INTRODUCTION TO RELATIONAL DATABASE SYSTEMS

DATENBANKSYSTEME 1 (INF 3131)

Torsten Grust
Universität Tübingen
Winter 2021/22

MINI WORLDS

Database systems are designed to capture well-defined subsets of the real world, the so-called **mini worlds**.

Mini World

A mini world contains the relevant **objects** (or: entities, things) of a real-world subset. Only the significant **attributes** (or: characteristics) of these objects are preserved. Objects may **relate** to each other. Specific **constraints** (or: rules of the world) are captured as well.

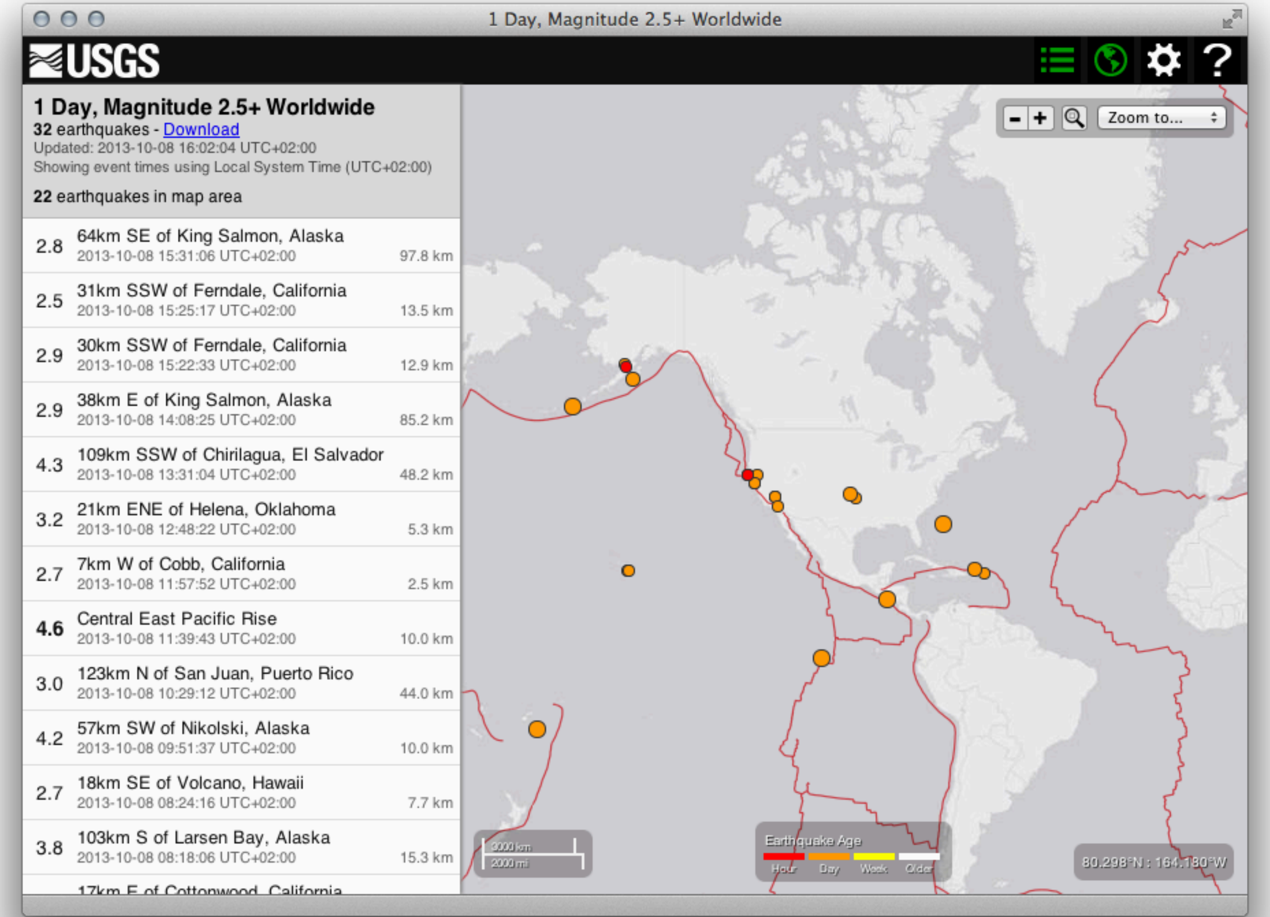
MINI WORLDS

Mini worlds may represent subsets of our (true, physical) environment as well as any of the many *virtual worlds* that we create.

Mini World Example: US Geological Survey Earthquake Maps

Real world
 The Earth
Subset covered
 Global real-time earthquake information
Relevant objects
 Quakes, locations, date/time
Significant attributes
 Magnitude, latitude, longitude, depth, day, hour, min, sec, ...

MINI WORLDS



Available at `http://earthquake.usgs.gov/earthquakes/map/`

MINI WORLDS

Mini World Example: Enterprise Data (→ TPC-H [1])

Real world
Company/Corporation
Subset covered
Ordering and Fulfillment, Client Relationships, Supply Chain
Relevant objects
Orders, Lineitems, Products, Suppliers, Customers, Shipments, …
Significant attributes
Product IDs, order/shipment dates, ordered quantities, prices, names, …
Constraints
“The price of an order must be the sum of the prices of its individual lineitems”

MINI WORLDS

Mini World Example: Web Sites (Amazon, Wikipedia, YouTube)

Real world

The World Wide Web

Subset covered

Web site (shop, encyclopedia, social networking)

Relevant objects

Store inventory, shopping baskets, payment data, wiki page contents, video stream data

Constraints

“When stock of item is below 10, that item has an order immediately notice”

MINI WORLDS

Mini World Example: Movie Script

Real world

Cinematography, movies, films

Subset covered

Movie scripts (story, setting, roles, scenes)

Relevant objects

Chapters, scenes, actors, characters, locations, character (co-)occurrence, dialogue, …

Relationships

Character *is played by* actor, scene *is part of* chapter, character *occurs in* scene, …

Contraints

"If an actor impersonates more than one character, these characters may not meet"

MINI WORLDS

Mini World Example: LEGO™ Sets, Bricks, Mini Figures

Real world
LEGO toys

Subset covered
Catalog of available LEGO sets (or: models) and their contents

Relevant objects
Categories (“space”, “city”, …), sets, individual bricks, mini figures, colors

Relationships
Set *contains* bricks, brick *is available in* color, brick_1 *is equivalent to* brick_2

Significant attributes
Names, product IDs, quantity, 3D size (measured in studs), weight, image, …

Contraints
“If a set contains a piece, details for the piece must be available either in the brick or mini figure listing”, “No two pieces share the same product ID”

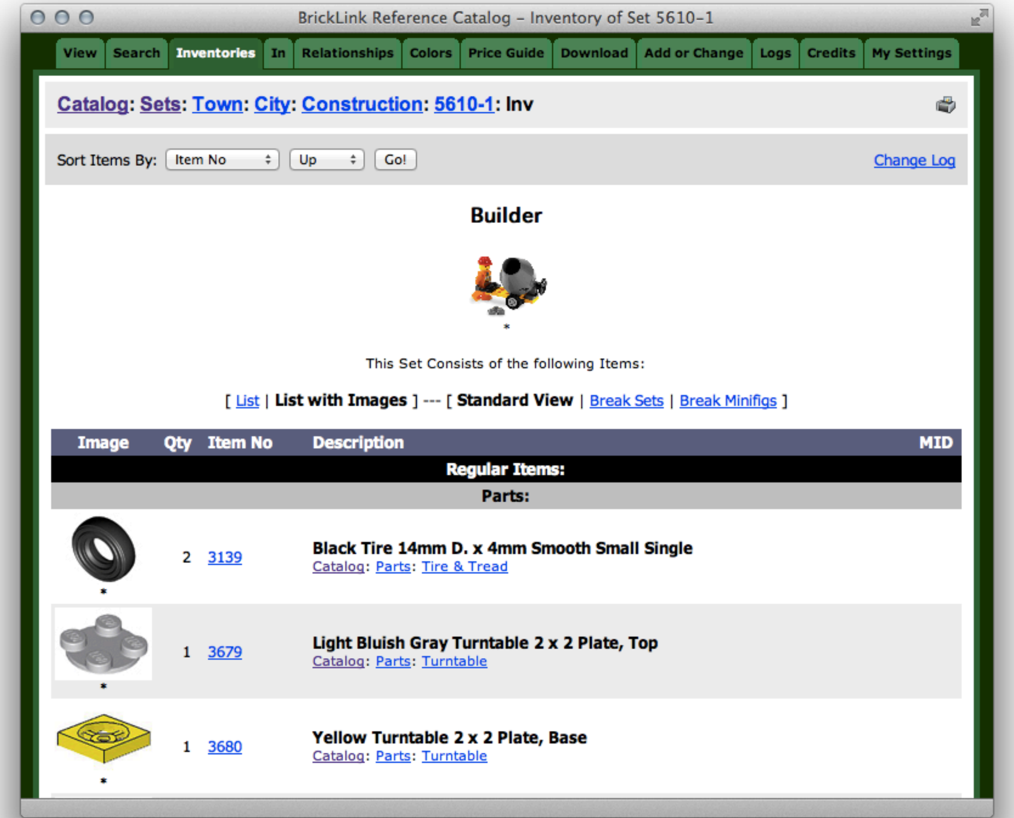
LEGO SET 5610

- A sample object of this particular mini world:

LEGO Set 5610 “Builder”, Category Town (City, Construction)

BRICKLINK

- Web site *BrickLink* hosts a database of the LEGO sets mini world:



Inventory of Set 5610-1 `http://www.bricklink.com/catalogItemInv.asp?S=5610-1`

DATA MODELS AND DATA LANGUAGES

Data Model
A data model defines a limited toolbox of constructs (or **types**) that can be used to represent the objects, attributes, and relationships of a mini world.

Data Language
Once the types are fixed, this also largely prescribes the **operations** we can perform with data of these types.

- Database systems are designed to effectively and efficiently support **a single data model and language** (we will see that support for “foreign” data models often feels awkward)

DATA MODEL: TEXT

Types

Text (strings of characters) in a particular encoding (e.g., Unicode/UTF-8).
Typically, newline characters '\n' are used to break the text into lines.

Besides the line-breaking convention, the text data model imposes no further structure on the data (→ **unstructured** data model).

Operations

1. **Iterate** over the lines of a given text (e.g. text contained in a file)
2. On each such line, use **pattern matching** to **extract** individual/groups of characters

DATA MODEL: TEXT

Example: GenBank (DNA Sequence Database)

- GenBank entry for *Saccharomyces cerevisiae* (Baker’s Yeast)

```
LOCUS       SCU49845     5028 bp    DNA             PLN       21-JUN-1999
DEFINITION  Saccharomyces cerevisiae TCP1-beta gene, partial cds, and Axl2p
            (AXL2) and Rev7p (REV7) genes, complete cds.
ACCESSION   U49845
VERSION     U49845.1  GI:1293613
KEYWORDS    .
SOURCE      Saccharomyces cerevisiae (baker's yeast)
  ORGANISM  Saccharomyces cerevisiae
            Eukaryota; Fungi; Ascomycota; Saccharomycotina; Saccharomycetes;
            Saccharomycetales; Saccharomycetaceae; Saccharomyces.
[...]
```

DATA MODEL: TEXT

- GenBank entry for *Saccharomyces cerevisiae* (Baker’s Yeast) [cont’d]

```
[...]
FEATURES             Location/Qualifiers
     source          1..5028
                     /organism="Saccharomyces cerevisiae"
                     /db_xref="taxon:4932"
                     /chromosome="IX"
                     /map="9"
     CDS             <1..206
                     /codon_start=3
                     /product="TCP1-beta"
                     /protein_id="AAA98665.1"
                     /db_xref="GI:1293614"
                     /translation="SSIYNGISTSGLDLNNGTIADMRQLGIVESYKLKRAVVSSASEA
                     AEVLLRVDNIIRARPRTANRQHM"
     gene            687..3158
                     /gene="AXL2"
[...]
```

DATA MODEL: TEXT

- GenBank entry for *Saccharomyces cerevisiae* (Baker’s Yeast) [cont’d]

```
[...]
ORIGIN
        1 gatcctccat atacaacggt atctccacct caggtttaga tctcaacaac ggaaccattg
       61 ccgacatgag acagttaggt atcgtcgaga gttacaagct aaaacgagca gtagtcagct
      121 ctgcatctga agccgctgaa gttctactaa gggtggataa catcatccgt gcaagaccaa
      181 gaaccgccaa tagacaacat atgtaacata tttaggatat acctcgaaaa taataaaccg
      241 ccacactgtc attattataa ttagaaacag aacgcaaaaa ttatccacta tataattcaa
[...]
//
```

- Aims for readability by humans *and* machines. Formatting conventions are obeyed to facilitate the construction of **parsers** for GenBank entries:[2]

```
/key=value
```

DATA MODEL: TEXT

Example: LEGO Set 5610 (BrickLink)

- Represent catalog information about LEGO Set 5610 (“Builder”) along with a detailed listing of the set contents (bricks, minifigures).
- This text file format primarily aims for **human readability.** The listing of the contents follows line-based formatting conventions that provide **hooks for parsing.**

```
LEGO™ Set "Builder" (set no 5610-1)
Category: Town (City, Construction)

Contains 20 pieces: 19 bricks, 1 minifigure

5610-1 Builder is a City impulse set released in 2008. It contains a construction
worker with a rolling cement mixer, along with 3 dark grey studs that resemble
mortar or concrete. When the mixer is pushed, the drum turns. The drum can also
tilt side-to-side, but not enough to dump the studs.
[...]
```

DATA MODEL: TEXT

- Catalog information for LEGO Set 5610 [cont'd, here: listing of set contents]

```
[...]
Brick#  Color/Weight  Name

1x 6157     Black/1.12g    Plate, Modified 2 x 2 with Wheels Holder Wide
2x 3139     Black/0.4g    Tire 14mm D. x 4mm Smooth Small Single
1x 3839b    Black/0.61g    Plate, Modified 1 x 2 with Handles - Flat Ends, [...]
1x 30663    Black/0.4g    Vehicle, Steering Wheel Small, 2 Studs Diameter
1x 6222     Dark Bluish Gray/3.57g    Brick, Round 4 x 4 with Holes
[...]

Minifig#  Weight  Name

1x cty052    3.27g    Construction Worker - Orange Zipper, Safety Stripes, Orange
[...]
```

DATA MODEL: TEXT

- Sample problem (or **query**):
  What is the overall weight (in grams) of LEGO Set 5610?

- Possible plan of attack:

1. **Iterate** over the lines of the catalog entry
2. Use **pattern matching** to identify lines of the form (␣ = space)

```
<quantity>x  …  /<weight>g …
<quantity>x  …  ␣<weight>g …
```

3. **Extract** quantity and weight in each such line
4. *Multiply* quantity and weight and *aggregate* (i.e. sum up) as needed

DATA MODEL: TEXT

The unstructured text data model provides poor support for queries even of this simple kind. One option: rely on UNIX' text processing tools like `grep`, `sed`, and `awk` to implement the plan.

`sed` (stream editor)

Operates over `'\n'`-separated lines of text, can filter lines based on **regular expressions**, can modify and then print selected lines. Example:

```
sed -E -e 's/regular expression/modification/p; …'
```

Good match for the text data model:
`sed` implicitly **iterates** over the lines (of its standard input), **pattern matches** and can **extract** select portions of matched lines.

DATA MODEL: TEXT

awk (Aho, Kernighan, Weinberger)

Iterate over `'\n'`-separated lines of text, apply rules of the following form to each line:

```
pattern { action }
```

- If *pattern* **matches**, *action* can **extract** the fields `$1`, `$2`, … of the matched line and perform (simple) computation.
  Specific patterns: `/<regular expression>/`, `//`, `BEGIN`, `END`.
- What constitutes a *field* is determined by field separator string `FS`

DATA MODEL: TEXT

- sed and awk script to compute the weight of LEGO Set 5610:

```
#! /bin/sh
# Compute the overall weight of all pieces in LEGO set 5610-1.

# Notes:
# - assumes one piece per line and input of the form:
#                     <quantity>x ... <weight>g ...
#   (everything else is considered noise and skipped over)
# - sed command `p': print pattern space, then process next line
#                `d': delete pattern space, next line

sed -E -e 's/^([0-9]+)x.+[ /]([0-9.]+)g.*$/\1 \2/p; d' |

awk '
  BEGIN { sum = 0        }
  //    { sum += $1 * $2 }
  END   { print sum      }
'
```

DATA MODEL: TEXT

- Sample problem (or **query**):
  Extract a subsequence, specificed by a location (from–to), from the DNA origin sequence of a GenBank entry
- Possible plan of attack:

1. **Iterate** over the lines of the GenBank entry
2. Use **pattern matching** to identify the start and end of the DNA sequence (`ORIGIN` and `//`, respectively)
3. Only when inside a sequence, use **pattern matching** to identify lines of the form

```
‹offset› …  ‹amino acids› ‹amino acids› ‹amino acids› …
```

4. **Extract** ‹amino acids› fields, *aggregate* (here: concatenate) the extracted fields
5. (*Cut* the requested subsequence from the concatenated result)

DATA MODEL: TEXT

- awk script to extract a DNA subsequence from a GenBank entry:

```
#! /bin/sh

FROM=$1
TO=$2

awk '
  BEGIN                 { ORS = ""; dna = 0; seq = "" }
  /ORIGIN/              { dna = 1 }
  dna && /^ *[0-9]+/    { for (i = 2; i <= NF; i++)
                             seq = seq $i
                        }
  /\/\//                { dna = 0 }
  END                   { print seq }
' |
cut -c $FROM-$TO
```

DATA MODEL: NESTED ARRAYS AND DICTIONARIES

Types

Container types: arrays and key/value pair dictionaries (or: hashes, association lists) and **basic atomic types** (e.g. numbers, strings, Booleans). Containers may contain atomic values as well as other containers (**nesting**).

Nested containers provide a multitude of data structuring options. Data models of this kind are commonly referred as being **semi-structured.**

Operations

1. Index- or key-based **lookup** into containers
2. (Nested) **iteration** over and **filtering** of container contents
3. **Construction** of new containers
4. **Computation** over basic types (comparison, arithmetics, …), **aggregration**

DATA MODEL: NESTED ARRAYS AND DICTIONARIES

Recent and now widespread instance of this data model: **JSON** (*JavaScript Object Notation*), excerpt of the JavaScript language definition (notation for literal JavaScript objects).

Find a complete and compact one-page(!) JSON definition on `http://json.org`. JSON has become popular as an inter-application data exchange format.

A recent proposal for a data language for this data model: **JSONiq** (`http://jsoniq.org`), derived from and interoperable with XQuery (the language for the XML data model).

DATA MODEL: NESTED ARRAYS AND DICTIONARIES

- JSON syntax (*string* and *number* follow usual syntactic conventions):

```
value     ::= string
            | number
            | true | false
            | null
            | dict
            | array
dict      ::= { }
            | { members }
members   ::= pair
            | pair , members
pair      ::= string : value
array     ::= [ ]
            | [ elements ]
elements  ::= value
            | value , elements
```

DATA MODEL: NESTED ARRAYS AND DICTIONARIES

- Sample JSON value **construction** (JSONiq):

```
let $set5610 := { "set": "5610-1",
                  "pieces": [ { "brick": "6157",  "quantity": 1 },
                              { "brick": "3139",  "quantity": 2 },
                              { "brick": "3839b", "quantity": 1 } ] }
return …
```

- Key-based **lookup** into dictionary *d* (via *d*.*k*, "dot notation"), index-based **lookup** (via *a*[[*n*]]) into array *a* (read "⟿" as "*evaluates to*"):

```
$set5610.set                  ⟿   "5610-1"
$set5610.pieces[[2]]          ⟿   { "brick": "3139",  "quantity": 2 }
$set5610.pieces[[2]].brick    ⟿   "3139"
```

DATA MODEL: NESTED ARRAYS AND DICTIONARIES

```
let $xs := [ 1, 2, 3 ]
let $ys := { "one": "eins", "two": "zwei", "three": "drei" }
return …
```

- **Iterating** over container contents. Need to convert contents into a **sequence** of values first.
  Iteration yields a sequence again:

```
for $x in members($xs)
return $x                           ⟶     (1,  2,  3)

for $y in keys($ys)
return $y                           ⟶     ("one",  "two",  "three")

for $y in keys($ys)
return $ys.$y                       ⟶     ("eins",  "zwei",  "drei")
```

DATA MODEL: NESTED ARRAYS AND DICTIONARIES

```
let $xs := [ 1, 2, 3, 4, 5, 6 ]
return …
```

- **Grouping** and **aggregation** of containers (+ computation, construction):

```
for $x in members($xs)                           ( { "even": true,
group by $even := $x mod 2 = 0          ⤏           "nums": [2, 4, 6] },
return { "even": $even, "nums": $x }               { "even": false,
                                                     "nums": [1, 3, 5] } )

for $x in members($xs)                           ( { "even": true,
group by $even := $x mod 2 = 0          ⤏           "sum":  12 },
return { "even": $even, "sum": sum($x) }           { "even": false,
                                                     "sum":  9 } )
```

- Note: In the group by clause, variable $x is bound to individual members of $xs. In the return clause, $x is bound to the **array of all group members.**

DATA MODEL: NESTED ARRAYS AND DICTIONARIES

Example: LEGO Set 5610 (BrickLink data modelled as JSON)

```
{ "set":          "5610-1",
  "name":         "Builder",
  "category":     "town",
  "year":         2008,
  "pieces": [
    { "brick":     "6157",
      "quantity": 1,
      "extra":     false,
      "color":     "Black",
      "weight":    1.12,
      "name":      "Plate, Modified 2 x 2 with Wheels Holder Wide" },
     ⋮
    { "minifig":   "cty052",
      "quantity": 1,
      "extra":     false,
      "weight":    3.27,
      "name":      "Construction Worker - Orange Zipper, ... } ] }
```

DATA MODEL: NESTED ARRAYS AND DICTIONARIES

- Sample problem (or **query**):
  What is the overall weight (in grams) of LEGO Set 5610?

- Possible plan of attack (JSONiq):
  1. **Access** the JSON representation of LEGO Set 5610
  2. **Iterate** over the set's pieces array:
     1. Inside each piece, **lookup** the values for the quantity and weight keys
     2. **Multiply** quantity and weight
  3. **Aggregate** (sum) the multiplied weights

DATA MODEL: NESTED ARRAYS AND DICTIONARIES

Example: USGS Real-Time Earthquake Data


```
{ "type":"FeatureCollection",
  "metadata":{
    "generated":1381237557000,
    "url":"http://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/2.5_day.geojson",
    "title":"USGS Magnitude 2.5+ Earthquakes, Past Day",
    "status":200,
    "api":"1.0.11",
    "count":31
  },
  "features":[
    { "type":"Feature",
      "properties":{
        "mag":2.9,
        "place":"38km E of King Salmon, Alaska",
        "time":1381234105000,
         ⋮
      } } … ] }
```

DATA MODEL: NESTED ARRAYS AND DICTIONARIES

- Sample **query:**
  What was the magnitude of the worst earthquake on the northern hemisphere?

- Possible plan of attack (JSONiq):

1. **Access** the JSON representation of USGS earthquake data
2. **Iterate** over the data's features array of quakes:
   - **Filter** quakes to retain only those that affected the northern hemisphere (**lookup** geometry to check whether latitude > 0)
3. **Iterate** over the qualifying quakes:
   - **Lookup** mag (magnitude) among the quake's properties
4. **Aggregate** (max) the collected magnitudes

DATA MODEL: NESTED ARRAYS AND DICTIONARIES

- A *slight* variation of the original sample **query:**
  What was the magnitude ***and place*** *of the worst earthquake on the northern hemisphere?*
- In a nutshell: we need `argmax()` not `max()`
- We require one of many possible different plans of attack here:

1. Once we computed the maximum magnitude `$mag`, iterate over all quakes again to find those with magnitude `$mag`.
2. Remember that for $m \in S$ we have
   $$\max(S) = m \Leftrightarrow \{\ y \mid y \in S,\ y > m\ \} = \phi$$
   for any set S of comparable elements.
3. Order all quakes by descending magnitude, then pick the first in that order.

DATA MODEL: NESTED ARRAYS AND DICTIONARIES

- These plans of attack represent typical query formulation techniques:
  - The use of **quantifiers** (here: ∀ / `empty()`, but ∃ is as important)
  - The use of **nested iteration** and **correlation**
  - The use of **ordered** containers and **positional lookup**
- We will revisit all of these in this course.

Query Equivalence?

Are all of these queries equivalent? Will they return the same earthquake regardless of the current earthquake data?

(DATA MODEL: ORDERED TREES)

Replace the role of arrays and dictionaries by **ordered trees** and obtain **XML**, another semi-structured data model in *wide* use today.

- XML defines a textual representation for data trees whose leaves contain strings:

```
<a>
  <b><c>foo</c></b>
  <d>bar</d>
</a>
```

```
     a
    / \
   b   d
   |   |
   c "bar"
   |
 "foo"
```

- Data languages for XML (**XPath**, **XQuery**) navigate trees (descend to child nodes, collect all nodes in subtree, collect all nodes on path to the root, …)

DATA MODEL: TABULAR

Types

Container types: tables of rows (or: records, tuples), each row having the same number of fields. Fields contain values of **basic atomic types** (e.g. numbers, strings, Booleans) only. Inside a row, entries are identified either by name or position.

We essentially obtain a **flat, tabular** data model comprised of strictly rectangular data grids.

Operations

1. (Nested) **iteration** over the rows of a table
2. **Filter** the rows based on given criteria (or: predicates)
3. **Access** one or more fields of a row
4. **Computation** over field values (comparison, arithmetics, …)

DATA MODEL: TABULAR

Thanks to its restrictiveness, the tabular data model can have particularly simple textual representations. Quite common: **CSV** (*comma-separated values*).

CSV (here: tab-separated values)

1. Table ≡ file, row ≡ line. Rows are separated by a newline (␊). Fields in a row are separated by a tab (␉).
2. First line of file contains field names, lines 2, 3,… contain data rows
3. Field value syntax uses usual conventions, strings are enclosed in "..."
4. A missing field value is represented by the string \N

DATA MODEL: TABULAR

Example: USGS Earthquake Data in CSV Format

```
time                 latitude longitude depth  mag  place
2013-10-08T12:08:25  58.6193  -156.005  85.2   2.9  "38km E of King Salmon, …"
2013-10-08T11:45:19  31.6814  131.7314  21.34  4.8  "35km ENE of Nichinan, Japan"
2013-10-08T11:31:04  12.2779  -88.4332  48.22  4.3  "109km SSW of Chirilagua, …"
2013-10-08T10:48:22  36.5863  -98.0361  \N     3.2  "21km ENE of Helena, Oklahoma"
```

(excerpt only, beautified)

- Note the regular row-wise/column-wise organization of data.
- This particular variant of CSV format is also used by PostgreSQL when relational data is to be exported/imported.

DATA MODEL: TABULAR

There is **no** agreed-upon (let alone standardized) **data language for CSV.** To touch CSV data, in this course we will build our own data language based on Python.

To name the beast, let's call it **Python QL** or **PyQL** for short (*pick•le* /'*pikəl*/: messy situation).

- Needed: a Python 3 installation
- A supporting Python module DB1 is available on the course homepage
- All other PyQL operations and constructs are in fact regular Python operations

DATA MODEL: TABULAR

- Sample PyQL query: access all rows of table earthquake.csv. **Iterate** over table, bind each row to **row variable** quake, print each row:

```
from DB1 import Table

earthquakes = Table('earthquakes.csv')
for quake in earthquakes:
  print(quake)                    # ⚠ indentation indicates block structure
```

- Function Table(*f*) reads CSV file *f* and returns a Python iterator. Iteration (e.g. via for … in …) yields each row in the file in the form of Python dictionary.

DATA MODEL: TABULAR

- In PyQL, use standard Python constructs to access a field in a row or to construct new rows:

```
from DB1 import Table

earthquakes = Table('earthquakes.csv')

for quake in earthquakes:
   print({ 'place': quake['place'], 'mag': float(quake['mag']) })
```

  - Compare to JSONiq's key-based dictionary lookup and dictionary construction syntax
  - Conversion from string (to numeric) is explicit (`float()`, `int()`)

DATA MODEL: TABULAR

- PyQL syntax:

```
e    ::=  Table(csv)                          Table access
       |  for v in e: e                       iteration
       |  [ e for v in e … ]                  list comprehension
       |  if e: e else: e                     conditional
       |  print(e, e, …)                      output
       |  e[f]                                field access
       |  { f: e, …, f: e }                   row construction
       |  float(e)  |  int(e)  |  …           type conversion
       |  e + e  |  e  ==  e  |  …            arithmetics, comparison
       |  float  |  int  |  string  |  …      literal values
       |  v                                   variable reference
       |  v = e                               variable assignment
       |  e … e                               sequence of PyQL statements
csv  ::=  CSV file name
f    ::=  field name
v    ::=  variable name
```

DATA MODEL: TABULAR

List Comprehensions

- PyQL's **comprehensions** provide elegant and compact notation for iteration and filtering:

```
# Magnitude of worst earthquake on the northern hemisphere
from DB1 import Table

earthquakes = Table('earthquakes.csv')

magnitudes = [ float(quake['mag'])
                 for quake in earthquakes
                   if float(quake['latitude']) >= 0.0 ]

print(max(magnitudes))
```

- Compare to set comprehensions as they are common in mathematics:

$$\{ f(x) \mid x \in S, p(x) \}$$

DATA MODEL: TABULAR

- Above we have used a **list-oriented style** of query formulation (list comprehensions, `max()` list aggregate).
- Same query in **imperative style.** Now, the focus is on updating the state of `float` variable `mag`:

```
earthquakes = Table('earthquakes.csv')

mag = 0.0                                   # ⚠ variable initialization

for quake in earthquakes:
  if float(quake['latitude']) >= 0.0:
    if float(quake['mag']) > mag:
      mag = float(quake['mag'])             # ⚠ variable update

print(mag)
```

DATA MODEL: TABULAR

- Consider the slight variation of the sample **query** again:
  *What was the magnitude **and place** of the worst earthquake on the northern hemisphere?*

```
# Modify/extend PyQL code of last slide
…
```

- The imperative query style can be convenient. It does, however, effectively allow writing programs that are arbitrarily complex to evaluate (or never terminate at all).
- For this (and more good) reason, by design **data languages are considerably more restricted than general programming languages.**

DATA MODEL: TABULAR

- The tabular data model is **flat:** fields contain **atomic** values
- In the absence of nesting, how to represent complex structured information? Recall:

```
{ "set": "5610-1",
  "pieces": [ { "brick": "6157",    …, "quantity": 1 },   ⚠ "pieces": non-flat
              { "brick": "3139",    …, "quantity": 2 },
              { "minifig": "cty052", …, "quantity": 1 } ] }
```

- One option: **flatten out** the nested data, attach set identifier 5610-1 to each row:

```
[ { "set": "5610-1", "brick": "6157",    …, "quantity": 1 }
  { "set": "5610-1", "brick": "3139",    …, "quantity": 2 }
  { "set": "5610-1", "minifig": "cty052", …, "quantity": 1 } ]
```

DATA MODEL: TABULAR

- In this flat model, where to keep the brick/minifigure (or: piece) details?
- Consider the full LEGO set mini-world. Brick 6157 occurs in many sets:

```
  ⋮
{ "set": "5610-1",  "brick": "6157", ⚑, "quantity": 1 }
  ⋮
{ "set": "10048-1", "brick": "6157", ⚑, "quantity": 4 }
  ⋮
{ "set": "1029-1 ", "brick": "6157", ⚑, "quantity": 2 }
  ⋮
```

- Keeping details for brick 6157 here (⚑) would **replicate** data — wastes space and comes with the risk that the copies go out of sync over time.
- Such **redundancy** is almost always to be avoided!

DATA MODEL: TABULAR

- Option adopted by the tabular data model:
 1. Keep brick details in a separate CSV file (i.e., a **separate bricks table**)
 2. Use brick **identifiers** (e.g., 6157) to locate bricks in this new table:

```
⋮
{ "piece": "6157", "type": "B", "name": "Plate, Modified …", "weight": 1.12 }
⋮
```

- There is **no redundancy.** The different LEGO sets share a single copy of the brick details.
  - If brick details change, a **single row** in the bricks table is affected.

Note: Brick identifier 6157 must indeed be **present** as well as **unique** in the bricks table. This is a typical mini-world rule (or **constraint**).

DATA MODEL: TABULAR

- Data of the LEGO mini-world is now spread over three tables:

contains

set	piece	color	extra	quantity
…	c	…	…	…

bricks

piece	type	name	cat	weight	img	x	y	z
b	…	…	…	…	…	…	…	…

minifigs

piece	type	name	cat	weight	img
m	…	…	…	…	…

- The two predicates $c = b$ and $c = m$ identify related rows in the three tables.

DATA MODEL: TABULAR

- Sample **query:**
  What is the overall weight (in grams) of LEGO Set 5610?

- Plan of attack (recall: a set's piece is either a brick or a minifig):
1. **Iterate** over contains, **filter** rows for pieces *c* in LEGO Set 5610
   1. **Iterate** over bricks, **filter** rows for *the* brick *b* that corresponds to *c*
      - **Multiply** quantity and weight of *b*, **aggregate** (sum up)
   2. **Iterate** over minifigs, **filter** rows for *the* minifig *m* that corresponds to *c*
      - **Multiply** quantity and weight of *m*, **aggregate** (sum up)
2. Return aggregate (sum) of steps 1.1 and 1.2

DATA MODEL: TABULAR

```
# Compute the overall weight of all pieces in LEGO set 5610-1 ("Builder")

from DB1 import Table

contains = Table('contains.csv')
bricks   = Table('bricks.csv')
minifigs = Table('minifigs.csv')

weight = 0

for c in contains:
  if c['set'] == '5610-1':
    for b in bricks:
      if c['piece'] == b['piece']:
        weight = weight + int(c['quantity']) * float(b['weight'])
    for m in minifigs:
      if c['piece'] == m['piece']:
        weight = weight + int(c['quantity']) * float(m['weight'])

print(weight)
```

1. A database system benchmark devised by the Transaction Processing Council, `http://www.tpc.org/tpch` ↩
2. Here and in the following, <var> denotes a *meta-syntactic variable* of name var (i.e. a placeholder). ↩