INTRODUCTION TO RELATIONAL DATABASE SYSTEMS

DATENBANKSYSTEME 1 (INF 3131)

Torsten Grust
Universität Tübingen

Summer 2023

TYPED / UNTYPED DATA

Valid Data?

1. In the **text data model**, *any* `'\n'`-separated file of (UTF8-encoded) text represents valid data.
   (All we can hope for is that the different lines contain the same patterns of text.)
2. For **JSON**, *any* value derivable from the non-terminal symbol ‹value› is valid data.
3. For the **tabular data model**, *any* CSV file in which each row (including the header row) contains the same number of `'\t'`-separated fields is valid data.

Valid, But Loose

The above are rather loose syntactic restrictions. It is still (too) easy to craft valid data that makes queries trip. Query execution may halt or (even worse) silently return non-sensical results.

TYPED / UNTYPED DATA

Example: Mangle the LEGO Set 5610 Text Data

- Slight edit in minifigure weight data (3.27g → 0.3.27g):

```
…
Minifig#  Weight  Name

1x cty052    0.3.27g    Construction Worker …     ⚠
```

  - Line ⚠ still matches the regular expression `^([0-9]+)x.+[ /]([0-9.]+)g.*$`
  - Rules of `awk`'s "best effort" string-to-number conversion apply. Overall result (weight of set) misleading.
- This is a so-called **silent error**. Particularly hard to detect in a sea of data.

TYPED / UNTYPED DATA

- JSON arrays and dictionaries may have **heterogeneous contents** (any *value* is like any other):

```
let $xs := [1, 2, "three", 4, 5]     ⚠
return
    for $x in members($xs)
    return $x + 1
```

  - JSONiq query will fail at **query runtime** (i.e., rather late; NB: query syntax is okay):

```
Code: [XPTY0004]
Message:  "+": operation not possible with parameters of type "xs:string" and
"xs:integer"
```

TYPED / UNTYPED DATA

- In JSONiq, failing index or key lookups **silently** evaluate to () (empty sequence):

```
{ one: 1 }.one            ⟶        1
{ one: 1 }.two            ⟶        ()
[1, 2, 3][[4]]            ⟶        ()
```

- Expressions involving () **propagate** (), which makes debugging of JSONiq queries hard:

```
() + 1                                              ⟶     ()
{ one: 1 }.()                                       ⟶     ()
[1, 2, 3][()]                                       ⟶     ()
for $x in members([1,2,3]) where () return $x       ⟶     ()
```

TYPED / UNTYPED DATA

- In PyQL and tabular CSV data, we rely on certain fields to contain data of a certain **type** (e.g., number or Boolean).
- If these assumptions are wrong, explicit conversions like `int()` or `float()` may fail at **query runtime** (after many seconds/minutes/hours, if we are unlucky).
- "Safe conversions" can help here but may introduce a noticable runtime overhead:

```
# convert string s to float (if that fails, return default float x instead)
def safe_float(s, x):
  try:
    f = float(s)
  except ValueError:
    f = x

  return f
```

TYPED DATA IN THE RELATIONAL DATA MODEL

Untyped Data Models

The text, JSON, and the tabular data models do *not* enforce values (container or atomic) to be of specific types. These data models are thus referred to as being **untyped**.

- The **relational data model** may be understood as a **typed** variant of the tabular data model:

1. There is only one container type: **table** (or: multisets) **of rows**,
2. all rows are of the **same row type** which is declared when the table is created,
3. A row type **consists** of a sequence of **atomic types**.

TYPED DATA IN THE RELATIONAL DATA MODEL

- **SQL:** Creating a table t and declaring its **row type:**

```
CREATE TABLE t (                -- t: table name and type name
    col₁ ty₁,
       ⋮
    colₙ tyₙ
);
```

  - Creates table t of n columns, col_i column name, *all* values in that column of atomic type ty_i.
  - Also implicitly declares row type $t = (col_1\ ty_1,\ \ldots,\ col_n\ ty_n)$.

TYPED DATA IN THE RELATIONAL DATA MODEL

- **Import** (or: load) correctly typed data from CSV file into a relational table (here: Relational Database Management System PostgreSQL):

```
\COPY t(col₁, …, colₙ) FROM csv;
```

  - Table *t* must have been created prior to the `\COPY` command.
  - CSV file *csv* must *not* contain a header row: all rows are interpreted as data rows.
  - Order and number of fields in the CSV file must match the columns col_1, …, col_n.
  - The **field values in the CSV file must have the declared types** ty_1, …, ty_n.

TYPED DATA IN THE RELATIONAL DATA MODEL

- Queries in the relational data model may rely on all rows having a known row type
  → **No expensive runtime checks and conversions** with safety measures (like `safe_float()`).
- Since types are known when the query is analyzed/compiled, the system may **specialize the executable query code** to (only) work for those types.
  (Even if we only save few milliseconds per row, the savings add up if we process many rows — as is typical.)
- Type-based errors in queries are detected at **query compile time** and thus early.
  Once a query has been compiled successfully, **no type errors will occur at runtime.**

DECLARATIVITY

- Once data instances (text files, JSON values, CSV files, relational tables) are of significant size and queries are of moderate complexity and beyond, **query performance** becomes a concern.
- Performance gains may be achievable in different ways:
  1. Carefully **exploit properties of the data** (mini-world constraints) to simplify your queries.
  2. Find entirely **different querying strategies** (beyond naive nested iteration, say) to process the data.
- ⚠️ Both options involve query modifications or whole query rewrites — **query equivalence** may *not* be sacrificed.
- Let us see whether such performance gains (here: **reduction of elapsed query time**) are achievable for the *weight of LEGO Set 5610* query.

DECLARATIVITY

- **Baseline:** Original PyQL program for the *weight of LEGO Set 5610* query:

```
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)
```

DECLARATIVITY

- As usual, let $|S|$ denote the *cardinality* of set S (i.e., $|t|$ denotes the number of rows in CSV table t)
- Measure the work performed by the PyQL programs in terms of the **numbers of rows** processed.
  - The work per row (field access, arithmetics, comparison) is assumed to be essentially constant, i.e. in $O(1)$.
  - Let $pieces(5610)$ denote the number of different piece types in LEGO Set 5610 (a one-line PyQL comprehension can compute function $pieces(s)$ for any given LEGO set s).

Work performed by baseline PyQL query:

$$|contains| + pieces(5610) \times (|bricks| + |minifigs|)$$

DECLARATIVITY

Optimzation #1 (Exploit Properties of the Data)

- Observations:
  1. Field piece is **unique** in table bricks: no two bricks share the same piece identifier. Once we have found a brick, we will not find another. (The same is true for table minifigs.)
  2. A given piece identifier *p* is **either** found in table bricks **or** in table minifigs. The piece field values of both tables are disjoint. (Question: What about the third option: *p* not present in any of the two tables?)
- **NB:** Both observations are consequences of **rules** (constraints) in the LEGO sets mini-world.
- We can use this to optimize the baseline PyQL query. Recall: **query equivalence** must be preserved.

DECLARATIVITY

Estimation of work performed by PyQL query after optimization #1:

$$|\mathrm{contains}| + \mathit{pieces}(5610) \times (\ b \times \tfrac{1}{2} \times |\mathrm{bricks}| + (1-b) \times (|\mathrm{bricks}| + \tfrac{1}{2} \times |\mathrm{minifigs}|))$$

where $0 \leqslant b \leqslant 1$ is the fraction of bricks in a set (b = 0.95 for LEGO Set 5610).

- In this estimate,
  - $\frac{1}{2} \times |\mathrm{bricks}|$ describes the average effort to find a piece in table bricks,
  - $|\mathrm{bricks}| + \frac{1}{2} \times |\mathrm{minifigs}|$ describes the effort to miss a piece in bricks and then find it in minifigs.

DECLARATIVITY

Optimization #2 (Change of Query Strategy)

- Idea: Proceed in two phases:
  - Phase 1: Iterate over contains and build a **small temporary data structure** that maps pieces to their quantity in LEGO Set 5610. (Only include pieces in Set 5610!)
  - Phase 2: Iterate over bricks and minifigs **once** and check for each brick / minifig in the data structure. If present, update overall weight.
- The auxiliary data structure essentially implements a function piece ↦ quantity. That function is partial (domain contains pieces of Set 5610 only).
  - Can think of this function as a two-column table itself:

piece	quantity
p_1	n_1
p_2	n_2
$\vdots$	$\vdots$

DECLARATIVITY

Estimation of work performed by PyQL query after optimization #2:

$$|contains| + |bricks| + |minifigs|$$

- It is essential that lookups in the temporary data structure (a variant of an **index**) are low-cost:
  - Size of index in $O(pieces(5610)) = O(1)$
  - $\Rightarrow$ We can expect $O(1)$ lookup cost

DECLARATIVITY

- Database systems are designed to **shield** users from performance considerations like we have just studied:
  - Exploitation of data properties (e.g. uniquenuess, disjointness) is automatic.
  - Decisions about the construction/removal of auxiliary data structures are automatic (or performed by database system administrators, DBAs).
  - Exploitation of such auxiliary data structures is automatic.

Declarativity ("What, not how")
Query languages are declarative. Users specify **what** results a query shall return, **not how** results are to be computed.

- **SQL is declarative:** Queries *cannot* refer to auxiliary data structures that are present for efficiency reasons. (There is no such query construct.)

DATA INDEPENDENCE

- In the LEGO sets mini-world, a *piece* is either a brick or a minifigure. All of our *Weight of LEGO Set 5610* queries were **explicitly** reflecting this fact. Cumbersome.
- Consider two options:

1. In the PyQL queries, define a **temporary pieces list** that combines the common features of bricks and minifigures. Then query pieces instead.
2. Construct a new **pieces table (= persistent file on disk)** from the existing bricks and minifigs tables. In PyQL queries, refer to the new pieces table just like any other table.

- Both options sound reasonable but require different pre-processing steps. However, once pre-processing is complete, **queries using pieces need not know the difference.**

DATA INDEPENDENCE

Pre-Processing for Option #1

- Perform pre-processing inside the PyQL program:-

```
 ⋮
bricks   = Table('bricks.csv')
minifigs = Table('minifigs.csv')

# A piece is either a brick or a minifig: build the union (make sure to
# only retain the features common for both piece types)
pieces = [ m for m in minifigs ]     \
         +                           \
         [ { 'piece':  b['piece'],
             'type':   b['type'],
             'name':   b['name'],
             'cat':    b['cat'],
             'weight': b['weight'],
             'img':    b['img']    } for b in bricks ]
 ⋮
```

DATA INDEPENDENCE

Pre-Processing for Option #2

- Prepare a new table pieces.csv on disk. Use UNIX shell utitlies to create the required CSV file:

```
> cut -f1-6 bricks.csv | tail +2 | cat minifigs.csv - > pieces.csv
```

  - cut -f1-6: project on the first six (tab-separated) fields of each line
  - tail +2: copy input to output but omit the first line (column name header)
  - cat: concatenate inputs and copy to output (- denotes standard input)
- Inside the PyQL program:

```
 ⋮
pieces = Table('pieces.csv')
 ⋮
```

DATA INDEPENDENCE

- **Regardless of option chosen,** the rest of the PyQL query reads:

```
weight = 0

for c in contains:
  if c['set'] == '5610-1':
    for p in pieces:
      if c['piece'] == p['piece']:
        weight = weight + int(c['quantity']) * float(p['weight'])
        break

print(weight)
```

Data Independence

The formulation of queries shall not depend on the details of physical data storage and layout. Customized **views of the data** are to be considered first-class data sources (just like tables).

PERSISTENCE

- For a typical program, the output is a function of the (current user) input only:
  - Today:

    ```
    5 ➜ factorial ➜ 120
    ```

  - Tomorrow:

    ```
    5 ➜ factorial ➜ 120
    ```

- Program `factorial` does not consult data besides the user input and values/objects created by itself.

PERSISTENCE

- For database-based programs (or: queries), the output is a function of the input **and the current state** of the mini-world (e.g., table contents):
  - Today:

```
                                        '5610-1'   ⎫
                                                   ⎬ ➔ weight of LEGO set ➔ 22.46
contains.csv, bricks.csv, minifigs.csv …           ⎭
```

  - Tomorrow:

```
                                        '5610-1'   ⎫
                                                   ⎬ ➔ weight of LEGO set ➔ 25.30
contains.csv, bricks.csv, minifigs.csv …           ⎭
```

PERSISTENCE

Persistence
Database systems maintain the state of data (tables) between invocations. Such *persistent* information outlives the process it was created by. In particular, the system persists the data across power outages and operating system reboots.

- Until now, we were using the **OS file system** to store persistent CSV files.
  - This is where most database systems hold their persistent data, too.
    ⚠️ Plain file management via operations `open()`, `close()`, `read()`, `write()` is not sufficient. (Why?)
  - Other storage options (e.g., raw disk devices) are common as well.