ABSTRACT

Database management systems deal with numerous aspects of data storage: data versioning, access control, validation, querying etc. They may do so by offering features that allow the user to specify the data model and express the desired system behaviour: SQL, tables, views, triggers, stored procedures, user-defined types and functions. These features add complexity to the DBMS as well as the data model. We propose the “functional database” as a more generic and elegant solution to dealing with various database aspects. The “functional database” leverages features of the functional paradigm, such as function composition and treating functions as “first-class citizens”.

We have adopted the relational data model as foundation for our functional database but we reckon other data models may be just as adequate. The relational model is implemented with a number of abstractions (context-dependence, lifting and decorating) to allow composition. Other database aspects, such as data validation and access control, may then be implemented with these abstractions, and composed (i.e., “stacked”) to create a more complete database system.

Keywords

Functional relational database, Relational data model, Access control, Data versioning, Context-dependence, Lifting, Decorating

1. INTRODUCTION

Databases are used to persist data about some state of affairs in the world, organised in some data model (e.g., the relational data model[4]). Database management systems (DBMSs) offer functionality beyond mere data persistence and an interface to create, read, update and delete (“CRUD”) data. Consider aspects such as a query language, data validation and versioning, user authentication and authorisation, transactions, user-defined types and functions and so on. Most modern DBMSs offer these features as separate components that require the user to deal with more complexity and new APIs. Moreover, data and operations thereon are usually separated across different systems: tables, views, triggers, stored procedures, user-defined types and functions etc.

We consider this lack of cohesion undesirable; we would rather see a generic way to describe data and all functions on it. It is our wish that database management systems (DBMSs) may someday be sufficiently expressive yet simple that the ubiquitous “user applications” as intermediary between human users and the DBMS may become obsolete. Consequentially, with this one generic solution to data persistence and interaction, data in databases could be shared with and browsed by the public as easily as they currently navigate the World Wide Web.

We think the functional paradigm can be used to model a “functional database” that allows many of the aforementioned database “aspects” to be implemented more generically and elegantly than in existing non-functional databases, by leveraging the concepts of function composition and first-class functions. Function composition is the application of one function to the results of another; or, in colloquial terms, “pipe-lining” of values through a chain of functions. The functional paradigm treats functions as “first-class citizens” which, in colloquial terms, means that “functions” are value types, like the plain integer, strings and floats are. Functional programming languages such as Haskell have a number of additional features such as lazy evaluation and purity. Lazy evaluation delays or prevents evaluation until and unless the result is absolutely required (e.g., doing I/O). This may allow for more compact expressions and performance benefits in some scenarios. Purity prevents functions from having side-effects, unless it is made explicit side-effects may occur.

When Codd introduced[4] the relational data model in 1970, state of the art database systems organised data with a tree structure or network model. Codd recognized inadequacies to these models and presented his relational data model as superior to graph and network data models, because the relational model does not “superimpose” any additional structure for machine representation purposes. Although we think our “functional database” can work with other data models, this paper builds upon the relational model as foundation for the “functional database” for the same reasons Codd presented. Thus, one may also say this paper describes a “functional relational database”.

1.1 Research questions

The purpose of this research is to get a holistic view on the strengths and feasibility of a functional (relational) database. The main question of this paper is: “How feasible is using the functional paradigm in databases and what are its strengths?”

To measure the feasibility of our composable functional database, we will express how a number of database “aspects” may be implemented with it. To measure the strengths of our composable functional database design, we will compare these implementations with their non-functional equiv-
alents on terms of compactness (more coherence of data and functions, less code), ease (fewer concepts and separate systems to manage) and flexibility (fewer modelling constraints).

We have identified a number of database aspects and selected: the relational data model, access control and data versioning. For every aspect we present an introduction, a way to represent each aspect in a functional database and a verification by means of examples and comparisons with non-functional representations. These sections answer the following questions:

1. What is known about the aspect? E.g.,
   (a) What trade-offs and challenges are known?
   (b) What existing (commercial) solutions are known?
   (c) Has this aspect been linked to functional programming already?

2. How could this aspect be expressed functionally?

3. What might such a functional expression look like? (Example)

4. How does the functional expression compare to existing non-functional implementations in terms of compactness, flexibility and ease?

We will investigate these aspects one at a time. The first and most fundamental aspect, the relational data model, will be investigated more thoroughly. In the elaboration on this aspect, we will also introduce the necessary concepts to implement the other aspects on top of this basis. The other aspects will be investigated less rigorously and serve to illustrate usefulness and feasibility of the greater concept: the functional (relational) database.

2. RELATION TO PREVIOUS WORK

Many programming languages have some library to interact with (external) database management systems. A Haskell library for this purpose is HaskellDB[9, 3]. We considered using HaskellDB as an API to our function database; however, HaskellDB assumes the common tables-with-records perception of databases, which is at odds with our composable and abstract design.

DAPLEX[13] is a “data definition and manipulation language for database systems” and uses the functional data model. In DAPLEX, data is modelled in terms of entities: relationships are functions; entity types are functions without arguments. We also use functions to represent data and relations, but our paper uses the relational data model and focusses on the design of a composable architecture.

Functional programming is used in database courses[8, 14] to teach students how to program a simple relational database management system. This illustrates the strength and feasibility of the functional paradigm to model ordinary relational databases. These implementations lack the compositability and abstractions we propose, however.

A completely different approach was taken by Kiyoki et al. They used[7] functional programming concepts “in order to manage processor resources and memory resources with the theoretical neatness of functional computation”. They use demand-driven (i.e., ‘lazy’) evaluation to parallelise the execution of database operations. Their research shows that a functional approach to databases may yield performance benefits.

3. STRUCTURE

We will elaborate on each of the selected aspects (i.e., relational data model, access control and data versioning) in the following three sections. Each section answers the research questions (see §1.1) in that order. The first question is answered in the section introductions; questions 2, 3 and 4 have their own subsection. The architecture of the functional relational database will be described in the aspect Relational data model. It will introduce the functional paradigm into the relational model and express the relational model into the functional paradigm; concepts such as “context-dependence”, “lifting” and “decorating” will be introduced to present a composable foundation of the functional relational database. The aspects Access control and Data versioning will then be implemented with the composable foundation described earlier.

Finally, we will conclude this paper in section 4 with a general assessment of the usability and feasibility of the functional relational database and provide pointers for further research.

4. RELATIONAL DATA MODEL

The relational data model[4] has a mathematical foundation and can be described as follows: a relation (or table) on n sets \((S_1, S_2, \ldots, S_n)\) is a set of n-tuples (with each tuple of the form \(v_1, v_2, \ldots, v_n\) and \(v_i \in S_i, \forall 1 \leq i \leq n\)). Each \(S_i\) is called a domain, column or attribute; each n-tuple is known as a row or record. A database has a collection of relations.

Commercial systems nowadays do not cover the full breadth of Codd’s concepts. Database models still include technical details (e.g., limited string length, indexed columns) and offer only a select set of primary data types (e.g., numeric, varchar, char(n)), although modern systems support more complex types and user-defined types. Functions and commands in databases may be supported through user-defined functions and stored procedures respectively. User-defined functions do not generally support partial application. Stored procedures may be used for data validation and access control (e.g., as “triggers”), but the procedure languages are often vendor-specific.

The relational data model does not constrain the domain types, such that one may define their own[11]. In essence, a functional database would treat “functions” as first-class citizens; i.e., functions are treated as values and can be saved and passed to other functions. Almendros-Jiménez and Becerra-Terón describe[1] how data relations, values and queries may be described functionally (e.g., record non-key values as function of the primary key). We could implement this by defining a “function-valued domain” as a data type. Existing (relational) database systems (that we know of) do not offer such a data type. There is an initiative[10] to translate functional expressions to Java byte code which could be saved in databases as binary data.

4.1 Design

In order to describe the essence of a functional relational database, we will express the relational model functionally and we will describe how functions can be expressed in the relational data model.
4.1.1 Expressing the relational model functionally
The relational data model was defined from the perspective of domains ("column-wise"), yet commercial database systems generally assume a more tuple-oriented perspective ("row-wise"). We see benefits to both perspectives, but have found the column-wise perspective to be more compatible for the purposes of this research. This perspective is adequate for our intention to generalise aspects and functionalities may be enabled this way. Fig.

\[ F_c. \] Each set of functions (\( F_{db} \), each \( F_r \) and each \( F_d \)) contains at least four characteristic functions: create, read, update and delete. We defined three abstractions concerned with these CRUD functions: context-dependence, lifting and decorating.

\textbf{Context-dependence.}\n
The CRUD functions must be provided arguments to determine control flow, such as a relation name and primary key value to locate the data to operate on. One might declare parameters for these arguments on the CRUD functions themselves, but we have opted to declare only one parameter of type Context for each CRUD function, such that they are of the form \( f: \text{Context} \rightarrow \ast \). This general Context type (e.g., a dictionary or a Haskell record type) may contain any necessary information for any database query or command. Some database features may require additional arguments to be supplied with queries and commands (e.g., user name, time and date of the query); we call such aspects context-dependent because that information is conveyed with the Context. This composable approach avoids the otherwise necessary rewriting of all CRUD functions to accept additional arguments.

\textbf{Lifting.}\n
Let us consider the read function for each "level", starting with the relation level (\( F_r \)) in the middle. The primary key of a relation is the domain that (by definition) uniquely selects a single tuple therein. In other words, each relation tuple is a function of the primary key. Therefore, the read function in \( F_r \) may be given a primary key value and return one tuple \( t \in r \). This tuple \( t \) consists of values in domains \( \mathcal{D}_r \), that is \( t = (t_1, t_2, \ldots, t_s) \subset d_1 \times d_2 \times \cdots \times d_s \) with \( t_i \in d_i \in \mathcal{D}_r \). So, how are the values of \( t_i \) deduced?

To that end, let us now consider the read function on the domain level (\( F_d \)). For each domain \( d \in \mathcal{D}_r \), this function abstractly represents all data therein. The read function for each non-primary-key domain is a function of the value for the primary key domain, following from the definition of "primary key". Given a primary key value, the function will return the according domain value (if the primary key exists and has a value for this domain) or throws an error (otherwise). The read function for primary key domains is an identity function: given the value of a primary key, it returns itself (if the record exists) or throws an error.

Notice how the read functions for the domains are “lifted” to the read function of the relation to form a tuple. The other functions (create, update and delete) are lifted analogously. We can also lift the CRUD functions from relations to the database. E.g., the read function on the database level (\( F_{db} \)) must be supplied with the name of some relation (i.e., \( \text{name}_r, \mathcal{D}_r \)) to look up the appropriate relation and invoke the read function on that relation (if it exists). See figure 1 for an illustration of “lifting” (the “context” is explained later).

\textbf{Decorating.}\n
Apart from lifting the CRUD functions to higher database layers, CRUD functions may also compose (or decorate) other CRUD functions on the same level. For example: one may decorate create and update functions on the domain level with other create and update functions; the latter functions validate the input and continue to the "original" function if and only if validation succeeds, thereby embedding validation logic in the domain. Various database aspects and functionalities may be enabled this way. Fig.
Implementations.

Jaco ter Braak has implemented\[2\] a proof-of-concept functional relational database that implements the aforementioned computability abstractions. Production-ready functional databases should offer a user interface to inspect and modify the “decoration chain” of CRUD functions and to inspect and modify the context values of queries and commands. With such an interface, the user can easily compose the database functionality he requires. Furthermore, the high-level declarative design that is discussed in this paper may benefit from modifications in practice, to optimise performance.

4.1.2 Expressing the function relationally

To treat a function as first-class citizen, we need to model some “function-valued domain” type and integrate it with ordinary database operations. Osborn and Heaven identified\[11\] three different types of operations that may be specified for an Abstract Data Type in their relational database system: primitive operations, aggregates and transformations. Although function-valued domains may require further provisions, we will borrow the set of requirements they defined to show what integration of a function-valued domain in the relational model may look like. According to their paper, primitive operations include constants (textual literals that may appear in queries), comparisons, inserting and updating values, displaying values and predicates (an n-ary function returning a Boolean). Aggregates are mappings from a single relation to a single value of a specified data type, such as count, sum, max, min and average. Transformations map one relation to another relation.

Functions can be easily specified textually (i.e., as “constants”) with the syntax of a functional language, such as Haskell or (LISP-like) S-expressions. Functions may be displayed as a string of the expression or as a string of the argument types and return type (the “signature”); alternatively, functions may first be evaluated and the result (either a primitive type or a partially applied function) may be displayed. There are no special predicates for this type, as any predicate \( P : s \rightarrow \text{Bool} \) is a function and may thus be described in the function type itself and simply evaluated.

Functions may be compared by comparing the evaluation of the function: if the result is a primitive type (integer, string etc.) evaluation is trivial; if the result is a partially applied function, comparisons may fail. Functions and values may be “physically” inserted and updated as a string or as binary data\[10\].

The vital function application operation (denoted explicitly by “\( \_ \_ \_ \_ \), or implicitly by whitespace) must be offered by the system. Suppose \( x : \text{Int} \rightarrow \text{String} \rightarrow \text{Bool} \) and \( y : \text{Int} \), then the application of \( y \) to \( x \) is \( x \_y \) : \( \text{String} \rightarrow \text{Bool} \). Moreover, functions that yield some primitive type must be compatible with the equivalent primitive type of the database system (e.g., the functional String must be compatible with the database primitive varchar), if the functional relational database system is built on top of a non-functional relational database system. Compatibility means comparability and implicit conversion between two equivalent types.

In functional languages, higher-order functions (e.g., map, foldr, filter) are ubiquitous. The existence of these functions rely on fundamental language features such as treating functions as first-class citizens and the function application operation.

Aggregates can use higher-order functions. For instance, \( \text{count } xs = \text{foldr} (+) 0 \ (\text{map} \ (x \rightarrow 1) \ xs) \) may express the \( \text{count} \) function is Haskell, using two higher-order functions (i.e., foldr and map). So, analogous to predicates, aggregates can be described in the function itself.

Transformations require a way to specify relations and domains. For instance, \( \text{toR} \ (a, b, c, d) = (d, a, c, b) \) may express a conversion from a relation \( R \) with domains \( a, b, c, d \) and converts it to a relation \( R^\prime \) with (in this case) a different order of those domains. Far more elaborate transformations are possible; relational algebra operations (e.g., projection, join) may be implemented as transformations.

If relations and domains can be specified from within the function ADT, such transformations can be described in the function itself too.

4.2 Example

Let there be a relation \( R \) describing persons with:

Domain id: \( \text{Nat} \) is the primary key and contains arbitrary natural numbers;

Domain \( \text{firstName}: \text{String} \) contains the first name of some person;

Domain \( \text{lastName}: \text{String} \) contains the last name of some person;

Domain \( \text{fullName}: \rightarrow \text{String} \) contains the concatenation of first and last name of some person, separated by a whitespace;

Domain \( \text{dateOfBirth}: \text{Date} \) contains the date some person was born;

Domain \( \text{age}: \rightarrow \text{Nat} \) describes the current age of some person;

Domain \( \text{dateOfDeath}: \rightarrow \text{Maybe} \ \text{DateTime} \) may contain a date of death for some person;

Domain \( \text{isAlive}: \rightarrow \text{Bool} \) determines whether a person is currently alive or not.

In the following three paragraphs we will transform a modelling of this relation from an ordinary table to (ultimately) a functional expression in accordance with the functional database architecture specified in § 4.1.2.

Ordinary database table.

In an ordinary relational database, this table (see figure 3) is finite and static. Each cell value must be precomputed in advance. Integrating functional domains (i.e., the values are ‘calculated’) in the table is problematic. \( \text{fullName} \) can be depicted without loss of information a non-functional String domain, although consistency guarantees will be compromised; what would happen if \( \text{firstName} \) or \( \text{lastName} \) changes? The domain \( \text{isAlive} \) could (analogously) be implemented as a mere flag denoting whether \( \text{dateOfDeath} \) is \text{Null} or not, but it cannot answer queries such as “What people were alive at 1950/01/01?” Moreover, it is necessary to update \( \text{isAlive} \) whenever \( \text{dateOfDeath} \) is modified. The domain \( \text{age} \) suffers from the same problem, but would also require a procedure to check and possibly increment the \( \text{age} \) for all people every 24 hours.
Figure 3. $R_1$ as an ordinary table. The dots (\ldots) represent some arbitrary but finite continuation of table records.

\begin{tabular}{|c|c|c|c|c|c|}
\hline
id & firstName & lastName & fullName & dateOfBirth & dateOfDeath \\
\hline
42 & Nomen & Nescio & Nomen Nescio & 1972/06/13 & Nothing \\
43 & Onbekende & Persoon & Onbekende Persoon & 1892/01/01 & Just 1936/12/31 \\
44 & A & B & A B & 1888/08/08 & Just 1958/08/08 \\
45 & Jaco & ter Braak & Jaco ter Braak & 1988/12/07 & Nothing \\
\hline
\end{tabular}

\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
id & firstName & lastName & fullName & dateOfBirth & age & dateOfDeath & isAlive \\
\hline
42 & Nomen & Nescio & Nomen Nescio & 1972/06/13 & & & \\
43 & Onbekende & Persoon & Onbekende Persoon & 1892/01/01 & & & \\
44 & A & B & A B & 1888/08/08 & & & \\
45 & Jaco & ter Braak & Jaco ter Braak & 1988/12/07 & & & \\
\hline
\end{tabular}

where

\begin{align*}
\text{fullName} &= \text{firstName}\_\text{lastName} \\
\text{age} &= \lambda \text{now} \to \text{max}\{0, \text{years}(\text{min}\{\text{now}, \text{dateOfDeath}\} - \text{dateOfBirth})\} \\
\text{isAlive} &= \lambda \text{now} \to (\text{dateOfBirth} \leq \text{now}) \land (\text{dateOfDeath} = \text{Nothing} \lor \text{now} < \text{fromJust dateOfDeath})
\end{align*}

Figure 4. $R_2$ as a table with functions. The dots (\ldots) represent some arbitrary but finite continuation of table records.

\begin{align*}
\text{id}_c &= \text{pk}(c) \ , \text{if} \ \text{pk}(c) \in \{42, 43, \ldots\} \ \
&= \text{error} \text{“Does not exist”} \ , \text{otherwise} \\
\text{firstName}_c &= \text{case} \ \text{pk}(c) \text{ of} \\
& \quad 42 \to \text{“Nomen”} \\
& \quad 43 \to \text{“Onbekende”} \\
& \quad \ldots \\
& \quad \to \text{error} \text{“Does not exist”} \\
\text{lastName}_c &= \text{case} \ \text{pk}(c) \text{ of} \ldots \\
\text{fullName}_c &= \text{firstName}(c) \_ \text{lastName}(c) \\
\text{dateOfBirth}_c &= \text{case} \ \text{pk}(c) \text{ of} \ldots \\
\text{age}_c &= \text{max}\{0, \text{years}(\text{min}\{\text{now}(c), \text{dateOfDeath}(c)\} - \text{dateOfBirth}(c))\} \\
\text{dateOfDeath}_c &= \text{case} \ \text{pk}(c) \text{ of} \ldots \\
\text{isAlive}_c &= [\text{dateOfBirth}(c) \leq \text{now}(c)] \land [\text{dateOfDeath}(c) = \text{Nothing} \lor \text{now}(c) < \text{fromJust dateOfDeath}(c)]
\end{align*}

Figure 5. $R_3$ as read functions for each domain, accepting a Context $c$. For readability purposes the functions have been named after their domain, instead of (projections on) read$_c$ for each domain.
A working solution would be to exclude those domains from the persisted table, and include the computed domains in a separate query. State of the art database systems allow for the definition of computed columns in the persisted table, which is an even better solution because it does not fragment the relation. Computed columns are specified as a function without parameters and yield a non-functional value. `full Name` can be properly expressed as a computed column. But queries such as “What people were alive at 1950/01/01?” still cannot be deduced from `is Alive`: how would one convey the reference date? By creating `user-defined functions` that do accept a reference date parameter for the computed domains, this query could be answered. However, one would need to define these functions and a query to invoke these functions, which again separates functions from the relation they belong to.

**Functional data type.**

With the function-valued data type, we would be able to store functions in cells and so we can describe the value of a cell as a function rather than a plain number (see figure 4). The functional domains could be included in the relation as computed columns specified with lambda-expressions. For example, `age` is a lambda expression that accepts a reference date and calculates the age of some person at the specified reference date. This form has sufficient expressiveness for the example. However, the modelling is not uniform: there are ordinary data columns and there are calculated columns. This special treatment for functions is undesirable for our purposes to present a compact generic approach to relational databases.

**Functional relations.**

One further transformation (see figure 5) yields a relation in which we have abstracted the tabular expression of a relation to a functional expression thereof, in accordance with §4.1.1. The expressive power of this functional representation of relations is even greater than the functional tabular representation. Due to describing domains with uniform CRUD functions, we may decorate these functions as described earlier to enable various database functionalities and aspects (e.g., a validation decorator or an authorisation decorator), completely invisible from the client side. We will further investigate these aspects in the rest of this paper.

### 4.3 Comparison

The implementation of the functional relational database is characterised by three abstractions on the elementary CRUD operations: context-dependence, lifting and decorating.

The functional specification of relations may be less compact than the non-functional specification in common scenarios. Each domain must be specified in terms of the CRUD actions. Moreover, each non-key domain must be explicitly specified as a function of its primary key. On the other hand, the expressive power of the functional specification and the abstractions remove the need for some other constructs (e.g., stored procedures, user-defined functions, views, calculated columns), improving compactness in other more advanced scenarios. Further improvements can be made in cases where domain values can be predicted or calculated from other values; one functional expression may then yield values for many fields.

Functionally specified relations may be easier to manage than non-functionally specified relations. The characteristic abstractions (i.e., context-dependence, lifting and decorating) enable the composability of a functional database. A composable system with a small set of supporting concepts and generic abstractions is easier to manage than a system without such generic building blocks (i.e., ordinary relational databases with many separate constructs and extensions).

Relations are more flexible specified functionally than non-functionally. Due to treating functions as first-class citizens, domain values may be plain values or calculations, indistinguishable to the client. Moreover, with support for function-valued domains the database may persist lambda abstractions and apply them in queries and commands. Also, the characteristic abstractions allows the behaviour of CRUD functions on databases, relations and domains to be flexible altered and adapted to new requirements.

In conclusion, the functional relational data model may compromise on compactness for common scenarios, but offers in return more ease, flexibility and compactness for more advanced scenarios.

### 5. ACCESS CONTROL

Multilevel relational database systems store data (e.g., a field, record, domain, relation etc.) with different security classifications (e.g., labels such as ‘Secret’, ‘Top secret’ etc.) and only allow users with proper clearances access thereto. Denning describes[5] a database system where a “trusted filter” is implemented, intercepting all access to the database and enforcing the multilevel security requirements: authenticating users and preventing unauthorised data exchange. Authentication is the act of verifying if identifying users are (indeed) who they claim to be. Authorisation is the act of giving users clearance to do certain actions, i.e., CRUD actions at various layers (database, relation, domain).

#### 5.1 Design

The processes of authentication and authorisation can be described and implemented as validation functions that yield a Boolean denoting success (true) or failure (false). The system administrator may implemented these functions to fit his requirements.

Decorations of CRUD functions on the database, relations or domains would invoke these validation functions and continue if and only if the validation (i.e., authentication or authorisation) succeeded. By decorating the CRUD functions, the access control logic becomes an integral part of the data it protects. The decoration also isolates the user interface from the data; the filter cannot be bypassed, unless it is explicitly disabled altogether. This holds true, even for subqueries: if some query reads some database, relation or domain, any and all decorations composed on the actual read-function of that item are evaluated first, including the access control filter.

**Authentication.**

The process of authentication may be modelled by decorating the CRUD functions at the database level ($F_a$). The authentication “check” function would likely require some “context variables”, such as the user’s name and password (hash) or certificate and IP address. Of course, the database management system must be adapted to supply these context variables, but the actual logic is fully embedded in the database itself. The check function might look-up whether the username exists and the passwords
match or not in some relation UserAccounts that the implementer defined. Alternatively, an external authentication service may be invoked.

**Privacy.**

Functional languages may support “partial application” (lambda expressions) and “lazy evaluation”. Lambda expressions are ubiquitous in functional databases and must surely be expressible in some format that can be shared between client and server, for example: a textual representation (e.g., S-expressions) of the expression tree. A database system may also benefit greatly from lazy evaluation, which is used in some functional languages such as Haskell. At some point, it may even be deemed profitable to (somehow) return a partially evaluated expression to clients instead of a fully evaluated answer, to balance computational load on clients and server. However, by revealing an expression tree instead of an evaluated answer, details about the inner workings of the database may be revealed (e.g., the authorisation mechanism). It must be guaranteed that any data transferred to the client is devoid of such sensitive data. By default, the system might only permit fully evaluated results; the client would only be allowed to query a fully applied expression.

Another major component of the aspect of privacy is which users may view what data. These constraints may be enforced with a proper authorisation scheme.

**Authorisation.**

Analogous to authentication, the process of authorisation may be modelled by decorating CRUD functions. Depending on the complexity of the authorisation policy, any of the database levels (database, relation, domain) may be modified. Policies may range from a simple “authorisation implies full access” policy to “read-only privileges for users with the role of ‘Clown’ for some specific domain in a specific relation” or could include validation of data security classifications and checksums, as Denning describes for a multilevel database. Alternatively, the system could connect to external databases or authorisation services.

### 5.2 Example

The following example models the essence of access control; checksums, data classifications and user credentials are omitted for simplicity.

Let $U = \{\text{uname} : \text{String}, \text{upassw} : \text{String}\}$ be the relation of registered user accounts in a system. In order to authenticate a user, let the Context include a field for the username $\text{uname}'$ and password-hash $\text{upassw}'$. Let us define a global helper function $\text{isAuthentic} : \text{String} \rightarrow \text{Bool}$ that queries relation $U$ for the existence of the user with $\text{uname} = \text{uname}'$ and determines whether $\text{upassw} = \text{upassw}'$ for that user. The read function of $F_{ab}$ may be:

$$read\ c = \text{actualRead} \ (c \& \{\text{authentic} = \text{authen} \ c\})$$

where

$$\text{authen} \ c \ = \ \text{uname}' \ c \ \neq \ \text{empty} \ \land \ \text{isAuthentic} \ (\text{uname}' \ c) \ (\text{upassw}' \ c)$$

$$\text{actualRead} \ c \ = \ \text{the \ real \ read \ function}$$

The create, update and delete functions are analogous to the read function.

Our example system includes an “invite only” registration system. We want to authorise every authenticated user to create a new user account in $U$; unauthenticated users may not do so. We want to authorise all authenticated users to read the entire domain $\text{uname} \in D_U$, but only their own password in domain $\text{upassw} \in D_U$. Authenticated users may only update and delete their own account.

In order to guarantee that only authenticated users may create accounts, the create function of $F_{ab}$ may be:

$$create\ c = \begin{cases} \text{actualCreate} \ c \ & \text{if author} \ c \\ \text{error} \ “\text{Author. fail}\” \ & \text{otherwise} \end{cases}$$

where

$$\text{author} \ c = \ \text{authentic} \ c$$

$$\text{actualCreate} \ c = \ \text{the \ real \ create \ function}$$

To ensure only authenticated users may read the user name, the read function of $F_{ab_{uname}}$ may be:

$$read\ c = \begin{cases} \text{actualRead} \ c \ & \text{if author} \ c \\ \text{error} \ “\text{Author. fail}\” \ & \text{otherwise} \end{cases}$$

where

$$\text{author} \ c = \ \text{authentic} \ c$$

$$\text{actualRead} \ c = \ \text{the \ real \ read \ function}$$

To make sure authenticated users may only read their own password, the read function of $F_{ab_{uname\ upassw}}$ differs from $F_{ab_{uname}}$ only in the function author:

$$\text{author} \ c = \ \text{authentic} \ c \ \land (\text{uname}' \ c) = (\pi_{uname}(U(c)))$$

The functions update and delete in $F_U$ are defined analogously to $F_{ab_{uname\ upassw}}$.

### 5.3 Comparison

Access control can be implemented with the decoration and context-dependence abstractions: a user includes his credentials in the “context” of a request and some CRUD authentication decoration validates these credentials. Authorisation decorations validate whether some user is authenticated and has sufficient privileges. The exact implementation of the validations is up to the user.

Functionally expressed access control is more compact than non-functional equivalents. No access control frameworks are required. The user only needs to implement access control on the databases, relations and/or domains that he wants to guard; the compactness of the implementation depends mainly on the complexity of the actual access control policy.

Access control is easier to manage functionally than non-functionally. The user is not required to learn and configure vendor-specific access control systems; he may roll his own. By leveraging the lifting abstraction, he may decorate any database level (database, relation, domains) with the same access control scheme.

The aforementioned features also indicate the increased flexibility of access control in a functional database in comparison to non-functional equivalents; access control is “just another function composition”. The implementation of the user validation function is unrestricted: it may
be a simple table-lookup or by using a public key infrastructure. In conclusion, access control benefits from increased compactness, ease and flexibility in functional databases.

6. DATA VERSIONING

Only very few databases are write-once archives; most databases must be updated, corrected and changed regularly to properly represent the state of affairs in the real world. However, by default databases have a timeless data model: data fields in records have values and modification thereof destroys the old values. By using versioning, one may retain old values by translating logical modifications of data to factual insertions of the updated instances. To differentiate between versions, a dimension must be introduced; usually, this is a version number or a timestamp.

One may keep all versions in one table (possibly breaking compatibility due to the newly created domain for version information) or create shadow tables[12] (keeping “current” data in the “base” table and backing up old data to another table, which has additional domains for version information). These solutions require modifications to the data model, require additional functions and triggers to facilitate the actual versioning, and possibly separates the data into multiple tables. Queries and stored procedures may need to be modified as well, to deal with the new domains for version information.

6.1 Design

With a functional database, one may implement versioning seamlessly by introducing context-dependencies for some time dimension (e.g., the valid time). Let the Context include a now variable. By default, the reference time equals the current system time but by explicitly setting the reference time to the past (or even the future!) the client should be able to “walk through time” by doing nothing more than changing a context variable. In other words: queries may be evaluated in another timeframe without needing to adapt the queries.

Versioning may be implemented by decorating the CRUD functions of domains. The read function would yield different results, depending on the value of now. For example:

\[
\text{read} \ c = \begin{cases} 
\text{newRead} \ c & \text{if } \text{Time}(\text{’20110520125100’}) \leq \text{(now)}c \\
\text{old read} \ c & \text{otherwise}
\end{cases}
\]

The functions for create, update and delete must be implemented to logically create, update and delete some value by actually creating new instances for the values the functions were supposed to create, update or delete. These new instances are functions like read’ (with newRead being the actual new value), such that the creation, update or deletion are only effective from the time of change onward; before then, the old unmodified read function must be returned. To avoid “lost modifications”, these functions must ensure that the “time-conditional modifications” are ordered time-descending. In other words: any later modification is evaluated before any earlier modification.

By setting the now context variable to a time before the moment of creation, update or deletion, the client may view old versions of data. Of course, a functional database may offer ways to disable the versioning and actually create, update or delete some entity forever.

6.2 Example

Let \( A = (\text{name} : \text{String}, \text{address} : \text{String}) \) be the relation of people’s addresses. Suppose a person named ‘Donald’ has moved from address ‘Sinistreet 1’ to ‘Dextreet 2’ by 2011/05/20. Then the new read function for the address domain might be read:

\[
\text{read}_2 \ c = \begin{cases} 
\text{Dextreet 2}' & \text{if } (\text{name}c) = \text{’Donald’} \wedge \\
\text{Date}(\text{’20110520’}) \leq \text{(now)c} \\
\text{read}_1 \ c & \text{otherwise}
\end{cases}
\]

\[
\text{read}_1 \ c = \begin{cases} 
\text{Sinistreet 1}' & \text{if } (\text{name}c) = \text{’Donald’} \\
\end{cases}
\]

Reading \( A \) and querying the address for Donald will then yield a different result, depending on the value of context variable now. Notice that versioning may be used to plan future changes too; a planned move by 2111/05/20 from ‘Dextreet 2’ to ‘Ridiculane 3’ may be expressed by committing an update with Date(’21110520’) \( \leq \) (now.c).

In practice, more advanced versioning systems may be required. E.g., [6] describes a model for a multiversion database using (but not restricted to) a bitemporal data model. A bitemporal data model introduces the dimensions valid time and transaction time on data; valid time describes the data is true in the real world whereas transaction time denotes the time of storage. By introducing both ‘times’ as separate context variables, multiversioning might be implemented.

6.3 Comparison

Versioning can be implemented with the decoration and context-dependence abstractions: some valid time context variable is used by a versioning decoration to select the proper version of some value. The functional expression of versioning above may be more compact than doing versioning with ordinary databases. There is no need to create a ‘shadow table’, nor must entire rows be backed-up if only a single value is changed. On the other hand, functional expressions with versioning logic are at itself likely to be more costly than saving plain values with a version number.

Functional versioning may be easier to manage than non-functional versioning. Versioned domains are indistinguishable by the client from non-versioned domains. Versioning can be enabled or disabled without breaking compatibility.

The functionally expressed versioning is more flexible than non-functional versioning, as already indicated by the aforementioned features; versioning is “just another function composition”. The user is not restricted in how to implement versioning: he may enable versioning for merely a subset of the domains in a relation or he may implement more advanced versioning systems.

In conclusion, versioning benefits from increased compactness, ease and flexibility in functional databases.

7. CONCLUSIONS

We have proposed a compositional design of a functional database to describe various database aspects more generically and elegantly than is done by currently available systems. This high-level declarative design may benefit from modifications in practice, to optimise performance.

We abstracted database systems to the essential operations (create, read, update and delete) and implemented the relational data model with those operations. Three abstractions on these operations were introduced. We introduced context-dependence to allow for control flow logic necessary for the implementation of various database aspects. Lifting CRUD functions was introduced, such that
the database layers (database, relation, domain) may let their operation \( X \) be implemented as a composition of the operation \( X \) of lower layers. \textit{Decorating CRUD} functions was introduced, such that database aspects could “intercept” calls to those operations and add program logic to model other database aspects. The functional relational data model showed strengths in ease, flexibility and, to some extent, compactness.

We have reviewed the aspect of \textit{Access control} and found that the aforementioned abstractions were sufficient to implement access control. The functional implementation of access control showed strengths in ease, flexibility and compactness.

The aspect of \textit{Data versioning} was also investigated and we have found that the aforementioned abstractions were sufficient to implement versioning. The functional implementation of data versioning showed strengths in ease, flexibility and compactness.

The main question was “\textit{How feasible is using the functional paradigm in databases and what are its strengths?”}\textit{ We have integrated the functional paradigm with the relational data model in two ways: we defined the functional-valued domain and we designed a composable database architecture. Ter Braak\cite{2} has shown the feasibility of this design. By summarizing the strengths of the three selected aspects’ functional expressions (i.e., the relational data model, access control and data versioning), we find that the functional paradigm has strengths in databases in terms of ease, flexibility and, to some extent, compactness.}

### 7.1 Future work

Early on, we identified many possible aspects to be interesting subjects for research on functional databases. We have only investigated a few, but others may yield interesting results nonetheless:

**Queries** Implementing (and possibly extending) a relational algebra or SQL in a functional database would be one of the most important aspects to investigate. Currently, our design reads values if a primary key is specified but queries are not supported. Expanding the \textit{CRUD} function set with a function \textit{All} might be sufficient to enable query evaluation, but more elaborate changes to the current design may be required.

**Keys** How would the functional database design support multiple keys in a relation? And what about keys that span multiple domains?

**Storage structures** The \textit{lifting} abstraction generalizes all three levels of a relational database (database, relation, domain). The uniform \textit{CRUD} function may be compatible with other data models; for example: the hierarchical tree structure, often used in file systems. The feasibility and strengths of (e.g.) \textit{functional hierarchical} databases would be interesting to research.

**Data validation** Whereas \textit{authorisation} validates whether the user has sufficient \textit{privileges} to execute some action or query, \textit{data validation} validates the \textit{new value} of some ‘\textit{CRUD-able entity’}.

**Data scrubbing/cleansing** Users may insert data in incorrect formats; e.g., too many spaces, wrong capitalization or ‘full name’ is interpreted as ‘First Last’ by person \( A \) and ‘Last, First’ by person \( B \). Instead of refusing to process incorrectly formatted data, the system may reformat such data instead.

**Data conversion** Suppose the database contains temperatures in Kelvin. Users may still (want to) enter temperatures in degrees Celsius or Fahrenheit. The database might automatically convert an input string of ‘10 °C’ to ‘283.15 K’. Another application: a database consists pictures in a specific format (e.g., PNG) and users should be able to either provide a URL or a byte-stream; the system could automatically download and convert the image.

**Encryption** For some applications, databases must not store data in plain unencrypted formats. A functional database might automatically encrypt and decrypt sensitive data, completely invisible to the (authorised) user.

**Transactions** How could multiple invocations of \textit{CRUD} functions form a logical unit of work (i.e., a transaction) in a functional database? Would it be less or more difficult than in ordinary databases? Might data versioning assist a transaction mechanism (namely, all \textit{CRUD} actions in a transaction share the same transaction time)?

**Concurrency** How could multiple transactions be executed simultaneously? Could the ACID properties be met? Could (again) data versioning assist (namely, by discriminating on \textit{valid time}, might we allow \textit{future} modifications to be executed simultaneously with \textit{past} queries)?

**Performance** How well could functional databases perform compared to ordinary databases? The functional database replaces the table-format of a relation by a more abstract representation. This may result in a loss of structure on byte-level; this would hurt performance. On the other hand: features such as lazy evaluation and referential transparency might yield performance \textit{benefits}.

**Schema versioning** Schema versioning might be implemented analogous to data versioning. The \textit{lifting} abstraction already permits any data versioning \textit{domain} decoration to be applied to a relation and thus yield schema versioning. Is this sufficient? Could old and new schemata exist and be used simultaneously and update each other completely invisible to the client?

**Infinite relations, open/closed-world assumptions** The abstractions we have provided seem to allow the specification of infinite relations: a relation \( \langle x, \sin x \rangle \) can be specified functionally. Moreover, one could model a relation with (by default) ‘all people in the universe’ \textit{except for} \( \ldots \): the \textit{read} function would always yield some answer except for a set of explicitly deleted records. Is the current design sufficient? What consequences would infinite relations and world assumption have for \textit{querying}?}

**Temporal Integrity Constraints** Data versioning uses \textit{valid time} to denote the logical time some value is active. Temporal logic might be implemented to re-use these values to allow for the specification temporal logic constraints.
Acknowledgements

We thank dr.ir. Maurice van Keulen and dr.ir. Jan Kuiper for their supervision and feedback during this research project. We also thank Christiaan Baaij M.Sc and students in our peer group for ideas and suggestions they contributed.

8. REFERENCES