How to get rid of the Relational Model, Relational Algebra, SQL, and ORMs (not only in Database Teaching)

Jens Dittrich

Saarland University

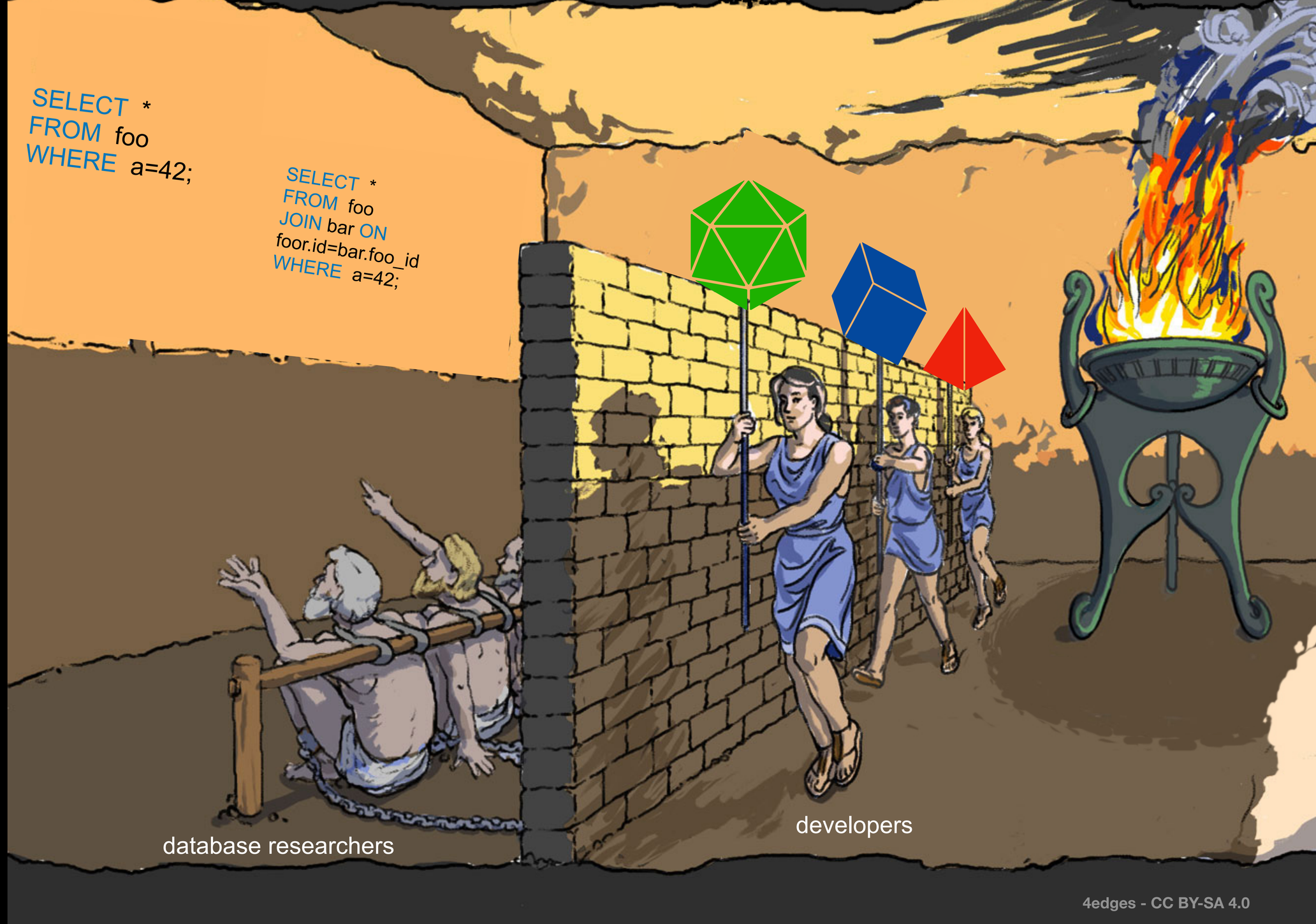
Agenda

Plato’s Allegory

Agenda

Plato’s Allegory

(database researcher’s version)

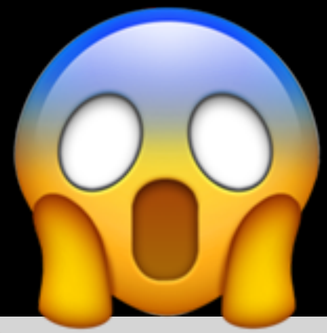


Backstory (1/2)

2019: I was head of the department's new Data Science and Artificial Intelligence B.Sc. and M.Sc. programs (DSAI)

Backstory (1/2)

- 2019: I was head of the department's new Data Science and Artificial Intelligence B.Sc. and M.Sc. programs (DSAI)
- many applications
- major issues with our existing application software OAS:
  - performance
  - usability
  - no software changes possible



The only alternative was an application software from SAP

Backstory (2/2)

2022: out of sheer desperation decided to build a new system myself

Well, I am the right person for this:

- ✅ working on “big data“,
- ✅ I know how to scale databases,
- ✅ do efficient query processing,
- ✅ I know some software engineering.

What it means to Develop a System

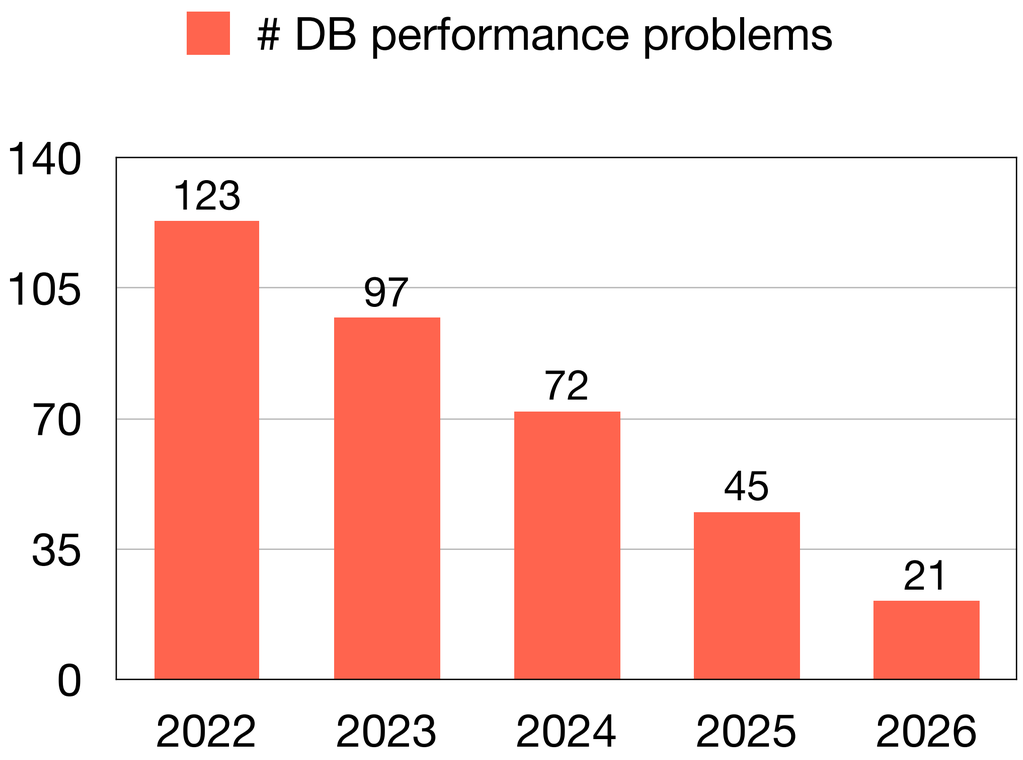
the outside world

the cave

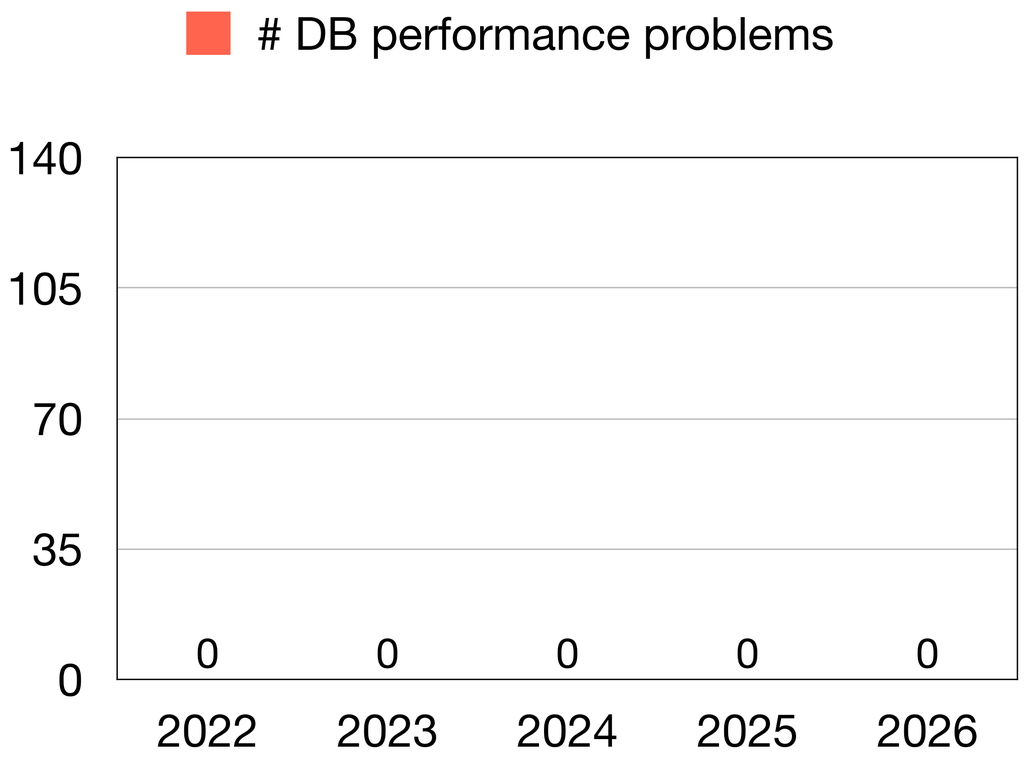
layers:	example technologies:
user	
UI	application-code written in Javascript, Vue
UI Application	UI-code written in javascript, PHP, React
Backend Application	application-code written in Python, Rust
Backend Framework	Django
ORM	object-relational mapper, Django ORM
Database System 	PostgreSQL, MySQL, Oracle, SQLite, DuckDB
Operating & File System	Linux, Windows, OS X, Android, iOS
Hardware	CPU, DRAM, SSD, hard disk

It turns out DB-related performance problems are “numerous“:

Expected:



Observed:



Agenda

Plato’s Allegory

(database researcher’s version)

Agenda

Plato's Allegory

(database researcher's version)

Two Major Developer Modes:

Developer Mode 1:

Developers do **NOT** use SQL-statements in Web development

-> “the people outside the cave do not see the shadows they cast in the cave“

The Full Stack

the outside world

the cave

layers:

user

UI

UI Application

Backend Application

Backend Framework

ORM

Database System

Operating & File System

Hardware

example technologies:

application-code written in Javascript, Vue

UI-code written in javascript, PHP, React

application-code written in Python, Rust

Django

object-relational mapper, Django ORM

PostgreSQL, MySQL, Oracle, SQLite, DuckDB

Linux, Windows, OS X, Android, iOS

CPU, DRAM, SSD, hard disk

ORM vs Database System

the outside world

the cave

layers:	example technologies:
user	
UI	application-code written in Javascript, Vue
UI Application	UI-code written in javascript, PHP, React
Backend Application	application-code written in Python, Rust
Backend Framework	Django
ORM	object-relational mapper, Django ORM
Database System	PostgreSQL, MySQL, Oracle, SQLite, DuckDB
Operating & File System	Linux, Windows, OS X, Android, iOS
Hardware	CPU, DRAM, SSD, hard disk

ORM vs Database System

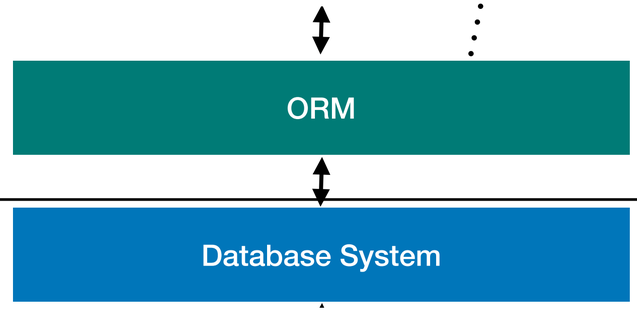
the outside world

ORMs bridge the gap between the OO-world and the flat table SQL-world

These libraries are called “mappers“, however, “wrapper“ or

“hide under the carpet“-er

would be a better name.



object-relational mapper,
Django ORM

PostgreSQL, MySQL, Oracle,
SQLite, DuckDB

the cave


ORM
Database System
Application
AI-generated

Example Roundtrip

Outside World

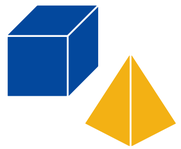
ORM

Database System

Example Roundtrip

Outside World

foo.objects.filter(a=42)



ORM

```
SELECT  *
FROM  foo
WHERE  a=42;
```

(7, 42, “Peter“),
(1, 42, “Horst“),
(4, 42, “Tom“),
(2, 42, “Jens“),

Database System

If SQL is so great, why are people building and heavily using these workarounds?

Why do people need

?

And why is there

in the first place?

Plus: Several More Problems with ORMs

n+1 queries
-> HUGE performance trap

broken transactional semantics for navigational queries
-> data in different version in the UI

data model management with migrations
-> sometimes migrations do not work: then manual migrations needed!

friction points and ugliness in ORM QL
-> swap data models inside the query,
e.g. try a simple GROUP BY in Django ORM

ORMs basically wrap the (limited) expressiveness of SQL into function calls
-> i.e., ORMs translate the expressiveness of SQL to the OO-world

Plus: Several More Problems with ORMs

n+1 queries
-> HUGE performance trap

broken transactional semantics for navigational queries
-> data in different version in the UI

data model management with migrations
-> sometimes migrations do not work: then manual migrations needed!

friction points and ugliness in ORM QL
-> swap data models inside the query,
e.g. try a simple GROUP BY in Django ORM

ORMs basically wrap the (limited) expressiveness of SQL into function calls
-> i.e., ORMs translate the expressiveness of SQL to the OO-world

Here is an example why wrapping SQL's expressiveness does not work:

[SIGMOD 2025]

[SIGMOD 2026]

Agenda

Plato's Allegory

(database researcher's version)

Two Major Developer Modes:

Developer Mode 1:

Developers do **NOT** use SQL-statements in Web development

-> “the people outside the cave do not see the shadows they cast in the cave“

Developer Mode 2:

Developers embed SQL directly in their programming language.

-> “the people outside the cave carry a mix of shadows and objects“

SQL Injection Explanation from my Undergrad Course

Mixing of Query and Parameter:

```
userName = "' OR '1'='1"
```

```
query = "SELECT * FROM users WHERE name = '" + userName + "';"
```

String concatenation

```
query = "SELECT  * FROM users WHERE name = '' OR '1'='1';“
```

call DBMS with:
`dbms.execute(query)`

query semantics:

```
"SELECT  * FROM users WHERE name = '' OR '1'='1';“
```

this will always evaluate to true!

DBMS parses the String and determines itself
which part is considered the query and
which part is considered the parameter
This is the SQL injection **problem**:
We injected additional SQL into the original query!

SQL Injection Fix in my Undergrad Course

Clear separation of Query und Parameter:

```
query = "SELECT * FROM users WHERE name = %s;"
```

```
userName = "' OR '1'='1"
```

call DBMS with:

```
dbms.execute(query, params = (userName,)
```

DBMS gets query string with placeholder and parameter **separately**
-> no confusion what the query is
-> no SQL injection possible

query semantics:

```
"SELECT  * FROM users WHERE name = "' OR '1'='1';"
```

This query will return all users where the name is equal to ' OR '1'='1'.

SQL Injection Fix in my Undergrad Course

Clear separation of Query und Parameter:

```
query = "SELECT * FROM users WHERE name = %s;"
```

call DBMS with:

```
dbms.execute(query, params = (userName,)
```

This / makes all the difference!

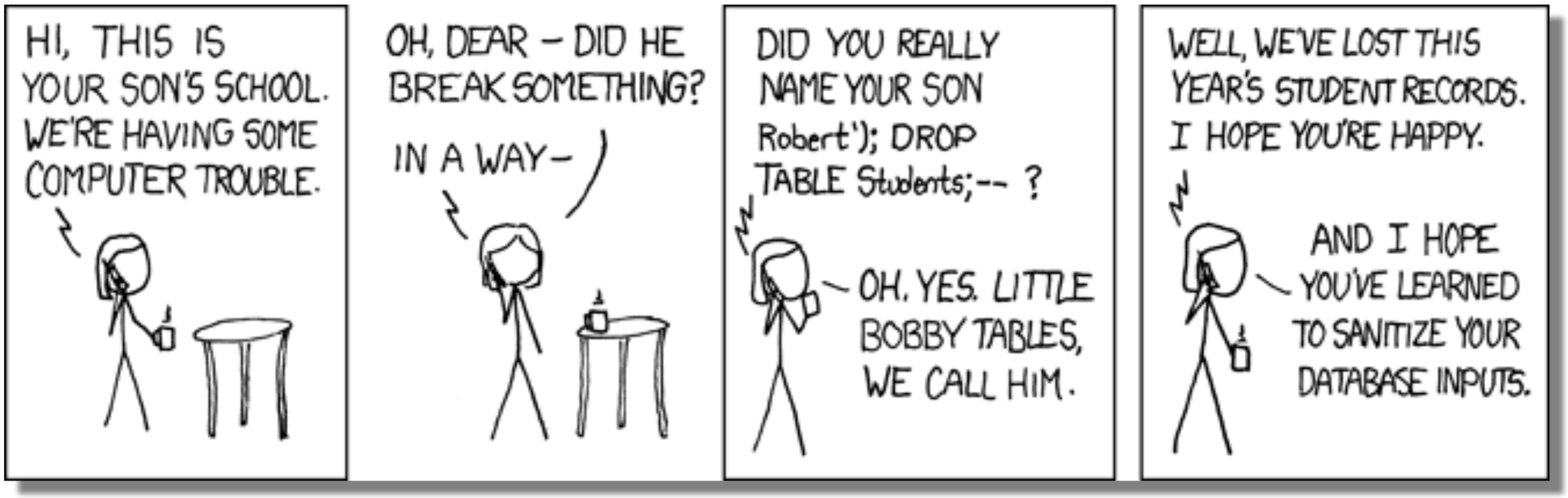
Did you spot it?

query semantics:

```
"SELECT  * FROM users WHERE name = "' OR '1'='1';"
```

This query will return all users where the name is equal to ' OR '1'='1'.

Little Bobby Tables



https://xkcd.com/327/

2024

cwe.mitre.org/top25/archive/2024/2024_top25_list

Common Weakness Enumeration
A community-developed list of SW & HW weaknesses that can become vulnerabilities

New to CWE? Start here!

Home > CWE Top 25 > 2024 | ID Lookup: Go

Home | About ▼ | Learn ▼ | Access Content ▼ | Community ▼ | Search ▼

2024 CWE Top 25 Most Dangerous Software Weaknesses

NOTICE: *This is a previous version of the Top 25. For the most recent version go **here**.*

Top 25 Home | Share via: X | View in table format | Key Insights | Methodology

1. Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
   CWE-79 | CVEs in KEV: 3 | Rank Last Year: 2 (up 1) ▲
2. Out-of-bounds Write
   CWE-787 | CVEs in KEV: 18 | Rank Last Year: 1 (down 1) ▼
3. Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
   CWE-89 | CVEs in KEV: 4 | Rank Last Year: 3
4. Cross-Site Request Forgery (CSRF)
   CWE-352 | CVEs in KEV: 0 | Rank Last Year: 9 (up 5) ▲

2025

cwe.mitre.org/top25/archive/2025/2025_cwe_top25.html

Common Weakness Enumeration
A community-developed list of SW & HW weaknesses that can become vulnerabilities

New to CWE? Start here!

Home > CWE Top 25 > 2025 | ID Lookup: Go

Home | About ▼ | Learn ▼ | Access Content ▼ | Community ▼ | Search ▼

2025 CWE Top 25 Most Dangerous Software Weaknesses

Top 25 Home | Share via: X | View in table format | Key Insights | Methodology

1. Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
   CWE-79 | CVEs in KEV: 7 | Rank Last Year: 1
2. Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
   CWE-89 | CVEs in KEV: 4 | Rank Last Year: 3 (up 1) ▲
3. Cross-Site Request Forgery (CSRF)
   CWE-352 | CVEs in KEV: 0 | Rank Last Year: 4 (up 1) ▲
4. Missing Authorizati…
   CWE-862 | CVEs in KE…

SQL Injection in my Undergrad Course

ORM (1/2)

Typical Properties of an ORM (Object-Relational Mapper)

An ORM shields the application developer from the database. It allows the application developer to

1. define an object-oriented schema
2. query for objects in that schema
3. insert, update, and delete those objects

Advantages

- much easier application development
- no more messing around with SQL
- automatic protection against SQL injection attacks (see below)

SQL Injection in my Undergrad Course

ORM (1/2)

Typical Properties of an ORM (Object-Relational Mapper)

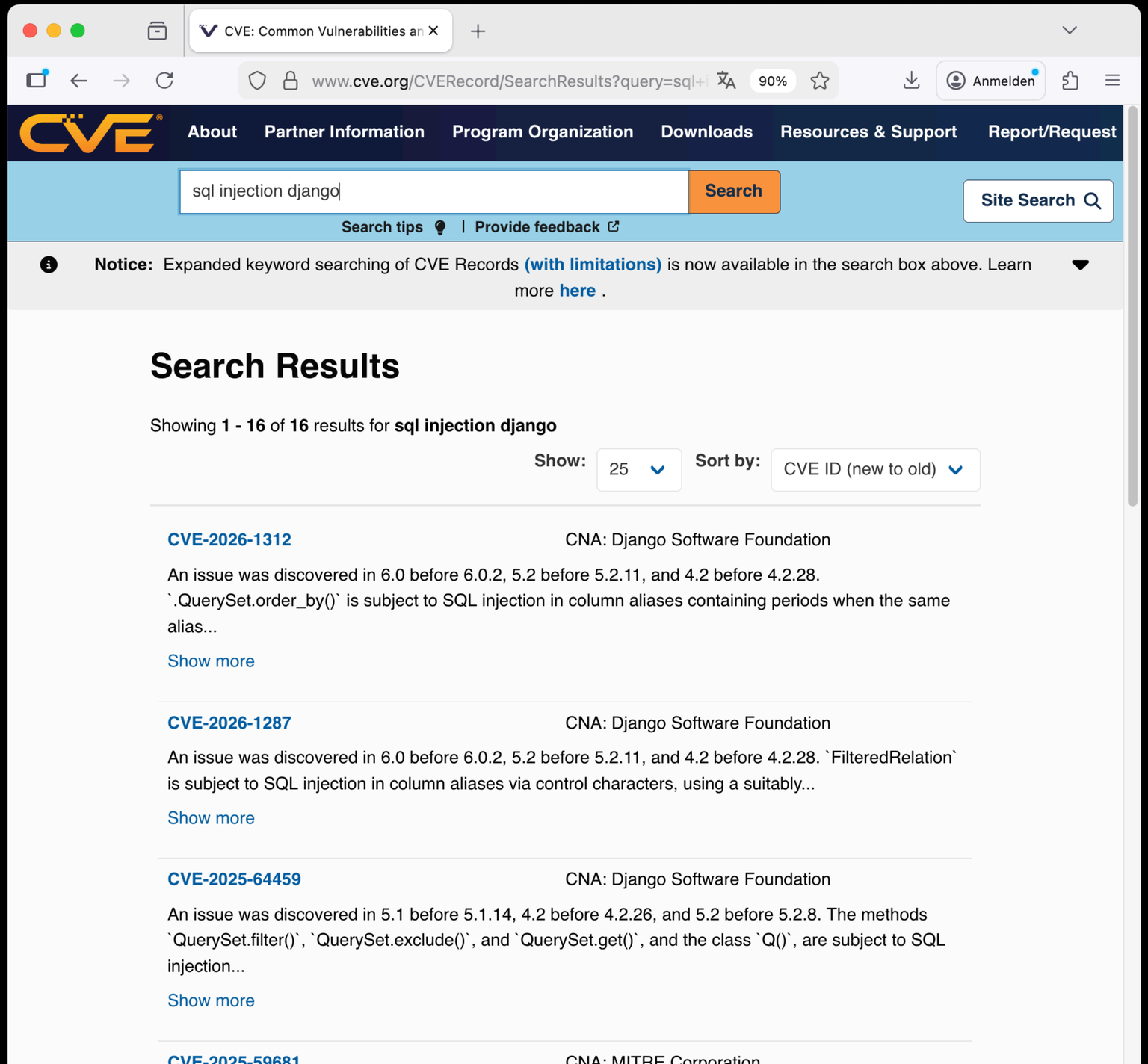
An ORM shields the application developer from the database. It allows the application developer to

1. define an object-oriented schema
2. query for objects in that schema
3. insert, update, and delete those objects

Advantages

- much easier application development
- no more messing around with SQL
- automatic protection against SQL injection attacks (see below)

Well…



16 SQL injection issues in Django ORM since 2019

SQL Injection in my Undergrad Course

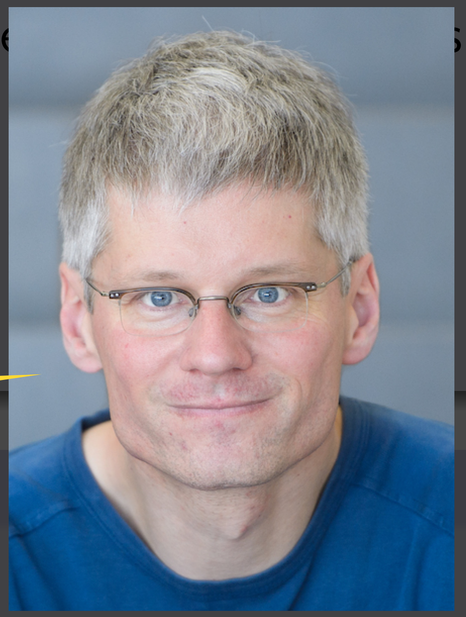
ORM (1/2)

Typical Properties of an ORM (Object-Relational Mapper)

An ORM shields the application developer from the [illegible] the application developer to

1. define an object-oriented schema
2. query for objects in that schema
3. [illegible] objects

!



TODO: fix this to “most of the time, but no guarantee!“

- much easier application development
- no more messing around with SQL
- automatic protection against SQL injection attacks (see below)

So, how to solve SQL injection?
Not as an afterthought but from the get go?

?

?

And, why isn’t this ONE language in the first place?

!

This is actually a fundamental software design problem!

!

Root of the problem: We combine two different languages: SQL and some programming language!

Enter: FDM and FQL

Vision paper:

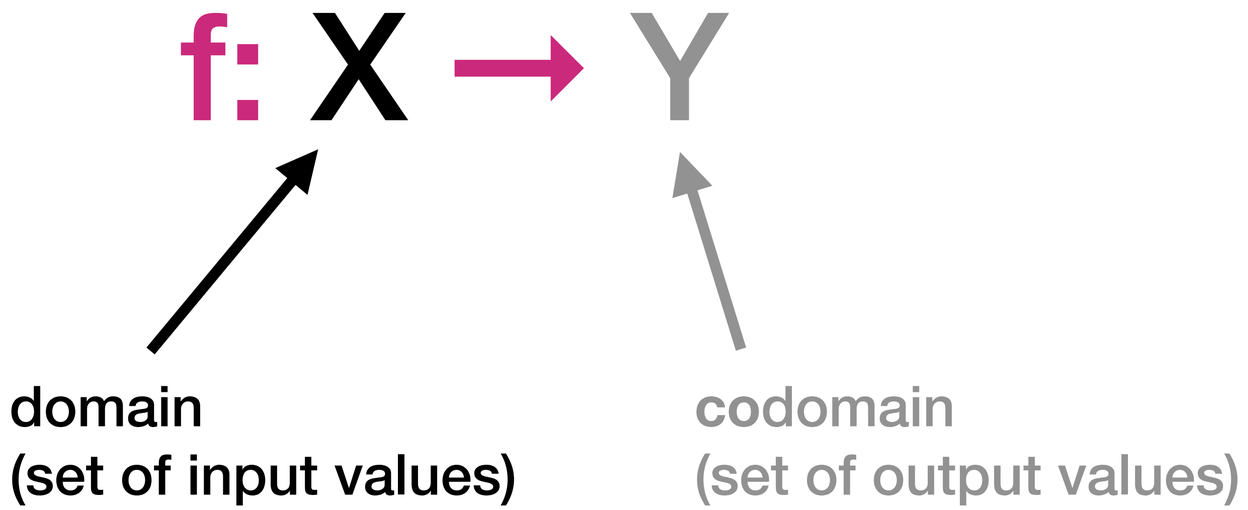
[EDBT 2026]

FDM and FQL: Core Ideas

- purely functional (key/value) data model
- same modeling concept at all levels, no matter whether we are looking at “tuples“, “relations“, or “databases“, ...
- all operators are unary: input is a function, output is a function
- query language is a façade and part of the embedding programming language
- same power for updates as for reading
- easily extensible
- the notion of an “index“ is built into the data model

None of this is true for SQL

Reminder: Functions

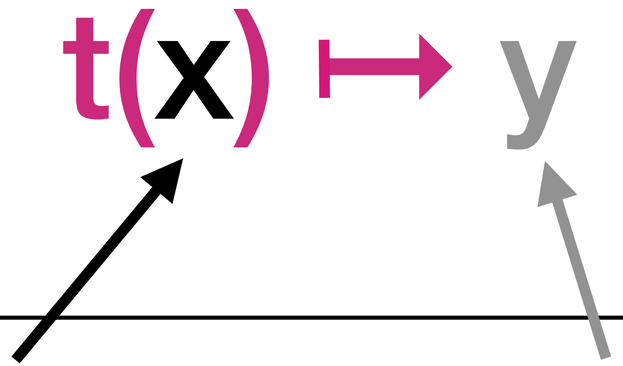


$$f(x) \mapsto y \quad \forall x \in X$$

FDM: Functions!

Concept:	$f(x) \mapsto y$

FDM: Tuple Functions

Concept:	
Special Case:	attribute name ↦ attribute value
Examples:	“first name“ ↦ “Alice“ “last name“ ↦ “Miller“ “age“ ↦ 42

FDM: Relation Functions

Concept:	$R(x) \mapsto y$
Special Case:	tuple key $\mapsto$ tuple function
Examples:	42 $\mapsto$ $t_{42}()$ 11 $\mapsto$ $t_{11}()$ 77 $\mapsto$ $t_{77}()$

FDM: Database Functions

Concept:	$DB(x) \mapsto y$
Special Case:	relation name $\mapsto$ relation function
Examples:	“users“ $\mapsto$ $R_{users}()$ “customers“ $\mapsto$ $R_{customers}()$ “orders“ $\mapsto$ $R_{orders}()$

FDM: Constraints

just like in relational database systems, yet way more powerful as expressed through predicates

You can attach arbitrary constraints:

- **(partial) schema**
- **cardinality**
- **…**

FQL: All Operators are Unary

In other words:

all operators are higher-order functions on FDM domains and codomains

$$Op: \mathbb{F}_{in} \rightarrow \mathbb{F}_{out}$$

domain of input FDM functions

codomain of **output** **FDM functions**

$$Op(f_{in}) \mapsto f_{out} \quad \forall f_{in} \in \mathbb{F}_{in}$$

FQL: All Operators are Unary

Concept:	$Op(f_{in}) \mapsto f_{out}$

FQL: All Operators are Unary

Concept:	$Op(RF_{in}) \mapsto RF_{out}$
Special Case:	relation function $\mapsto$ relation function
Examples:	one input relation $\overset{\sigma}{\mapsto}$ filtered input relation one input relation $\overset{\Gamma}{\mapsto}$ group by AND aggregate one input relation $\overset{\pi}{\mapsto}$ projection result

≈ unary relational algebra

FQL: All Operators are Unary

Concept:	$Op(DB_{in}) \mapsto RF_{out}$
Special Case:	database function $\mapsto$ relation function
Examples:	n relations $\xmapsto{Q}$ query result
	2 relations $\xmapsto{\bowtie}$ binary join result
	n relations $\xmapsto{\bowtie}$ n-ary join result
	n relations $\xmapsto{\cup}$ n-ary intersection result

≈ n-ary relational algebra

FQL: All Operators are Unary

Concept:	$Op(DB_{in}) \mapsto DB_{out}$
Special Case:	database function $\mapsto$ database function ≈ [SIGMOD 25]
Examples:	database $\xmapsto{\text{resultDB}}$ result database

Landscape of Input and Output Functions

co-domain $\mathbb{F}_{out}$ → **↓ domain $\mathbb{F}_{in}$**	TF	RF	DBF	SDBF
TF				
RF				
DBF				
SDBF				

Subset of $\mathbb{F}$	**Meaning**
TF	a set of tuple functions
RF	a set of relation functions
DBF	a set of database functions
SDBF	a set of sets of databases functions

Landscape of Input and Output Functions

co-domain $\mathbb{F}_{out}$ → ↓ domain $\mathbb{F}_{in}$	TF	RF	DBF	SDBF
TF				
RF		**unary** relational algebra operators, e.g. filter, group by		
DBF		**binary** relational algebra operators (kind of), e.g. join, union, intersect n-ary operators		
SDBF				

Subset of $\mathbb{F}$	Meaning
TF	a set of tuple functions
RF	a set of relation functions
DBF	a set of database functions
SDBF	a set of sets of databases functions

Landscape of Input and Output Functions

co-domain $\mathbb{F}_{out}$ → **↓ domain $\mathbb{F}_{in}$**	TF	RF	DBF	SDBF
TF	?	?	?	?
RF	?	**unary** relational algebra operators, e.g. filter, group by	?	?
DBF	?	**binary** relational algebra operators (kind of), e.g. join, union, intersect n-ary operators	?	?
SDBF	?	?	?	?

Subset of $\mathbb{F}$	**Meaning**
TF	a set of tuple functions
RF	a set of relation functions
DBF	a set of database functions
SDBF	a set of sets of databases functions

Landscape of Input and Output Functions

co-domain $\mathbb{F}_{out}$ → ↓ domain $\mathbb{F}_{in}$	TF	RF	DBF	SDBF
TF	?	?	?	?
RF	?	**unary** relational algebra operators, e.g. filter, group by	?	?
DBF	?	**binary** relational algebra operators (kind of), e.g. join, union, intersect n-ary operators	resultDB, subdatabase [SIGMOD 2025]	?
SDBF	?	?	?	?

Subset of $\mathbb{F}$	Meaning
TF	a set of tuple functions
RF	a set of relation functions
DBF	a set of database functions
SDBF	a set of sets of databases functions

Landscape of Input and Output Functions

co-domain $\mathbb{F}_{out}$ → ↓ domain $\mathbb{F}_{in}$	TF	RF	DBF	SDBF
TF	filter, map, project (per tuple λ-functions)	fake/test data generation	fake/test data generation	fake/test data generation
RF	aggregate a relation to a tuple, e.g., compute statistics for input like count the number of tuple functions, size, ...	unary relational algebra; updates in relational algebra and SQL filter, map, project (per tuple)	horizontal or vertical partitioning; replication; fake/test data generation	replicate **and** partition relation into shard relations
DBF	aggregate a database to a tuple, e.g., compute statistics for input like count the number of relation functions, size, ...	binary relational algebra any n-ary relation algebra	result database [47]; filter, map, project (per relation)	replicate **or** partition database into shard databases
SDBF	aggregate a set of databases to a tuple, e.g., compute statistics for input like count the number of database functions, size, ...	aggregate a set of databases to a relation, e.g., compute statistics for each database function, compute a size distribution over all databases	aggregate a set of databases to a database, e.g., merge two databases into one; compute statistics for each database function, compute a size distribution over all relations of all databases	result set of databases; filter, map, project (per database)

Table 1: A classification of FQL operators by the order of $\mathbb{F}_{in}$ and $\mathbb{F}_{out}$ (Definition 8). Green visualizes the same order ($\equiv$). Red means the output has a lower order ($\succ$). Yellow means a higher order ($\prec$). Each cell shows some example operators and/or entire subclasses like relational algebra. Many of these cells offer exciting opportunities for future work. The red boxes (▢) mark the space covered by relational algebra and SQL. The entry marked with (▪▪▪) denotes where relational algebra and SQL allow for updates, inserts, and deletes. In contrast, in FDM and FQL, the entire landscape can be used for updates, inserts, and deletes.

[EDBT 2026]

FQL: Same Power for Updates and Reads

co-domain $\mathbb{F}_{out}$ → ↓ domain $\mathbb{F}_{in}$	TF	RF	DBF	
TF	filter, map, project (per tuple λ-functions)	fake/test data generation	fake/test data ge	
RF	aggregate a relation to a tuple, e.g., compute statistics for input like count the number of tuple functions, size, ...	unary relational algebra; updates in relational algebra and SQL filter, map, project (per tuple)	replication; fake/ tion	relation
DBF	aggregate a database to a tuple, e.g., compute statistics for input like count the number of relation functions, size, ...	binary relational algebra any n-ary relation algebra	result database project (per relat	base into
SDBF	aggregate a set of databases to a tuple, e.g., compute statistics for input like count the number of database functions, size, ...	aggregate a set of databases to a relation, e.g., compute statistics for each database function, compute a size distribution over all databases	aggregate a set database, e.g., me into one; compute statistics for each database function, compute a size distribution over all relations of all databases	ter, map,

Updates in SQL: only possible in the dotted area

vs

Updates in FQL: entire space

Table 1: A classification of FQL operators by the order of $\mathbb{F}_{in}$ and $\mathbb{F}_{out}$ (Definition 8). Green visualizes the same order ($\equiv$). Red means the output has a lower order ($\succ$). Yellow means a higher order ($\prec$). Each cell shows some example operators and/or entire subclasses like relational algebra. Many of these cells offer exciting opportunities for future work. The red boxes (□) mark the space covered by relational algebra and SQL. The entry marked with (▪▪▪) denotes where relational algebra and SQL allow for updates, inserts, and deletes. In contrast, in FDM and FQL, the entire landscape can be used for updates, inserts, and deletes.

[EDBT 2026]

FQL is a Façade

Concept:	$Op(F_{in}) \mapsto F_{out}$
PL Concept:	`f_out = Op( f_in )`
Longer Example:	`f_out = Op( Op( Op( f_in ) ) )`

virtually all PLs have some syntax to express functions

-> no need to change the PL's parser

linear **logical FQL** plan **declares** the query in Python, C++., Rust, …

This does **NOT** necessarily execute all of this in the PL: **may** delegate to a backend/optimizer.

FQL is Lazy

Example:

`f_out` = Op(Op(Op(`f_in`)))

prepares an FQL pipeline, but does not
execute anything
(similar to Django ORM execution model)

`users` = `f_out`[“users“]

gets a handle to the value mapped to from key “users“,
whatever `users` maps to, we don’t care at this point (still lazy)

`horst` = `users`[“Horst“]

gets a handle to the value mapped to from key “Horst“,
whatever `horst` maps to, we don’t care at this point (still lazy)

funqdb on github

BigDataAnalyticsGroup/funqdb

github.com/BigDataAnalyticsGroup/funqdb

BigDataAnalyticsGroup / **funqdb** 🔒

Type / to search

Code | Issues | Pull requests | Agents | Actions | Projects | Security | Insights | Settings

funqdb Private

Edit Pins | Watch 0 | Fork 0 | Star 0

main | 1 Branch | 0 Tags | Go to file t | Code

Jens Dittrich Merge branch 'extend_ci' into 'main' 8ac19c2 · 4 days ago 129 Commits

ci	fix	4 days ago
docs	marburg	4 days ago
fdm	related_values	2 weeks ago
fql	schema validation	2 weeks ago
store	sentinel replacement on load; fixed tests; more tests	3 weeks ago
tests	related_values	2 weeks ago
.gitignore	fixed gitignore	2 months ago
.gitlab-ci.yml	fix tag	4 days ago
LICENSE header.txt	color test	3 weeks ago
LICENSE.txt	color test	3 weeks ago
README.md	readme	5 days ago
TODO.md	color test	3 weeks ago
poetry.lock	lol	4 days ago

About

FDM and FQL

- Readme
- AGPL-3.0 license
- Activity
- Custom properties
- 0 stars
- 0 watching
- 0 forks
- Audit log

Releases

No releases published
Create a new release

Packages

No packages published
Publish your first package

Contributors 1

SimRi99 Simon Rink

[EDBT 2026]

A Functional Data Model and Query Language is All You Need

Jens Dittrich
jens.dittrich@bigdata.uni-saarland.de
Saarland University
Germany

Abstract

We propose the vision of a functional data model (FDM) and an associated functional query language (FQL). Our proposal has far-reaching consequences: we show a path to come up with a modern query language (QL) that solves (almost if not) all problems of SQL (NULL-values, type marshalling, SQL injection, missing querying capabilities for updates, etc.). FDM and FQL are much more expressive than the relational model and SQL. In addition, in contrast to SQL, FQL integrates smoothly into existing programming languages. In our approach both QL and PL become the 'same thing', thus opening up several interesting holistic optimization opportunities between compilers and databases.

Keywords

Data Models, SQL Criticism, Relational Model, Relational Algebra, Functional Data Model, Functional Query Language, SQL Injection, ORM, Indexing, FDM, FQL

1 Introduction

This paper is inspired by two very recent papers which we believe give a new spin to what databases can and should be. The first, published at CIDR in January 2025 [16] (best paper award), is a vision paper that proposes to keep the entity relationship abstraction as a DDL interface to the DBMS. So rather than *first* translating an entity-relationship model (ERM) to the relational model (RM) to *then* CREATE tables, that paper allows a DBMS to

none of these abstractions and the constructs used in FQL are specific for tuples, relations, and/or databases anymore. With FQL, you can query any relation as if it were a tuple; you can query any database as if it were a tuple; you can query any set of databases as if it were a tuple, a relation, or a database, and so forth (Sections 2.2 and 2.8).

(3) FDM tears down the boundary between data that is stored and data that is computed (Section 2.4).

(4) FDM includes features of key, integrity constraints, and logical indexing as part of its conceptual definition already rather than as an afterthought (Sections 2.5 and 3).

(5) FQL is not limited to returning a single result table as in SQL or relational algebra-inspired languages (Section 2.7).

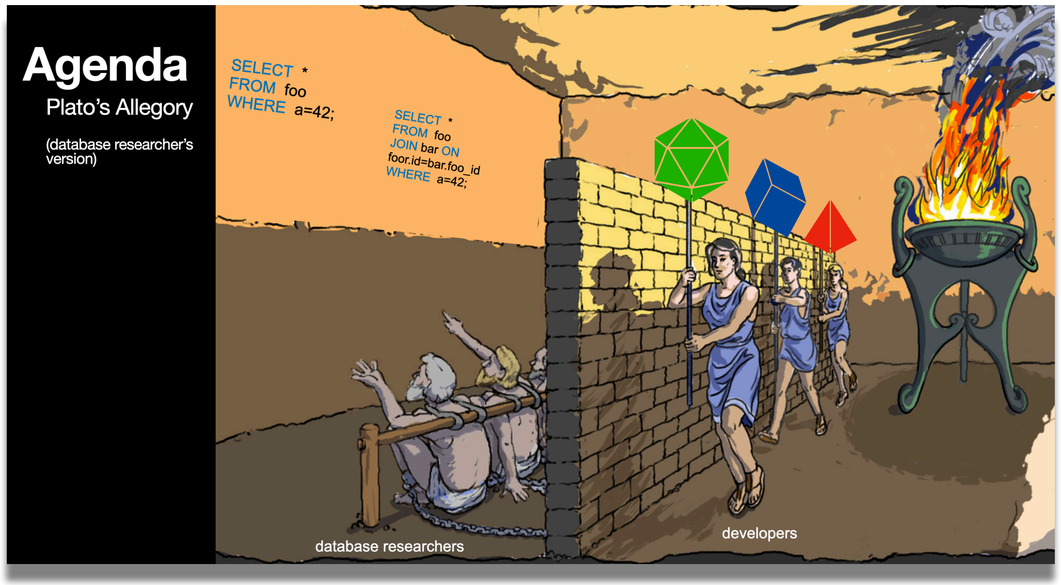
(6) FQL never leaves the data model in order to work around and/or compensate data model representation problems (Section 4.2).

(7) FQL is as powerful for querying as it is for changing data in contrast to SQL where reading data is much more powerful than writing data (Section 4.3).

(8) FQL is easily extensible. Whether a function is defined by 'a user' or by 'a library', FQL allows for using functions defined outside the realm of the database (Sections 4.1 and 4.2).

(9) FQL seamlessly blends into host programming languages (PLs). Everything in FQL is expressible through operators. From the point of view of a programmer, FQL looks like programming constructs of the PL, however, the PL may decide to delegate parts of these constructs to the database

Conclusions (1/2)



$$f(x) \mapsto y$$

tuple name $\mapsto$ tuple function

$$Op: F_{in} \rightarrow F_{out}$$

domain of input FDM functions

codomain of **output** FDM functions

Conclusions (2/2)

co-domain $\mathbb{F}_{out}$ → ↓ domain $\mathbb{F}_{in}$	TF	RF	DBF	SDBF
TF	filter, map, project (per tuple λ-functions)	fake/test data generation	fake/test data generation	fake/test data generation
RF	aggregate a relation to a tuple, e.g., compute statistics for input like count the number of tuple functions, size, ...	unary relational algebra; updates in relational algebra and SQL filter, map, project (per tuple)	horizontal or vertical partitioning; replication; fake/test data generation	replicate **and** partition relation into shard relations
DBF	aggregate a database to a tuple, e.g., compute statistics for input like count the number of relation functions, size, ...	binary relational algebra including any form of aggregation; any n-ary relation algebra	result database [47]; filter, map, project (per relation)	replicate **or** partition database into shard databases
SDBF	aggregate a set of databases to a tuple, e.g., compute statistics for in-	aggregate a set of databases to a relation, e.g., compute statistics for function, compute a on over all databases	aggregate a set of databases to a database, e.g., merge two databases into one; compute statistics for each database function, compute a size distribution over all relations of all databases	result set of databases; filter, map, project (per database)