A Functional Data Model and Query Language is All You Need

[Vision Paper]

Jens Dittrich

Saarland University

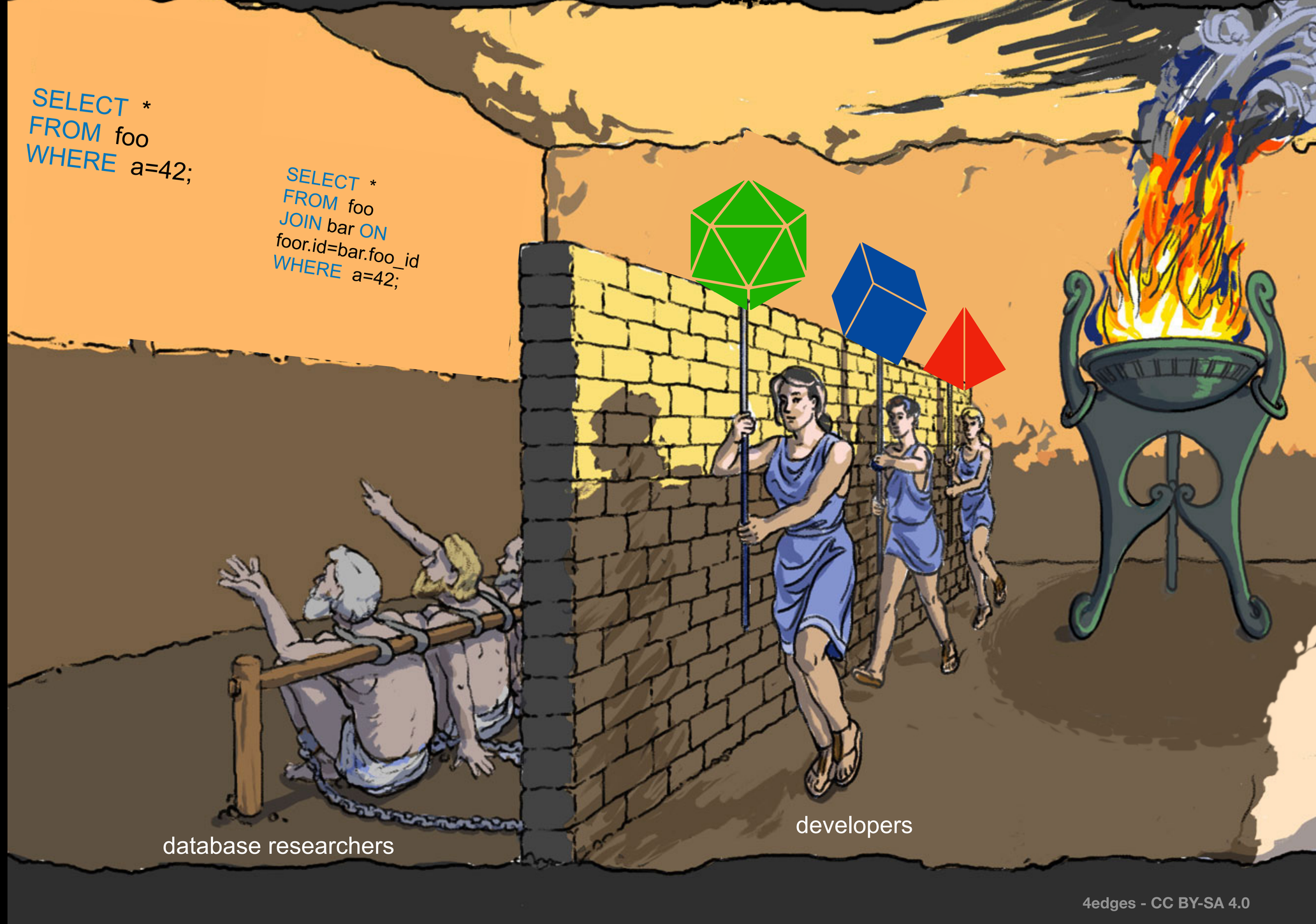
Agenda

Plato’s Allegory

Agenda

Plato’s Allegory

(database researcher’s version)



Agenda

Plato’s Allegory

(database researcher’s version)

Two Major Developer Modes:

Developer Mode 1:

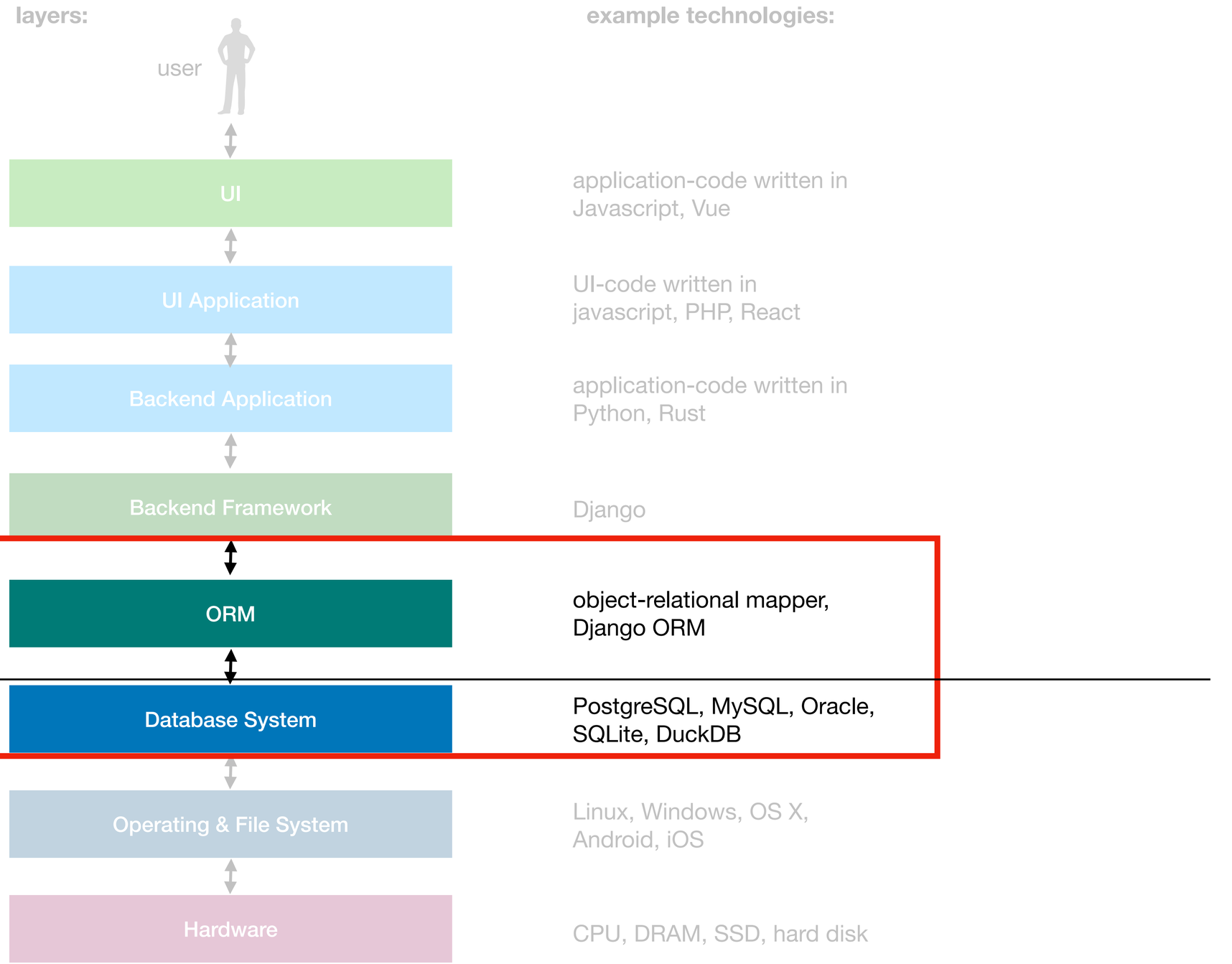
Developers do **NOT** use SQL-statements in Web development

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

ORM vs Database System

the outside world

the cave




ORM
Database System
Application
AI-generated

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

Plus: Several More Problems with ORMs

😫 n+1 queries
-> HUGE performance trap

😲 broken transactional semantics when fetching additional data
-> data in different versions in the UI

Plus: Several More Problems with ORMs

😫 n+1 queries
-> HUGE performance trap

😵 broken transactional semantics when fetching additional data
-> data in different versions in the UI

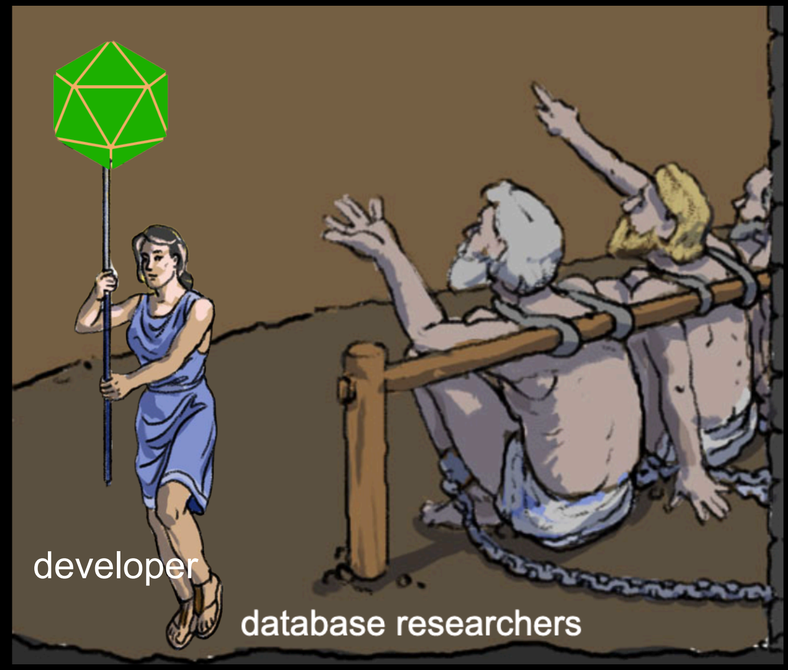
🤤 schema management in ORM
-> sometimes does not work

Plus: Several More Problems with ORMs

n+1 queries
-> HUGE performance trap

broken transactional semantics when fetching additional data
-> data in different versions in the UI

schema management in ORM
-> sometimes does not work



Plus: Several More Problems with ORMs

n+1 queries
-> HUGE performance trap

broken transactional semantics when fetching additional data
-> data in different versions in the UI

schema management in ORM
-> sometimes does not work

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

Plus: Several More Problems with ORMs

n+1 queries
-> HUGE performance trap

broken transactional semantics when fetching additional data
-> data in different versions in the UI

schema management in ORM
-> sometimes does not work

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

ORMs suffers from limited expressiveness of SQL
-> [SIGMOD 2025] [SIGMOD 2026]

so far:

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“

now:

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:

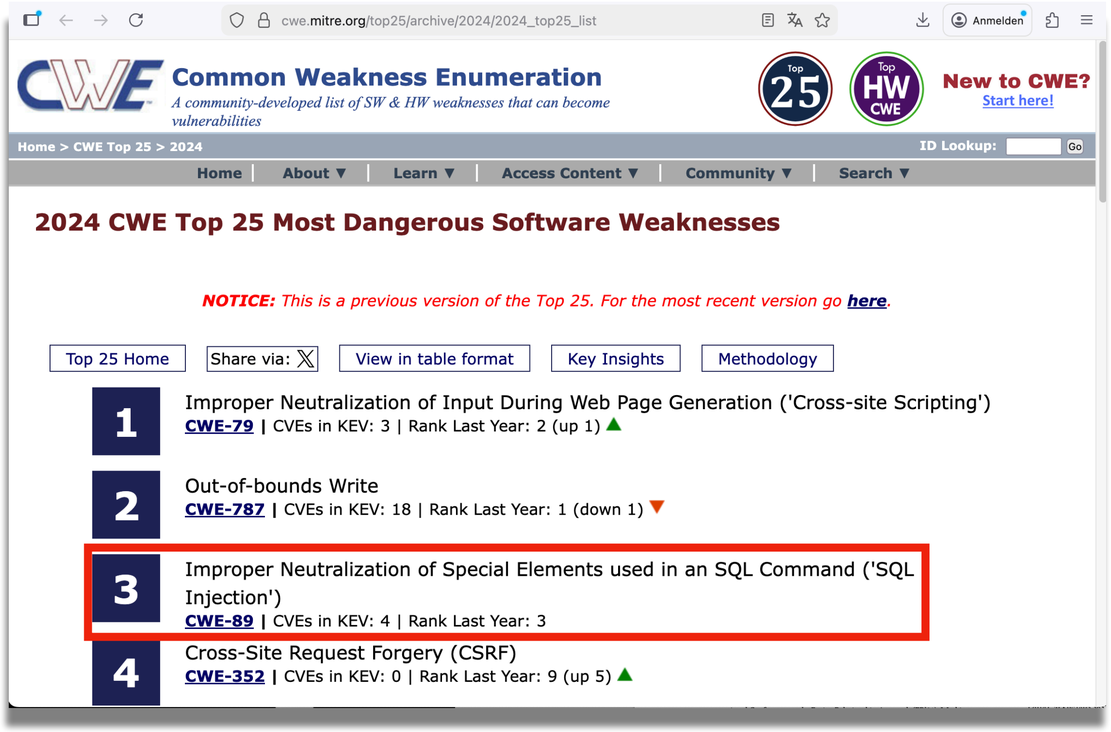
```
"SELECT  * FROM users WHERE name = "' OR '1'='1';"
```

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

see notebook

https://github.com/BigDataAnalyticsGroup/bigdataengineering/blob/master/SQL%20Injection.ipynb

2024


cwe.mitre.org/top25/archive/2024/2024_top25_list
Anmelden
Common Weakness Enumeration
A community-developed list of SW & HW weaknesses that can become vulnerabilities
Top 25
Top HW CWE
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:
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)

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…

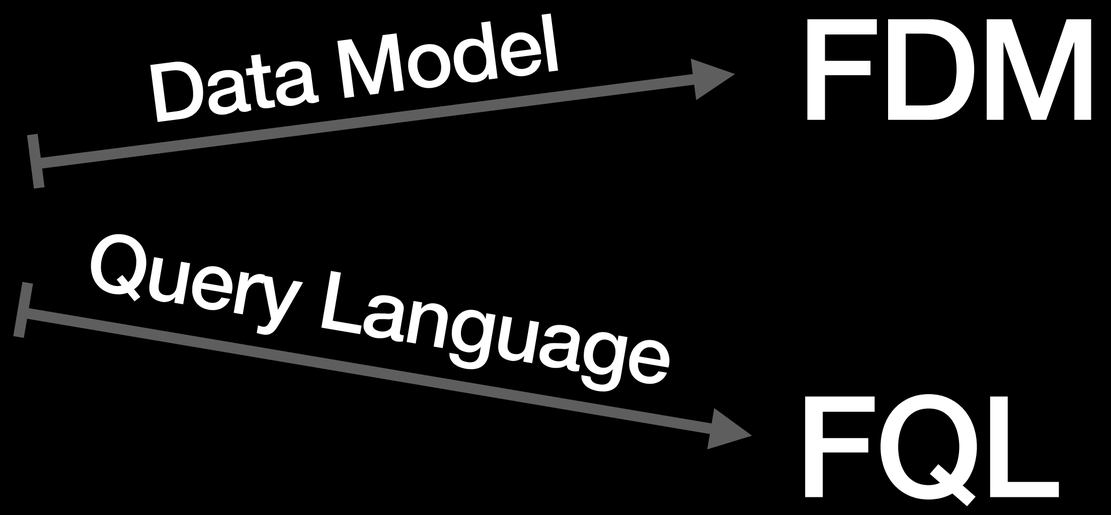
So, how to solve SQL injection?
Not as an afterthought but from the get go?

?

?

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

Functional

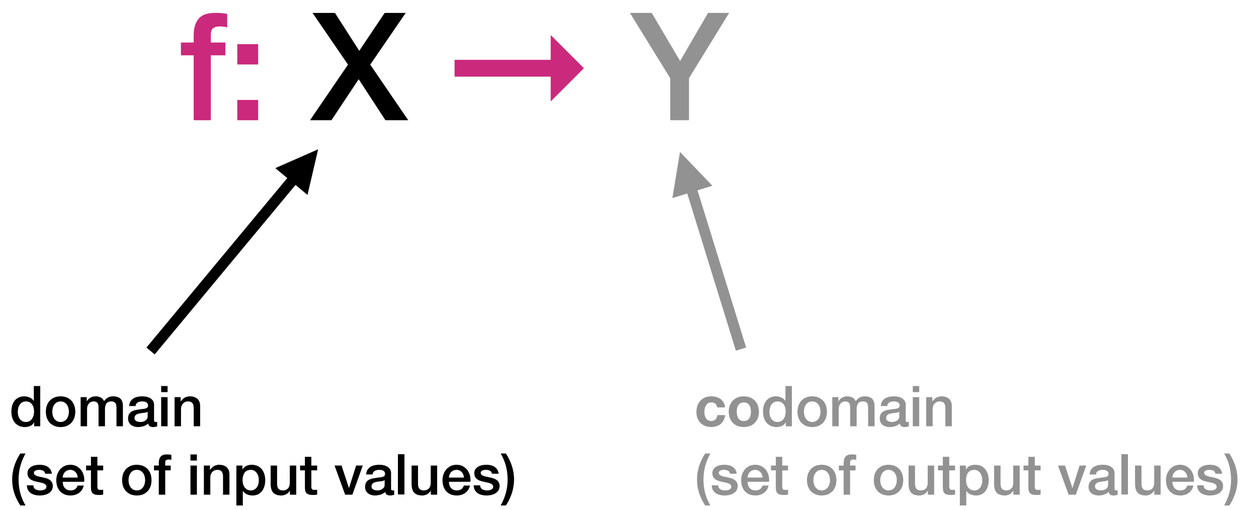


FDM and FQL: Core Ideas

- 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

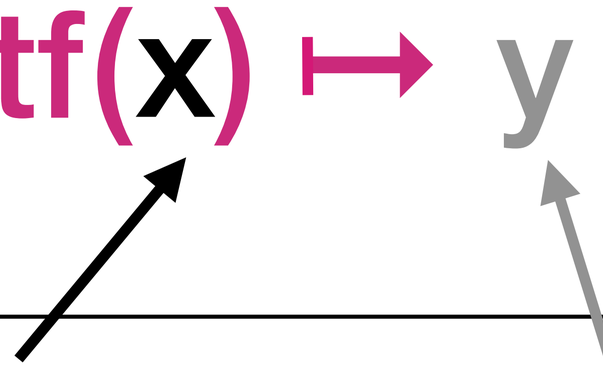


$$\forall x \in X \; f(x) \mapsto y$$

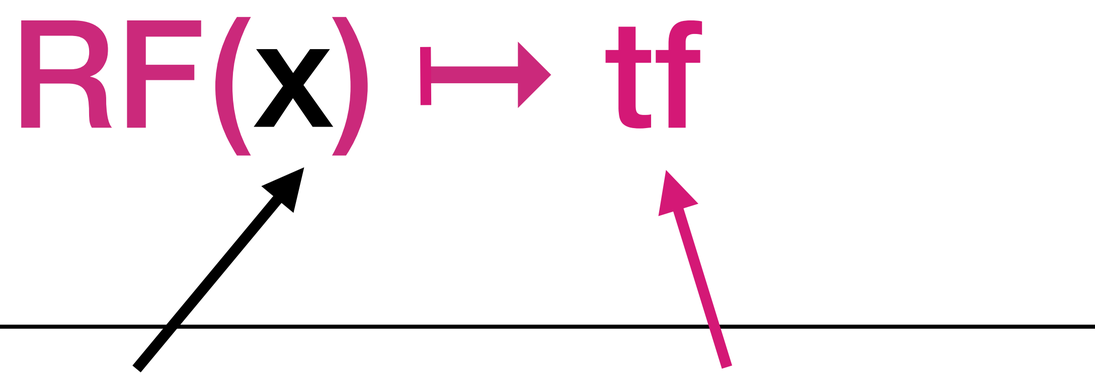
FDM: Functions!

Concept:	$f(x) \mapsto y$

FDM: Tuple Functions

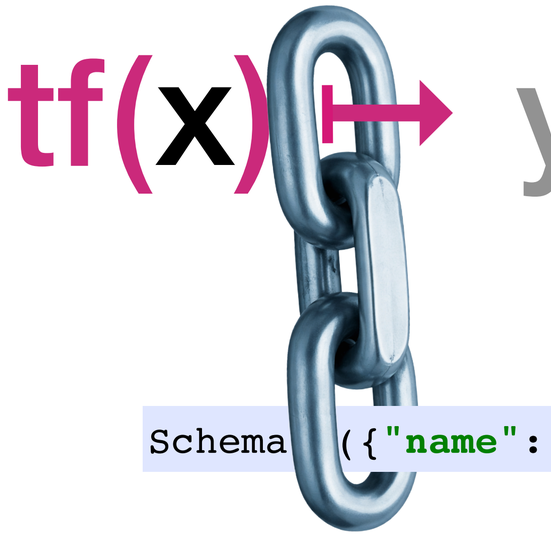
Concept:	
Special Case:	attribute name ↦ attribute value
Examples:	“first name“ ↦ “Alice“ “last name“ ↦ “Miller“ “age“ ↦ 42

FDM: Relation Functions

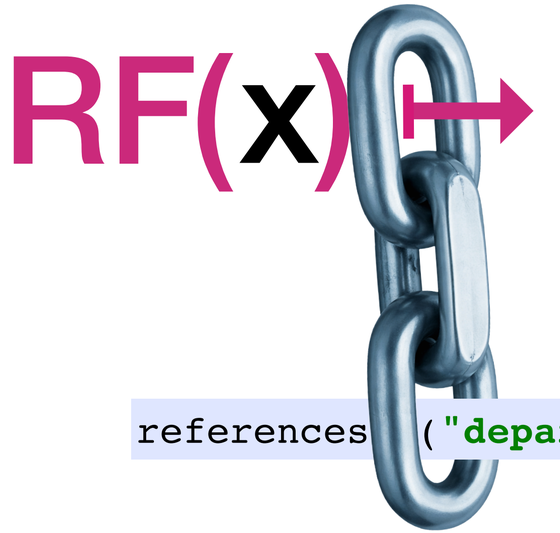
Concept:	
Special Case:	tuple key ↦ tuple function
Examples:	42 ↦ $tf_{42}()$ 11 ↦ $tf_{11}()$ 77 ↦ $tf_{77}()$

FDM: Database Functions

Concept:	$DBF(x) \mapsto RF$
Special Case:	relation name $\mapsto$ relation function
Examples:	“users“ $\mapsto$ $RF_{users}()$ “customers“ $\mapsto$ $RF_{customers}()$ “orders“ $\mapsto$ $RF_{orders}()$




```
Schema({"name": str, "yob": int, "department": TF})
```




```
references("department", departments)
```

We may attach arbitrary predicates to any function.

These predicates must be fullfilled no matter what insert/update/delete is performed.

This generalizes SQL's:

- CREATE TABLE
- CHECK
- NOT NULL
- FOREIGN KEY REFERENCES

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**

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

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	$\overset{\Gamma}{\mapsto}$	(flat, relational) group by AND aggregate
	one input relation	$\overset{\sigma}{\mapsto}$	filtered input relation
	one input relation	$\overset{\pi}{\mapsto}$	projection result

≈ unary relational algebra

FQL: All Operators are Unary

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

≈ n-ary relational algebra

FQL: All Operators are Unary

Concept:	$Op(DBF_{in}) \mapsto DBF_{out}$
Special Case:	database function $\mapsto$ database function ≈ [SIGMOD 25]
Examples:	database $\overset{resultDB}{\mapsto}$ result database database $\overset{DB}{\mapsto}$ arbitrary database

FQL: All Operators are Unary

Concept:	$Op(\ RF_{in}\ ) \mapsto DBF_{out}$
Special Case:	relation function $\mapsto$ database function
Examples:	one input relation $\overset{\Gamma}{\mapsto}$ (true) grouping sets, cube one input relation $\overset{\Gamma}{\mapsto}$ (true) grouping, partitioning, replication one input relation $\overset{\bowtie}{\mapsto}$ (true) outer joins

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.

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	er, 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.

funqdb on github

BigDataAnalyticsGroup/funqdb

github.com/BigDataAnalyticsGroup/funqdb

BigDataAnalyticsGroup / **funqdb** Private

Type / to search

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

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


LOUDER CROWDER
LOUDER CROWDER
SQL? LOL!
CHANGE MY MIND

Comparing SQL with FQL: TPC-H Query 5

```
select
    n_name,
    sum(l_extendedprice * (1 - l_discount)) as revenue
from
    customer,
    orders,
    lineitem,
    supplier,
    nation,
    region
where
    c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and l_suppkey = s_suppkey
    and c_nationkey = s_nationkey
    and s_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and r_name = ':1'
    and o_orderdate >= date ':2'
    and o_orderdate < date ':2' + interval '1' year
group by
    n_name
order by
    revenue desc
LIMIT 1;
```

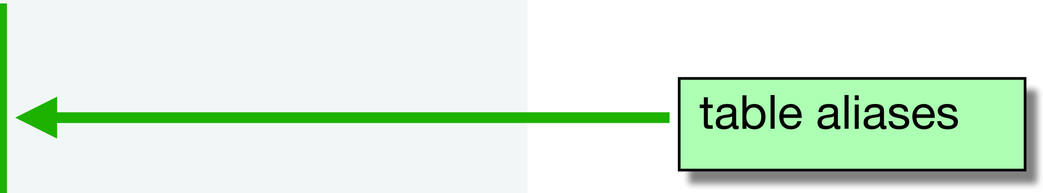
No aliases for tables

Attributes in the database schema must be unique to make them referrable.

Comparing SQL with FQL: JOB Query 10c

SQL

```
SELECT MIN(chn.name) AS character,
       MIN(t.title) AS movie_with_american_producer
FROM char_name AS chn,
     cast_info AS ci,
     company_name AS cn,
     company_type AS ct,
     movie_companies AS mc,
     role_type AS rt,
     title AS t
WHERE ci.note LIKE '%(producer)%'
  AND cn.country_code = '[us]'
  AND t.production_year > 1990
  AND t.id = mc.movie_id
  AND t.id = ci.movie_id
  AND ci.movie_id = mc.movie_id
  AND chn.id = ci.person_role_id
  AND rt.id = ci.role_id
  AND cn.id = mc.company_id
  AND ct.id = mc.company_type_id;
```



Comparing SQL with FQL: JOB Query 10c

SQL

```
SELECT MIN(chn.name) AS character,
       MIN(t.title) AS movie_with_american_producer
FROM char_name AS chn,
     cast_info AS ci,
     company_name AS cn,
     company_type AS ct,
     movie_companies AS mc,
     role_type AS rt,
     title AS t
WHERE ci.note LIKE '%(producer)%'
  AND cn.country_code = '[us]'
  AND t.production_year > 1990
  AND t.id = mc.movie_id
  AND t.id = ci.movie_id
  AND ci.movie_id = mc.movie_id
  AND chn.id = ci.person_role_id
  AND rt.id = ci.role_id
  AND cn.id = mc.company_id
  AND ct.id = mc.company_type_id;
```

1. SQL redundantly redefines foreign keys already present in the database schema and also transitive join predicates

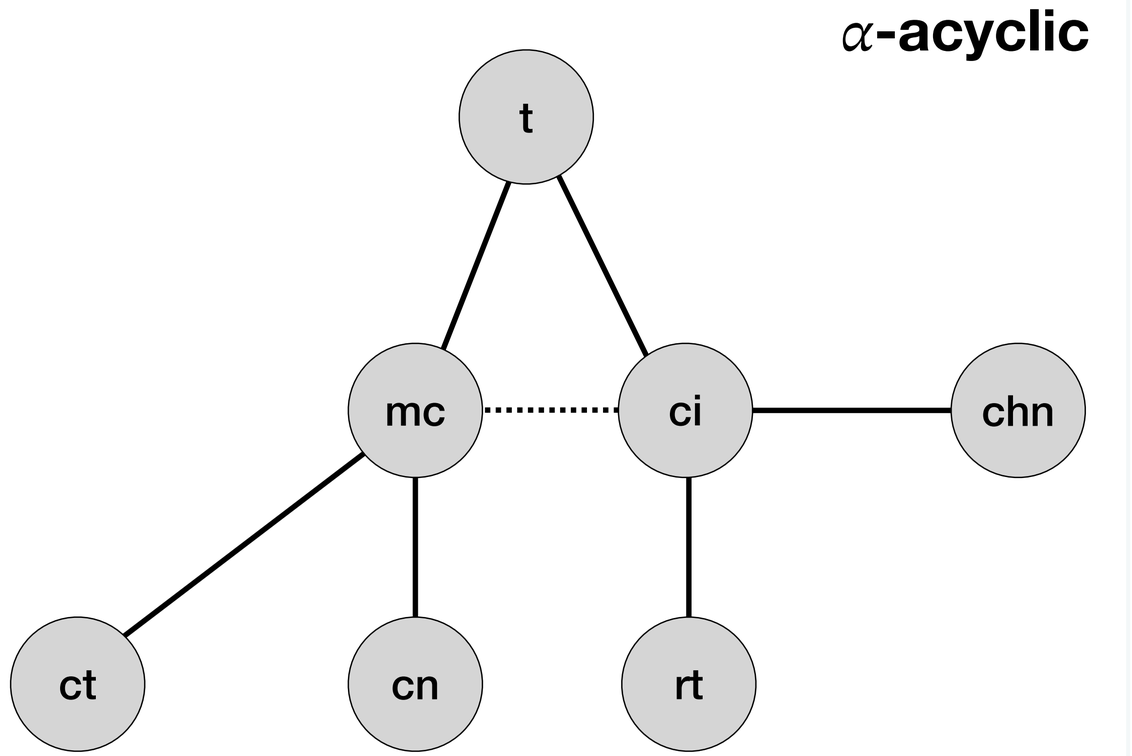
Comparing SQL with FQL: JOB Query 10c

SQL

```
SELECT MIN(chn.name) AS character,
       MIN(t.title) AS movie_with_american_producer
FROM char_name AS chn,
     cast_info AS ci,
     company_name AS cn,
     company_type AS ct,
     movie_companies AS mc,
     role_type AS rt,
     title AS t
WHERE ci.note LIKE '%(producer)%'
  AND cn.country_code = '[us]'
  AND t.production_year > 1990
  AND t.id = mc.movie_id
  AND t.id = ci.movie_id
  AND ci.movie_id = mc.movie_id
  AND chn.id = ci.person_role_id
  AND rt.id = ci.role_id
  AND cn.id = mc.company_id
  AND ct.id = mc.company_type_id;
```

1. SQL redundantly redefines foreign keys already present in the database schema and also transitive join predicates

SQL query graph:



Comparing SQL with FQL: JOB Query 10c

SQL

```
SELECT MIN(chn.name) AS character,
       MIN(t.title) AS movie_with_american_producer
FROM char_name AS chn,
     cast_info AS ci,
     company_name AS cn,
     company_type AS ct,
     movie_companies AS mc,
     role_type AS rt,
     title AS t
WHERE ci.note LIKE '%(producer)%'
  AND cn.country_code = '[us]'
  AND t.production_year > 1990
  AND t.id = mc.movie_id
  AND t.id = ci.movie_id
  AND ci.movie_id = mc.movie_id
  AND chn.id = ci.person_role_id
  AND rt.id = ci.role_id
  AND cn.id = mc.company_id
  AND ct.id = mc.company_type_id;
```

2. SQL defines table filters separate from relations: the user needs to mentally connect this in their head (split-attention)

Comparing SQL with FQL: JOB Query 10c

SQL

```sql
SELECT MIN(chn.name) AS character,
       MIN(t.title) AS movie_with_american_producer
FROM char_name AS chn,
     cast_info AS ci,
     company_name AS cn,
     company_type AS ct,
     movie_companies AS mc,
     role_type AS rt,
     title AS t
WHERE ci.note LIKE '%(producer)%'
  AND cn.country_code = '[us]'
  AND t.production_year > 1990
  AND t.id = mc.movie_id
  AND t.id = ci.movie_id
  AND ci.movie_id = mc.movie_id
  AND chn.id = ci.person_role_id
  AND rt.id = ci.role_id
  AND cn.id = mc.company_id
  AND ct.id = mc.company_type_id;
```

FQL (syntax in progress)

```
result: RF = aggregate(
    join(
        DBF(
            {
                "chn": char_name,
                "ci": cast_info.where(note__like="%(producer)%"),
                "cn": company_name.where(country_code="[us]"),
                "t": title.where(production_year__gt=1990),
            }
        )
    ),
    character=Min("chn.name"),
    movie_with_american_producer=Min("t.title"),
)
```

3. some tables that are neither filtered nor used in the aggregates are repeated in the query even though they are part of the database schema

Comparing SQL with FQL: JOB Query 10c

SQL

```sql
SELECT MIN(chn.name) AS character,
       MIN(t.title) AS movie_with_american_producer
FROM char_name AS chn,
     cast_info AS ci,
     company_name AS cn,
     company_type AS ct,
     movie_companies AS mc,
     role_type AS rt,
     title AS t
WHERE ci.note LIKE '%(producer)%'
  AND cn.country_code = '[us]'
  AND t.production_year > 1990
  AND t.id = mc.movie_id
  AND t.id = ci.movie_id
  AND ci.movie_id = mc.movie_id
  AND chn.id = ci.person_role_id
  AND rt.id = ci.role_id
  AND cn.id = mc.company_id
  AND ct.id = mc.company_type_id;
```

FQL (syntax in progress)

```
result: RF = aggregate(
    join(
        DBF(
            {
                "chn": char_name,
                "ci": cast_info.where(note__like="%(producer)%"),
                "cn": company_name.where(country_code="[us]"),
                "t": title.where(production_year__gt=1990),
            }
        )
    ),
    character=Min("chn.name"),
    movie_with_american_producer=Min("t.title"),
)
```

4. SQL is a language separate from the embedding programming language
-> high risk of SQL injection

Comparing SQL with FQL: JOB Query 10c

SQL

```sql
SELECT MIN(chn.name) AS character,
       MIN(t.title) AS movie_with_american_producer
FROM char_name AS chn,
     cast_info AS ci,
     company_name AS cn,
     company_type AS ct,
     movie_companies AS mc,
     role_type AS rt,
     title AS t
WHERE ci.note LIKE '%(producer)%'
  AND cn.country_code = '[us]'
  AND t.production_year > 1990
  AND t.id = mc.movie_id
  AND t.id = ci.movie_id
  AND ci.movie_id = mc.movie_id
  AND chn.id = ci.person_role_id
  AND rt.id = ci.role_id
  AND cn.id = mc.company_id
  AND ct.id = mc.company_type_id;
```

1. SQL redundantly redefines foreign keys already present in the database schema and also transitive join predicates

FQL (syntax in progress)

```
result: RF = aggregate(
    join(
        DBF(
            {
                "chn": char_name,
                "ci": cast_info.where(note__like="%(producer)%"),
                "cn": company_name.where(country_code="[us]"),
                "t": title.where(production_year__gt=1990),
            }
        )
    ),
    character=Min("chn.name"),
    movie_with_american_producer=Min("t.title"),
)
```

FQL does **NOT** require redefining foreign keys (which in FDM are foreign objects anyways); the query will however report about the assumptions and constraints used

Comparing SQL with FQL: JOB Query 10c

SQL

```sql
SELECT MIN(chn.name) AS character,
       MIN(t.title) AS movie_with_american_producer
FROM char_name AS chn,
     cast_info AS ci,
     company_name AS cn,
     company_type AS ct,
     movie_companies AS mc,
     role_type AS rt,
     title AS t
WHERE ci.note LIKE '%(producer)%'
  AND cn.country_code = '[us]'
  AND t.production_year > 1990
  AND t.id = mc.movie_id
  AND t.id = ci.movie_id
  AND ci.movie_id = mc.movie_id
  AND chn.id = ci.person_role_id
  AND rt.id = ci.role_id
  AND cn.id = mc.company_id
  AND ct.id = mc.company_type_id;
```

FQL (syntax in progress)

```
result: RF = aggregate(
    join(
        DBF(
            {
                "chn": char_name,
                "ci": cast_info.where(note__like="%(producer)%"),
                "cn": company_name.where(country_code="[us]"),
                "t": title.where(production_year__gt=1990),
            }
        )
    ),
    character=Min("chn.name"),
    movie_with_american_producer=Min("t.title"),
)
```

2. SQL defines table filters separate from relations: the user needs to mentally connect this in their head (split-attention)

relations are defined with their filters, much like a traditional query graph

Comparing SQL with FQL: JOB Query 10c

SQL

```sql
SELECT MIN(chn.name) AS character,
       MIN(t.title) AS movie_with_american_producer
FROM char_name AS chn,
     cast_info AS ci,
     company_name AS cn,
     company_type AS ct,
     movie_companies AS mc,
     role_type AS rt,
     title AS t
WHERE ci.note LIKE '%(producer)%'
  AND cn.country_code = '[us]'
  AND t.production_year > 1990
  AND t.id = mc.movie_id
  AND t.id = ci.movie_id
  AND ci.movie_id = mc.movie_id
  AND chn.id = ci.person_role_id
  AND rt.id = ci.role_id
  AND cn.id = mc.company_id
  AND ct.id = mc.company_type_id;
```

FQL (syntax in progress)

```
result: RF = aggregate(
    join(
        DBF(
            {
                "chn": char_name,
                "ci": cast_info.where(note__like="%(producer)%"),
                "cn": company_name.where(country_code="[us]"),
                "t": title.where(production_year__gt=1990),
            }
        )
    ),
    character=Min("chn.name"),
    movie_with_american_producer=Min("t.title"),
)
```

3. some tables that are neither filtered nor used in the aggregates are repeated in the query even though they are part of the database schema

intermediate tables do **NOT** have to be repeated as they exist in the schema anyway;

Again: FQL reports/“explains“ the tables and constaints used anyways

Comparing SQL with FQL: JOB Query 10c

SQL

```sql
SELECT MIN(chn.name) AS character,
       MIN(t.title) AS movie_with_american_producer
FROM char_name AS chn,
     cast_info AS ci,
     company_name AS cn,
     company_type AS ct,
     movie_companies AS mc,
     role_type AS rt,
     title AS t
WHERE ci.note LIKE '%(producer)%'
  AND cn.country_code = '[us]'
  AND t.production_year > 1990
  AND t.id = mc.movie_id
  AND t.id = ci.movie_id
  AND ci.movie_id = mc.movie_id
  AND chn.id = ci.person_role_id
  AND rt.id = ci.role_id
  AND cn.id = mc.company_id
  AND ct.id = mc.company_type_id;
```

FQL (syntax in progress)

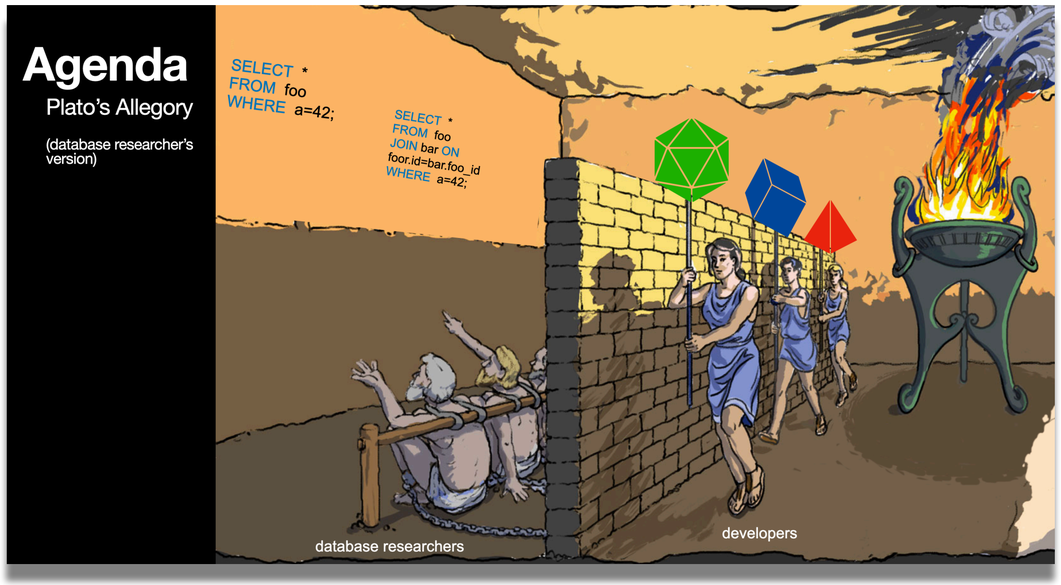
```
result: RF = aggregate(
    join(
        DBF(
            {
                "chn": char_name,
                "ci": cast_info.where(note__like="%(producer)%"),
                "cn": company_name.where(country_code="[us]"),
                "t": title.where(production_year__gt=1990),
            }
        )
    ),
    character=Min("chn.name"),
    movie_with_american_producer=Min("t.title"),
)
```

4. SQL is a language separate from the embedding programming language
-> high risk of SQL injection

FQL is a programming language is a façade, the query can be executed by any backend

Here the façade is in Python, but this can be almost any other programming language.

Conclusions (1/2)



$$RF(x) \mapsto tf$$

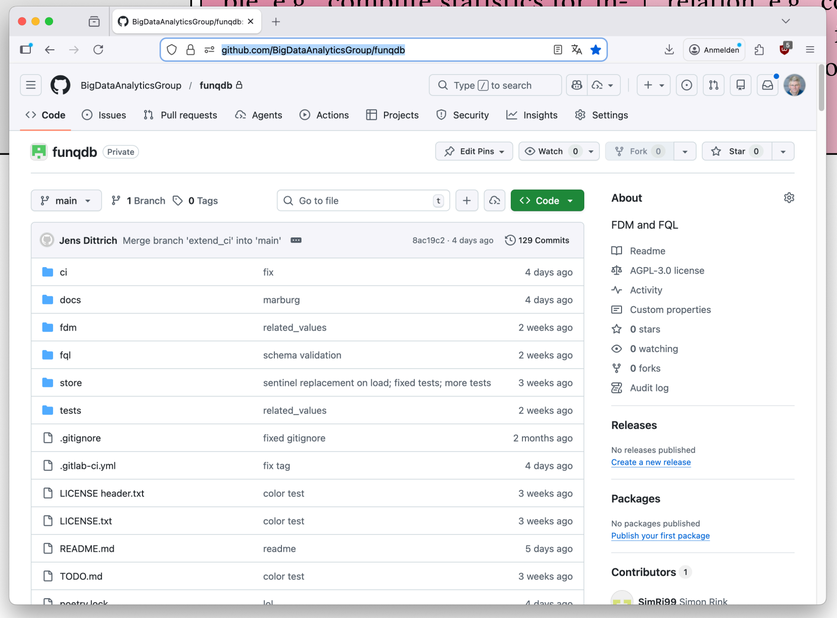
tuple key $\mapsto$ tuple function

$$Op(DBF_{in}) \mapsto RF_{out}$$

database function $\mapsto$ relation function

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)



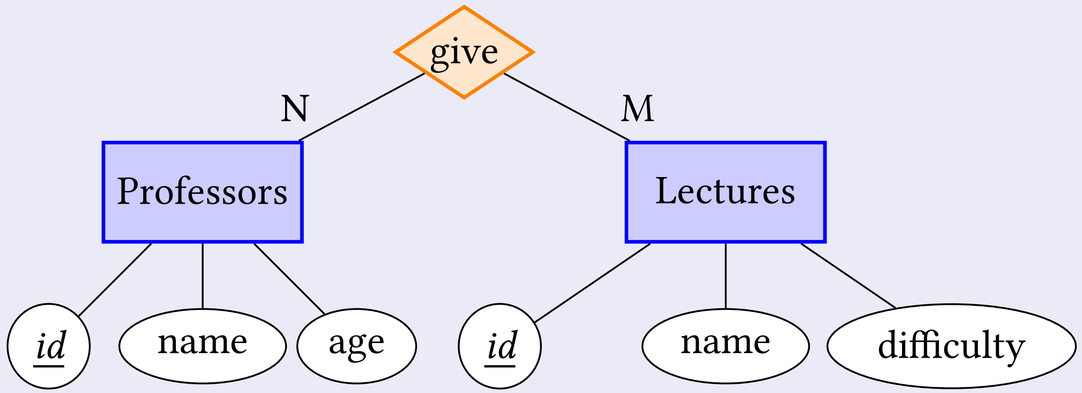
Backup Slides

Example: Getting Data from the DBMS for the UI

Problem Statement

Assume we want to DISPLAY a table with an N:M-relationship like this in our UI:

Professors	Lectures	
name	name	difficulty
Prof. A	Computer Science	low
	Databases	low
Prof. B	Computer Science	low
	Databases	low



(a) Entity-relationship model.

Professors

id	*name*	*age*
0	Prof. A	49
1	Prof. B	60
2	Prof. C	32

give

pid	*lid*
0	0
1	0
0	1
1	1
0	2
1	3

Lectures

id	*name*	*difficulty*
0	Computer Science	low
1	Databases	low
2	Mathematics	high
3	Artificial Intelligence	high

(b) Relational model and sample data.

How do we retrieve the data for that table from the database?

The Solution is Easy: One SQL Statement!

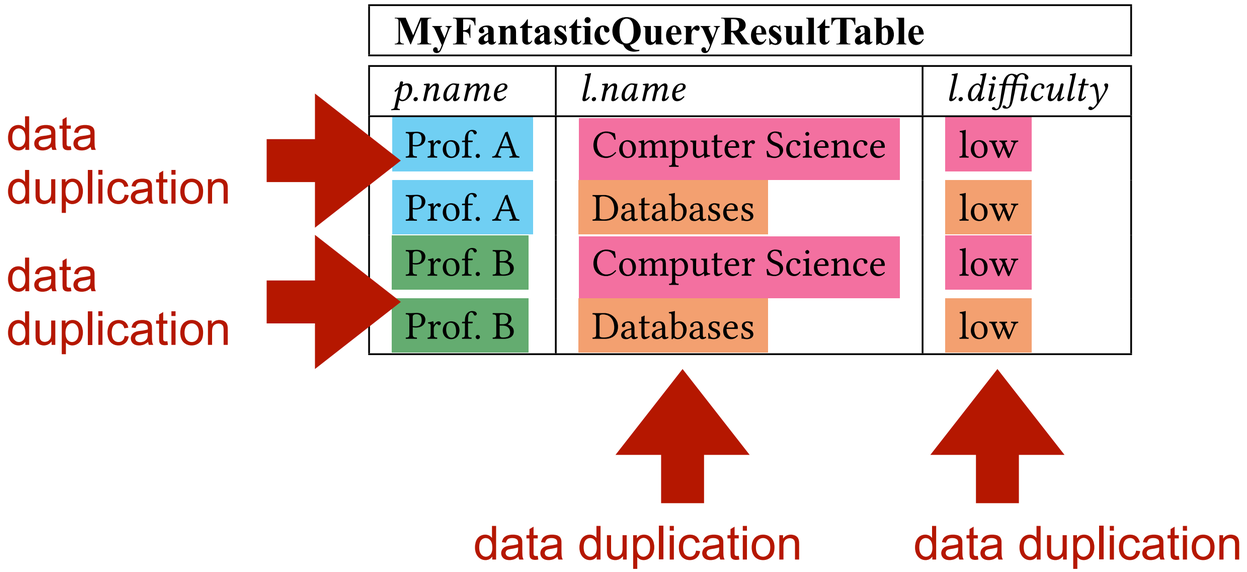
```sql
SELECT p.name, l.name, l.difficulty
FROM professors AS p, give AS g, lectures AS l
WHERE l.difficulty = 'low' AND
      p.id = g.pid AND
      l.id = g.lid
ORDER BY p.name;
```

Job done! Right?

Unfortunately, this leads to a couple of problems:

Problem 1/7: Denormalization ⇒ Data Duplication

SQL Output:



Desired Output:

Professors	Lectures	
name	name	difficulty
Prof. A	Computer Science	low
	Databases	low
Prof. B	Computer Science	low
	Databases	low

Why? Join operation **denormalizes** the data.

denormalization:
inverse operation to E/R and relational modeling

Our Solution: Allow SQL to Return a Result Subdatabase

SELECT RESULTDB SQL Extension:

```
SELECT RESULTDB p.id, p.name,
                g.pid, g.lid,
                l.id, l.name, l.difficulty
FROM professors AS p, give AS g, lectures AS l
WHERE l.difficulty = 'low' AND
      p.id = g.pid AND
      l.id = g.lid;
```

Result (conceptually):

{

Professors

id	*name*	*age*
0	Prof. A	49
1	Prof. B	60
2	Prof. C	32

,

give

pid	*lid*
0	0
1	0
0	1
1	1
0	2
1	3

,

Lectures

id	*name*	*difficulty*
0	Computer Science	low
1	Databases	low
2	Mathematics	high
3	Artificial Intelligence	high

}

Impact: 7 Problems Solved at the Same Time

- no denormalization problems whatsoever
- schema of the result? same as the database!
- no relational/key/schema information loss
- the result already contains objects, I mean entities, no: objects, wait
- ahh right: objects=entities!

[SIGMOD 2025]

[SIGMOD 2026]