# Towards conceptual and logical modelling of NoSQL databases

***Jaroslav Pokorný***
*Charles University*
*Prague, Czech Republic* *pokorny@ksi.mff.cuni.cz*

***Karel Richta***
*College of Polytechnics*
*Jihlava, Czech Republic* *karel.richta@vspj.cz*

**Abstract**

NoSQL databases support the ability to handle large volumes of data in the absence of an explicit data schema. On the other hand, schema information is sometimes essential for applications during data retrieval. Consequently, there are approaches to schema construction in, e.g., the JSON DB and graph DB communities. The difference between a conceptual and database schema is often vague in this case. We use functional constructs – typed attributes for a conceptual view of DB that provide a sufficiently structured approach for expressing semantics of document and graph data. Attribute names are natural language expressions. Such typed functional data objects can be manipulated by terms of a typed λ-calculus, providing powerful nonprocedural query features for considered data structures. The calculus is extendible. Logical, arithmetic, and aggregation functions can be included there. Conceptual and database modelling merge in this case.

**Keywords:** typed lambda calculus, functional data objects, NoSQL databases, relational databases, database schema, conceptual schema, database integration.

## 1. Introduction

Systems that store and process Big Data have become a common component of data management architectures. NoSQL databases (DB) are often used for storing Big Data. Generally, Big Data can be a combination of structured, semi-structured, and unstructured data, i.e.

- structured data in database (DB) and data warehouses based on SQL,
- unstructured data, such as text and document files held in Hadoop clusters or in NoSQL database systems (DBS), and
- semi-structured data, such as web server logs or streaming data from sensors.

To deal with such data in a meaningful way, its users must have a detailed understanding of the available data and possess some semantics of what answers they are looking for to make sure the information they get is valid and up to date.

Today, NoSQL DBs are often used for storing Big Data. Typically, no predefined schema is enforced before writing data into a NoSQL DB. NoSQL DBS are mostly schema-less. This type of data handling is more flexible in the case of Big Data, unstructured data, or frequent schema changes. Emphasize that this does not mean that NoSQL DBs do not have a schema. As opposed to statically typed schemas in SQL DB, a schema-less DB contains rather dynamically typed schema. A schema is here required for reading and understanding data and not for optimizing storage. On the other hand, people working with Big Data are making the typical mistake. They want to access data in the NoSQL DB without any knowledge of its schema, and then often find that their queries do not work, that they do not know the keys for identifying subsets of the required data. In this scenario, the data comes without a schema and its multiple data models coexist. Therefore, there is a gap between data modelling and physical data aspects of NoSQL DB.

Traditionally, data projects that start with data models and data schemas usually behave better than those that do not. That observation remains true in the NoSQL world. However, the big difference with NoSQL is that, because the DB often has no conceptual/database

view at disposal, it is more likely developers will just throw data in and solve problems with its semantics later on. The situation becomes even more complicated in the case when the DB is multi-model, i.e., more NoSQL DBS of different categories are integrated in one DBS. This case was simpler when the integrated databases were relational and equipped with a conceptual/database schema.

There are approaches documenting that conceptual/database data modelling is possible also in NoSQL world. A systematic literature review of the current state of research regarding database design methods in the new database era is performed in [21]. Some NoSQL DBS are based on a Model-Driven Architecture (MDA), i.e., on transformation rules starting from UML diagrams and generating a NoSQL physical data [1]. A generic logical model that describes data according to the common features of the three types of NoSQL systems: column-oriented, document-oriented, and graph-oriented is used here as an intermediate link. Of the newer ones, let us mention the NoAM (NoSQL Abstract Model) [2] presenting a novel abstract data model for NoSQL DB, which exploits the commonalities of various NoSQL systems. NoAM describes in a uniform way the features of many NoSQL systems, and so can be effectively used for an intermediate representation in a NoSQL DB design methodology. A favourite in this area is the Concept and Object Modelling Notation (COMN) [7] enabling to cover the full spectrum of analysis and design, i.e. to represent the objects and concepts in a problem space, logical data design, and concrete NoSQL and SQL DBs implementation.

The traditional problem in the database world is querying. Usual approaches based on SQL and integrated non-relational data types are imported there in some way. Only a few attempts exist enabling to query a database through E-R schemas, e.g., [11]. However, strong means for querying NoSQL database data are still missing. Here we use functional constructs – attributes for a conceptual view of DB and sufficiently structured approach for expressing semantics of NoSQL data considered in this architecture. Conceptual structures have a database role in this approach and can serve as the basis of a query language. In other words, conceptual and database modelling querying merge in this case. We will introduce a formal apparatus for typing attributes and languages of terms based on a typed lambda calculus. Consequently, we obtain a tool covering all NoSQL data models in a unified way, i.e., also a possibility of querying multi-model data in a unified way. Indeed, any relational database can be a member of such multi-model architecture.

The rest of the paper is organized as follows. Section 2 illustrates the features of the main categories of NoSQL DB. In Section 3, we briefly present a history of conceptual modelling. Section 4 is focused on conceptual/databases schemas for some NoSQL data models, particularly JSON and database graphs. In Section 5, we describe a functional type system appropriate for typing conceptual constructs. We also introduce a version of a typed λ-calculus for manipulating these constructs in a query style. Section 6 introduces a variant of conceptual/database schemas for JSON and graph NoSQL data models. Section 7 concludes the paper.

## 2. Categories of NoSQL DB

NoSQL DBS are usually classified into four basic categories in terms of the supported data model: key-value stores, document-based stores, column-based stores, and graph-based stores. Due to the vague semantics of the name NoSQL, XML DB, JSON DB, object stores, and multi-model DB are often considered also as NoSQL categories. Typically, rather low-level tools are used for processing NoSQL data of the first three categories, let us name, e.g., a software framework and programming model MapReduce in the Hadoop framework and the architecture style REST (e.g., [14]).

(i) *Key-value stores* are the simplest NoSQL DB. The *key* is a unique identifier associated with a value. The *value* may have different formats for different key-value systems. In some cases, the value is an array, a JSON structure or a simple string of bytes. However, key-value stores typically abstract from the type of value stored, therefore providing only a simple query functionality rather than using a query language. Searches are performed on keys, not values, and they are restricted to exact matches. Rather, a set of operations usable for application programming is at disposal. Key-value stores are suitable for

efficient reading and writing of extensive amounts of data. They store all key-value pairs together in a single namespace, which is analogous to a relational schema.

As examples, an open-source Redis[1] in-memory DB and Amazon DynamoDB[2] can be considered. SQL-like and MapReduce querying are not supported in Redis, REST querying is supported by some APIs. DynamoDB is a key-value store with added support for JSON data to provide document-like data structures that better match with objects in the application code.

(ii) *Document-oriented* or *document-based* NoSQL systems store data as *collections* of similar documents. Each document reminds a row in relational DB, but in a schema-free context. It is specified as self-describing data. In document-based DB, a record, called a *document*, has its own internal structure and can be stored as a value for a key (*document ID*). In a document DB, documents can be nested. The users or processing applications have the schematic responsibility in such database. This may result in some disadvantages, such as the lack of *integrity constraints* (IC). However, the lack of schema provides high flexibility to store a wide range of data, which makes it suitable for storing Big Data. Documents do not have to be free text, may be structured in various formats, e.g., XML or (mostly) JSON (JavaScript Object Notation), or BSON (Binary JSON). It is possible to store a set of heterogeneous documents inside the same collection.

The MongoDB[3] is a well-known representative of document-oriented NoSQL systems storing data in BSON. MongoDB offers a rich query language having powerful filtering capabilities. MapReduce and REST querying are also supported by MongoDB. Similarly, Apache CouchDB[4] stores JSON documents and offers optionally MapReduce for data retrieval and REST API.

(iii) *Column-based DB* or *wide-column* DB also utilize the key-value concept. Column-based DBs are designed to address the large number of *columns* and frequent changes in the schema. Column-based stores use a table form but in a flexible and scalable way. Each row consists of a key and one or more related columns, which are called *column families*. Each column family has a name that should be declared when the table is created and cannot be changed subsequently. Each column family is associated with column *qualifiers*. Column qualifiers are not predefined but can be dynamically created and inserted into the table. Each column is stored in a separate file or region in the system's storage. Each row's key-column family can have different numbers of columns and the columns can contain different kinds of data. Data is retrieved using a query language. The column structure is appropriate to fast aggregation queries.

The most famous open-source column-oriented DBS are HBase[5] and Hypertable[6]. Both DBS have no SQL-like query language, but MapReduce and REST querying is supported there. On the other hand, Apache Cassandra[7] offers the Cassandra Query Language (CQL) whose query style reminds SQL. Recall that we can meet column-oriented stores also in the relational DBMS world. For example, Vertica[8] is the main representative of this DB category. However, columns are used here only as the way how DBMS stores data.

*Remark*: There are some soft differences between basic constructs of (i) and (iii). For example, DynamoDB supports two types of primary keys: partition keys and composite primary keys given by couples (partition key, sort key). Here partition keys and sort keys can contain only one attribute. On the other hand, Cassandra allows including more than one column (attribute) into partition keys and clustering columns.

(iv) *Graph databases* (GDB) are storage systems that use graph structures to represent and store data. Their *nodes* and *edges* can be labelled to indicate the types of objects and associations they represent. An edge (or a relationship) connects two nodes and may be directed. When directions are added, the relationships between nodes are identified by the

[1] https://redis.io/ (retrieved on 7. 6. 2021)
[2] https://aws.amazon.com/dynamodb/ (retrieved on 7. 6. 2021)
[3] https://www.mongodb.com/ (retrieved on 7. 6. 2021)
[4] https://couchdb.apache.org/ (retrieved on 7. 6. 2021)
[5] https://hbase.apache.org/ (retrieved on 7. 6. 2021)
[6] https://hypertable.org /(retrieved on 7. 6. 2021)
[7] https://cassandra.apache.org/ (retrieved on 7. 6. 2021)
[8] https://www.vertica.com/overview/ (retrieved on 7. 6. 2021)

names of the nodes and can be traversed according to these directions. A *mixed graph* is a graph in which some edges may be directed, and some may be undirected. In each GDB data model, the database is represented as a network structure containing edges between nodes to illustrate the relationships among nodes. GDBs therefore focus on relationships, rather than data. Nodes and edges may also contain *properties* that describe the real data items contained within each object. At the scale of the whole GDB, the data model changes as edges are stored with new kinds of relationship data, and nodes are stored with new kinds of data items. The graphs use the technique, in which each node has explicit references to its adjacent node.

The most popular type of GDB is the open source Neo4j[9] with the query language Cypher. REST querying is supported by Neo4j but MapReduce not.

## 3. Conceptual modelling

In the history of database management systems (DBMS), it is typically their focus on so-called *semantic* (*conceptual*, *entity-oriented*, or *object-oriented*) *database models*. The strength of semantic database models is that they are conceptually oriented. Let us recall the E-R model, which occurs in many variants and is traditionally the basis for the design of most relational databases. It is interesting that the expressive power of a given semantic database model depends on a few number of basic constructs only. Most of the interesting "semantic" constructs in semantic database models are redundant, i.e., they do not increase the expressive power of these models.

The possibility of various ways of modelling the same object system is called *semantic relativism*. The existence of this phenomenon was observed (only as a hypothesis) by Klein and Hirschheim who noted in [9] that the possibilities of all E-R approaches are approximately the same. A fundamental theme arising in semantic database models focuses on the development of formal methods for comparing of the information capabilities of conceptual schemas expressed in these models. For example, work [16] discusses the possibility how to reach it with a functional approach based on typed functional objects and typed λ-calculus.

Another authors give a similar (but imprecise) opinion [8]: semantic database models distinguish each other in that, how many ICs they absorb into basic constructs and how many of them must be formulated explicitly. Many people mean that with powerful tools for formulation of explicit ICs semantic database models offer practically the same chances for conceptual modelling. Unfortunately, this is not true. Really, explicit ICs only "filter" data coming into a DB. The expressiveness of a database model is given by its basic modelling constructs.

Experiences from the last years show that traditional design techniques, e.g., E-R model and UML, cannot be easily used with new database technologies like NoSQL DB. For example, it is not possible to represent data structures that are interrelated without using keys in E-R model. There are some approaches using, e.g., UML [22]. The paper [1] presents an MDA-based approach to implement UML conceptual model describing Big Data in column-oriented NoSQL systems. These approaches are rather conservative and not sufficiently general for NoSQL data modelling. There are also approaches to conceptual modelling document data. For example, the authors of [4] propose a model for the conceptual representation of a document data store based on UML class diagrams and mapping rules for its implementation.

In the era of Big Data, it is essential that we have the tools we need to understand the data coming to us faster than ever before, and to design databases and data processing systems that can adapt easily to changing data schemas and business requirements. Novel NoSQL data organization techniques must be used side-by-side with traditional SQL DB. The above-mentioned COMN model can represent the objects and concepts with simple and familiar graphical notation (boxes and lines). COMN models can also represent the static structure of software and the predicates that represent the patterns of meaning in databases. They enable:

[9] https://neo4j.com/ (retrieved on 7. 6. 2021)

- to think about objects, concepts, types, and classes in the real world, using the ordinary meanings of English words,
- to express logical data designs that are freer from implementation considerations than is possible in any other notation,
- to understand key-value, document, columnar, and table-oriented database design in logical and physical terms,
- COMN is a modelling technique, not just notation. COMN provides a notation to handle all constructs that E-R techniques do not do well, and it steps up to the problem of linking physical and conceptual models.

## 4. Conceptual/database schemas of NoSQL DB

In the NoSQL world, there is a problem for those who are not developers, e.g., of MapReduce programs, to understand the DB content. This is because the schema is hidden in MapReduce program. Therefore, most database administrators and data analysts cannot access and understand these schemas. The user‘s dependence on developers to understand the code significantly reduces his/her possibility to propose changes and expansions of data in such DB. Therefore data modelling is the key for the better understanding of enterprise data and some level of DB schema design is inevitable. In other words, NoSQL follows *query-driven design*. In the style *schema-for-read*, data does not follow any internal schema.

Concerning an integration of data from different databases, i.e., in the multi-model case, two approaches based on database schema management occur [18]:

- top-down – starting with a global schema to design schemas for particular data stores,
- bottom-up – through middleware, i.e., to use schema mapping for schemas of data stores in sites with a middleware (e.g., OLE DB[10], JDBC[11]) and then use a query transformation. Data is loosely integrated and managed by multiple servers.

We remind that the former concerns rather homogenous DB models used in integrated data stores, while the latter supports heterogeneous DB models and consequently heterogeneous DBS, particularly when some of them are NoSQL. The latter variant is more relevant in practice (for detailed discussion, see [18]). Moreover, the relational data model is one of these models. The integration of relational and NoSQL DB is one of the possible ways of bringing them [4]. In the case of NoSQL, even more than one database model is included in one DBMS architecture. For example, Cassandra combines column-based data model and key-value data model, DynamoDB combines document-oriented and key-value data models. ArangoDB[12] represents also a multi-model approach, meaning that it can even address documents, graphs, and key values.

Often XML and JSON data are also considered in the context of NoSQL DB. Both JSON and XML models are “self-describing” and hierarchical. However, JSON is easier to read, less verbose, and brings less constraints than XML. For example, a conceptual model for XML called XSEM is proposed in [13]. It divides the conceptual modelling process into conceptual and structural levels.

Both relational and NoSQL databases offer support for JSON, meaning that hybrid solutions, in the context of the schema, are already at disposal [6]. In Sections 4.1 and 4.2, we shortly summarize some information about JSON and graph schemas, respectively. The conceptual model behind graphs is a variant of E-R model. Functional conceptual variants for JSON and graph data will be presented in Section 6.

### 4.1. JSON schemas

To manage JSON data collections, we can rely on several schema languages, like JSON Schema[13], Mongoose,[14] and Joi[15]. Remark that the JSON Schema specification is

[10] https://searchsqlserver.techtarget.com/definition/OLE-DB (retrieved on 7. 6. 2021)
[11] https://docs.oracle.com/javase/7/docs/technotes/guides/jdbc/jdbc_41.html (retrieved on 7. 6. 2021)
[12] https://www.arangodb.com/ (retrieved on 7. 6. 2021)
[13] http://json-schema.org (retrieved on 7. 6. 2021)
[14] https://mongoosejs.com (retrieved on 7. 6. 2021)
[15] https://github.com/sideway/joi (retrieved on 7. 6. 2021)

somewhat complex and is not too used in practice. A short overview of these languages can be founded in [3]. A formal approach based on typed functional JSON data objects proposed in [20] will be presented in Section 6.1. Fig. 4.1 shows an example of JSON schema expressed in the JSON Schema language describing books with a title, authors, and issue date.

```
{
  "$schema":"http://json-schema.org/draft-04/schema",
  "type":"object",
  "properties":{
     "book":{
        "type":"object",
        "properties":{
           "title":{"type":"string"},
           "authors":{"type":"array",
                      "items":{"type":"string"}},
           "issued_in":{"type":"number"},
                     }  }
           }  }
}
```

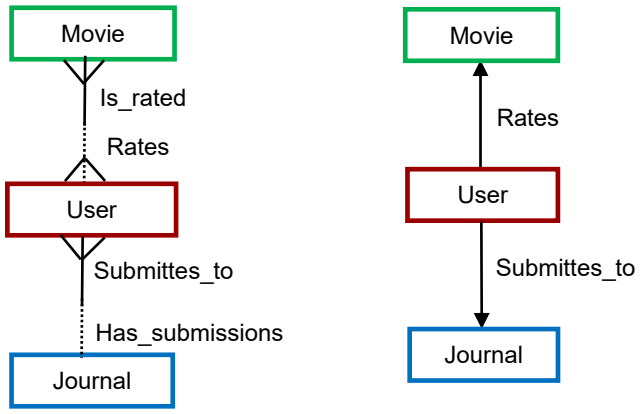
**Fig. 4.1.** JSON schema fragment for documents containing book objects.

## 4.2. Graph schemas

The database model used for GDB can be characterized by the following three features:

- Data and the graph database schema are represented by a graph or by data structures generalizing the notion of graph.
- Manipulation of data is expressed by graph transformations like graph-oriented operations and type constructors.
- ICs enforce data consistency.

The *database graph* is mostly a directed, labelled, attributed multigraph. Database graphs with different sets of attributes for nodes/edges of the same types are accepted. This situation occurs in practice, e.g., when no graph database schema is at disposal, i.e., in a schema-less case, which makes such graphs suitable for semi-structured data storage. This is in line with the NoSQL concept.




(a) graph conceptual schema (b) graph database schema

**Fig. 4.2.** Schemas in graph databases environment

In [17] we used a binary E-R model as a variant for graph conceptual modelling. A correct graph conceptual schema may be mapped into an equivalent (or nearly equivalent) graph database schema with the straightforward mapping algorithm but with a weaker notion of a database schema, i.e. some inherent ICs at the conceptual level will be neglected to satisfy the usual notation of directed, labelled, attributed multigraphs. Both nodes and edges are defined by a unique identifier (Id). In Fig. 4.2 (a), we present a conceptual schema without attributes of entity and relationship types, where a user submits at most to one journal. Explicit conceptual M:N cardinality between movies and journals is lost in the graph database schema (b). We can see that a difference between conceptual view and database view is rather negligible here.

The typed graph model presented in [10] enables using hyper-nodes and hyper-edges,

in which a relationship (called a hyper-edge) can connect any number of nodes. This feature better supports graph conceptual modelling.

## 5. Functional modelling

Our approach to conceptual/database modelling is based on the notion of function. This notion covers naturally all structures allowed in the recent database models. As a manipulating tool for functions, we have a typed lambda calculus at disposal. At the conceptual level, the approach is inspired by transparent intensional logic, see, e.g., [12], [15], and [5]. Conceptual functions will be named by expressions in a natural language and will be directly used for querying. By using these functions for NoSQL data, we obtain query tools at a conceptual rather than the logical level.

Section 5.1 shortly summarizes functional typing. Attributes are introduced in Section 5.2. Section 5.3 describes the traditional manipulation language for functions – typed lambda calculus. The possibility of querying attributes is shortly presented in Section 6.2.

### 5.1. Types

Each conceptual/database model is based on a typing system. For example, in the relational DB model, such a system contains one type – relation. In our basic functional type system we use functional types and tuple types. In other database models, we will also use types as arrays, sets, and more.

*Definition 5.1*. The existence of some (*primitive*) types $S_1,...,S_k$ ($k \geq 1$) is assumed. They constitute a base **B**. More complex types are obtained in the following way:

(a) Every member of the base **B** is a (*primitive*) *type over* **B**.
(b) If $S, R_1,...,R_n$ (n≥1) are types over **B**, then $(S:R_1,...,R_n)$ ($n \geq 1$) is a (*functional*) *type over* **B**.
(c) If $R_1,...,R_n$ (n ≥ 1) are types over **B**, then $(R_1,...,R_n)$ is a (*tuple*) *type over* **B**.

The *functional type system* **T** *over* **B** (or **T** if **B** is assumed) is the least set containing types given by (a)-(c).

For example, *Number*, *String*, *Bool* ϵ **B**. If members of **B** are interpreted as mutually disjoint nonempty sets, then $(S:R_1,...,R_n)$ denotes the set of all (total or partial) functions from $R_1\times...\times R_n$ into $S$, $(R_1,...,R_n)$ denotes the Cartesian product $R_1\times...\times R_n$. An object $o$ of the type $T$ is called a *T-object* and can be denoted *as o/T*. $(R_1,...,R_n)$-objects contain an ordered set of respective $R_1,...,R_n$ -objects.

A primitive type *Bool* is defined as the set {`TRUE`, `FALSE`}. The type *Bool* allows typing some objects as sets and relations. They will not consider here as separate units. Both can be modelled as unary and n-ary characteristic functions, respectively. Thus, both notions are redundant in **T**.

Mathematical functions may be easily typed. Arithmetic operations +, -, *, / are examples of (*Number*:*Number*,*Number*)-objects. Logical connectives, quantifiers, and predicates are also typed functions: e.g., **and**/(*Bool*: *Bool*, *Bool*), $R$-identity $=_R$ is (*Bool*: $R$,$R$)-object, universal $R$-quantifier $\Pi_R$, and existential $R$-quantifiers $\Sigma_R$ are (*Bool*:(*Bool*:$R$)) - objects. As for our notational convention, we use an infix notation for arithmetic and logical functions. Similarly, we write '∀x...' and '∃x...' for application of the universal and existential quantifier, respectively. With **T**, it is possible to type functions of functions, nested tables, ISA-hierarchies, etc.

Summarizing the notions given above, we will distinguish between object types and objects. Objects are of object types. Because object types are functions, objects are their values. Indeed, the values need mark-up given by the name of the object.

### 5.2. Attributes

Object structures usable in building a database can be described by some expressions of a natural language. Each base **B** will consist of descriptive and entity types. Consider **B** = {*User*, *Movie*, *U_ID*, *Name*, *Birth_year*,…, *String*, *Number*}. For example for GDBs, we can conceive entity types as sets of node IDs. Descriptive types *String*, *Numbers*, etc., serve for domains of properties of entities. Then, e.g., the expression "the movies rated by a given user" (shortly MRU) denotes a ((*Bool*:*Movie*):*User*)-object, i.e. a (partial) function

f:*User*→(*Movie*→*Bool*), "the name of a given user" (shortly NU, e.g.) denotes a (*Name*:*User*)-object. Such (named) functions represent attributes. Attribute names are natural language expressions. This fact increases their conceptual features. More formally, *attributes* are functions of type ((*S*:*T*):*W*), where *W* is the logical space (possible worlds), *T* contains time moments, and $S \in \mathbf{T}$. Thus, attributes are partial empirical (data) functions. For simplicity, we will not explicitly consider types *W* and *T* for attributes in this paper. Other conceptual constructions are propositions of type *Bool*. Some of them are connected with attributes. Attributes bear a basic information concerning an application domain. We say that attributes generate certain basic propositions. For example, "The name of a given user is Baker" denotes such a proposition. It is generated by the attribute NU.

For database users ICs are also important. Essentially, each IC is a proposition that introduces redundancy into the conceptual/database schema. On the other hand, we need ICs for filtering erroneous data. ICs can be expressed again as simple propositions or, on a more formal level, by a logical language, e.g., by a typed lambda calculus (see Section 5.3).

A lot of functions need even no possible world. For example, aggregation functions like COUNT, AVARAGE, and arithmetic functions provide such functions. They have the same behaviour in all possible worlds and time moments. We call such functions *analytical*. On the other hand, attributes are *empirical functions*. Range of these functions are again functions.

Typing system **T** over **B** and a set of attribute specifications enable to consider a *functional conceptual/database schema* as a couple $\mathbf{S}_{LT} = (\mathbf{A}, \mathbf{I})$, where **A** is a tuple $(A_1,...,A_n)$ of typed variables of LT called *attribute identifiers* and **I** is a tuple of other terms of LT representing ICs. With attributes we can obtain basic true propositions (in an actual world and a given time moment). These propositions form an *information base* for $\mathbf{S}_{LT}$.

### 5.3. Typed lambda calculus

Our version of the typed lambda calculus (language LT) directly supports manipulating objects typed by **T** introduced in Section 5.1. We use it in Section 5.4 as a tool for querying a database described as a set of attributes. We also present a syntactic sugar for LT and examples of queries.

*Definition 5.2.* We will suppose a collection **Func** of constants, each having a fixed type, and denumerable many variables of each type. Then the *language of* (λ-)*terms* (LT) is defined as follows:

Let types $R, S, R_1,\ldots,R_n$ $(n \geq 1)$ be elements of **T**.

(1) Every variable of type *R* is a *term* of type *R*.
(2) Every constant of type *R* is a *term* of type *R*.
(3) If M be a term of type $(S{:}R_1,\ldots, R_n)$, and $N_1,\ldots, N_n$ are terms of types $R_1,\ldots, R_n$, respectively, then $M(N_1,\ldots, N_n)$ is a *term* of type *S*. /*application*/
(4) If $x_1,\ldots, x_n$ are different variables of the respective types $R_1,\ldots, R_n$ and M is a term of type *S*, then $\lambda x_1,\ldots, x_n(M)$ is a term of type $(S{:}R_1,\ldots, R_n)$ /*lambda abstraction*/
(5) If $N_1,\ldots,N_n$ are terms of type $R_1, \ldots, R_n$, respectively, then $(N_1,\ldots, N_n)$ is a term of type $(R_1,\ldots, R_n)$. /*tuple*/
(6) If M is a term of type $(R_1,\ldots,R_n)$, then M[1],…,M[n] are terms of respective types $R_1,\ldots, R_n$. /*components*/

### 5.4. Querying with terms

The language LT with at least applications of functions and lambda abstractions provides a powerful tool for querying database data conceived as functions. A query can be expressed by a term of LT. In Section 2, we have mentioned a lot of query languages in particular categories of NoSQL DB. Using LT as a universal query language in this case can be useful in two directions:

(1) increasing the expressive power of querying data in NoSQL DB,
(2) querying data over DBS considered in a multi-model environment.

Regardless of the fact that NoSQL DBS are considered schema-less, some explicit database schemas are needed in our approach. Very natural possibilities for the

construction of such schemas are offered by GDB. In [19] we used a functional approach in which a property graph is represented by typed partial functions. We were inspired by the HIT Database Model (e.g., [15]), as a functional alternative of a binary E-R model. The functions considered here are of two kinds: *single-valued* and *multivalued*.

*Example 5.*1: Consider attributes MRU/((*Bool*:*Movie*):*User*) and NU/(*Name*:*User*) from Section 5.2 and the query "Give a set of couples associated to each user given by his/her name and the number of movies rated by him/her". An associated LT term providing a relation looks as follows:

λ n, c (c=COUNT(λ m (MRU(u)(m) and NU(u) = n)) )

or in a more readable version as:

$\lambda\, n^{Name}, c^{Number}\ (c^{Number}=COUNT(\lambda\, m^{Movie}(MRU(u^{User})(m^{Movie}) \text{ and } NU(u^{User}) = n^{Name})))$

More structured answers could be achieved by terms like:

λ x, y,… λ u … λ z …(M)

where M/*Bool*. We would get another syntactic sugar, e.g., by introducing the constructs SELECT, WHERE, etc.

We can also suppose that other data models used as typed functions used in LT, e.g., those derived from JSON documents [20]. Then, e.g., comparing to MongoDB querying, LT is more powerful concerning its expressivity due to logical connectives, quantifiers, arithmetic operations, and aggregation functions included in **Func**. LT enables to obtain classical relations, tree structures of String values, and even new data as query answers.

*Example 5.2*: In a multi-model environment integrating NoSQL DB together with a relational DB, we can consider relations as Boolean functions [19], e.g.,

Actors/(*Bool*:*A_ID*, *Name*, *Address*)
Plays(*Bool*:*Title*, *Actor_ID*)

## 6. Functional conceptual/database schema design

Now we focus on the functional approach to two NoSQL models: a document model using JSON objects (Sections 6.1-6.2) and a graph model (Section 6.3). We will define functional schemas for both cases. Such schemas can be considered as conceptual schemas, but, on the other hand, they can serve as database schemas with the LT language enabling querying. In a multi-model environment, relational data can be considered too. For each NoSQL model, we must first define an appropriate type system supporting its data structures.

### 6.1. JSON functional schemas

First, we introduce a general functional type system supporting the structure of JSON data. We start with its basic version extending Definition 5.1. The tuple type is not considered.

*Definition 6.1.* The existence of some (primitive) types $S_1,...,S_k$ (k≥1) is assumed. They constitute a base **B**. More complex types are obtained in the following way:

(a) Every member of the base **B** is a (*primitive*) *type over* **B**.
(b) If $T_1$, $T_2$ are types over B, then ($T_2$:$T_1$) is a (*functional*) *type over* B.
(c) If $T_1,...,T_n$ (n ≥ 1) are types over **B**, then $\{T_1,...,T_n\}$ is a *set type* over B.
(d) If $T_1,...,T_n$ (n ≥ 1) are types over **B**, then $[T_1,...,T_n]$ is an *array type* over B.

The *JSON type system* **T** *over* **B** (or **T** if **B** is assumed) is the least set containing types given by (a)-(d).

If members of **B** are interpreted as mutually disjoint nonempty sets, then $\{T_1,...,T_n\}$-objects contain an unordered set of respective $T_1,...,T_n$ objects. $[T_1,...,T_n]$-objects are arrays of respective $T_1,...,T_n$ objects.

For the description of JSON data format we need also a type *Name* in **B** for naming JSON types and regular expressions. Then we define the type system $\mathbf{T}_{reg}$ over **B** as follows.

(e) Let name ϵ *Name*. Then name:*String* is a *named character data over* **B**.
(f) Let *T* be a set type or named character data. We consider *regular expressions T**, *T*+, or *T*?. They can occur in array types, *T*? can occur in set types.
(g) Let *T* be a regular expression. Let name ϵ *Name*. Then name:*T* is a *named type over* **B**.

A move from character objects to abstract objects is given by the following definition.

*Definition 6.2.* Let $\mathbf{T}_{reg}$ over **B** be the type system from Definition 6.1 and **O** be the set of abstract objects. Then the object type system $\mathbf{T}_O$ induced by $\mathbf{T}_{reg}$ (or $\mathbf{T_O}$ if $\mathbf{T}_{reg}$ is understood) is the least set containing the object types given by the following rule:

- Let name:T be from $T_{reg}$. Replace all names in name:T by their upper-case version. Then NAME:T is a member of $T_O$.

*Example 6.1.* Object types from $\mathbf{T}_O$ related to the JSON schema fragment in Fig. 4.1 include

TITLE:*String*
ADDRESS:*String*
NAME:*String*
ISSUED:*Number*
BOOK:{TITLE, AUTHORS, ISSUED?}
AUTHORS:[AUTHOR*]
AUTHOR:{NAME, ADDRESS?}

They extend the original JSON schema by a structured author, author's name, and consider more authors including none.

A *JSON-database schema*, $\mathbf{S}_{JSON}$, is then a set of variables of types from $\mathbf{T}_O$. Given a JSON-database schema $\mathbf{S}_{JSON}$, a *JSON-database* is any valuation of variables in $\mathbf{S}_{JSON}$. For convenience, we denote the variables from $\mathbf{S}_{JSON}$ by the same names as names from $\mathbf{T}_O$, e.g., BOOK, AUTHOR, etc. Then object types in Example 6.1 represent a JSON-database schema. Explicit ICs on the JSON-database are not considered here.

### 6.2. Quering JSON data

Querying JSON data uses a variation of the LT language from Definition 5.2. Tuples and components are omitted and sets and arrays are considered.

Let types $R$, $S$, $R1,\ldots,R_n$ ($n \geq 1$) are elements of **T**.

(1) Every variable of type $R$ is a *term* of type $R$.
(2) Every constant of type $R$ is a *term* of type $R$.
(3) If M is a term of type ($S$:$R_1,\ldots, R_n$), and $N_1,\ldots, N_n$ are terms of types $R_1,\ldots, R_n$, respectively, then $M(N_1,\ldots, N_n)$ is a *term* of type $S$. /*application*/
(4) If $x_1,\ldots, x_n$ are different variables of the respective types $R_1,\ldots, R_n$ and M is a term of type $S$, then $\lambda x_1,\ldots, x_n(M)$ is a term of type ($S$:$R_1,\ldots, R_n$) /*lambda abstraction*/
(5) If M is a term of type $\{T_1,...,T_n\}$ and N is of type $T \in \{T_1,...,T_n\}$. Then
N(M) is a term of type $T$. /*set element*/
(6) If $N_1,...,N_n$ are terms of types $T_1,...,T_n$ , respectively, then
$[N_1,...,N_n]$ is a term of type $[T_1,...,T_n]$. /*array*/
(7) If M is a term of type $[T_1,...,T_n]$, then
$M[1],...,M[n]$ are terms of types $T_1,...,T_n$, respectively. /*elements of array*/

Then, supposing a book abstract object b, we can specify a path in a JSON document via composition of functions, e.g.,

NAME(AUTHORS(b)[1]) = 'Gaspar'

or, in a more friendly way,

b.AUTHORS[1].NAME = 'Gaspar'

More generally, queries will be expressed by lambda abstractions, similarly as in Example 5.1. Observe also that our LT language allows to consider in a query attributes belonging to different categories of NoSQL databases. Finally, the relational data model can be taken into account, too.

### 6.3. Graph functional schema

The notion of attribute applied in GDBs could be restricted to attributes of types ($R$:$S$), (*Bool*($R$):$S$), or (*Bool*:$R$,$S$), where $R$ and $S$ are entity types. This strategy simply covers binary functional types, binary multivalued functional types, and binary relationships described as binary characteristic functions. The last option corresponds to M:N relationship types. For modelling directed graphs, the first two types are sufficient, because

M:N relationship types can be expressed by two „inverse“ binary multivalued functional types. Here we will consider always one of them, i.e., mixed graphs are not used in our graph conceptual modelling. For graphical expressing a graph conceptual schema, we use two types of arrows according to associated binary functional types. Comparing to Figure 4.2 (a), this notation is simpler.

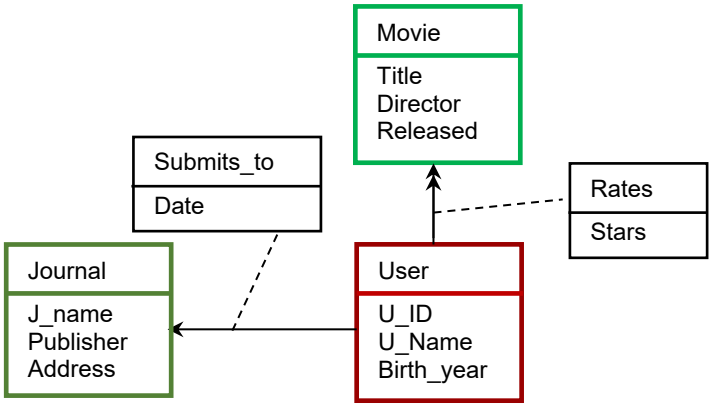



**Fig. 6.1.** GDB conceptual schema Movies

*Example 6.2*: We describe multivalued functions by double arrows in a conceptual schema graph. Single arrows denote directional functional relationships, e.g., Submits_to. Tuple types with $S_1,...,S_m$ used for properties of edges contain as a last component the entity type of the end node of the given edge, e.g., (Date, Journal) for Submits_to. Then a functional database schema corresponding to the GDB schema in Fig. 6.1 can look as:

Movie/((*Title*, *Director*, *Released*):*Movie*)
User/((*U_ID*, *U_Name*, *Birth_year*):*User*)
Journal/((*J_name*, *Publisher*, *Address*):*Journal*)
Rates/((*Bool:Stars*, *Movie*):*User*)
Submits_to/((*Date*, *Journal*):*User)*

## 7. Conclusions

In this paper, we have shown a database-oriented view on NoSQL data based on a functional approach to data typing. A conceptual schema can be understood as a set of typed functions. This enables to model the world of relational and NoSQL DB in a more or less unified way. We have also introduced a version of typed λ-calculus, the language LT, which can be used as a formal background for:

- functional data integration from various NoSQL sources with different data models (multi-model environment),
- querying functional data in such integrated environment.

A significant feature of such approach is higher expressive power of query tools based on LT. For example, features supporting constructs of a predicate calculus with arithmetic and aggregate functions can be considered in such LT. Moreover, querying over attributes enables to users to see queries more conceptually than querying over a database schema. Clearly, the line between “conceptual” and “logical” is rather blurred in this approach. All depends on the way how attributes are modelled and named.
Consequently, the responsibility of developers (programmers) for the optimization of DB is considerably increased.

Some challenges arise for a future research, e.g.:

- how design an adequate user-friendly (maybe extensible) functional query language,
- how to transform LT queries into expressions of NoSQL query languages (e.g., MongoDB, CQL, Neo4j),
- based on LT to specify and formally prove and compare the expressive power of NoSQL query languages.

**Acknowledgments.** This work was supported by the Charles University project Q48.

## References


1. Abdelhédi, F., Brahim A.A., Atigui, F., Zurfluh, G.: MDA-Based Approach for NoSQL Databases Modelling. In: Proc. of International Conference on Big Data Analytics and Knowledge Discovery, DaWAK 2017, LNCS 10440, pp. 88-102 (2017)
2. Atzeni, P., Bugiotti, F., Cabibbo, C., Torlone, R.: Data modelling in the NoSQL world. Comput. Stand. Interfaces, Elsevier, 67, pp.103-149 (2020)
3. Baazizi, M.A., Colazzo, D. Ghelli, G., Sartiani, C.: Schemas and Types for JSON Data: From Theory to Practice. In: Proc. of SIGMOD Conference 2019, pp. 2060-2063 (2019)
4. Chaves, D., Malinowski, E.: Document Data Modelling: A Conceptual Perspective. In: Proc. of ADBIS 2019 Conference, CCIS 1064, pp. 19–27 (2019)
5. Duží, M., Pokorný, J.: Semantics of General Data Structures. In: Information modelling and knowledge bases IX. P. J. Charrel, H. Jaakkola, H. Kangassalo (Eds.), IOS Press, Amsterdam, Netherlands, pp. 115-130 (1998)
6. Gaspar, D., Coric, I.: Bridging Relational and NoSQL Databases. ADMDM Book Series, IGI Global (2017)
7. Hills, T: NoSQL and SQL Data Modelling: Bringing Together Data, Semantics, and Software. 1st Edition, Technics Publications (2016)
8. Hull, R., King, R.: Semantic Database modelling: Survey, applications and research issues. ACM Comp. Survey 19(3) 201–260 (1987)
9. Klein, H.K., Hirschheim, R.: A Comparative Framework of Data Modelling Paradigms and Approaches. The Comp. Journ. 30(1) 8-15 (1987)
10. Laux, F.: The Typed Graph Model. In: Proc. of DBKDA 2020, Lisbon, Portugal, pp. 13-19 (2020)
11. Lawley, M., Topor, R.: A Query Language for EER Schemas. In: Proc. of the 5th Australasian Database Conference, ADC '94, Christchurch, New Zealand, pp. 292-304 (1994)
12. Materna, P., Pokorný, J: Applying the simple theory of types to data bases. Information Systems 6(4), 283-300 (1981)
13. Nečaský, M.: XSEM - A Conceptual Model for XML. In: Proc. of Asia Pacific Conference on Conceptual Modelling, Ballarat, Australia. CRPIT, 67, pp. 37-48 (2007)
14. Pautasso, C., Wilde, E., Alarcon, R.: REST: Advanced Research Topics and Practical Applications, Springer (2014)
15. Pokorný, J.: A Function: Unifying Mechanism for Entity-Oriented Database Models. In: Proc. of ER 1988 Conf., pp. 165-181 (1988)
16. Pokorný, J.: Semantic Relativism in Conceptual Modelling. In: Proc. of DEXA 1993 Conference, pp. 48-55 (1993)
17. Pokorný, J.: Modelling of Graph Databases. Journal of Advanced Engineering and Computation 1(1), 4-15 (2017)
18. Pokorný, J.: Integration of relational and NoSQL databases. Vietnam Journal of Computer Science 6(4), 389-405 (2019)
19. Pokorný, J.: Integration of Relational and Graph Databases Functionally. Foundations of Computing and Decision Sciences 44 (4), 427-441 (2019)
20. Pokorný, J.: JSON Functionally. In: Proc. of ADBIS 2020 Conference, LNCS 12245, Springer Nature Switzerland AG, Cham, pp. 139-153 (2020)
21. Roy-Hubara, N., Sturm, A.: Design methods for the new database era: a systematic literature review. Software & Systems Modeling 19:297–312 (2020)
22. Shin, K., Hwang, Ch., Jung, H.: NoSQL Database Design Using UML Conceptual Data Model Based on Peter Chen's Framework. Int. Journal of Applied Engineering Research 12(5), 632-636 (2017)