyond relational algebra or relational calculus

They are different from other functions, e. g. `substring()`!

# SQL Aggregate Functions

SQL aggregate functions

- count(*) counts the tuples
- sum(x) sums the attribute values `x`
- avg(x) computes the average value of `x`
- min(x) compute the minimum value of `x`
- max(x) compute the maximum value of `x`

`sum()` and `avg()` are only defined for numerical values. `count()` can be used for any attribute, and it also accepts the `distinct` keyword.

# SQL Aggregate Functions: Syntax

Aggregate functions are written in the `SELECT`-clause of a query.
If there are also *regular* attributes in the SELECT-clause, then

- the query must have a `GROUP BY`-clause
- the query may have a `HAVING`-clause

GROUP BY groups the result relation by the specified attributes,
HAVING applies a condition to the group.

# Aggregate Function Example

How many speakers are there in the table speaker?

```
select count(*) from speaker;
```

```
count(*)
162
```

GROUP BY is not necessary because there is no regular attribute in the SELECT-clause.

# Aggregate Function Example

How many speakers of either sex are there in the table speaker?

```
select sex, count(*) from speaker;
```

```
sex count(*)
M 162
```

This is not the expected result – the count for female speakers is missing!

# Aggregate Function with GROUP BY

There is both a regular attribute and an aggregate function in the SELECT-clause. Thus, we need to use a GROUP BY clause to group the result by the values of the regular attribute.

```
select sex, count(*) from speaker group by sex;
```

```
sex count(*)
F 77
M 85
```

# GROUP BY and HAVING

Now, let's get only the count for speakers where there are more than 80 in a group.

```
select sex, count(*)
from speaker
group by sex
having count(*) > 80;
```

```
sex count(*)
M 85
```

HAVING takes a condition as argument – this condition usually contains an aggregate function.

# Aggregate Function in Nested Query

Get the speakers with a height larger than the average height

```
select id, sex, age, height
from speaker
where height > (select avg(height) from speaker)
order by id, height, age, sex
```

Why isn't GROUP BY needed?

# Aggregate Function in Nested Query

Get the speakers with a height larger than the average height of their sex

```
select spk1.id, spk1.sex, spk1.age, spk1.height
from speaker spk1
where spk1.height >
    (select avg(spk2.height)
    from speaker spk2
    where spk1.sex = spk2.sex)
order by spk1.id, spk1.height, spk1.age, spk1.sex
```

Why do we need the alias variable?

# Aggregate Function and Nested Query

Get the average height per age group as a percentage of the average height of all speakers.

```
select s1.age, round(avg(s1.height), 2) as avg_group,
   round((select avg(s2.height) from speaker s2), 2) as avg_all,
   round(avg(s1.height) / (select avg(s2.height) from speaker s2) * 100, 2)
     as percent,
   count(*), min(s1.height), max(s1.height)
from speaker s1
where s1.age > 0
group by s1.age
order by s1.age
```

What's special about this query?

# Query Analysis

The columns in the SELECT-clause contain a nested query returning an aggregate value.

```
round(avg(s1.height) / (select avg(s2.height) from speaker s2) * 100, 2)
  as percent,
```

- `round(..., 2)` rounds the number to two digits after the decimal point
- `as percent` labels the column with `'percent'`
- `avg(s1.height)` computes the average size of the speaker in the current group
- `(select avg(s2.height) from speaker s2)` is a nested query returning a single value

# Extending this Aggregate Query

Now calculate this separately for female and male speakers

```
select s1.sex, s1.age, round(avg(s1.height), 2) as avg_group,
   round((select avg(s2.height) from speaker s2 where s2.sex = s1.sex), 2)
     as avg_all,
   round(avg(s1.height) / (
     select avg(s2.height)
     from speaker s2
     where s2.sex = s1.sex) * 100, 2)
   as percent,
   count(*), min(s1.height), max(s1.height)
from speaker s1
where s1.age > 0
group by s1.sex, s1.age
order by s1.sex, s1.age
```

# Summary Data Manipulation

In the relational model, almost everything is a relational table

- results of queries are again tables
- queries can be nested inside conditions
- unrestricted commands affect entire tables
- alias names allow disambiguation of attributes
- virtual tables are a means to customize queries

In SQL, you do not need to think about *how* the result is computed – you write a query expression and the database system finds *ALL* matching records.

# SQL Data Definition

Create, alter and drop tables

- CREATE databases or tables
- ALTER modify or rename tables, attributes, etc.
- DROP databases or tables

Commonly, create all tables of a database first and only then enter your data. Because of inter-table dependencies, it may be necessary to do this in a specific order.

# Data Types

SQL knows at least the following data types

smallint, int small resp. large whole numbers

real, double precision normal resp. double precision floating point numbers

char(N), varchar(N) strings of fixed resp. variable length

date, time, timestamp Date, Time and timestamp (with time zone)

Most database system support further data types, e. g. text.

# Non-Standard Data Types

Many SQL systems support additional data types.

- **Array** structured sequences with numerical index
- **BLOB** Binary Large Object, a stream of bytes for storing e. g. media data
- **Geometric types** geometric data
- **XML** structured XML data
- **JSON** JavaScript object notation

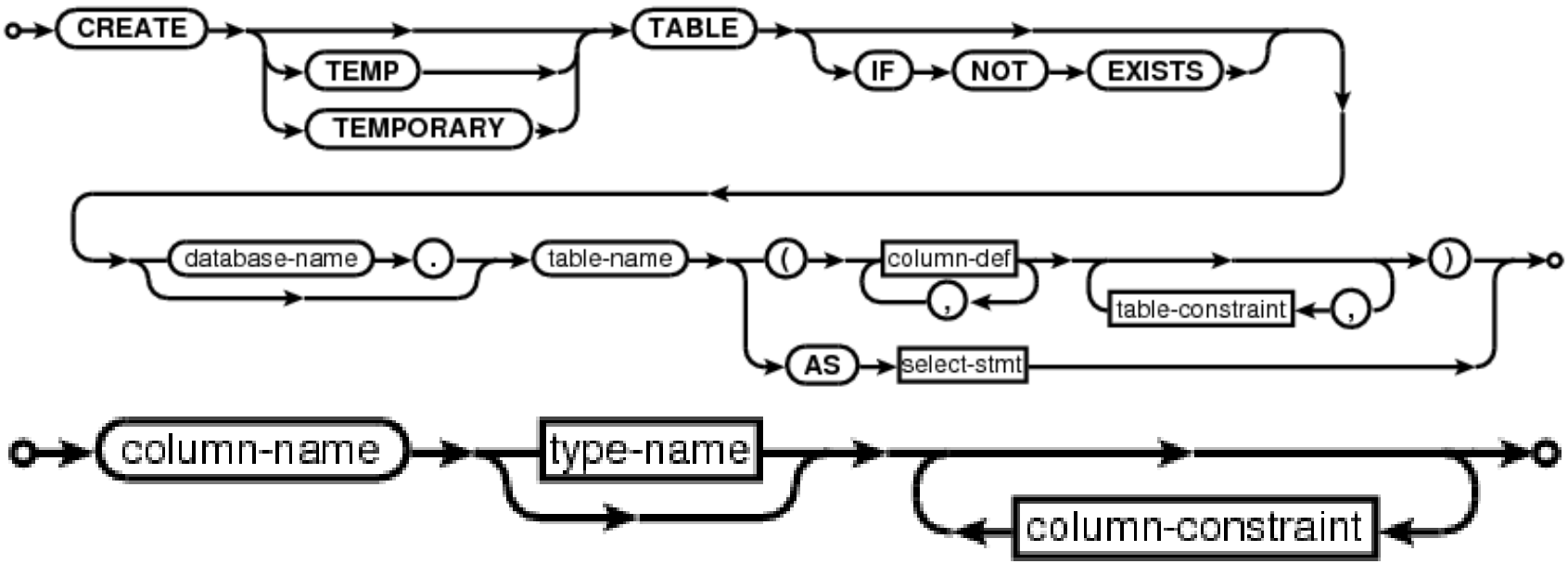
These data types do not fit the relational operators well.

# Grammar of the CREATE TABLE Command

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name ( [
  { column_name data_type [ DEFAULT default_expr ] [ column_constraint [ ... ]
    | table_constraint
    | LIKE parent_table [ { INCLUDING | EXCLUDING } { DEFAULTS | CONSTRAINTS }
    [, ... ]
])
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace ]
where column_constraint is:
[ CONSTRAINT constraint_name ]
...
```

from: online-Help in the SQL Terminal of postgreSQL

# Graphical Representation of the Grammar




from: SQLite Documentation
`http://www.sqlite.org/langcreatetable.html`

# Creating a Table in SQL

Create a table describing geographical locations

```
create table geolocation (
    id int primary key,
    label text,
    latitude float,
    longitude float,
    type text,
    comment text
)
```

# Creating a Linked Table in SQL

```
create table student (
    id int primary key,
    familyName text,
    firstName text,
    sex char(1),
    dateOfBirth date,
    isFrom int references geolocation (id)
)
```

`references` establishes the link of the attribute `isFrom` to the `id` attribute of the table `geolocation`. The database system will now only accept entries with a valid reference.

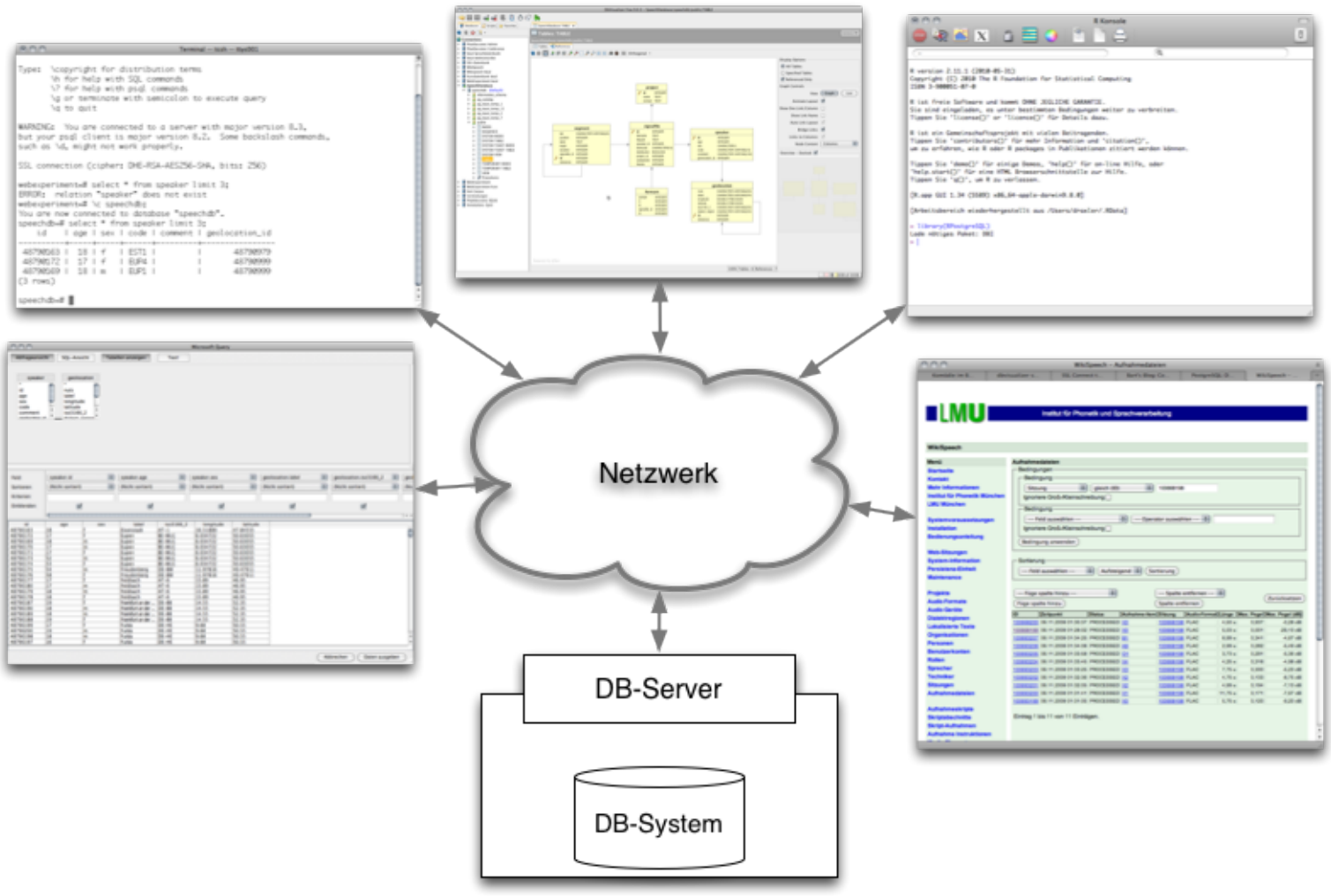
# Accessing SQL from External Software

Many programming languages, statistical packages, or spreadsheet applications have database interfaces.
In general, access to the database is as follows:

1. install the interface library (or *driver*) for your database system
2. open the connection to your database
3. send an SQL-query
4. retrieve the result data, one record after the other or all at once

Common names for database interfaces are ODBC (*Open Database Connectivity*), JDBC (*Java*), DBI (*Database Interface*).

# Database Architecture




Commonly, a database system is a *client-server* system (exception: SQLite)

# Accessing SQLite from R

Here's an example for the statistics package R and SQLite:

```
> install.packages("RSQLite")
Installing package into '/Users/draxler/Library/R/3.2/library'
...
The downloaded source packages are in...

> library(DBI)
> con = dbConnect(RSQLite::SQLite(),  "WORK_DIRECTORY/speechdb.sqlite")
> myQuery = dbSendQuery(con, "SELECT * FROM speaker")

> myData = dbFetch(myQuery, n = -1)

> myData
     id age sex code comment geolocation_id weight height accent
1  4254  16   f                    1044036     53    170     BY
2  4256  14   f                    1044039     56    174     BY
3  4260  15   f                    1044036     60    175     BY
4  4261  13   f                    1044039     45    160     BY
5  4262  13   f                    1044039     50    165     BY
...
```

The > is the prompt of the R interpreter. Commands are entered after the prompt.

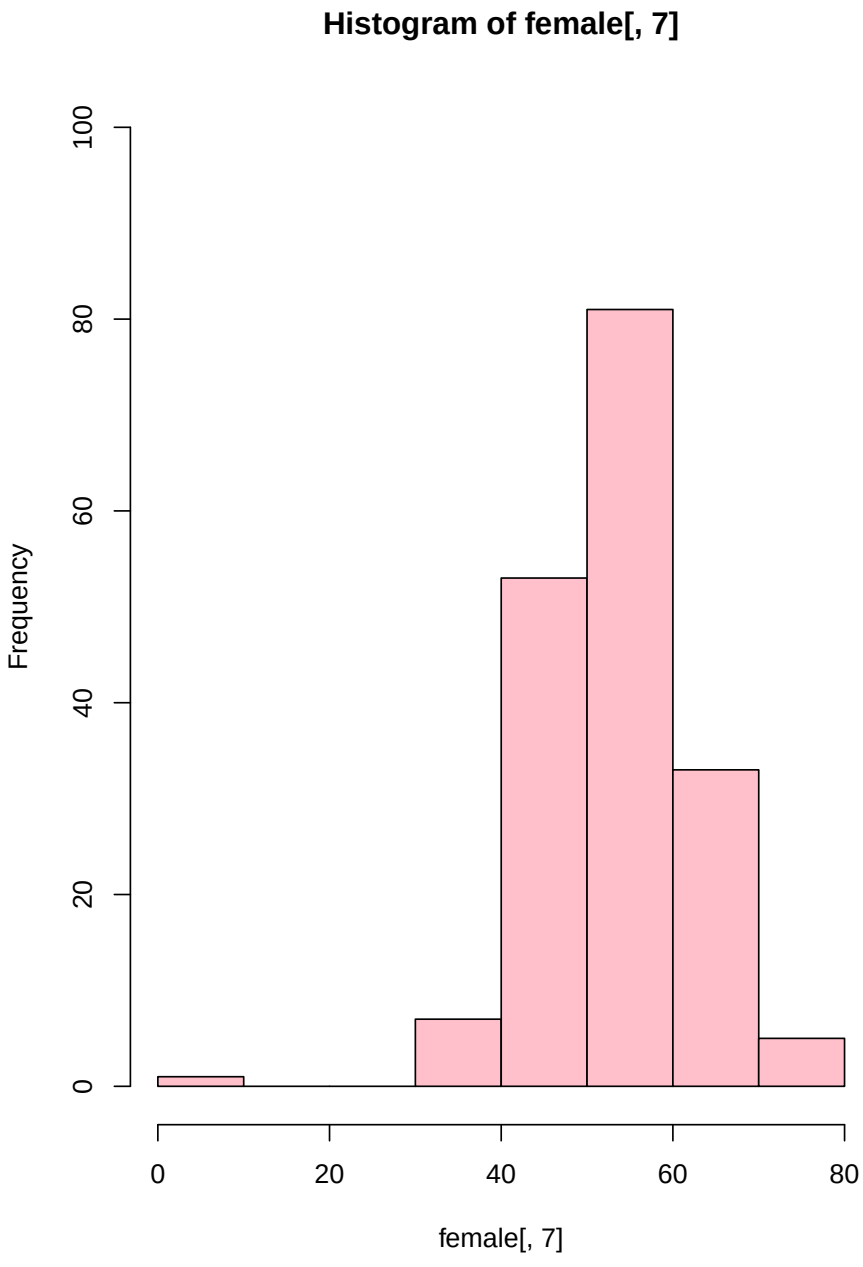
# Accessing SQLite from R (cont'd)

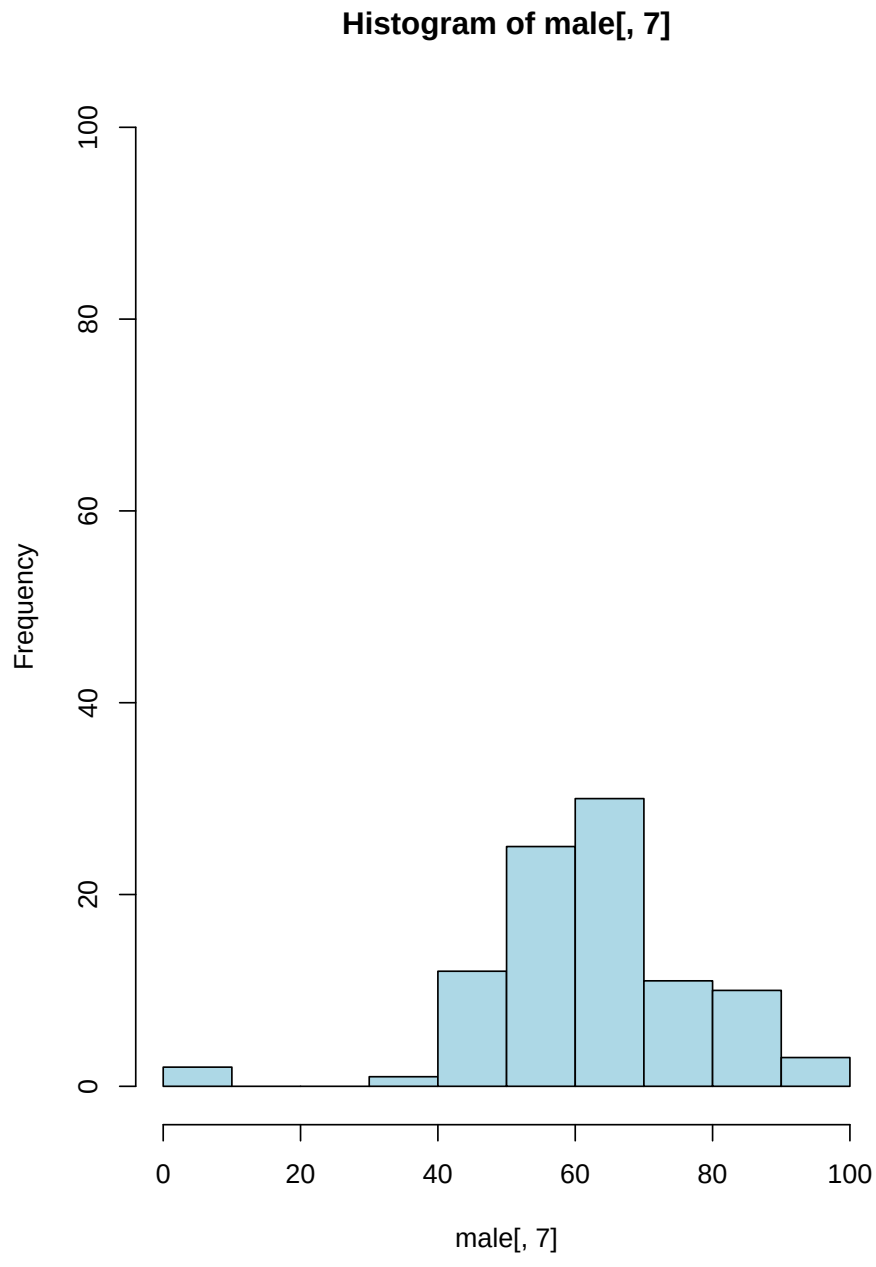
```
> female = subset(myData, sex == 'f')
> male = subset(myData, sex == 'm')

> par(mfrow = c(1,2)

> hist(female$weight, col="pink", ylim = c(0, 100), main="Weight (f)")
> hist(male$weight, col="lightblue", ylim = c(0, 100), main="Weight (m)")
```

# Accessing SQLite from R (cont'd)



# SQLite Access from Python

Here's how to do it from Python

```
import sqlite3
connection = sqlite3.connect("DATABASE_NAME")

cursor = connection.cursor()

cursor.execute("SELECT * FROM speaker")
print("fetchall:")
result = cursor.fetchall()
for r in result:
    print(r)
```

As an alternative, you can retrieve a single record using `fetchone()`

# Access to Databases via the Web

Often, web pages offer access to databases via customized user forms. Examples are repositories, information system, corpus pages. Such systems are generally based on the following technologies:

- Tomcat web server with Java and JDBC
- node.js plus database interface
- php and phpadmin with a web server
- Shiny package and R
- content management systems such as Typo3, Drupal, Wordpress and others require a database installation

Ask your friendly administrator for help!

# Discussion of Database Interfaces

Pros

- supported by many external programs
- standardised mode of operation
- dynamic generation of queries
- use database for what it can do best: quick access to large amounts of data

Cons

- datastructures of external program and database often do not match
- 'thinking in two languages' is necessary
- system or database administator privileges may be required

# Warning

Database access from the outside is always a security risk.

- restrict access to the minimum necessary
- keep your software up to date
- use proven technology
- get professional support, e. g. from your local admin

Hackers are not interested in your data, but in getting into your institution's network!

# Graph Databases

Basic idea: nodes are connected via relations

- nodes and relations are named
- nodes and relations have attributes
- relations may exist between any pair of individual nodes
  relations are directed

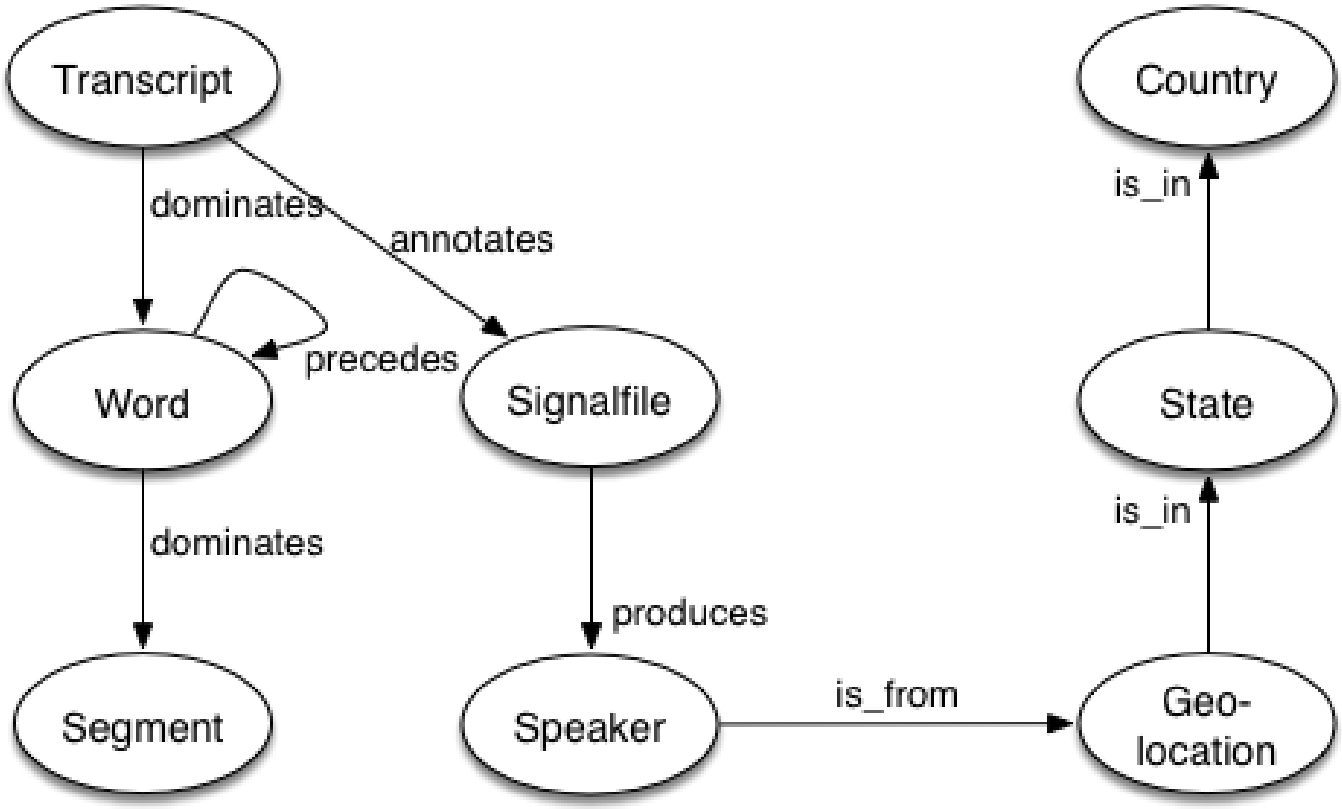
Queries via pattern matching

- allows recursion, limited or unlimited
- in queries, relations may be directed or bidirectional

Server, desktop and embedded versions are available ([RWE15])

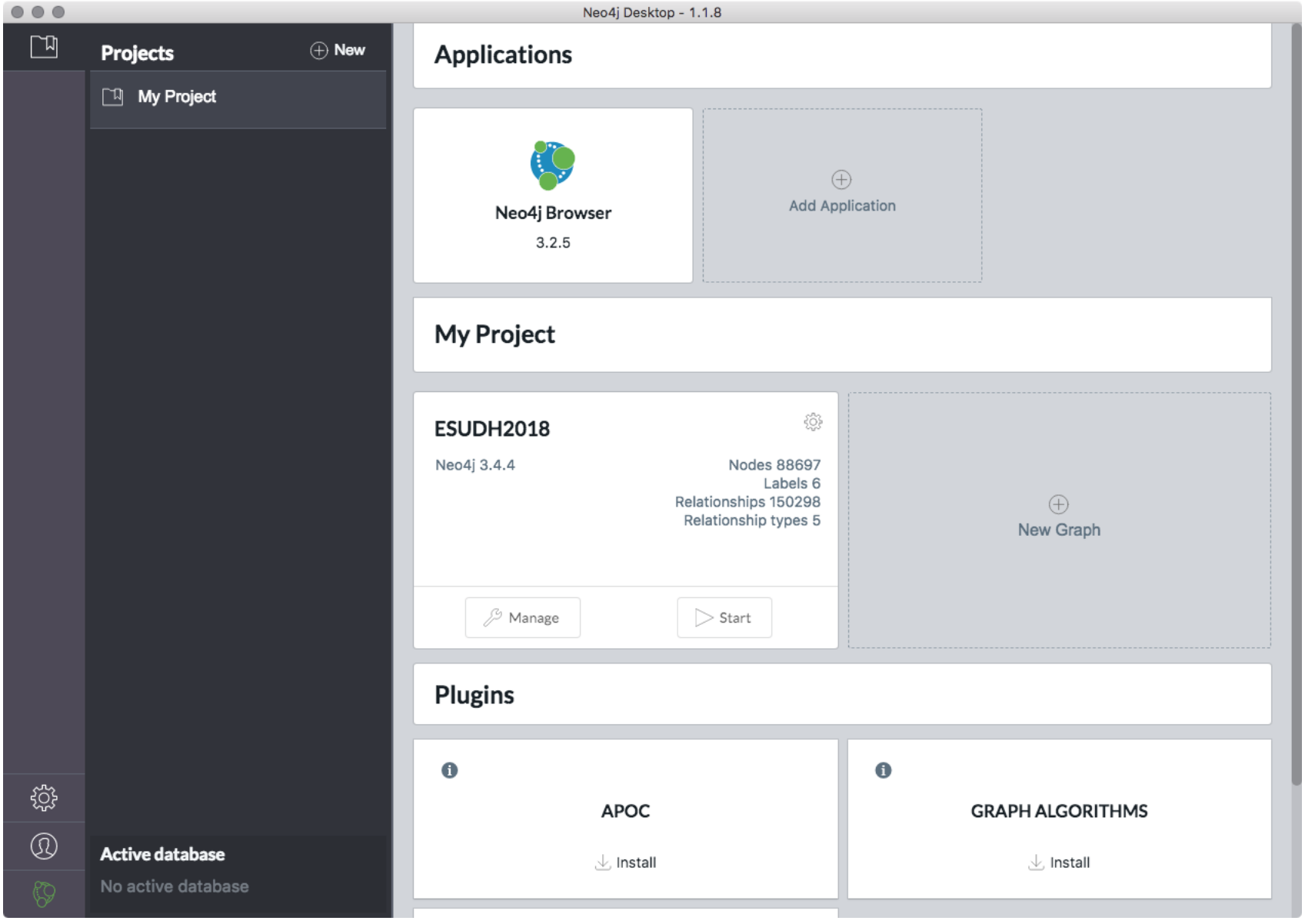
- commercial product (neo4j.com)
- restricted version available for free (Neo4j Desktop)

# Phonetic Database as Graph DB




Note that instead of one general purpose relational table for segments there are now three node types, `Segment`, `Word`, and `Transcript`, linked by the relation `dominates`. Similarly for `Geolocation`, which now is in a `State` which in turn is in a `Country`.

# Neo4JDesktop

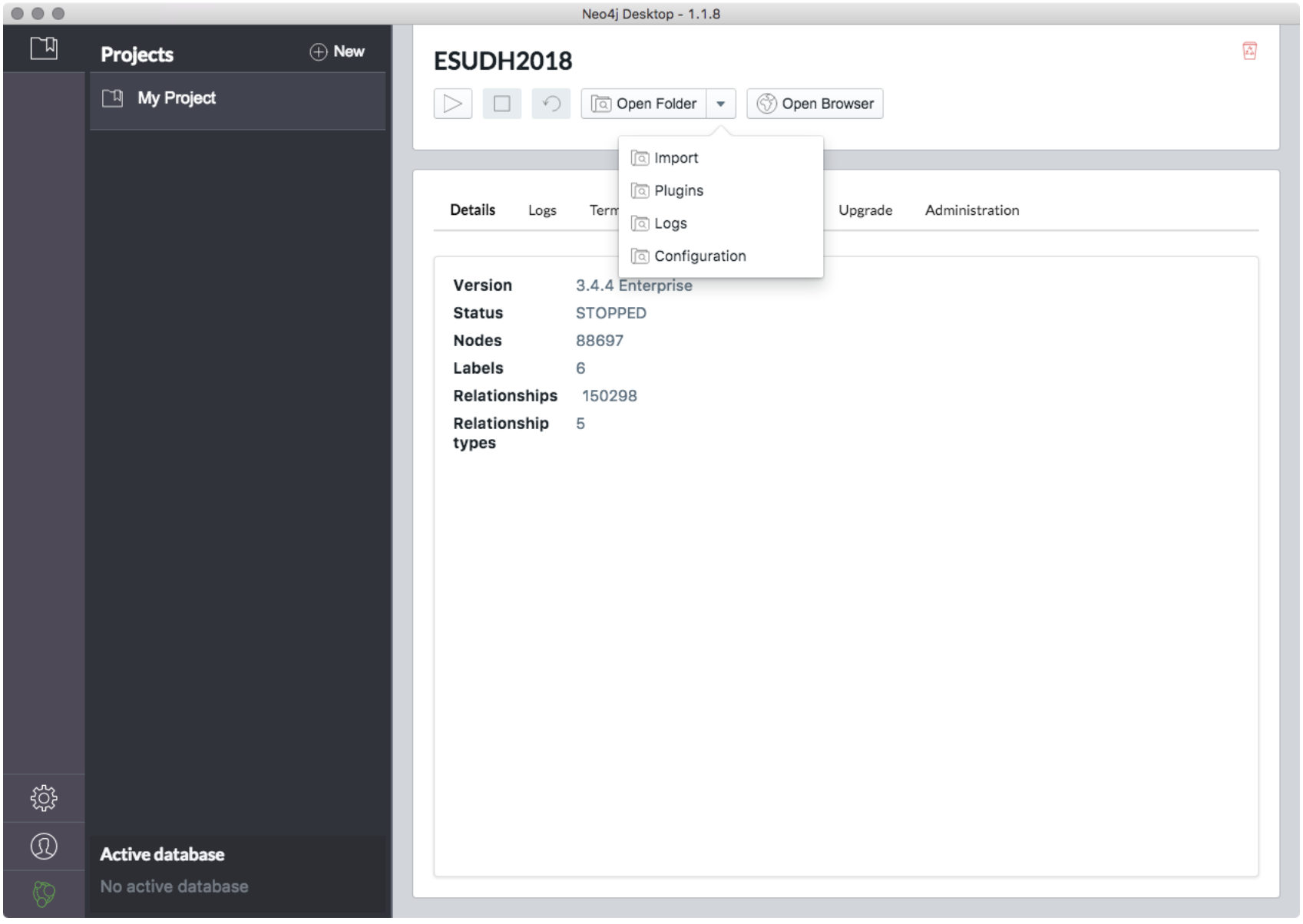


# Import Data into Neo4j

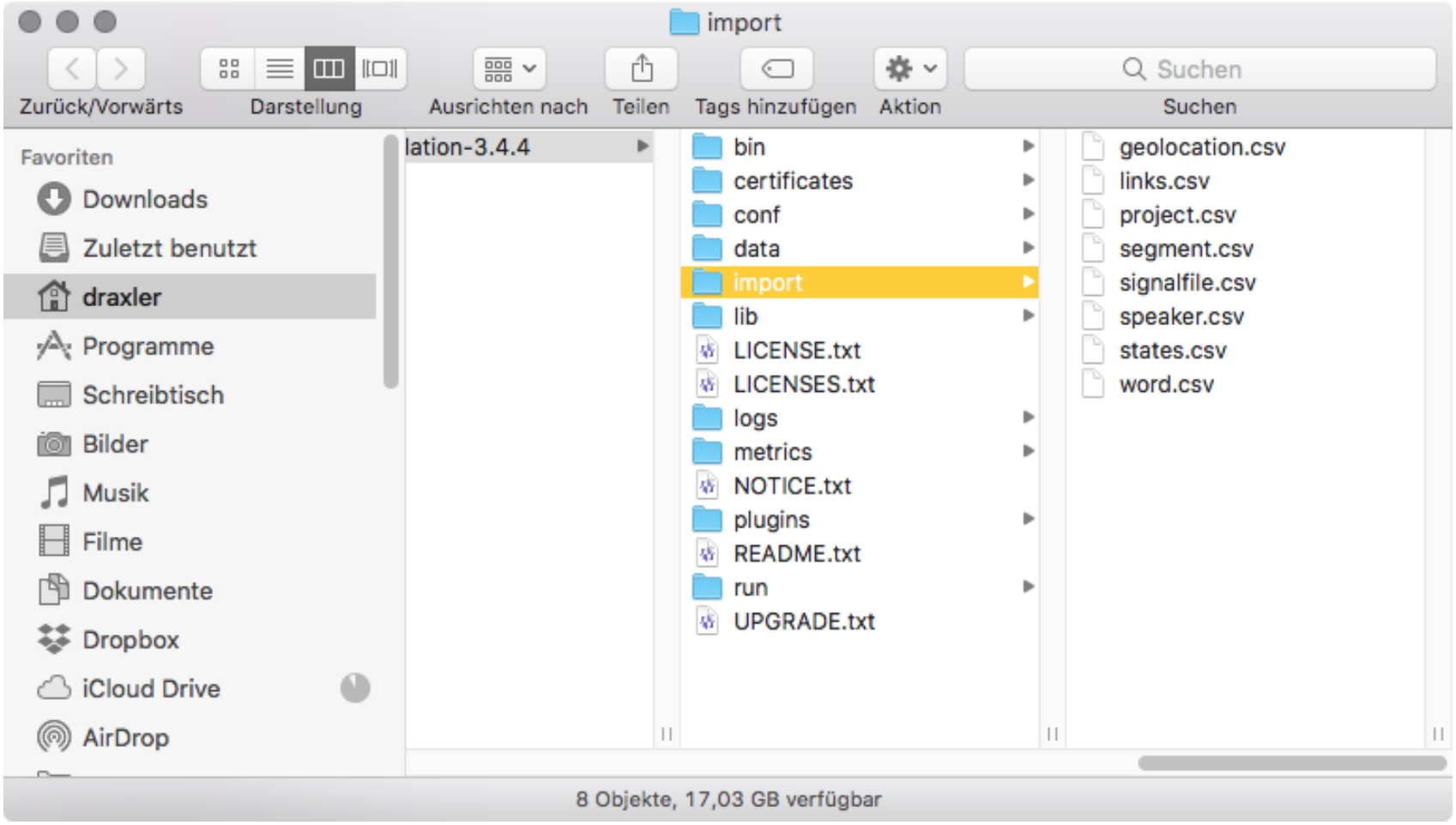
Neo4j allows importing data from comma- or tab-separated files using the command load csv.

- files should be in a folder named import in the local Neo4j-folder
- use `using periodic commit` before the `load csv` command for large imports
- you must explicity convert items to the appropriate type when importing; default is string
- you may map csv header labels to graph DB attribute names, e.g. `set g.name = row.label`
- you can create relations after importing nodes

# Neo4j Import Data: Manage



# Neo4j Import Data: Set Import Folder




Drop the files that you want to import into this folder

# Import Data into Neo4j

Go back to the start screen and open the browser by clicking on the browser icon. In the top area of the screen, enter the IMPORT command.

```
load csv with headers from "file:/state.csv"
as row create (st:State)
set st.code = row.code,
st.name = row.label,
st.population = toFloat(row.population)
```

`"file:/state.csv"` is a URL, the initial '/' tells the Neo4jBrowser that the file is in the import folder. An explicit type conversion is necessary for numerical data.

# Import Data into Neo4j

Go back to the start screen and open the browser by clicking on the browser icon. In the top area of the screen, enter the IMPORT command.

```
load csv with headers from "file:/geolocation.csv"
as row create (g:Geolocation)
set g.key = row.key, g.label = row.label,
g.longitude = toFloat(row.longitude),
g.latitude = toFloat(row.latitude),
g.state = row.state
```

Note that the last line assigns the column named `code` in the CSV-file to the property `state` in the node `Geolocation`. The database now contains nodes of type `State` and `Geolocation`.

# Database Content after Import

States nodes have three properties:

| | |
|---|---|
| `code` | ISO 3166-2 code for the federal state, e. g. 'DE-BY' for 'Bayern' (*Bavaria*). |
| `name` | name of the federal state |
| `population` | floating point value of the population of the state |

Geolocation nodes have five properties:

| | |
|---|---|
| key | unique numerical ID |
| label | name of the geolocation |
| longitude | floating point value of the longitude |
| latitude | floating point value of the latitude |
| state | ISO 3166-2 code of the federal state |

# Generate Relationships between Nodes

The `Geolocation` node contains the property `state` which refers to the matching entry in a `State` node. In relational databases, this is a *foreign key*. In a graph database, a relationship links two nodes.

```
match (g:Geolocation), (st:State)
where g.state = st.code
create (g)-[i:is_in]->(st)
```

Once this relationship has been created, the foreign key property is deleted.

```
match (g:Geolocation)
remove g.state
```

# Import speaker Data

```
load csv with headers from "file:/speaker.csv"
as row create (s:Speaker)
set s.key = toInteger(row.key),
s.sex=row.sex, s.age=toInteger(row.age),
s.geolocation_id = row.geolocation_id
```

# Create Relationships between Speakers and Geolocations

```
match (spk:Speaker), (geo:Geolocation) where
spk.geolocation_id = geo.key
create (spk)-[i:is_from]->(geo)

match (spk:Speaker) remove spk.geolocation_id
```

Here, the `geolocation_id` property is a foreign key to the matching `Geolocation` item. After having created the relationship, the property `geolocation_id` is deleted.

# Continue Import

Import signalfile data from the CSV-files in the same manner:

1. import the file `signalfile.csv` into nodes of type `Signalfile`
2. generate the relationship `-[p:produces]->` between `Speaker` and `Signalfile`
3. delete the foreign key property linking `Signalfile` and `Speaker`

The database now contains quite a number of node items and relationships!

# Continue Import

Continue importing data from the CSV-files in the same manner. Check how the tables are linked via foreign keys, and make sure to use type conversion where necessary:

1. import the file `transcript_segments.vsv` into nodes of type `Transcript`
2. generate the relationship `-[p:annotates]->` between `Transcript` and `Signalfile`
3. import `ort_segments.csv` into nodes of type `Word`
4. generate the relationship `-[d:dominates]->` between `Transcript` and `Word`
5. import `mau_segments.csv` into nodes of type `Segment`
6. generate the relationship `-[d:dominates]->` between `Word` and `Segment`

Finally, remove all redundant foreign key properties.

# Cypher Graph Query Language

Cypher is Neo4j's query language for graph DBs

- nodes are represented by round brackets, e. g. `(st:State)`
- relations by ASCII-art arrows, e. g. `-[i:is_in]->`
- node and relation names are case-sensitive, i. e. `State` is not the same as `state`
- a colon : separates variables from node and relation names, e. g. `st:State`
- variables represent nodes or relations, e. g. `(o:ORT) -[d:dominates]-> (m:MAU)`

# Cypher Queries Strucure

A query consists of the following parts

- **match** starts search using pattern matching; there may be more than one match clause!
- **where** conditions that must be met
- **return** lists the items to display in the result

The scope of variables is the entire query

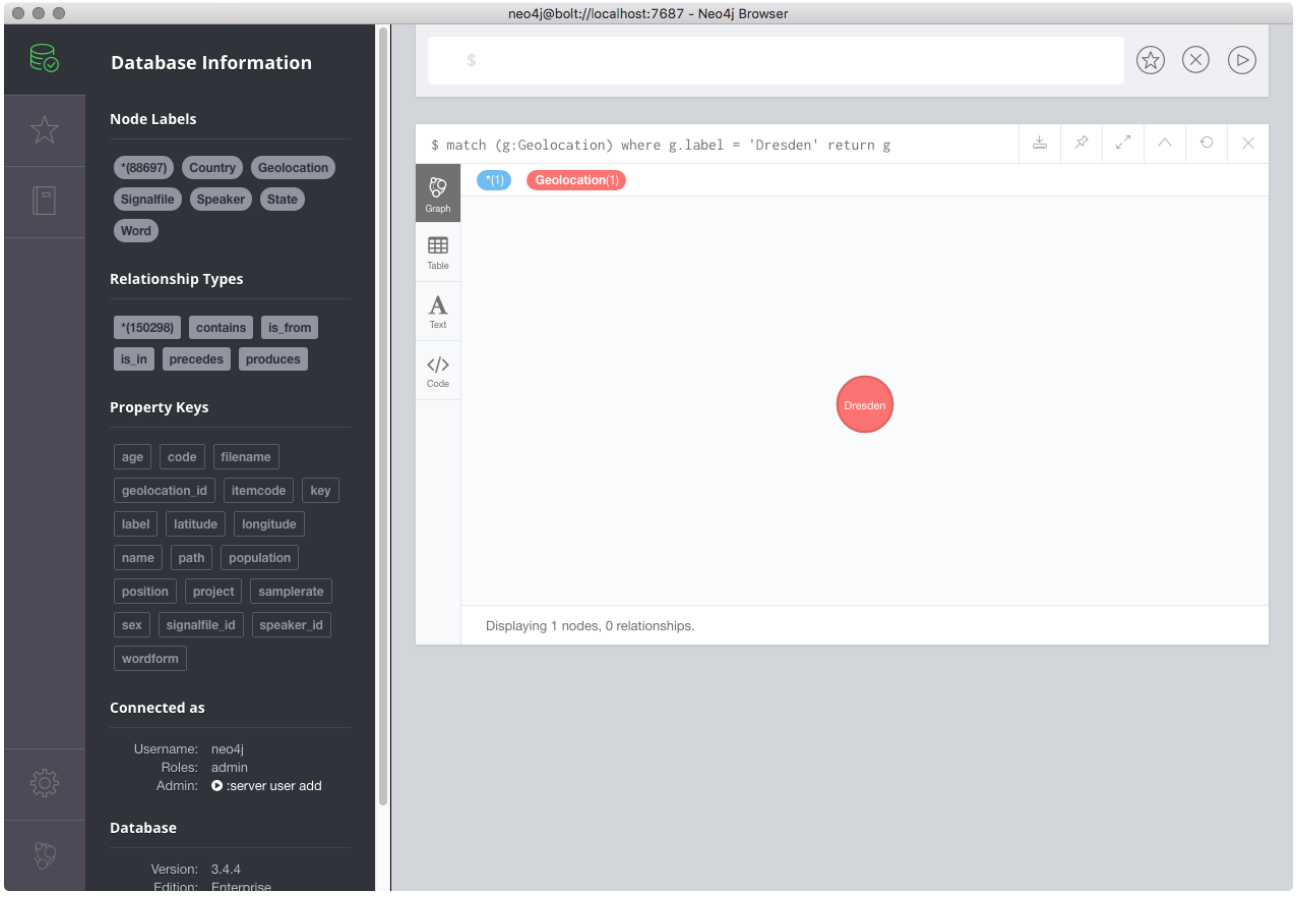
# Sample Cypher Query

"Get geolocations named Dresden"

```
match (g:Geolocation)
where g.label = "Dresden"
return g
```

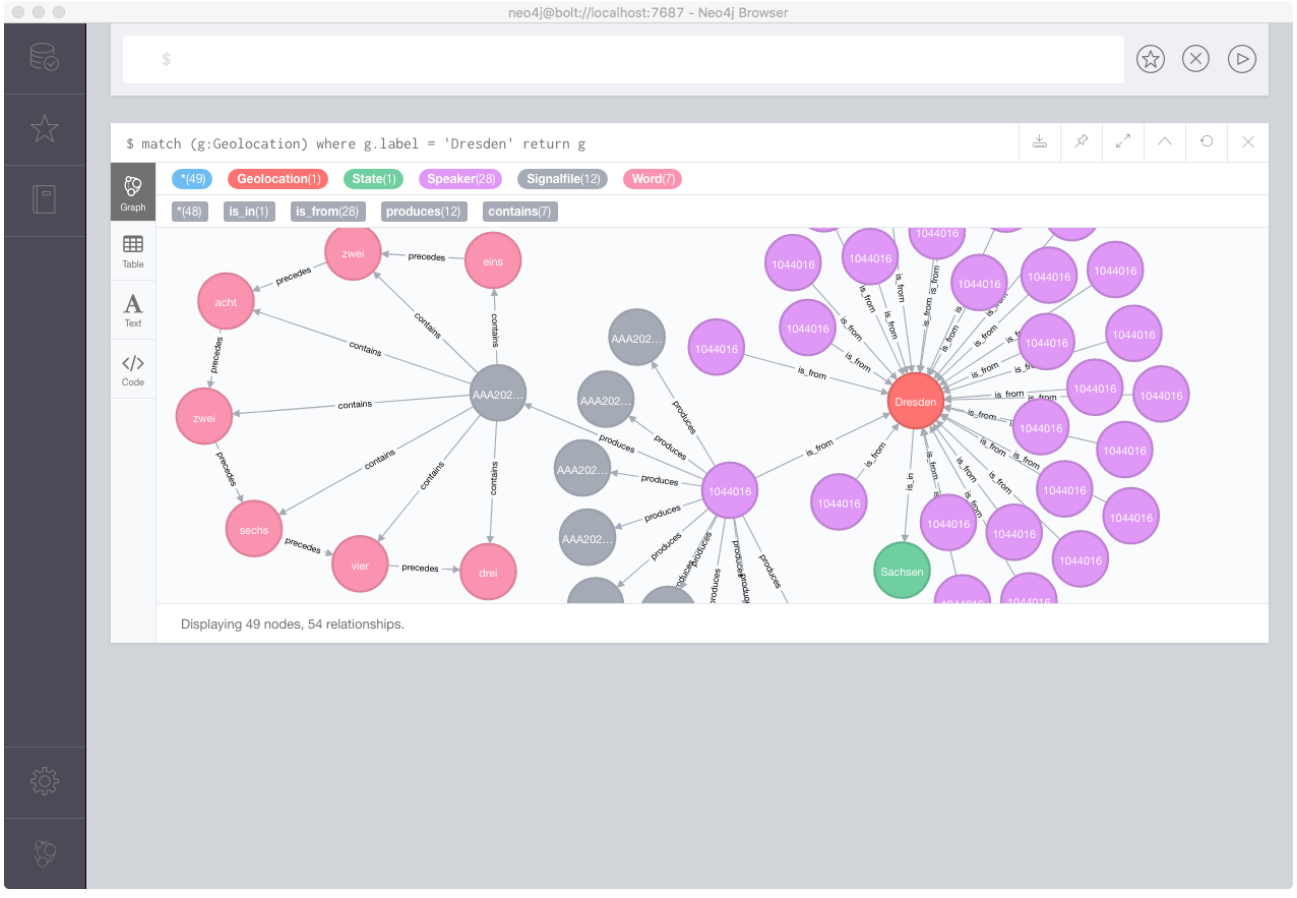
There are many comparison and other operators available to restrict queries.

# Explore the Database




Click on the oval labelled "Geolocation" under the query field. A status bar opens at the lower end of the graphics panel. There you can select which property of the node to display and which size or colour the node should have.

# Explore the Database




A double click on the node opens all linked nodes. Select a `Speaker` node and double click. Continue...

# Sample Cypher Queries

Instead of browsing the graph you can query the database, e. g. "Get all geolocations in Saxony"

```
match (g:Geolocation) -[:is_in]-> (st:State)
where st.name = "Sachsen" return g
```

Relations may be empty if their content is not of interest, e. g.

```
match (g:Geolocation) --> (st:State)
where st.name = "Sachsen" return g
```

# Sample Cypher Queries

"Get all speakers from cities in Saxony who are male and 18 years old"

```
match (spk:Speaker) --> (g:Geolocation) --> (st:State)
where st.name = "Sachsen" and spk.age = 18
and spk.sex = 'm'
return spk, g, st
```

# Sample Cypher Queries

Link the words in the transcript of a sentence:

```
match (w1:Word)<--(t:Transcript)-->(w2:Word)
where (w1.position + 1 = w2.position)
create (w1)-[p:precedes]->(w2)
```

Note how compact Cypher expresses the fact that two words belong to the same transcript.

# Comparison of SQL and Cypher

The following SQL query searches for the count and average duration of the diphthong /aI/ in the words 'eins', 'zwei' and 'drei'.

```
select ort.label, mau.label, count(*), avg(mau.duration)
from segment ort
   join links l on l.lto = ort.id
   join segment mau on l.lfrom = mau.id
where mau.label = 'aI'
and ort.label in ('eins', 'zwei', 'drei')
group by mau.label, ort.label
```

# Comparison of SQL and Cypher

This is the same query in Cypher

```
match (w:Word)-[:dominates]->(mau:Segment)
where (w.wordform = "eins" or w.wordform="zwei"
   or w.wordform="drei")
   and mau.label = "aI"
return w.wordform, mau.label, count(mau),
avg(mau.duration)
```

The main difference is in the FROM/JOIN-clause, and the lack of a GROUP BY in Cypher.

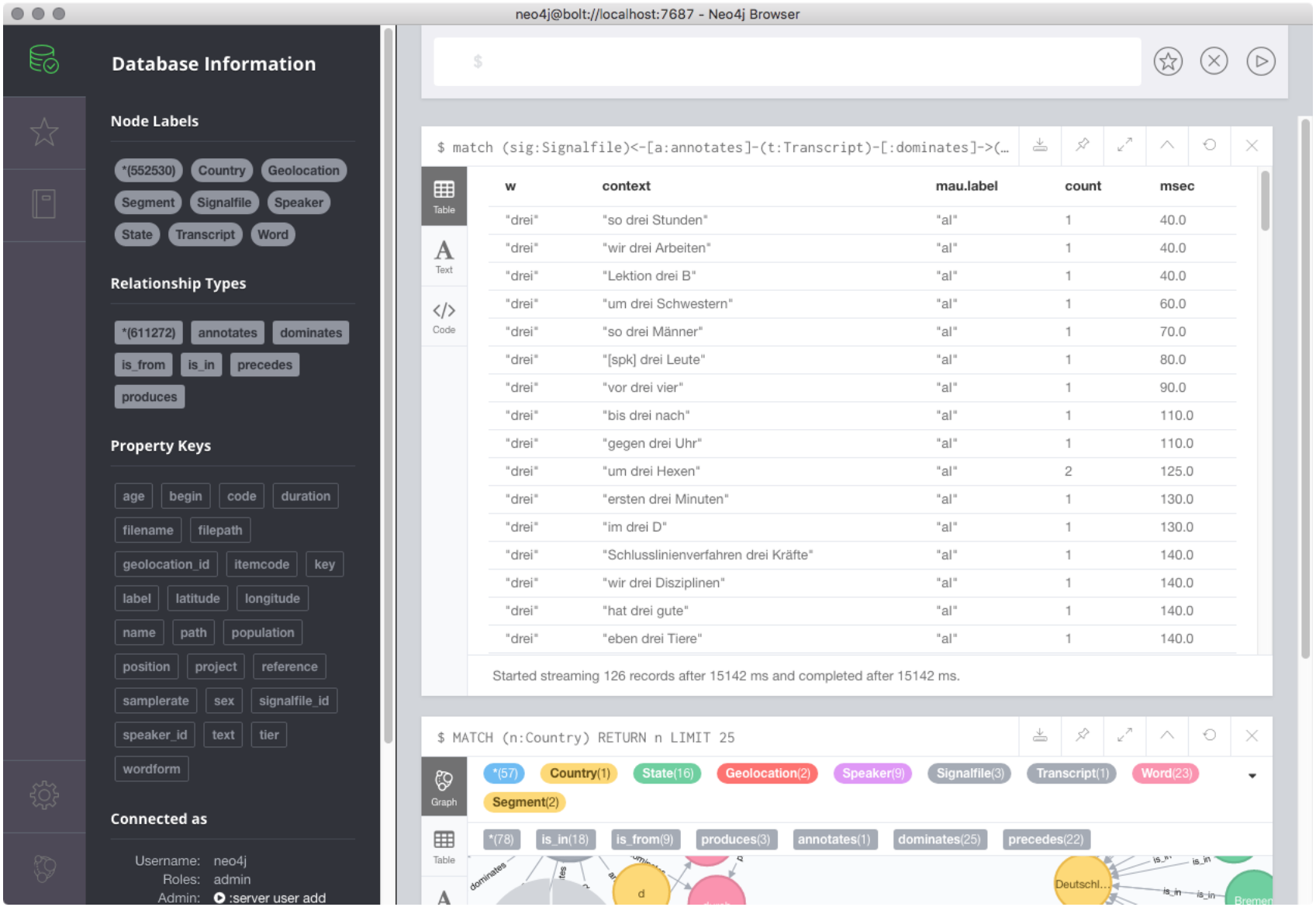
# Extending the Query

The query can be extended to show the duration in ms instead of samples, and the context of the word:

```
match (sig:Signalfile)<-[a:annotates]-(t:Transcript)-[:dominates]->
   (w:Word)-[:dominates]->(mau:Segment)
match (w1:Word)-[:precedes]->(w:Word)-[:precedes]->(w2:Word)
where (w.wordform = "eins" or w.wordform="zwei"
   or w.wordform="drei")
   and mau.label = "aI"
return w.wordform as w, w1.wordform + ' ' + w.wordform
   + ' ' + w2.wordform as context,
   mau.label, count(mau) as count,
   round(avg(mau.duration)/sig.samplerate * 1000) as msec
order by w.wordform, msec
```

Note that there are now two MATCH-patterns, and the variable `w` is shared between the two patterns.

# Result of the Query



# References



E. F. Codd.
A relational model for large shared data banks.
*Communications of the acm*, 13(6):377–387, 1970.


R. Elmasri and S. Navathe.
*Fundamentals of Database Systems*.
Benjamin Cummings, Redwood City, 3rd Edition, 1999.


Ian Robinson, Jim Webber, and Emil Eifrem.
*Graph Databases*.
O'Reilly & Associates, 2015.