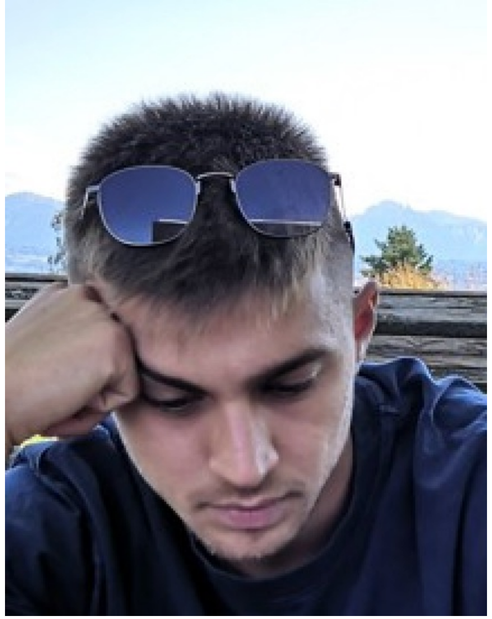
User-Defined Functions in Relational Databases: Challenges and Promising Solutions based on YeSQL

Yannis Ioannidis
University of Athens & Athena Research Center

Team

(past and present)

Yannis Foufoulas



Kostas Chasialis

Alkis Simitsis

Lefteris Stamatogiannakis

Motivation

Since the beginning of (database) time and still in modern times

general applications and specialized data science pipelines

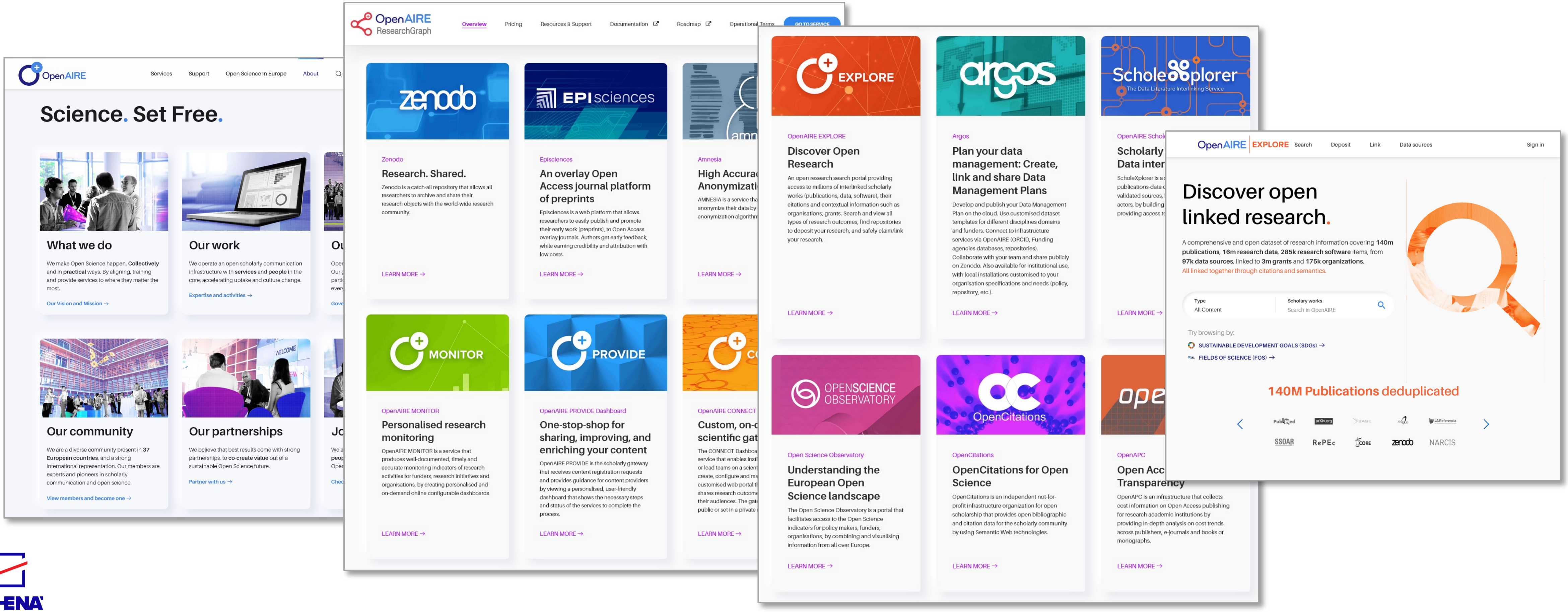
involve complex processing on data stored in

databases and other diverse data sources

Use case: OpenAIRE

https://www.openaire.eu

- OpenAIRE Research Graph uses a semantic graph database to aggregate a collection of research data properties (metadata, links) for funders, organizations, researchers, communities and publishers
- Auto-discovery, collective research, advanced analysis, trustable data and indexing, interoperability
- 65 European institutions / 1,000+ data providers / 42M services last year / 140M pubs deduplicated



OpenAIRE production query (snippet abridged)

- Text mine publication's fulltext to extract links to projects from various funders

```
select docid, id, fundingclass1, grantid,
    jdict('documentId', docid, 'projectId', id, 'confidenceLevel’,
    sqroot(min(1.49,confidence)/1.5), 'textsnippet', context) as C1, context
  from (
       select docid,id,confidence, docid, id,  fundingclass1, grantid, context
       from (
          select 0.8 as confidence, docid, id, fundingclass1, grantid, context from (select docid, regexpr("(\d+)",middle) as middle,
               comprspaces ( j2s(prev1,prev2,prev3,prev4,prev5,prev6,prev7,prev8,prev9,prev10,prev11,prev12,prev13,"<<<",middle,">>>",next)) as context
          from (
              setschema 'docid,prev1,prev2,prev3,prev4,prev5,prev6,prev7,prev8,prev9,prev10,prev11,prev12,prev13,middle,next’
                   select c1, textwindow(lower(c2),-13,0,1, '\b\d{5,6}\b') from pubs where c2 is not null
           )
           where CAST(regexpr("(\d+)",middle) AS INT)>70000
       ), grants
       where fundingclass1="AKA" and  (regexprmatches("[\b\s]academy of finland[\b\s]", context) or  regexprmatches("[\b\s]finnish (?:(?:programme for
          )?cent(?:re|er)s? of excellence|national research council|funding agency|research program)[\b\s]", context)
         or regexprmatches("[\b\s]finnish academy[\b\s]", context))
         and grantid=middle
  )
  group by docid,id
);dueio5k[5ktfmcl65eqey wb
```

The complete query can be found here:
https://github.com/openaire/iis/blob/34895ac6d9eec71050537224100d944576a38e23/iis-wf/iis-wf-referenceextraction/src/main/resources/eu/dnetlib/iis/wf/referenceextraction/project/main_sqlite/oozie_app/lib/scripts/projects.sql

Motivation

- Many programming language tools to assist developers design pipelines
- **But:** **C**omplicated ecosystem, unscalable processing

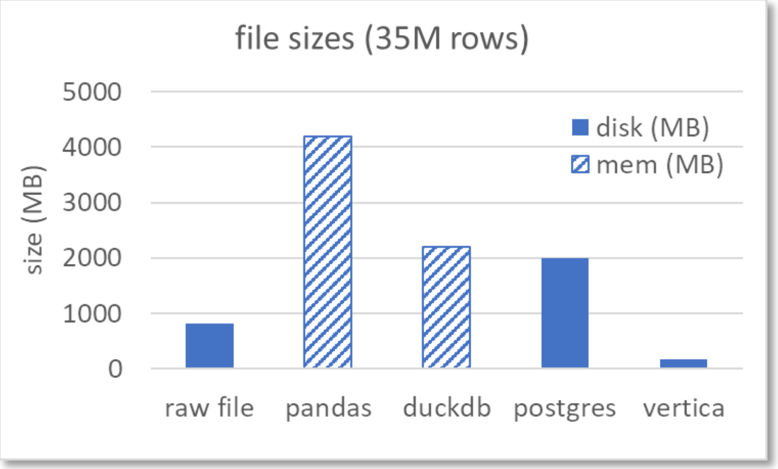
- Relational DBMSs offer efficient large data processing
- **But:** SQL has limited expressive power

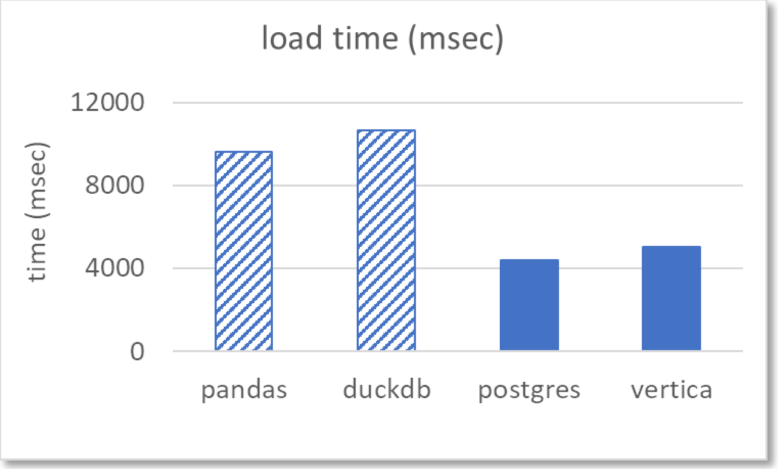
	Expressivity	Debug	Opt/tion	Execution
Pandas API	High	Easy	No	Slow
SQL	Limited	Hard	Yes	Fast
SQL + UDFs	High	Easy	**??**	**??**

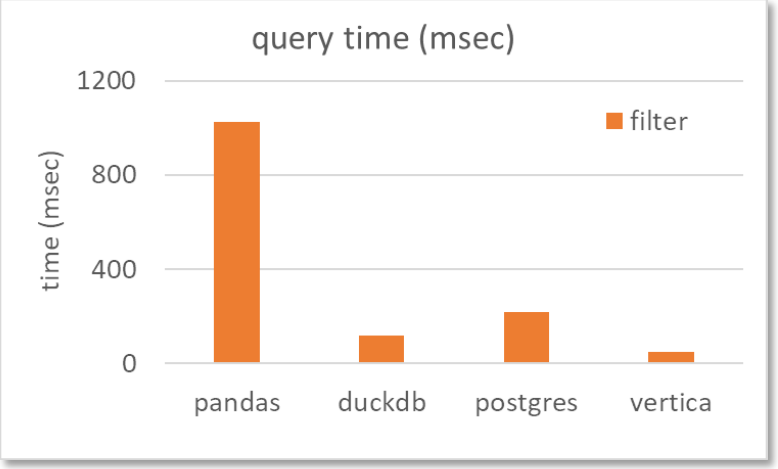
- UDFs in SQL merge relational and programming syntax and semantics
- **But:** Impedance mismatch between declarative (SQL) and procedural (e.g., Python) operation

Example Task: Find #records for year 2018

- Census csv data: 35M records, 818MB disk size
- Exec options
  - Pandas – in-memory
  - DuckDB – embedded DB, in-memory
  - PostgreSQL – server DB, disk, row-store
  - Vertica – server DB, disk, column-store
- Load
  - Pandas: df = pd.read_csv(“census.csv")
  - SQL: create table / copy table
- Query
  - Pandas: df[df.Year==2018].count()
  - SQL: select count(year) from census where year = 2018



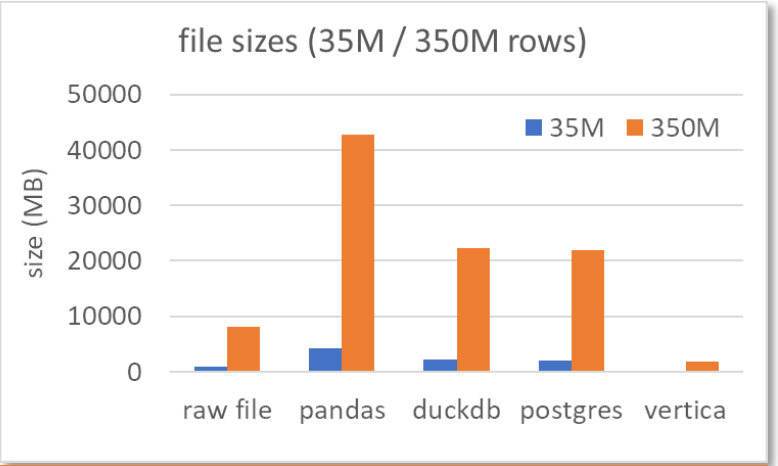




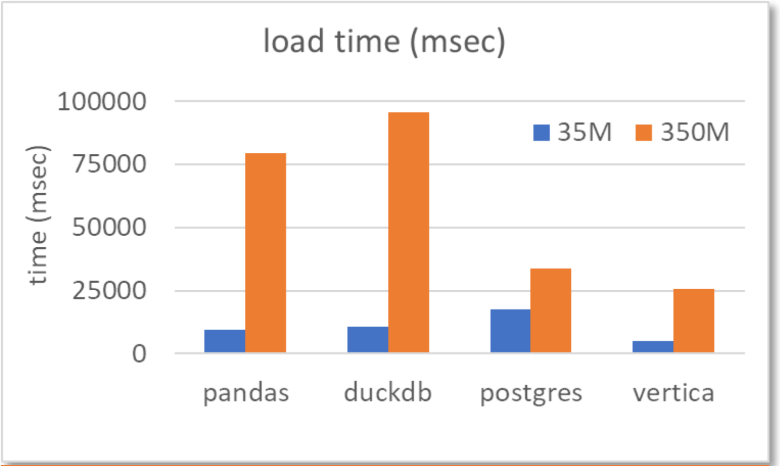
ATHENA

Example Task: Find #records for year 2018

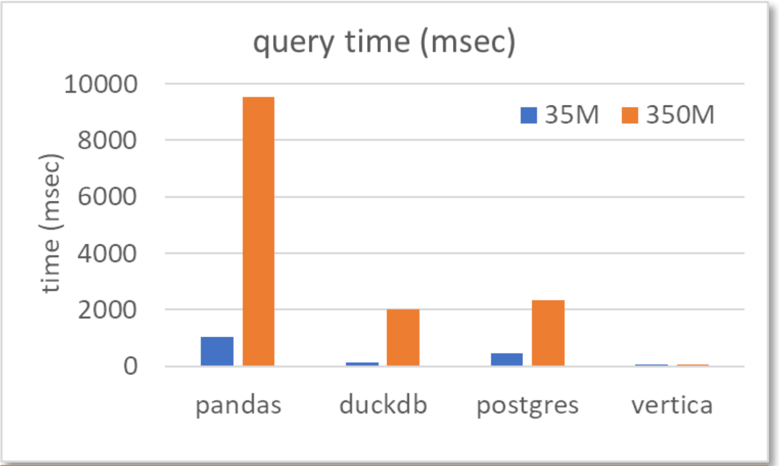
- Census csv data: ~~35M~~ 350M records, ~~818MB~~ 8GB disk size
- Exec options
  - Pandas – in-memory
  - DuckDB – embedded DB, in-memory
  - PostgreSQL – server DB, disk, row-store
  - Vertica – server DB, disk, column-store
- Load
  - Pandas: df = pd.read_csv(“census.csv")
  - SQL: create table / copy table
- Query
  - Pandas: df[df.Year==2018].count()
  - SQL: select count(year) from census where year = 2018



lower storage



faster load



faster query

Example Task: Find #records for year 2018

- So, let's move the "functions" count and filter inside the data engine
  - select count(year) from census where year = 2018

```
CREATE AGGREGATE pycnt (integer) (
   INITCOND = 0,
   STYPE = integer,
   SFUNC = _pycnt
);

CREATE FUNCTION _pycnt (s
  RETURNS integer
AS $$
  global s
  if b is not None:
    s = s + 1
  return s
$$ LANGUAGE plpython3u;
```

count in PostgreSQL

```
CREATE LIBRARY pylib AS 'pylib.py' LANGUAGE 'Python';
CREATE TRANSFORM FUNCTION flt2 AS
   LANGUAGE 'Python' NAME 'flt2_factory' LIBRARY pylib fenced;
class flt2(vertica_sdk.TransformFunction):
   def processPartition(self, server_interface, input, output):
      while True:
         if input.getInt(0) == input.getInt(1):
            output.setInt(0,col)
            output.next()
         if not input.next():
            break
class flt2_factory(vertica_sdk.TransformFunctionFactory):
   def getPrototype(self, server_interface, arg_types, return_type):
      .....
      return_type.addInt()
   def getReturnType(self, server_interface, arg_types, return_type):
      return_type.addColumn(arg_types.getColumnType(0), "result")
   def createTransformFunction(cls, server_interface):
      return flt2()
```

filter in Vertica

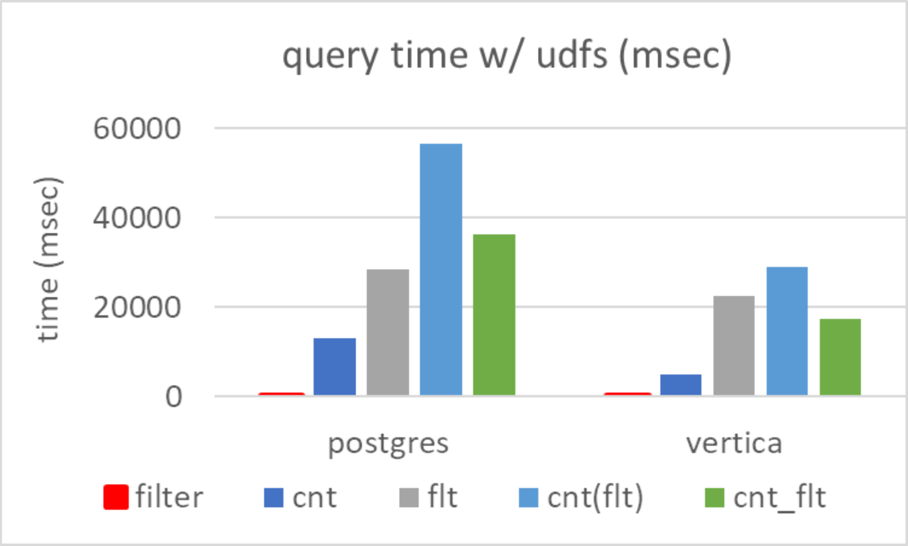
filter: select **count**(year) from census where **year = 2018**;

cnt: select **pycnt(year)** from census where **year = 2018**;

flt: select **count**(**pyflt(year,2018)**) from census;

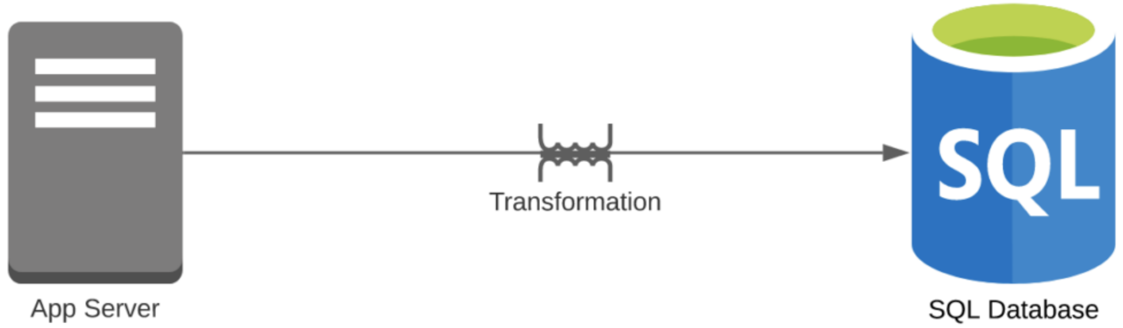
cnt(flt): select **pycnt**(**pyflt(year,2018)**) from census;

cnt_flt: select **pycntflt(year,2018)** from census;



Challenges: Performance

- Impedance mismatch between declarative (SQL) and procedural (e.g., Python) operation



- Frequent context switches
- Data conversions and copies
- Excessive function calls
- Inefficient compilation
- Limited query optimization
- Long UDF pipelines
- Materialization of intermediate results (some engines)

Challenges: Fragmented space

- UDFs in many languages
  - C/C++, Java, R, Matlab, Scala, Python, …
- Case in point: Python UDFs
  - Most popular amongst growing communities of data science and data analytics
  - Fragmented space: too many libraries, frameworks
  - Performance challenges due to conversions between Python and C/C++ (DBMS implementation choice)

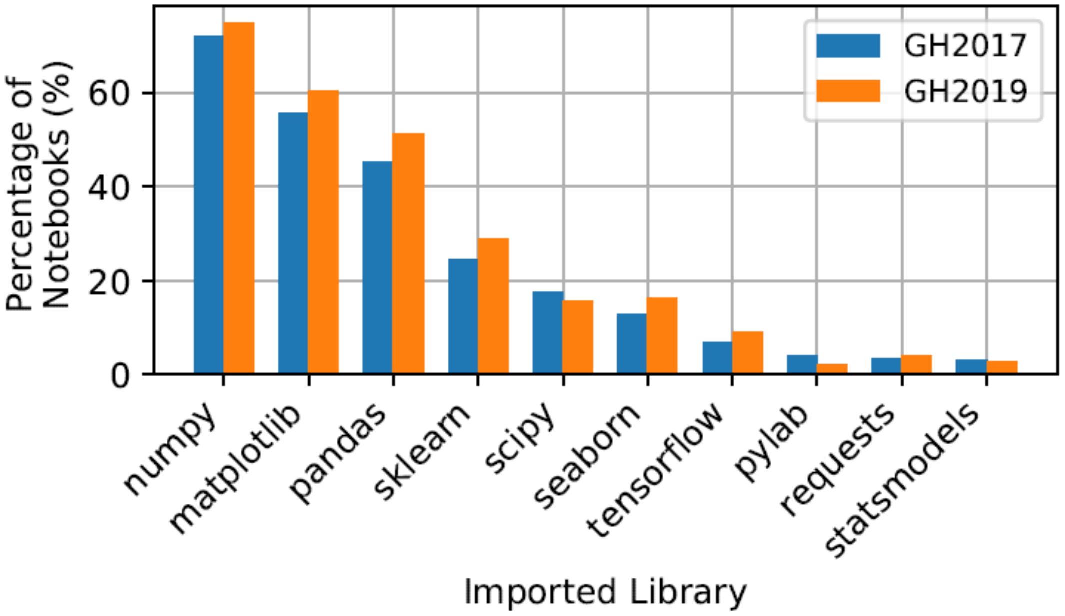


Figure 2. Top-10 used libraries across notebooks.

[src: Psallidas et al. arXiv 2019]

State of the art

State of the art: UDF translation

- Python compilers
    - Generate bytecode for JIT compilation
    - Examples: GraalPython, Truffle, Numba, Pyston, …
    - Several limitations in practice re library support, performance, etc.

- Python transpilers
    - Translate Python into other languages, typically C/C++
    - Examples: Cython, Nuitka, …
    - Limitations include a priori data type declaration, slow compilation time, etc.

A taxonomy of solutions

- Translation of UDFs into SQL
  - **Pros:** Holistic UDF + Query optimization
  - **Cons:** Library/framework specific, very hard to support a full-fledged language

- Translation of UDFs (and rel ops) into common Intermediate Representation (IR)
  - **Pros**: Advanced fusion, loop fusion, traditional query optimization applies to some extent
  - **Cons:** Library/framework specific, very hard to support a full-fledged language

- Embed UDFs Into data engine
  - **Pros**: Existing arbitrary code, optimizations such as fusion/loop-fusion, vectorization, function inlining, performance boosters such as JIT/LLVM compilation
  - **Cons**: Query optimization is trickier (still doable!); requirs analysis of user code, cost prediction, etc.

A taxonomy of solutions

- UDF optimization
  - Parallelization
  - Vectorization
  - Function inlining
  - In/Out-process
  - Tracing/method JIT
- Execution model
  - Tuple/Vector/Operator
  - Column/Row based
- Query optimization
  - Reordering
  - Fusion
  - Rules, Cost
- Usability / Expressivity
  - Engine/Lib specific
  - Static/Dynamic

UDF solutions			UDF optimization							Execution model					Qry optimization				Usability & expr.			
			parallelization	vectorization	func inlining	in-process	out-process	tracing JIT	method JIT	tuple-at-a-time	vector-at-a-time	oper-at-a-time	column-based	row-based	reordering	fusion	rules	cost model	engine spec	lib spec	static	dynamic
UDF-to-SQL	T-SQL - PL/SQL	Froid [34]	x			x				x				x	x	x	x	x			x	
UDF-to-SQL	T-SQL - PL/SQL	Aggify [15]	x			x				x				x	x	x	x	x			x	
UDF-to-SQL	T-SQL - PL/SQL	PLSQL/AWAY [8]	x			x				x					x	x	x	x	x		x	
UDF-to-SQL	T-SQL - PL/SQL	Duta et al. [7]	x			x				x						x		x			x	
UDF-to-SQL	Python UDFs	Blacher et al. [1]			x					x	x	x	x	x	x		x	x		x		x
UDF-to-SQL	Python UDFs	Snakes on a plan [11]			x					x	x	x	x	x	x		x	x		x		x
UDF-to-SQL	Python UDFs	Grizzly [17]			x					x	x	x	x	x	x		x	x		x	x	
UDF-to-SQL	Python UDFs	AIDA [6]	x		x							x	x				x					
UDF-to-SQL	Python UDFs	Schüle et al. [38]		x	x				x	x				x				x	x	x	x	
UDF-to-SQL	Scala UDFs	CLIS [43]	x			x										x						
UDF-to-IR	Python UDFs	Weld [28, 29]	x	x		x						x	x			x			x	x	x	
UDF-to-IR	Python UDFs	Raven [23]	x			x				x				x	x	x	x	x	x	x	x	
UDF-to-IR	Scala - Java UDFs	Flare [10]		x	x				x	x			x					x	x	x	x	
UDF-to-IR	Scala - Java UDFs	Boehm et al. [2]		x	x						x		x			x	x	x	x	x	x	
UDF-to-IR	Scala - Java UDFs	Emani et al. [9]			x					x				x				x	x	x	x	
UDF-to-IR	Matlab UDFs	HorsePower [3]	x		x	x						x	x			x	x		x	x		
UDF-to-IR	Multiple languages	Babelfish [14]	x			x			x	x						x	x		x			
UDF-to-IR	Multiple languages	BabbleFlow [21, 22]	x		x	x				x		x		x	x	x	x	x	x		x	
In-engine UDF	Data engines	PostgreSQL [30]	x				x			x				x				x	x		x	
In-engine UDF	Data engines	MonetDB/NumPy [33]	x	x		x						x	x		x		x		x		x	
In-engine UDF	Data engines	Schüle et al. [37]	x			x				x				x	x	x	x	x	x		x	
In-engine UDF	Java UDFs	Rosenfeld et al. [36]	x			x				x				x	x		x	x	x		x	
In-engine UDF	Python UDFs	Kläbe et al. [25]	x	x			x				x		x				x				x	
In-engine UDF	Python UDFs	YeSQL [13]	x		x	x	x	x		x	x	x	x	x	x	x	x	x			x	x
In-engine UDF	Python UDFs	Tuplex [42]	x		x	x		x		x				x		x	x		x			x
In-engine UDF	C++ UDFs	UDOs [39]	x			x				x				x							x	
In-engine UDF	Multiple languages	Tupleware [5]	x		x	x		x		x				x	x	x	x	x	x		x	

[Y. Foufoulas, A. Simitsis – tutorials @IEEE ICDE'23, @PVLDB'23]

ATHENA

A taxonomy of solutions

In-engine UDF	System																				
Data engines	PostgreSQL [30]	x				x			x				x				x	x		x	
Data engines	MonetDB/NumPy [33]	x	x		x						x	x		x		x		x		x	
Data engines	Schüle et al. [37]	x			x				x				x	x	x	x	x	x		x	
Java UDFs	Rosenfeld et al. [36]	x			x				x				x	x		x	x	x		x	
Python UDFs	Kläbe et al. [25]	x	x			x				x		x				x				x	
Python UDFs	YeSQL [13]	x	x	x	x	x	x		x	x	x	x	x	x	x	x	x			x	x
Python UDFs	Tuplex [42]	x		x	x		x		x				x		x	x		x			x
C++ UDFs	UDOs [39]	x			x				x				x							x	
Multiple languages	Tupleware [5]	x		x	x		x		x				x	x	x	x	x	x		x	

[Y. Foufoulas, A. Simitsis – tutorials @IEEE ICDE'23, @PVLDB'23]

Standard SQL interpretation (non-JIT)
[interpreted plan w/ UDFs]

JIT-compiled UDFs
[interpreted plan w/ JIT-cmp'd UDFs]

JIT-compiled queries
[JIT-cmp'd plan w/ UDFs]

Commercial/production data engines

UDF Characteristic	PostgreSQL	MonetDB /NumPy	SQLite	Duck DB	SQL Server	Vertica
Polymorphic output	+/-	-	+	-	+/-	+/-
Dynamic typing	-	-	+	-	-	-
Statefulness	+/-	-	+	+	-	-
Performance boosting (e.g., JIT)	-	-	-	-	-	-
Vectorization	-	+	-	+	-	+
Optimization	-	-	-	-	+/-	-

Caveat: The data shown in the table is based on our understanding of publicly available documentation

ATHENA

The YeSQL approach

YeSQL name

You extend SQL

Enough with NoSQL!

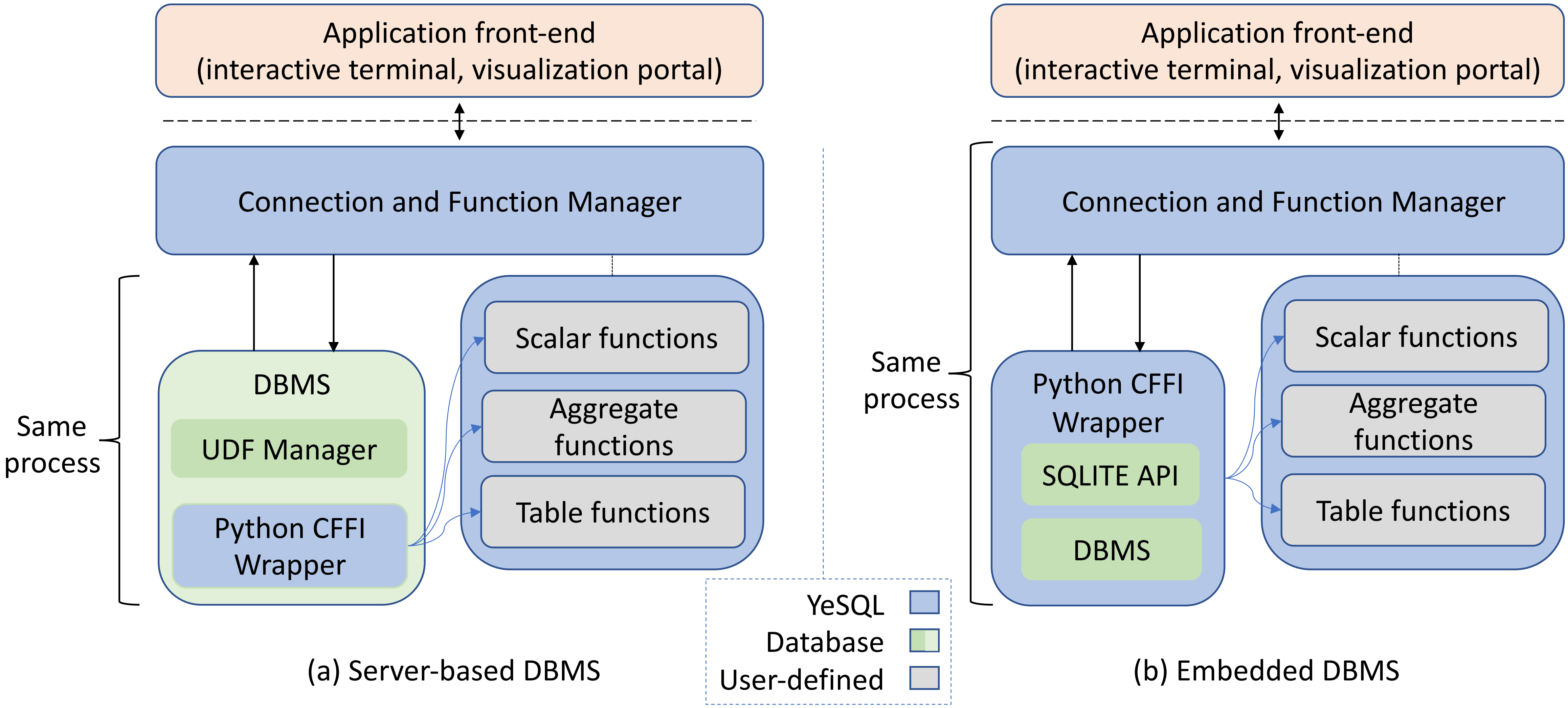
Pronounced ‘yes’ ‘q’ ‘l’

YeSQL characteristics

- SQL extension to more usable, more expressive, and more performant Python UDFs
- Expressiveness
  - Stateful UDFs
  - Dynamically typed UDFs
  - Scalar, aggregate, and table UDFs
- Performance
  - JIT-compilation (with seamless DBMS/UDF exchange)
  - Parallelization
  - Statefulness
  - Fusion for optimization
- Support for both server-based and embedded data engines
- Usability
  - Parametric polymorphic UDFs
  - Functional syntax for UDFs

ATHENA

YeSQL architecture



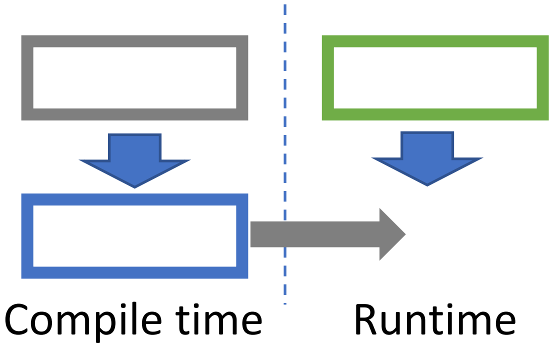
ATHENA

Color Coding of Code Frame

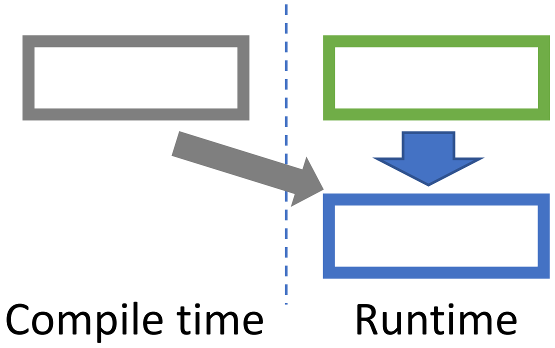
UDF definition

Code automatically produced by YeSQL
(e.g., python wrapper or rewritten SQL statements)

YeSQL query submitted



(a) Static



(b) Dynamic

UDF Registration

Python UDF

```
@scalar_udf
def remove_punc(text: str): -> str
   return " ".join(["".join([ch for ch in word if ch.isalnum()]) for word in text.split(' ')]) if text is not None else None
```

what the UDF dev implements

PyPy / CFFI code [db agnostic] @ Wrapper

```
/* remove_punc.h */
extern void remove_punc_pypy_wrap(char **input, int count, char **result);

import udf_funcs as udf

@ffi.def_extern()
def remove_punc_pypy_wrap(input, count, result):
   for i in range(count):
      _tmp = ffi.string(input[i]).decode()
      _res = udf.remove_punc(_tmp).encode()
      result[i] = lib.strdup(ffi.from_buffer(memoryview(_res))
```

MonetDB C-UDF [db specific] @ Function Manager

```
CREATE OR REPLACE FUNCTION remove_punc(input STRING)
RETURNS STRING
LANGUAGE C
{
#pragma CFLAGS -I<path>
#pragma LDFLAGS -L<path> -l<lib>
#include "remove_punc.h"
  result->initialize(result, input.count);
  remove_punc_pypy_wrap(input.data, input.count, result->data);
};
```

YeSQL – UDF registration

Python UDF

```
@scalar_udf
def remove_punc(text: str): -> str
   return " ".join(["".join([ch for ch in word if ch.isalnum()]) for word in text.split(' ')]) if text is not None else None
```

what the UDF dev implements

PyPy / CFFI code [db agnostic] @ Wrapper

```
/* remove_punc.h  */
extern void remove_punc_pypy_wrap(char **input, int count, char **result);

import udf_funcs as udf

@ffi.def_extern()
def remove_punc_pypy_wrap(input, count, result):
   for i in range(count):
      _tmp = ffi.string(input[i]).decode()
      _res = udf.remove_punc(_tmp).encode()
      result[i] = lib.strdup(ffi.from_buffer(memoryview(_res))
```

C part   Python part   CFFI part

MonetDB C-UDF [db specific] @ Function Manager

```
CREATE OR REPLACE FUNCTION remove_punc(input STRING)
RETURNS STRING
LANGUAGE C
{
#pragma CFLAGS -I<path>
#pragma LDFLAGS -L<path> -l<lib>
#include "remove_punc.h"
  result->initialize(result, input.count);
  remove_punc_pypy_wrap(input.data, input.count, result->data);
};
```

YeSQL Usability and Expressiveness

ATHENA

YeSQL – real world query

- Example Query: Text-mine a corpus of publications to identify which publications have been funded by NSF and identify the respective NSF project identifier (7-digit string)

```
select var(`pos`,(select toregex(term) from positives));

select texts.id, projects.id
  from (select id, textwindow(keywords(text),10, 1, 5, '\d{7}')
           from (sample 100 file 'publications.json') as input_pubs)
         as texts, projects
  where texts.middle = projects.grantid and
         regexprmatches($pos, lower(texts.prev||” “||texts.next));
```

(polymorphic) table UDF   scalar UDF   aggregate UDF

UDF fusion: “textwindow(keywords(text), …)” and “sample .. file” / sample(file(publications.json))

Syntax inversion: “sample 100 file ‘publications.json’”
instead of “select * from sample (100 selection * from publications.json)”

ATHENA

Expressiveness and Usability

- Expressiveness
  - Dynamically typed UDFs
  - Stateful UDFs
  - Scalar, aggregate, and table UDFs

- Usability
  - Parametric polymorphic UDFs
  - Functional syntax for UDFs

Dynamically typed UDFs

```
def add(arg1, arg2):
   return arg1+arg2
```

what the UDF dev implements

```
SELECT CAST(add(3, 2) AS int);
```

```
SELECT CAST(add('Hello ', 'World') AS string);
```

Stateful UDFs
e.g., via a global dictionary

```
globaldict = {}
def var(arg1, arg2 = None):
  if arg2 is not None:
     globaldict[arg1] = arg2
     return True
  else:
     return globaldict[arg1]
```

```
SELECT var('a', 'HELLO WORLD');
>> 1
SELECT lower(var('a'));
>> hello world
```

Functional syntax

```
SELECT * FROM json_parse('select authors from publications');
SELECT * FROM file('data.csv');
```

```
json_parse select authors from publications;
file 'data.csv';
```

Parametric polymorphic

authors:

```
{ "name": "Peter" }
```

```
{ "name": "Peter",
  "citedby": 100 }
```

Usability: User Study

- Setup
  - 380 undergrads were asked to develop two algorithms:
    - Document similarity with TF-IDF
    - Document classification using a preexisting training set with weighted terms
  - Using (a) YeSQL and (b) Python and SQL but without UDFs

- Results
  - 328 (86.3%) completed successfully the task w/ YeSQL
  - 165 (43.4%) scored an excellent grade w/ YeSQL

Significantly higher rates than w/o YeSQL

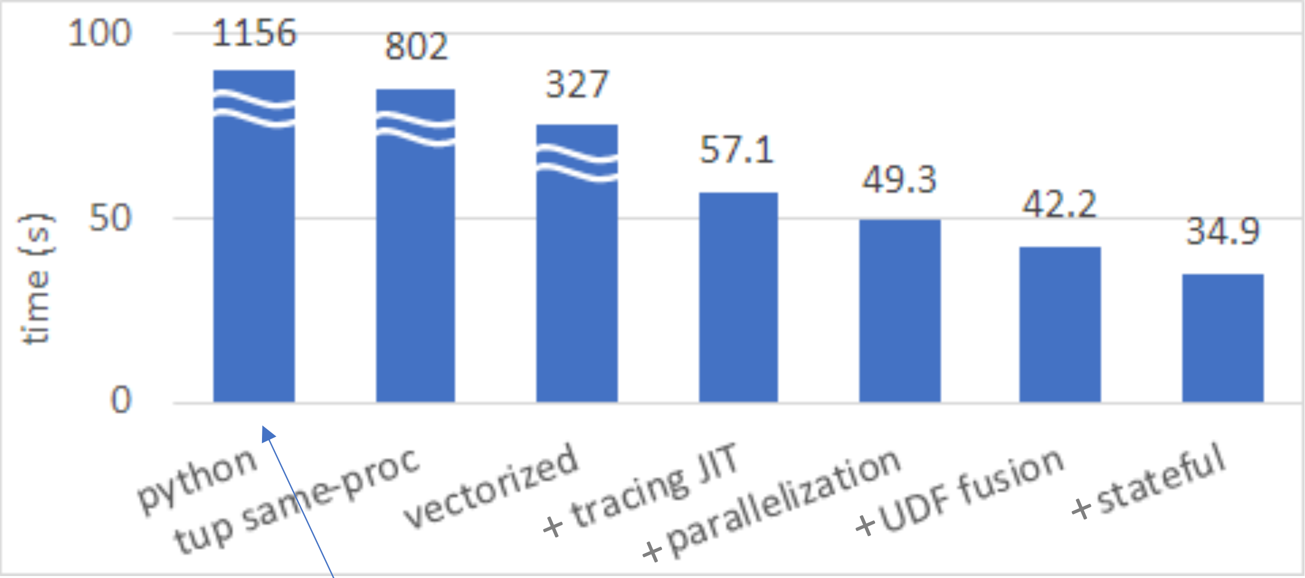
ATHENA

YeSQL Performance Enhancements

Performance: Where does the time gain come from?

- Query with four UDFs

```
select udf1(postal_code), udf2(facts_and_features), udf3(udf4(url))
from zillow;
```

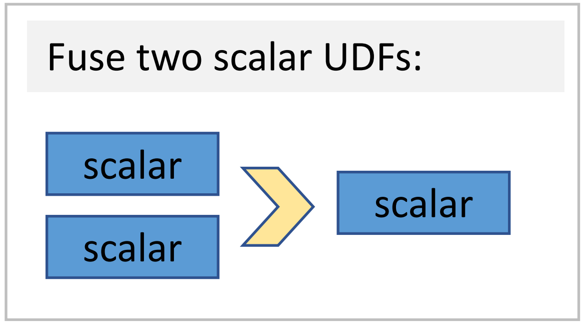


typically, data scientists start here
with PostgreSQL or PySpark

Steps:
1. Spawned CPython process as tuple-at-a-time
2. In-process tuple-at-a-time execution
3. Vectorized execution using embedded NumPy
4. Tracing JIT-compilation with PyPy
5. Employ parallelism with multi-threaded execution
6. Fusion on JIT
7. Stateful UDF execution

Performance: UDF fusion

- Fusable UDFs
  - The second UDF’s input data is the same as the first UDF’s output
  - The argument data types are available in the query plan
- Fusion happens at the CFFI level
  - A wrapper function is created just-in-time and pipelines the UDFs
  - CFFI conversions are eliminated
  - Longer sequences of instructions exploit tracing JIT
- Example: fuse two scalar UDFs




```
def len(input):
   return len(input)

def add(input):
   return add+1
```

```
SELECT add(len(c1))
from table;
```

```
@ffi.def_extern()
def fused_add_len(input, count, result):
   for i in range(count):
      val = ffi.string(input[i])
      result[i] = add(len(val))
   return 1
```

```
CREATE OR REPLACE FUNCTION
add_len(input STRING) RETURNS INT
LANGUAGE C
{
 #include ”udfs.h"
  result->initialize(result, input.count);
  fused_add_len(input.data, input.count,
result->data);
};
```

```
SELECT add_len(c1) from table;
```

Performance: UDF and relational operator fusion

(1) udf → udf ⇨ fused_udf

(2) udf → rel ⇨ fused_udf or rel → udf ⇨ fused_udf

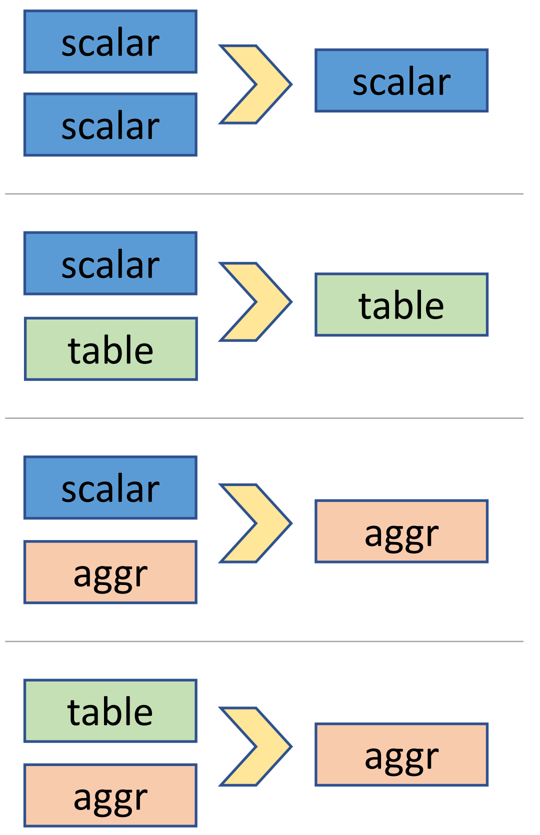
- Remove optimization barriers caused by black-box treatment of UDFs
- Result of fused_rel also possible [Blacher et al.@CIDR22]

(3) udf → rel → udf ↬ rel → udf → udf ⇨ rel → fused_udf

- Additional fusion and optimization opportunities

[→ , ⇨, ↬ : data pipeline, operator fusion, and operator reordering, respectively

Cases of (1):
udf → udf ⇨ fused_udf



Experimental Performance Evaluation

Evaluation

Implementation – YeSQL codebase

- ~66K lines of Python and C++
- ˜18.5K lines for code definitions of 150+ Python UDFs

Setup

- Intel Core (Ivy Bridge E) i7-4930K, 3.40GHz, 6cores/12CPUs
- 64GB mem
- Ubuntu 20.04

Baselines

- **Tuplex**
- **MonetDB** (v.11.41.11)
- **PostgreSQL** (v.12.9)
- **dbX**
- **Pandas** (v.1.3.5)
- **Spark** (PySpark, v.2.4.7)

Used

- Cython (v.0.29.25) / Numba (v.0.54.1) / Nuitka (v.0.6.19.1)
- Compiled_UDF_engine (TU Ilmenau, 2022)
- PyPY (v.7.3.6 with GCC 7.3.1)
- CFFI (v.1.14.6)
- CPython (v.3.8.10)
- SQLite (v.3.31.11)

Datasets

Zillow*
- listings from Boston, MA
- #columns: 10
- size/#rows: 1GB/5.6M, 5GB/28.6M, 10GB/56M

Flights*
- airport and airline data
- #columns: 110
- size/#rows: 1.6GB/5M, 3.2GB/10M, 6.4GB/20M

Text mining
- plain texts of research publications
- pipeline:
  - mining UDFs: tokenization, stopwords removal, pattern extraction
  - join with a db table
  - pattern matching: remove false positives

(*: obtained by the Tuplex repo, as they used in the SIGMOD’21 paper)

ATHENA

Evaluation: Queries

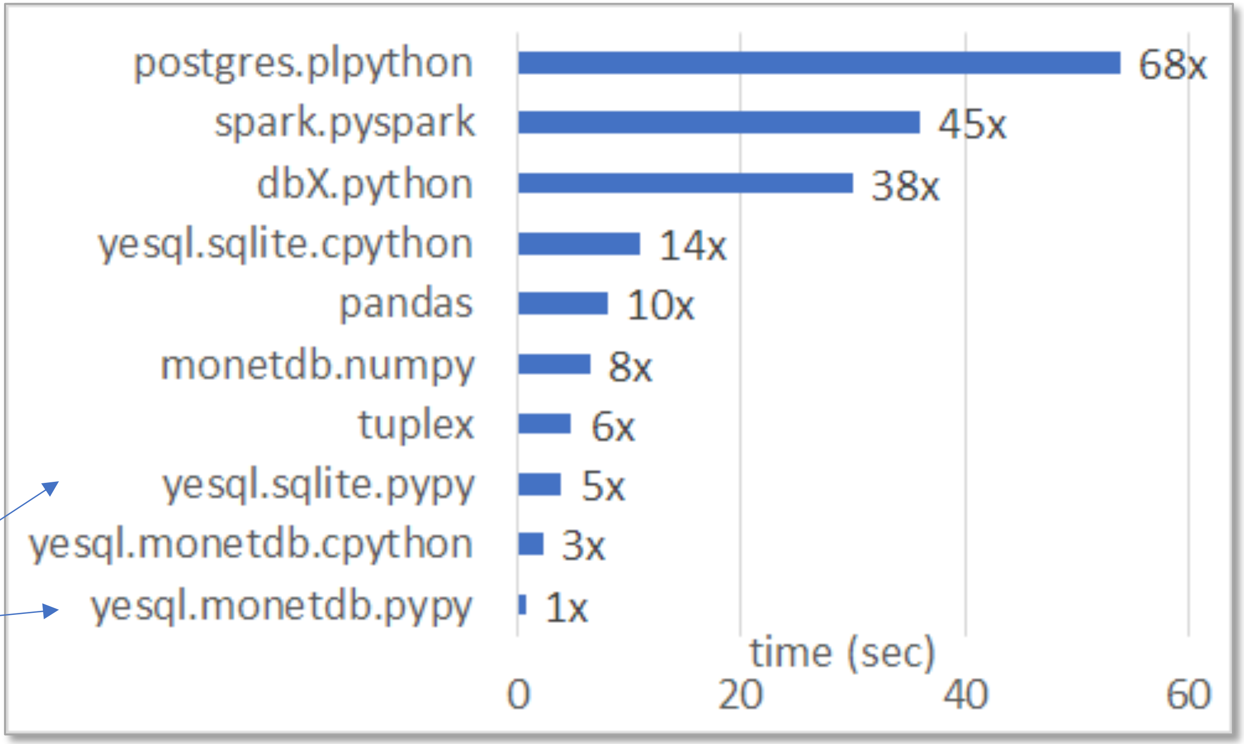
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13
#udfs	6	14	4	5	9	9	1	1	4	1	4	32	5
#scalar	4	9	4	4	8	6	0	0	2	0	3	26	3
#aggr	1	5	0	1	0	2	0	0	1	0	1	8	1
#table	1	0	0	0	1	1	1	1	1	1	0	0	1
nest-dpth	4	3	4	5	5	5	1	1	2	1	3	4	3
#fused	4	9	3	4	8	8	0	0	2	0	2	23	4
fused-optim	< 1 msec												
code-gen	< 1 msec												
compile	661	728	718	602	387	534	585	515	577	543	508	597	521
runtime	8075	61977	18087	15639	139079	109532	23092	6156	16161	5495	1349	2573	52183

YeSQL overheads

Evaluation: YeSQL vs. Data engines

- Comparison w/ SOTA tracing JIT system Tuplex @SIGMOD'21 and other popular choices

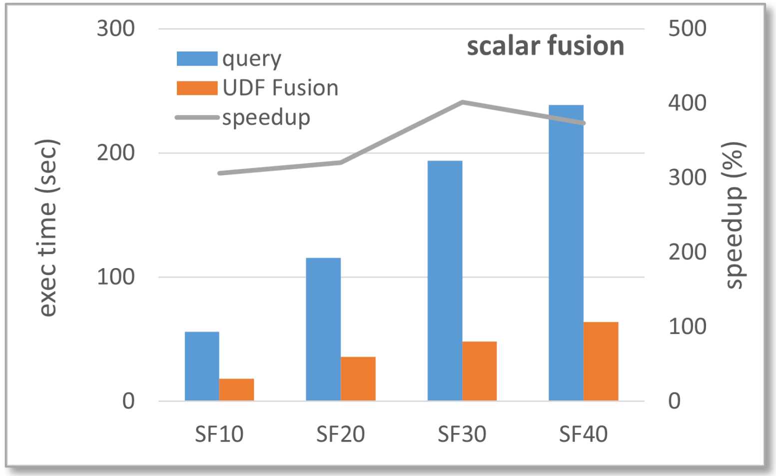
JIT-compiled YeSQL

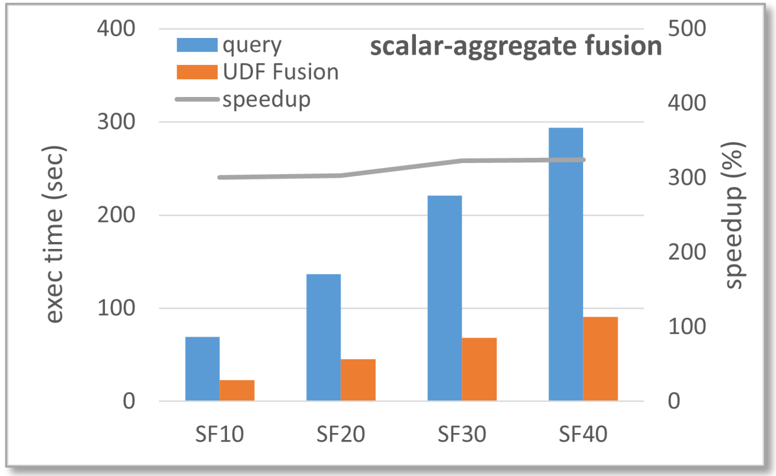


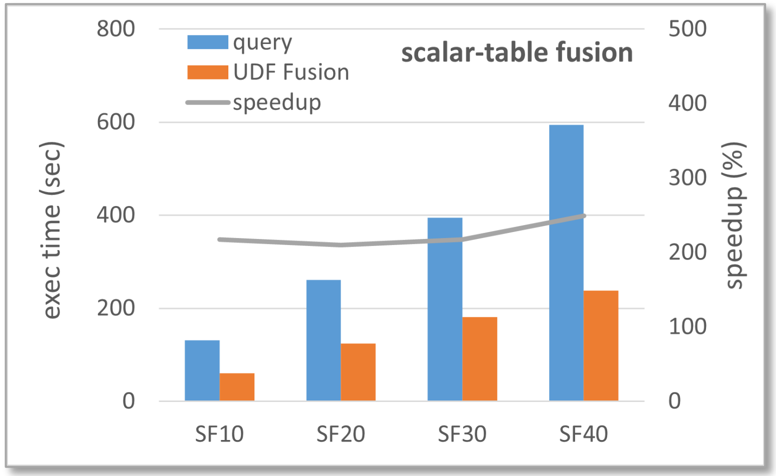
execution time [less is better]

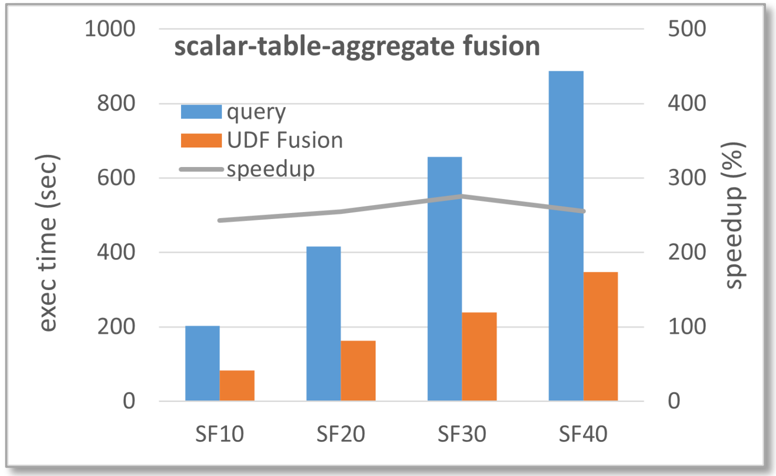
Evaluation: Fusion of UDF types

- TPC-H queries on the orders table
- 3x speedups









ATHENA

Conclusions and open problems

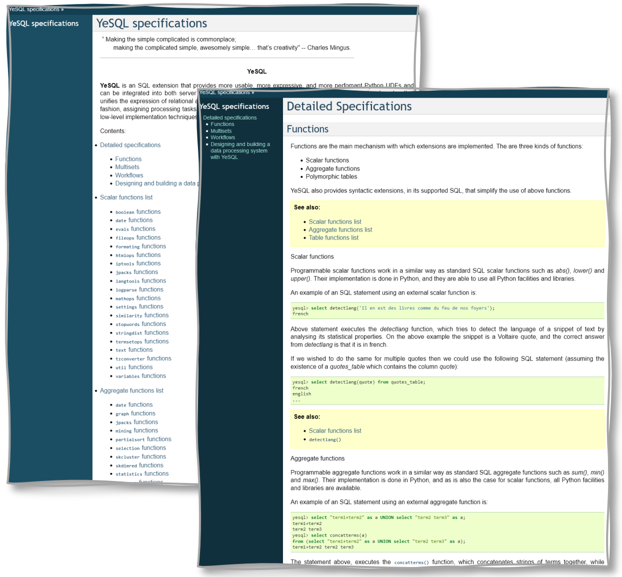
- SQL/programming language impedance mismatch still challenging
  - Pushing computation into data systems for scalability
- YeSQL promises high expressiveness, usability, and performance
- Fusion of UDFs and relational operators is key factor in YeSQL performance

- On-going work and next steps
  - Deeper fusion-based optimization
  - Provably-correct Python-2-YeSQL translation
  - Federated YeSQL query processing

YeSQL – Reading material, code bits

- Relevant publications
  - Y. Foufoulas, A. Simitsis, L. Stamatogiannakis, Y. Ioannidis: YeSQL: "You extend SQL" with Rich and Highly Performant User-Defined Functions in Relational Databases. In PVLDB 2022.
  - Y. Foufoulas, A. Simitsis, Y. Ioannidis: “YeSQL: Rich User-Defined Functions without the Overhead”. In PVLDB 2022.
  - Y. Foufoulas, A. Simitsis: “User-Defined Functions in Modern Data Engines”. In IEEE ICDE 2023.
  - Y. Foufoulas, A. Simitsis: “Efficient Execution of User-Defined Functions in SQL Queries”. In VLDB 2023.

- Repo:
  - Specs: https://athenarc.github.io/YeSQL/
  - Code, datasets: https://github.com/athenarc/YeSQL



YeSQL in practice

- YeSQL in R&D projects
  - Geospatial ontologies, text mining and information extraction, data cleaning and exploration, and machine learning on medical data

- YeSQL in production inside OpenAire
  - OpenAIRE implements the **Open Science** policies of Europe
  - 65 European institutions / 1,000+ data providers / 42M services last year / 140M pubs deduplicated
  - YeSQL used daily to harvest, classify, text mine, and extract information from all data providers to create the OpenAIRE Research Graph

Open Access → Open Science

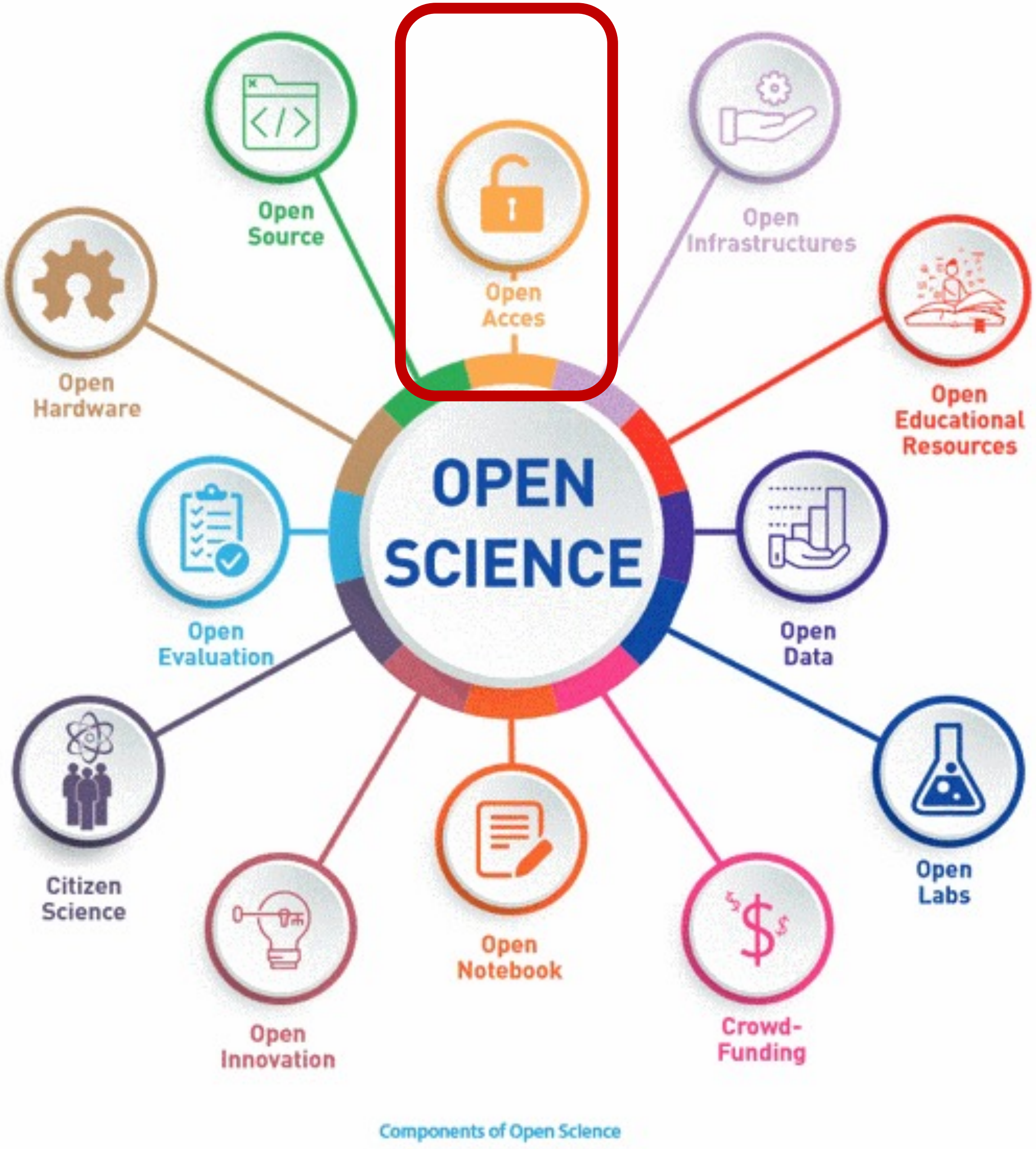
- Open Access: A policy on access to publications (mostly) and cost models

- Open Science: A new paradigm of the scientific process

ATHENA

Open Science

- Collaboration
- Reproducibility
- Transparency
- Trust



The picture was taken from the UNESCO Open Science brochure.

Open Science Dimensions

Open

paper publications

research data / FAIR data

software / lab books

methodologies / prot

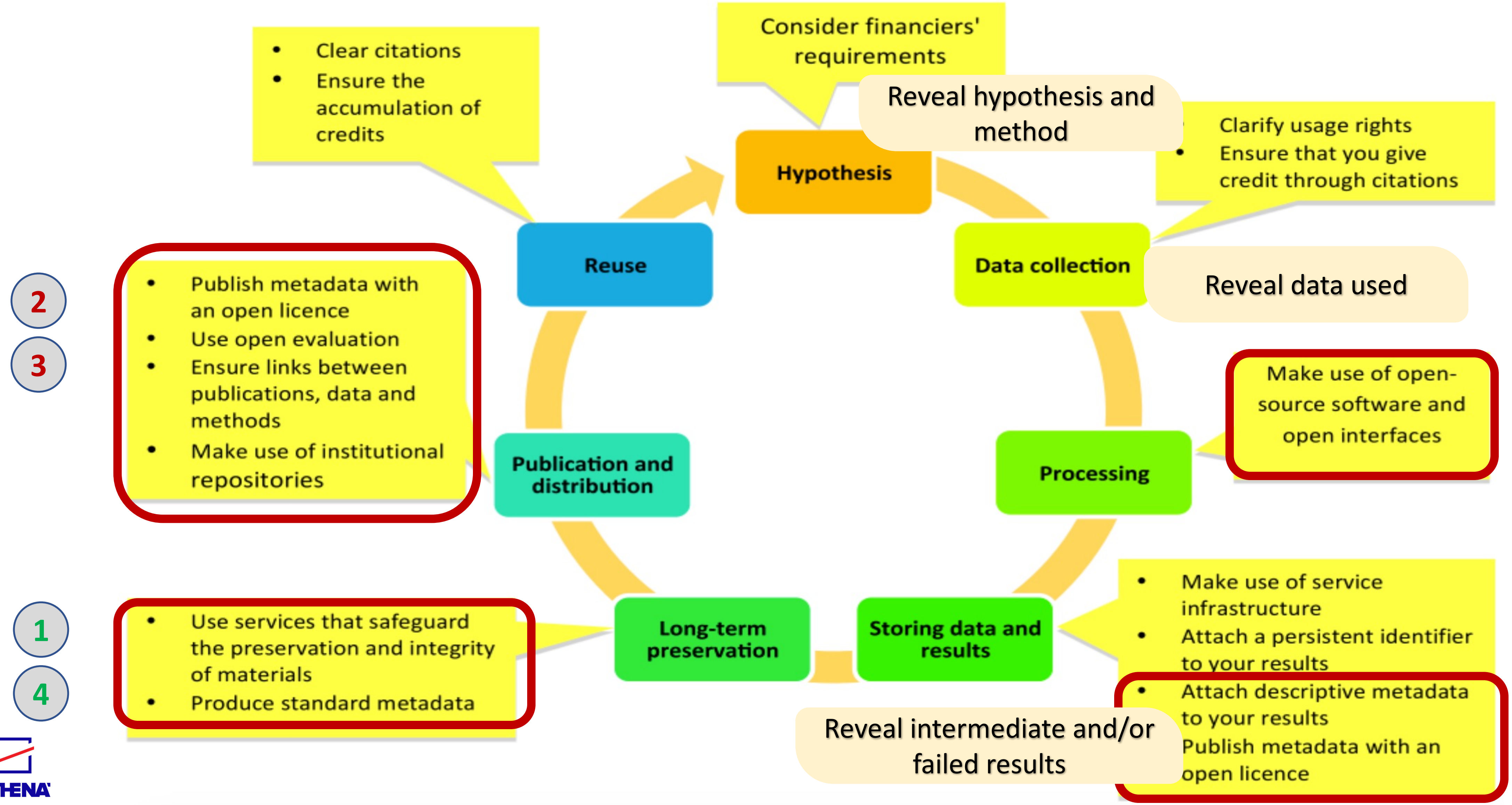
educational re

proce review)

s to resources for analytics

Changing the very definition of a publication:
A live graph with interconnected explorable objects

Open Research Lifecycle



ATHENA

Open Science: Open Evaluation/Reviews

- Reviews are a ***scientific contribution***
- Instead of 2-anonumous, 0–anonymous
- ***Reviewers sign their reviews***

- Transparency
- Accountability
- Quality

- Mitigation for potential senior-to-junior bulling

ATHENA

OPENAIRE

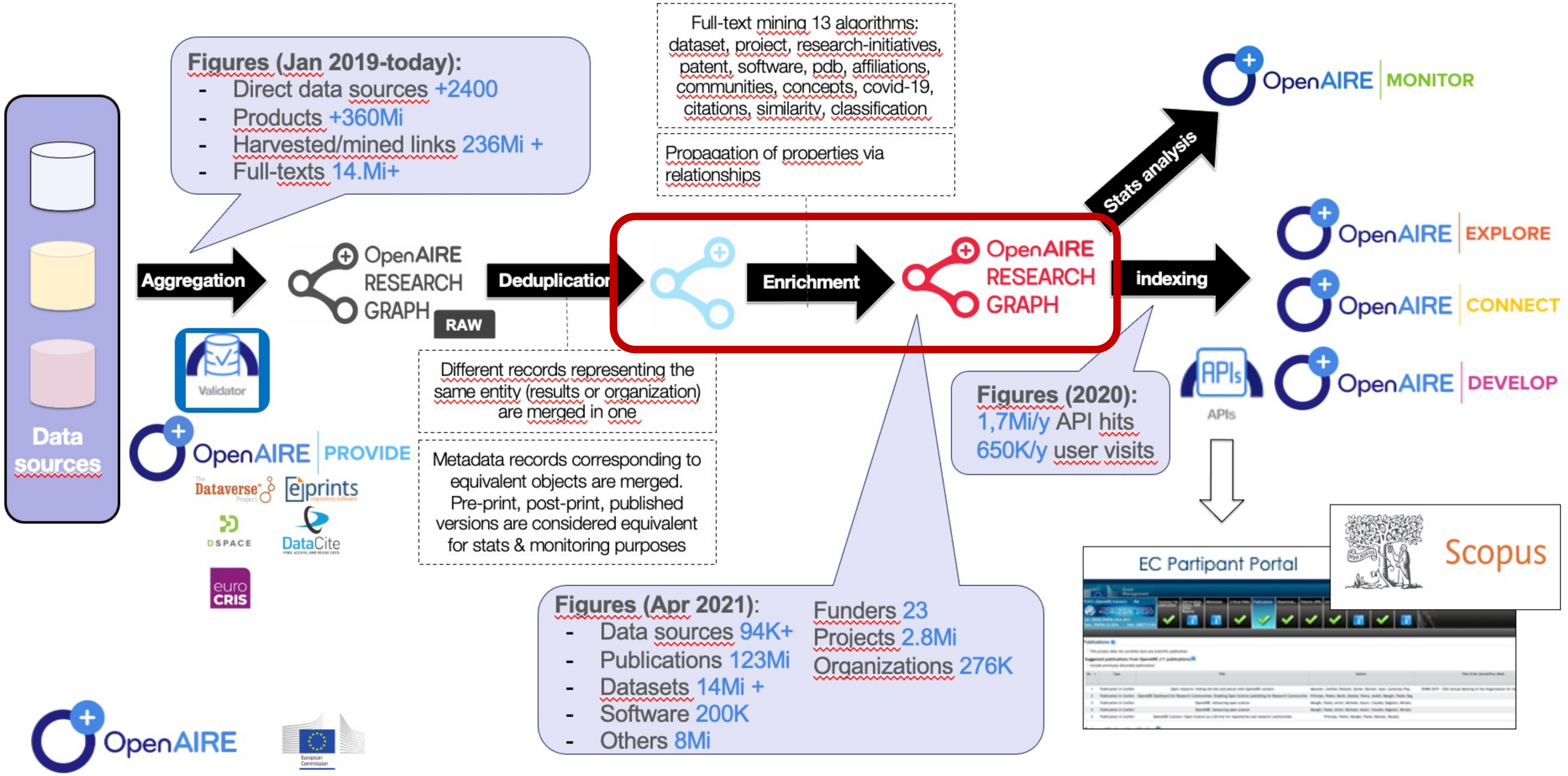
- European infrast
  open scholarly c
- **Non-profit orga**
  - Established Oct 20
  - Headquarters Gree
  - 47 members
  - From 34 countries

Natlia Manola – CEO
Athena RC - Athens

Paolo Manghi – CTO
CNR ISTI – Pisa

ATHENA

OpenAIRE Graph Dataflow



1947 75 2022

acm

Association for Computing Machinery

ATHENA

ΕΥΧΑΡΙΣΤΩ/**GRÀCIES**/HVALA/DĚKUJI/TAK/**DANK
JEWEL**/AITÄH/KIITOS/MERCI/**DANKE**/KÖSZNÖNÖ/
GRAZIE/PALDIES/**AČIŪ**/GRAZZI/TAKK/DZIĘKUJĘ
OBRIGADO/MULTUMESC/**СПАСИБО**/ХВАЛА
HVALA/**БЛАГОДАРЯ**/THANKYOU/TAK/GRACIAS
/**KIITOS**/TACK/TEŞEKKÜREDERIM/**СПАСИБІ**/
JUFALEMINDERIT/**ΕΥΧΑΡΙΣΤΩ**/DANKJEWEL/TAK
TACK/**GRAZZI**/DANKJEWEL/MULTUMESC/AITÄH
KÖSZNÖNÖ/СПАСИБО/**ХВАЛА**/AČIŪ/**THANKYOU**