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 work with the ERM abstraction *directly*. This has many positive implications including: all semantics of the ERM are preserved. However, in terms of query language, [16] suggests to resort to approaches like SQL++[49] which however leads to a couple of additional problems including the impedance mismatch with JSON. A second recent paper (our own previous work), which appeared in June 2025 at SIGMOD [47], identifies a long list of problems with SQL which all have the same root cause: SQL forces all result data into a single (possibly denormalized) result relation. Therefore we showed how to extend SQL to allow it to return a result *subdatabase*, i.e., the relations with their subset of tuples from all the input relations of the query that contribute to the result. Those two works heavily inspired our paper. Originally, we planned to simply combine those two ideas. But while doing so, we observed that we can come up with a much more versatile data model and query language. Therefore, our paper is more than the sum of [16] and [47]. It is the product. This vision paper and the proposed FDM and FQL come with a long list of contributions and opportunities:

(1) We present the vision of a functional data model (FDM) and a functional query language (FQL) (Sections 2, 3, and 4).
(2) FDM tears down the boundaries between tuples, relations, and databases when modeling and querying data. Thus, 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 system (Section 4.2).
(10) FQL makes SQL injection close to impossible *by design* and not as an afterthought as in SQL (Section 4.2).
(11) We present practical challenges and an agenda how to adopt FDM and FQL (Section 5).
(12) Finally, we present an initial research agenda on FDM and FQL (Section 7).

2 Overview of FDM

2.1 Foundations

We start with the common function definition taken from [67]:

Definition 1 [Function (Set-based definition)]. A function $f: X \rightarrow Y$ from a set X (called the domain) to a set Y (called the codomain) assigns to each element $x \in X$ exactly one element $y \in Y$: $f(x) \mapsto y$.

According to Definition 1, a function is just a special case of a relation! However, the term "function" may have multiple meanings which often get confused. Throughout this paper, we will use the *function machine metaphor* [40] to define a function:

Definition 2 [Function (Function Machine definition)]. A function f is a *blackbox* that takes as its input an element from a set X and returns an element from a set Y.

Though the difference of the two definitions sounds subtle, this is actually a big change: in Definition 2, the focus shifts away from relating pairs of values to an *arbitrary blackbox computation*. We

EDBT '26, Tampere (Finland)

will later see how this is key to unifying static data, i.e., elements from X and Y, and computation, i.e., queries and views.

Definition 3 [Higher-Order Function]. A function f is called a higher-order function if its domain or codomain is a set of functions.

2.2 A Functional Data Model (FDM)

Rather than modeling relations as sets, a central idea of our paper is the following: we **model tuples, relations, databases, and sets of databases as functions**. We start with tuples at the lowest level, here the natural input to a function is the attribute name returning the attribute value as the result[1]. For relation functions, a natural input is either a 'row'- or 'tuple'-id or their primary key. Calls to relation functions (Section 2.4) return tuple functions. For database functions, a natural input is the name of a 'table'. Calls to database functions (Section 2.7) return relation functions. And so forth.

	Modeling Concept	
Abstraction	**Relational Model**	**FDM**
tuple	sequence of attribute/value-pairs, alternatively: function	function
relation	set of tuples	function
database	set of relations	function
set of databases	set of databases	function
index	n/a	function call

This model has three main effects: First, we tear down the boundaries between tuples, relations, and databases when modeling, representing, and querying data. Thus, none of these abstractions and the constructs used in a query language operating on FDM are specific for tuples, relations, and/or databases anymore. Second, in this model each function call already conceptually reflects an index lookup (Section 2.5). Third, it allows us to hide whether data is static (retrieved) or computed.

2.3 Tuple Functions

As of Definition 2, a function maps elements from an input domain X to an output codomain Y. Thus, a natural conceptual, but discrete, way to define a *tuple function* representing data of a single tuple is:

$$t_1(attr : \text{string}) := \big\{(\text{'name'} : \text{'Alice'}), (\text{'age'} : 12)\big\}.$$

Function $t_1(X) \rightarrow Y$ has the domain $X = \{\text{'name'}, \text{'age'}\}$ and the codomain $Y = \{\text{'Alice'}, 12\}$. Thus, looking up an attribute value is equivalent to calling tuple function t_1 with the attribute name, e.g., $t_1(\text{'age'}) = 12$.

The idea to model a tuple as a function is in line with database theory [1, 41]. Yet, all these works and also Codd himself [12] then proceed to model a relation as a **set** of tuples. In contrast to that, we proceed to model relations and databases as *functions*.

Computed Functions. A function does not have to explicitly enumerate the concrete mappings. For instance, we could define a tuple function t to return a computed attribute value:

$$t(attr : \text{string}) := \begin{cases} 1000 \cdot t_1(\text{'age'}), & \text{if } attr = \text{'salary'}, \\ t_1(attr), & \text{otherwise.} \end{cases}$$

This implies, that the value of an attribute can be computed and is indistinguishable from an attribute that is not computed. Technically, we could also go as far as to model every attribute as a function, but we believe that that would overcomplicate our model[2].

In other words, under this functional model the boundary between data that is *stored* and data that is *computed* is removed. This is a bit unusual from the point of view of a traditional relational model. It implies that a computed attribute value may return something that was never 'inserted' into the database in the sense of a 'stored data item'. In FDM, this may happen at all levels of the data model.

Similarly, traditional NULL-values in RM and SQL, are represented in FDM through a tuple function that is not defined for certain inputs, e.g., if we call $t_1(\text{'profession'})$, t_1 is not defined. This makes it superfluous to introduce artificial NULL-values for non-existing values.

2.4 Relation Functions

Basic Form. Let's assume a function[3] R_1 mapping an input attribute reflecting a customer id *cid* to a tuple function t_{cid}. Here we assume *cid* to be a primary key, however, we could also use any other suitable candidate key or row-id: $R_1(cid : \text{int}) := t_{cid}$.

R_1 is a higher-order function that we call a *relation function*. It represents the data that in the relational model would be kept in a *set* of tuples: a relation. We may call R_1 with an integer input *cid* and receive a tuple function mapped to from that *cid* value. For instance, assume we have two tuple functions:

$$t_1(attr : \text{string}) := \big\{(\text{'name'} : \text{'Alice'}), (\text{'age'} : 12)\big\},$$
$$t_2(attr : \text{string}) := \big\{(\text{'name'} : \text{'Bob'}), (\text{'age'} : 25)\big\}.$$

Now, a call to $R_1(1)$ returns t_1, a call to $R_1(2)$ returns t_2. Calls with $cid \notin \{1, 2\}$ are not defined.

Constraining Relation Functions. In order to represent which concrete tuple functions 'exist' in any relation function R, we may constrain the input domain to allow for specific *cid* values only, e.g.: $R(cid) : X \rightarrow Y$, where $X = \{1, 2\}$. X does not have to be discrete but may be continuous: $R(cid) : X \rightarrow Y$, where $X = [7; 12]$. This implies that a relation function may represent a continuous data space that is not just a discrete set (as in the relational model) but a continuous subspace of generated 'tuple functions'. In general, domain constraints can be used to *type* functions and express integrity constraints on their domains and codomains.

2.5 Logical Indexes

The relational model comes with the possibility to add additional constraints like unique constraints. In addition, relational DBMSs allow for adding indexes. Both concepts are generalized by relation functions which provide a *logical index*. Assume we define a second relation function: $R_2(\text{age} : \text{int}) := t_{age}$. This provides an alternative view on the tuple functions already returned by $R_1(cid : \text{int})$, however organized by attribute *age*. This implies, that different relation functions may map to the same tuple function. This is in sharp contrast to the relational model. In fact, in FDM the same tuple function may even be returned for different parameters of the **same** relation function (**non injective mapping**). We can easily translate this feature of FDM back

[1] or for pivot tables it may be the individual data values of an attribute of the underlying column; for horizontal partitions of an input relation it could be the partitioning key, e.g., the attributes specified in GROUP BY.

[2] In the 1980ies there were a couple of works proposing exactly that. See our discussion in Related Work.

[3] In database literature there is the convention to represent relations with a capital letter and tuples with a lowercase letter. We borrow this notation convention for our functions to facilitate reading even though our functional approach unifies both concepts into one.

to the relational model: simply define tuples to contain all attributes from domain and codomain. This implies uniqueness in the relational model.

For our example relation function R_2, the mathematical definition of a function already guarantees that for each *age*, R_2 may only point to at most one tuple. Thus, *Definition 2 already implicitly provides the unique constraint.* If we want to allow for assigning multiple tuples with a specific input, i.e., duplicates, we need to *explicitly* nest the results returned by the relation function. For instance, assume a third tuple function with a duplicate *age* value:

$$t_3(attr : \text{string}) := \{(\text{'name'} : \text{'Thomas'}), (\text{'age'} : 25)\}.$$

Then, we need to define $R_3(\text{age} : \text{int}) \rightarrow \{TF\}$. In other words, the relation function R_3 returns a set of tuple functions. In a relational DBMS, this is exactly what logical indexes on attributes with duplicates do! FDM includes this idea on a conceptual level already and not as an afterthought or physical design add-on on top of the data model. To be precise, FDM has an inbuilt notion of a *logical* index independent from the index's *physical* realization (B-tree, hash table, etc).

2.6 Computed Relations

Let's define a relation function R_4 as:

$$R_4(cid : \text{int}) := \begin{cases} t_{cid}, & \text{if } cid \in \{1, 2\}, \\ \lambda cid.\{(\text{'name'}: \text{rnd\_str}(cid)), (\text{'age'}:42 \cdot cid)\}, & \text{otherwise.} \end{cases}$$

Here, λ expresses a λ function. In other words, if a predefined tuple function does not exist in R_4, i.e., *cid* is neither 1 nor 2, R_4 returns an anonymous function: a λ-tuple function where the value returned by the name attribute is a pseudo-random (seeded) string. If 'age' is used as a parameter in that tuple function, that tuple function will return $42 \cdot cid$. For instance, let's retrieve a tuple function from R_4 and access one of its attributes in a single expression: $R_4(10)(\text{'age'}) = 420$. In turn, $R_4(2)(\text{'age'}) = 25$.

Relation function $R_4 : X \rightarrow Y$ has the domain $X = int$. Y is a set of tuple functions which can be further typed to be restricted to a specific type of tuple function.

2.7 Database Functions

Let's assume a function mapping an input attribute rel_name to a relation function. For instance,

$$DB(rel\_name : \text{string}) := \{(\text{'myTab'} : R_3), (\text{'Table1'} : R_1), (\text{'Table2'} : R_2)\}.$$

Thus, given the name of a relation, DB returns a relation function. We coin this a *database function.* Just like relation functions, a database function may also return a computed *lambda* function. In other words, DB may return computed relation functions which were never 'stored'. Similar functions can be defined for sets of databases which we omit for space constraints.

2.8 Blurring the lines between the Different Functions

Let's extend our example from above. We keep t_1 but change t_2:

$$t_1(attr : \text{string}) := \{(\text{'name'} : \text{'Alice'}), (\text{'age'} : 12)\},$$
$$t_2(attr : \text{string}) := \{(\text{'name'} : \text{'Bob'}), (\text{'age'} : t_1)\}.$$

Now, t_2 is a higher-order function. Would we still call t_2 'a tuple'? Definitely not in the sense of the first normal form which mandates that attribute values shall be atomic. Yet tuple and even relation nesting has been enabled in SQL starting with SQL 99. Let's extend our example and add a tuple t_4 with an attribute returning a relation function when accessing attribute 'age':

$$t_4(attr : \text{string}) := \{(\text{'name'} : \text{'Tom'}), (\text{'age'} : R)\}.$$

Tuple t_4 semantically feels like adding metadata to a relation. Yet, in the relational model we would still call t_4 'a tuple' and use the modeling constructs available for tuples: *'a relation is a set of tuples'.*

In contrast, in our approach, we use the same modeling construct *at all levels*: functions. Thus, the artificial boundary between tuples, relations, and databases is gone and we could promote t_4 to become part of the codomain of a database function. Whether your 'table' *age* is an actual relation or: *age* is nested in some other table, or *age* represents a collection of databases, you can use the same exact abstractions and thus query language constructs. There is no need anymore to work around this with `ARRAY` [3] domains, MAP [42] domains or other workarounds in your SQL table definitions.

3 Relationship Functions

In FDM, we can express relationships actually very easily.

Definition 4 [Relationship Functions and Predicates]. Given k functions $\text{F}_1, \ldots, \text{F}_k$ with domains $X_1, \ldots, X_k$, a relationship among these functions can be expressed through a *relationship function*: $\text{RF} : X_1 \times X_2 \times \ldots \times X_k \rightarrow Y_{\text{RF}}$.. If $Y_{\text{RF}} == \texttt{bool}$, we call RF a *relationship predicate* indicating whether a relationship exists among $\text{F}_1, \ldots, \text{F}_k$ for a given input.

Figure 2 shows the general idea of a relationship function. In FDM, any relationship among functions can be expressed by creating a *relationship function* having as its input the combined inputs of the participating functions. For our running example, in Figure 1, we have two relation functions customers(*cid*) and products(*pid*). To express a relationship among these two relation functions, we can simply define a function order(*cid*, *pid*). Note that the keys *cid* and *pid* are not part of the returned attributes. Also note that we may use any suitable key. In addition, we do not necessarily imply that order(*cid*, *pid*) is *stored* as a table as in a relational database. The latter is a physical design decision, i.e., FDM may be mapped to a more physical or lower-level data model similar to mapping ERM to the relational model as a separate physiological design step as suggested in [16].

In the conceptual example in Figure 2, there are three functions $f : X_f \rightarrow Y_f$, $g : X_g \rightarrow Y_g$, and $h : X_h \rightarrow Y_h$. Now we define a *relationship function* $m : X_f \times X_g \times X_h \rightarrow Y_m$. It shares domains with all three functions f, g, and h. This already implies the traditional foreign key constraints we would have to enforce in the relational model/SQL as separate constraints: there, the intermediate m-relation, e.g., 'order' in Figure 1, may only use foreign keys that exist in f, e.g., 'customers', and g, e.g., 'products'. In FDM, we enforce these constraints as a side effect by simply making functions share the same domains. Note that Definition 4 supports relations between attributes, relations (**not** only their tuples), databases, etc. See Figure 3 for an example: we use a database function DB: $\text{str} \rightarrow \text{Y}_{\text{DB}}$ where Y_{DB} is a set of relation functions, and a relation function users: $\text{int} \rightarrow \text{Y}_{\text{users}}$. Both functions are connected through a relationship function

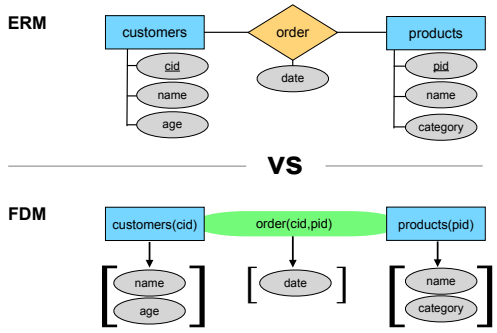

Figure 1: Traditional entity-relationship diagram vs relationship function representation. We use [.] as a shorthand for a type definition, i.e. the co-domain mapped to.

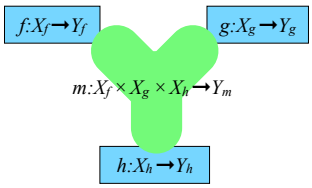

Figure 2: General idea of a relationship function

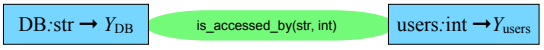

Figure 3: FDM can express a relationship between a database and a relation, cf. with ERM which can only model the metadata of a database.

is_accessed_by: str × int → $Y_{\text{is\_accessed\_by}}$. This relationship function expresses which relation (**not** the relation's metadata!) in the database is accessed by which user, without having to work around this through the database's metadata.

4 FQL

4.1 FQL Operators and FQL Algebra

Definition 5 [FQL Operator]. The general form of an FQL operator is: $Op : \mathbb{F}_{\text{in}} \rightarrow \mathbb{F}_{\text{out}}$ with $\mathtt{f}_{\text{in}} \mapsto \mathtt{f}_{\text{out}}$.

Here, $\mathbb{F}_{\text{in}}$ denotes a domain of input functions, $\mathtt{f}_{\text{in}} \in \mathbb{F}_{\text{in}}$. Likewise, $\mathbb{F}_{\text{out}}$ denotes a co-domain of output functions, $\mathtt{f}_{\text{out}} \in \mathbb{F}_{\text{out}}$. In other words, an FQL operator is simply a higher-order function transforming an input function into an output function. Note that this definition departs from the one used in relational algebra where for instance a join operation has (at most) two input relations. In FQL, this is expressed by using a database function as input that returns two (or more for n-ary joins) relation functions. An FQL operator may take additional parameters.

This is in sharp contrast to relational algebra operators (e.g., π, σ, $\cup$, $\times$, $\bowtie$, ...). FQL operators are neither restricted to representing a single output relation nor restricted to up to two input relations as in relational algebra. FQL operator inputs and outputs can represent any granule including tuples, relations, databases, sets of databases, etc. See Table 1 for the principal classes of FQL operators we can derive from this. Notice that our approach also goes beyond our own recently proposed RESULTDB SQL extension [47][59] in that we cannot only return a relational subdatabase but can also return a database with a completely different schema, e.g., for aggregations.

Definition 6 [FQL Algebra Composition]. Given two FQL operators

$$Op_1 : \mathbb{F}^i_{\text{in}} \rightarrow \mathbb{F}^j_{\text{out}} \text{ with } \mathtt{f}^i_{\text{in}} \mapsto \mathtt{f}^j_{\text{out}}.$$
$$Op_2 : \mathbb{F}^k_{\text{in}} \rightarrow \mathbb{F}^l_{\text{out}} \text{ with } \mathtt{f}^k_{\text{in}} \mapsto \mathtt{f}^l_{\text{out}}.$$

If $\mathbb{F}^j_{\text{out}} = \mathbb{F}^k_{\text{in}}$. Then,

$$Op_2 \circ Op_1 : \mathbb{F}^i_{\text{in}} \rightarrow \mathbb{F}^l_{\text{out}} \text{ with } \mathtt{f}^i_{\text{in}} \mapsto \mathtt{f}^l_{\text{out}}$$

is a valid FQL algebra expression.

We do not imply that FQL expressions are executed in the order the FQL operators are nested. FQL expressions may be rewritten in any way as long as the semantic of the expression is preserved. In the following, we differentiate between important (co-)domains and how this leads to a (natural) classification of FQL operators.

Definition 7 [Subsets of $\mathbb{F}$]. (Co-)domains may be restricted to be easily mappable to existing data models like the relational model. We differentiate between the following important subsets of $\mathbb{F}$:

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

Definition 8 [Order among $\mathbb{TF}$, $\mathbb{RF}$, $\mathbb{DBF}$, $\mathbb{SDBF}$]. Given two sets $\mathbb{F}_{\text{in}}, \mathbb{F}_{\text{out}} \in \{\mathbb{TF}, \mathbb{RF}, \mathbb{DBF}, \mathbb{SDBF}\}$, the order $\text{order}(\mathbb{F}_{\text{in}}, \mathbb{F}_{\text{out}}) \mapsto \{\succ, \prec, \equiv\}$ among these sets is defined as $\text{order}(\mathbb{RF}, \mathbb{TF}) \mapsto \succ$, $\text{order}(\mathbb{DBF}, \mathbb{RF}) \mapsto \succ$, $\text{order}(\mathbb{SDBF}, \mathbb{DBF}) \mapsto \succ$, and $\text{order}(\mathbb{A}, \mathbb{B}) \mapsto \equiv$ if $\mathbb{A} == \mathbb{B}$. If $\text{order}(\mathbb{A}, \mathbb{B}) \mapsto \succ$ then $\text{order}(\mathbb{B}, \mathbb{A}) \mapsto \prec$.

This leads to a landscape of three classes of FQL operators based on the operator's relationship of $\mathbb{F}_{\text{in}}$ and $\mathbb{F}_{\text{out}}$, see Table 1.

4.2 FQL Operator Costumes and SQL Injection

FQL does not impose any new syntax on programmers. We envision FQL as a 'bunch of functions' that use the type system of the embedding programming language (PL). So the same FQL expression may look syntactically different in Python and C++ or Rust. Yet, conceptually that PL syntax can express the same query semantics. We call an FQL operator in a host PL a *function costume*. These functions are **not necessarily** compiled to machine code or interpreted by the PL's compiler and/or runtime. Instead the entire FQL expression or any suitable part of it **may be pushed down** to the database system which can then optimize the expression and return a function (through some streaming interface: ONC, generators, vectorized, etc.). So from the point of view of the developer it looks like as if all those functions were executed in the PL — and in the order specified. However, the underlying team of PL compiler/runtime and DBMS may decide differently. In fact, the entire artificial boundary between SQL and the PL embedding SQL-expressions vanishes. Python code examples for this can be found in a previous version of this paper [17]. That artificial boundary between the PL and the textual SQL string is the root source for **SQL injection** which is ~~the 3rd~~ [44] now: the 2nd [45] most dangerous software weakness. As FQL is embedded into the PL, there is no way to change the boundary between the text belonging to the query and the text belonging to user-provided parameters. Thus, there is no way to

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.

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)

change the semantics of a query. Thus, there is no SQL injection opportunity.

This also opens up interesting future work as now the optimizations done by the PL compiler/interpreter/runtime overlap with the optimizations done with the DBMS and may be done in the **same optimization space**. Thus, we now have the option to delegate parts of the query expression down to the DBMS and leave some to the PL, e.g., depending on runtime statistics or the likelihood that DB optimizations may even have a benefit. In addition, this implies that the **artificial boundary between an embedded PL-statement (a UDF) and the outer query is gone**, too.

4.3 Powerful Updates Through In-Place FQL Usage

Any FQL expression can be used in two different ways:
(1.) Out-of-place Usage: The FQL expression offers a different *perspective* in the sense of a database *view* on the input. It does not change the input function in the underlying relation or database, but based on that input function, the FQL operator returns an output function conceptually reflecting a consistent snapshot. This corresponds to a classical read-only (`SELECT`) query copying data from the database to the outside by means of producing a result set. This leaves the data in the database unchanged.
(2.) In-place Usage: This FQL expression replaces a function in the input FDM. In this mode, the FQL expression is used as a "data rewrite rule". Such a rule may mimic SQL's database DML-operations like `INSERT/UPDATE/DELETE`. However, in SQL, the latter operations are limited to modifying one or multiple tuples of a relation at a time: see the entry marked with (▪▪▪) in Table 1. In contrast, in our approach, we can transform and replace the underlying functions arbitrarily. Conceptually, we can replace entire relations or even databases by simply redefining a function.

5 Practical Challenges

How can we adopt FDM and FQL? The relational model and SQL have been a huge success. They have always been challenged by alternative approaches, e.g., key-value or key-document stores, Hadoop, XML, object-oriented DBMS, etc. Relational DBMSs and SQL also shaped component boundaries/APIs. Yet, they did not shape programming languages (PLs). The dominant PLs are procedural and object-oriented, heavily enriched by features originally invented by and exclusive to functional PLs. As of today, to connect RDBMSes and PLs, the dominant technology is to wrap RDBMS and make them look like object-oriented stores: they hide SQL. This is coined object-relational mapper (ORM). There is a long list of ORMs, e.g., Django ORM [51], SQLAlchemy [53], Hibernate [52], sequelize [61], drizzle [22], etc. The query language constructs they provide are sometimes quite hacky. For instance, to express a simple group by in Django ORM you have to use the `values()` statement which leaves the object-oriented data model and switches to a dictionary [21] Given the history of ORMs and their idea to hide the relational model and SQL, this yields a plan for adopting FDM and FQL. It could look as follows (see Section 7 for the research opportunities that come with that):

(1) **Functional relational mapper (FRM)**: Develop a replacement for an ORM mapping FDM models and FDM queries to an existing relational DBMS (just like an ORM but providing FQL operator costumes which are mapped to the DBMS).
(2) **DBMS interface**: Open up an open source DBMS to directly accept FQL expressions through an API (not as text) mapping them to SQL internally.
(3) **DBMS Storage**: Open up an open source DBMS store to optimize for FDM and FQL and allow for joint optimization.
(4) **Native FDM and FQL**: Provide a native FDM/FQL system.

On Jan 29th, 2026, we started an **open source project** following approach (4), see [24].

6 Related Work

Everything anyone does or proposes in data modeling is build on the shoulders of giants. While some parts of the things put forward in this vision paper have been proposed (multiple) times in previous work, several key aspects have not. Overall, we are

not aware of any previous work framing the idea of a functional data model and (invisible) query language as our vision paper does. In more detail:

Beyond Tables and Single Table Results. See our discussion of [16] and [47] in the Introduction. Note that the idea of returning a subdatabase was also explored from an information retrieval viewpoint outside SQL in [64, 65].

Query Language Alternatives. In terms of relational QLs there were other proposals that basically boil down to proposing a hybrid of RA operators and SQL-style syntax in the PL [38, 48, 57, 63]; a variant of RA in the PL, e.g., XXL [6]; or offer both RA and SQL [68], and/or combine that with a pipe syntax [57, 63]. The most recent call to replace SQL with something more functional is [46]. However, that work builds heavily on relational algebra and allows developers to build pipelines on the relational model through a dot syntax, very similar to Django ORM QuerySets which always starts with one relation and the appends additional relations through joins. Such a syntax is too limited for our data model as we want our operators to be able to express transformations on entire (arbitrary) 'subdatabases' (any nested function expression) rather than just single relations. Moreover, in that work, the underlying RM and its limitations are not questioned. Complementary to our proposal, another recent proposal [25] makes the case for a higher-level, abstract relational query language. There were a couple of other approaches trying to break the RA operator abstraction into suboperators [39], [19], and [4, 36]. In fact, any FQL operator doing 'less' than producing a single output relation can be coined a **suboperator**. Similarly, any FQL operator returning something bigger than a relation could be called a **superoperator**.

Previous Functional Data Models and Languages. In the 1970ies and 1980ies, there has been quite a number of approaches on developing functional data models and languages. See Peter M. Gray's excellent encyclopedia articles [29–31] and book [28] for a start. Though the terms FDM [37, 62] and FQL [7] were the same as the ones we use, FDM back then was more of a triplet data model and RDF "[was] very similar to the Functional Data Model" [30]. Though that principle functional view of data was similar to what we are proposing now; back then, most of the query languages proposed were still textual and considered separate from the programming language, e.g., OSQL [5]. An exception was Adaplex [10] which integrated a query language into Ada, however made use of nested loop syntax to express query semantics rather than more declarative operators like in our FQL. Many of these works can be seen as predecessors to which then became object-oriented DBMS, object-relational DBMSes, and ORMs [56] as workarounds for the mismatch of the relational and the object-oriented world. A more recent approach is glowdust [27] which shares our vision of blending data and computation. Glowdust also models tuples as functions, which is in line with database theory [1, 41], however, similar to database theory, does not provide any higher order functional abstractions for relations and databases. In addition, glowdust proposes another query language which is separate from existing PLs. Moreover, the query capabilities do not go beyond simple filter operations on relations. Other proposals suggest to annotate tuples and relations and then re-define relational algebra operations as mathematical operations on a semiring [32] or ring [35]. However, those works are orthogonal to what we propose and could actually be adapted to work on top of our model and language. More recent approaches integrate lambda-functions into SQL [55, 60]. However, they neither touch the underlying data model nor do they depart from SQL and relational algebra as we are proposing in this paper.

Mixed model approaches. Another line of work tries to marry the relational world with JSON, e.g., SQL++[49] and Oracle [34, 50]. However, as outlined by [47], these approaches cannot represent N:M result sets. All those approaches mix up the underlying conceptual (mathematical) data model (tree structured data) with its representation (the syntax). This common confusion was already discussed in other proposals for unifying data models like [20]. Rel [2, 58] has a very similar motivation as our work. However, in contrast to FDM, Rel keeps the abstraction of relations to be *sets of tuples* and tuples to be *"an ordered immutable sequence of data values"* [58] which we both discard to be functions. In addition, in Rel it is unclear how attributes are typed. In contrast, our approach can directly leverage the typing mechanisms of the embedding PL, e.g., the type hint system in Python wich can even be checked at runtime [33]. Syntax-wise, Rel is quite different from the most common PLs like Rust, C++, Python, etc. and thus is much harder to integrate into those PLs. In the 1980ies, a number of interesting proposals were made to develop query languages directly on ERM [8, 14, 23, 54]. Yet all of these works fall behind our approach in terms of expressiveness.

Criticism of SQL, RM, ERM, and RA. Since the dawn of the relational model [12], SQL [9], and the entity-relationship model [11], there has been criticism [13, 15, 18, 46, 63] as well as several extensions e.g., [66]. However, we are not aware of any other proposal that is as simple and at the same time as powerful as FDM and FQL.

7 A Functional Research Agenda and Conclusion

A lot of exciting **future work** lies ahead, including exploring:

(1) a full-blown implementation of FQL and integration into programming languages like Python and other languages, we just started developing an initial prototype [24].
(2) the list of new FQL operators (Table 1) going beyond SQL,
(3) the joint optimization space of compilers and databases and its impact on program/query optimization and UDFs,
(4) the joint transactional space of compilers and databases and their impact on program/query concurrency control,
(5) the powerful update capabilities of FQL,
(6) how to integrate database optimizers architecturally,
(7) how to integrate database concurrency control architecturally,
(8) how to replace ORMs like Django ORM,
(9) how FDM and FQL may be leveraged for polyglot data stores [26],
(10) how FDM and FQL may be leveraged for tensors (an n-dimensional tensor can be seen as a function having a composite key of its n dimension-indices), and
(11) the usability aspects of our data model [43].

This paper presented a vision for a functional data model (FDM) and a functional query language (FQL). We believe that our proposal — though challenging the successful status quo of the relational model, relational algebra, SQL, and ORMs — comes with a lot of opportunity to view data management from a different viewpoint.

Acknowledgments. I would like to thank the anonymous reviewers; my Ph.D. students Su Yilmaz, Simon Rink, and Luca Gretscher; as well as Wolfgang Gatterbauer for feedback on earlier versions of this work.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley. http://webdam.inria.fr/Alice/

[2] Molham Aref, Paolo Guagliardo, George Kastrinis, Leonid Libkin, Victor Marsault, Wim Martens, Mary McGrath, Filip Murlak, Nathaniel Nystrom, Liat Peterfreund, Allison Rogers, Cristina Sirangelo, Domagoj Vrgoč, David Zhao, and Abdul Zreika. 2025. Rel: A Programming Language for Relational Data *(SIGMOD/PODS '25)*. Association for Computing Machinery, New York, NY, USA, 283–296. doi:10.1145/3722212.3724450

[3] DuckDB Array-syntax. 2025. DuckDB Array. https://duckdb.org/docs/stable/sql/data_types/array. [Online; accessed 28-July-2025].

[4] Maximilian Bandle and Jana Giceva. 2021. Database Technology for the Masses: Sub-Operators as First-Class Entities. *Proc. VLDB Endow.* 14, 11 (2021), 2483–2490.

[5] David Beech. 1988. A foundation for evolution from relational to object databases. In *Advances in Database Technology—EDBT '88*, J. W. Schmidt, S. Ceri, and M. Missikoff (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 251–270.

[6] Jochen Van den Bercken, Björn Blohsfeld, Jens-Peter Dittrich, Jürgen Krämer, Tobias Schäfer, Martin Schneider, and Bernhard Seeger. 2001. XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 39–48.

[7] Peter Buneman and Robert E. Frankel. 1979. FQL - A Functional Query Language. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, Philip A. Bernstein (Ed.). ACM, 52–58. doi:10.1145/582095.582104

[8] Douglas M. Campbell, David W. Embley, and Bogdan D. Czejdo. 1985. A Relationally Complete Query Language for an Entity-Relationship Model. In *Entity-Relationship Approach: The Use of ER Concept in Knowledge Representation, Proceedings of the Fourth International Conference on Entity-Relationship Approach, Chicago, Illinois, USA, 29-30 October 1985*, Peter P. Chen (Ed.). IEEE Computer Society and North-Holland, 90–97.

[9] Donald D. Chamberlin and Raymond F. Boyce. 1974. SEQUEL: A Structured English Query Language. In *Proceedings of 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Michigan, USA, May 1-3, 1974, 2 Volumes*, Gene Altshuler, Randall Rustin, and Bernard D. Plagman (Eds.). ACM, 249–264.

[10] A. Chan, U. Dayal, and S. Fox. 1987. An Ada-compatible distributed database management system. *Proc. IEEE* 75, 5 (1987), 674–694. doi:10.1109/PROC.1987.13781

[11] Peter P. Chen. 1975. The Entity-Relationship Model: Toward a Unified View of Data. In *Proceedings of the International Conference on Very Large Data Bases, September 22-24, 1975, Framingham, Massachusetts, USA*, Douglas S. Kerr (Ed.). ACM, 173.

[12] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (1970), 377–387. doi:10.1145/362384.362685

[13] E. F. Codd. 1971. Further Normalization of the Data Base Relational Model. *Research Report / RJ / IBM / San Jose, California* RJ909 (1971).

[14] Bogdan D. Czejdo, Ramez Elmasri, Marek Rusinkiewicz, and David W. Embley. 1990. A Graphical Data Manipulation Language for an Extended Entity-Relationship Model. *Computer* 23, 3 (1990), 26–36. https://doi.org/10.1109/2.50270

[15] C. J. Date. 1984. A Critique of the SQL Database Language. *SIGMOD Rec.* 14, 3 (1984), 8–54. doi:10.1145/984549.984551

[16] Amol Dehspande. 2025. Beyond Relations: A Case for Elevating to the Entity-Relationship Abstraction. *CIDR '25* (2025).

[17] Jens Dittrich. 2025. A Functional Data Model and Query Language is All You Need. arXiv:2507.20671 [cs.DB] https://arxiv.org/abs/2507.20671

[18] Jens Dittrich. 2025. How to get Rid of SQL, Relational Algebra, the Relational Model, ERM, and ORMs in a Single Paper – A Thought Experiment. arXiv:2504.12953 [cs.DB] https://arxiv.org/abs/2504.12953

[19] Jens Dittrich and Joris Nix. 2020. The Case for Deep Query Optimisation. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org.

[20] Jens-Peter Dittrich and Marcos Antonio Vaz Salles. 2006. iDM: A Unified and Versatile Data Model for Personal Dataspace Management. In *VLDB*. 367–378.

[21] group by example Django ORM. 2025. Django ORM group by example. https://www.pythontutorial.net/django-tutorial/django-group-by/. [Online; accessed 4-December-2025].

[22] drizzle ORM. 2025. drizzle ORM. https://orm.drizzle.team/. [Online; accessed 19-December-2025].

[23] Ramez Elmasri and Gio Wiederhold. 1981. GORDAS: A Formal High-Level Query Language for the Entity-Relationship Model. In *Entity-Relationship Approach to Information Modeling and Analysis, Proceedings of the Second International Conference on the Entity-Relationship Approach (ER'81), Washington, DC, USA, October 12-14, 1981*, Peter P. Chen (Ed.). North-Holland, 49–72.

[24] funqDB. 2026. funqDB github. https://github.com/BigDataAnalyticsGroup/funqdb. [Online; accessed 09-Feb-2026].

[25] Wolfgang Gatterbauer and Diandre Miguel Sabale. 2026. Database Research needs an Abstract Relational Query Language. In *16th Conference on Innovative Data Systems Research, CIDR 2026, Chaminade, CA, USA, January 18-21, 2026*. www.cidrdb.org. https://vldb.org/cidrdb/2026/database-research-needs-an-abstract-relational-query-language.html

[26] Daniel Glake, Felix Kiehn, Mareike Schmidt, Fabian Panse, and Norbert Ritter. 2022. Towards Polyglot Data Stores – Overview and Open Research Questions. arXiv:2204.05779 [cs.DB] https://arxiv.org/abs/2204.05779

[27] glowdust. 2025. glowdust. https://codeberg.org/glowdust/glowdust. [Online; accessed 18-September-2025].

[28] Peter Gray, Larry Kerschberg, Peter King, and Alexandra Poulovassilis. 2004. *The Functional Approach to Data Management: Modeling, Analyzing and Integrating Heterogeneous Data*. doi:10.1007/978-3-662-05372-0

[29] Peter M. D. Gray. 2018. AMOSQL. In *Encyclopedia of Database Systems, Second Edition*, Ling Liu and M. Tamer Özsu (Eds.). Springer. doi:10.1007/978-1-4614-8265-9_1111

[30] Peter M. D. Gray. 2018. Functional Data Model. In *Encyclopedia of Database Systems, Second Edition*, Ling Liu and M. Tamer Özsu (Eds.). Springer. doi:10.1007/978-1-4614-8265-9_173

[31] Peter M. D. Gray. 2018. Functional Query Language. In *Encyclopedia of Database Systems, Second Edition*, Ling Liu and M. Tamer Özsu (Eds.). Springer. doi:10.1007/978-1-4614-8265-9_1092

[32] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Beijing, China) *(PODS '07)*. Association for Computing Machinery, New York, NY, USA, 31–40. doi:10.1145/1265530.1265535

[33] Alex Grönholm. 2025. Typeguard. https://github.com/agronholm/typeguard?tab=readme-ov-file. [Online; accessed 11-July-2025].

[34] Shashank Gugnani, Zhen Hua Liu, Hui Chang, Beda Hammerschmidt, Srinivas Kareenhalli, Kishy Kumar, Tirthankar Lahiri, Ying Lu, Douglas McMahon, Ajit Mylavarapu, Sukhada Pendse, and Ananth Raghavan. 2025. JSON Relational Duality: A Revolutionary Combination of Document, Object, and Relational Models. In *Companion of the 2025 International Conference on Management of Data* (Berlin, Germany) *(SIGMOD/PODS '25)*. Association for Computing Machinery, New York, NY, USA, 431–443. doi:10.1145/3722212.3724441

[35] Christoph Koch. 2010. Incremental query evaluation in a ring of databases. In *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Indianapolis, Indiana, USA) *(PODS '10)*. Association for Computing Machinery, New York, NY, USA, 87–98. doi:10.1145/1807085.1807100

[36] André Kohn, Viktor Leis, and Thomas Neumann. 2021. Building Advanced SQL Analytics From Low-Level Plan Operators. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1001–1013.

[37] K. G. Kulkarni and Malcolm P. Atkinson. 1986. EFDM: Extended Functional Data Model. *Comput. J.* 29, 1 (1986), 38–46. doi:10.1093/COMJNL/29.1.38

[38] Linq. 2025. Linq. https://learn.microsoft.com/en-us/dotnet/csharp/linq/. [Online; accessed 1-April-2025].

[39] Guy M. Lohman. 1988. Grammar-like Functional Rules for Representing Query Optimization Alternatives. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 1-3, 1988*, Haran Boral and Per-Åke Larson (Eds.). ACM Press, 18–27.

[40] Function machine metaphor. 2025. Function Machine Metaphor. https://mathinsight.org/function_machine. [Online; accessed 17-September-2025].

[41] David Maier. 1983. *The Theory of Relational Databases*. Computer Science Press. http://web.cecs.pdx.edu/%7Emaier/TheoryBook/TRD.html

[42] DuckDB Map-syntax. 2025. DuckDB Map. https://duckdb.org/docs/stable/sql/data_types/map.html. [Online; accessed 28-July-2025].

[43] William C. McGee. 1976. On user criteria for data model evaluation. *ACM Trans. Database Syst.* 1, 4 (Dec. 1976), 370–387. doi:10.1145/320493.320504

[44] Mitre. 2024. 2024 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2024/2024_top25_list. [Online; accessed 11-February-2026].

[45] Mitre. 2025. 2025 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2025/2025_cwe_top25.html. [Online; accessed 11-February-2026].

[46] Thomas Neumann and Viktor Leis. 2024. A Critique of Modern SQL and a Proposal Towards a Simple and Expressive Query Language. In *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*.

[47] Joris Nix and Jens Dittrich. 2025. Extending SQL to Return a Subdatabase. *PACMMOD* (2025).

[48] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig latin: a not-so-foreign language for data processing *(SIGMOD '08)*. 1099–1110.

[49] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. 2014. The SQL++ Semi-structured Data Model and Query Language: A Capabilities Survey of SQL-on-Hadoop, NoSQL and NewSQL Databases. *CoRR* abs/1405.3631 (2014). arXiv:1405.3631 http://arxiv.org/abs/1405.3631

[50] Oracle. 2025. JSON Relational Duality. https://blogs.oracle.com/database/post/json-relational-duality-app-dev. [Online; accessed 24-March-2025].

[51] Django ORM. 2025. Django ORM. https://docs.djangoproject.com/en/5.1/topics/db/queries. [Online; accessed 13-March-2025].

[52] Hibernate ORM. 2025. Hibernate ORM. https://hibernate.org/. [Online; accessed 19-December-2025].

[53] SQL Alchemy ORM. 2025. SQL Alchemy ORM. https://www.sqlalchemy.org/. [Online; accessed 19-December-2025].

[54] Christine Parent and Stefano Spaccapietra. 1984. An Entity-Relationship Algebra. In *Proceedings of the First International Conference on Data Engineering, April 24-27, 1984, Los Angeles, California, USA*. IEEE Computer Society, 500–507.

[55] Linnea Passing, Manuel Then, Nina C. Hubig, Harald Lang, Michael Schreier, Stephan Günnemann, Alfons Kemper, and Thomas Neumann. 2017. SQL- and Operator-centric Data Analytics in Relational Main-Memory Databases. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*. OpenProceedings.org, 84–95. doi:10.5441/002/EDBT.2017.09

[56] William J. Premerlani, James E. Rumbaugh, Michael R. Blaha, and Thomas A. Varwig. 1990. An object-oriented relational database. *Commun. ACM* 33, 11 (Nov. 1990), 99–109. doi:10.1145/92755.92772

[57] PRQL. 2025. PRQL. https://github.com/PRQL/prql. [Online; accessed 1-April-2025].

[58] Rel. 2025. Rel. https://docs.relational.ai/rel. [Online; accessed 2-April-2025].

[59] Simon Rink and Jens Dittrich. 2026. Query Optimization for Database-Returning Queries. *PACMMOD* (2026).

[60] Maximilian E. Schüle and Jakob Hornung. 2024. Higher-Order SQL Lambda Functions. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 5622–5628. doi:10.1109/ICDE60146.2024.00450

[61] sequelize ORM. 2025. sequelize ORM. https://sequelize.org/. [Online; accessed 19-December-2025].

[62] David W. Shipman. 1979. The Functional Data Model and the Data Language DAPLEX (Abstract). In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, Philip A. Bernstein (Ed.). ACM, 59. doi:10.1145/582095.582105

[63] Jeff Shute, Shannon Bales, Matthew Brown, Jean-Daniel Browne, Brandon Dolphin, Romit Kudtarkar, Andrey Litvinov, Jingchi Ma, John D. Morcos, Michael Shen, David Wilhite, Xi Wu, and Lulan Yu. 2024. SQL has problems. We can fix them: Pipe syntax in SQL. *Proc. VLDB Endow.* 17, 12 (2024), 4051–4063.

[64] Alkis Simitsis, Georgia Koutrika, Yannis Alexandrakis, and Yannis Ioannidis. 2008. Synthesizing structured text from logical database subsets. In *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology* (Nantes, France) *(EDBT '08)*. Association for Computing Machinery, New York, NY, USA, 428–439. doi:10.1145/1353343.1353396

[65] Alkis Simitsis, Georgia Koutrika, and Yannis Ioannidis. 2008. Précis: from unstructured keywords as queries to structured databases as answers. *The VLDB Journal* 17, 1 (Jan. 2008), 117–149. doi:10.1007/s00778-007-0075-9

[66] Bernhard Thalheim. 2000. *The Entity-Relationship Model*. Springer Berlin Heidelberg, Berlin, Heidelberg, 29–54. doi:10.1007/978-3-662-04058-4_3

[67] Wikipedia. 2025. Function. https://en.wikipedia.org/wiki/Function_(mathematics). [Online; accessed 24-July-2025].

[68] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, Steven D. Gribble and Dina Katabi (Eds.). USENIX Association, 15–28.