ON THE ROLE OF NOVEL EXTENDED RELATIONS IN DOCUMENT CENTRIC INFORMATION SYSTEM DESIGN

András Benczúr, Bálint Molnár and Gyula I. Szabó
(Budapest, Hungary)

Communicated by Péter Racskó
(Received 16 January 2025; accepted 1 March 2025)

Abstract. In the document-centric modeling of information systems, the flow of data between individual documents and the collections stored in the database plays an important role. The types of data that can be stored in the collections and their constraints are given by the data models supported by the database management software. We show the semantically fundamental usage difference between the documents and the database. In addition to managing relational rows, the relational data model that is the most important starting point also contains a basic document type, namely the tabular document type, as a representation of a relation consisting of relatively few rows. With this dual task in mind, we introduce the extended relational model. In the standard relational model, each relation name identifies a collection of occurrences of a single row type. Therefore, in schema-level instructions and operations, each relation name/table name has a fixed set of attributes (ordered or unordered). In our paper, we introduce a collection containing tables of different but similar schemas. Actions can be specified at this level, which applies to several tables at the same time. The similarity of the schemes is given by belonging to a formal language over a finite set of attribute names. Every sentence of a given formal language can be a potential schema. An attribute name can therefore occur more than once in a specific schema, and the order is fixed.

Key words and phrases: Data models, formal languages, relational databases, documents, information systems, graphs.
2010 Mathematics Subject Classification: 68U35.
This research was supported by the Thematic Excellence Programme TKP2021-NVA-29 (National Challenges Subprogramme) funding scheme, and by the COST Action CA19130 - "Fintech and Artificial Intelligence in Finance Towards a transparent financial industry" (FinAI)
https://doi.org/10.71352/ac.58.010325
First published online: 8 March 2025

An instance of a schema collection consists of a finite number of table types. This is called an extended relation. Among the formal languages, we first discuss the use of regular languages. In the case of context-free languages, we use the language's terminal and non-terminal symbols as attribute names. With this, embedded relational schemas can also be handled. We use graph representations of formal languages as schema graphs. With the help of this, we specify the operations at the schema and instance level. The definition and implication problems of functional dependencies are investigated too. We show the close connection with the XML ELEMENT declaration. We also specify an implementation solution. After introduction, we shortly examine its use in the document centric modelling of information systems. A novel X-merge solution is specified between free documents and X-relations.

1. Introduction

At the beginning, we make a small detour to the Information Technology background. Our paper brings together three areas of our research conducted over the last ten years in the form of a study paper. The two parts related to technologies, the part related to the topic of database management and information systems, are framed by a novel approach to the concept of information, according to which the owner relates the information carrier to the meaning during an information event. The proposed approach outlines a formal framework in which information theory, relational data bases, document-centric perception of information systems, and the lingua franca representation language, namely XML, are integrated. The emergence of computing and digital technologies brought new possibilities for the physical carriers of information. A new era in the development of humanity's infosphere has begun [17] and is being built at an extraordinary speed. It is very important to emphasize that any management of information is always related to the representational forms of the information. Considering that, the carriers of information are the focus of our paper, in the following we distinguish the carriers of information from the information itself, and call the carriers of information *formations*.

After the general information technology detour, we provide a brief historical overview of the development of the structure of two defining formations of modern information systems, namely the development of the data models of databases and the document structures of modern information systems. The introduction and connection of our new data model to the world of documents was motivated by this background.

The notion of information, no matter how central it is in information technologies, is under intense debate and has no agreed definition. Therefore, bypassing the definition of the notion of information in the article [5] and [6], the authors placed the emphasis on the carriers of the information. We briefly outline the basis for this.

No information exist without material form, it is carried by something, that can be observed / perceived / shaped. This carrier is called a formation. During an information event the formation is observed / created, and the information appears to the observer / creator and the meaning is associated with the formation.

An information event or instance can be represented as a three-part tuple. The information event consists of an observer/producer, formation, and referent. The triplet can be temporally connected in three ways:

(a) Sensing (new referent) - Observer - (new) formation (R,O,F)

(b) Formation - Observer - Referent (F,O,R)

(c) Referent - Producer - Formation (R,P,F)

We give two basic laws:

The first basic law: Different meanings require different formations (distinguished by the observer/creator).

The second basic law: Detection/creation of a formation means information if it is also possible to detect a previously created formation to which the meaning is connected. The second basic law brings in the time dependence of information and the quality of understanding and utilization. The modeling of the semantics must also take this into account. This applies equally to information events in the human sphere and the artificial sphere.

What does it mean that we have these two information processing machines: the brain and the computer? (See [24, 7]) What do they process as a material substance? They process the carriers of information, the formations. Information processing of the brain is always belongs to some information event. Usually, information processing inside the computers does not belong to information event. It is carried out according to algorithmic semantics. See [15] the paradox of Denning and Bell.

During the unfolding of the possibilities of /electronic/machine computation, the development history of database management systems and automated information systems began at the same time. The representations of information in both systems are based on digital coding that matches the capabilities of computers.

Both systems are characterized by the fact that the past most important carriers of information, the formations consisting of written characters appearing on “paper”, have been replaced by the possibility of formations based on digital signals. The use of digital formations was made necessary by computers, and their use is also supported by computers. In automated information systems, two types of digital information carriers, i.e. formations, can be clearly distinguished. One, the digital document is directly related to activities and procedures, the other, the database is a collection of the most important factual data.

Returning to the formations of documents and databases, we also show the differences in their role in information events. The document format is suitable for direct display (for human perception) or as input for further tasks performed by computers. In the case of databases, the necessary sub-formations (e.g., relational rows) must first be extracted, possibly as a result of additional operations, and typically queries (which must also be given to the machine as a formation) must be performed and finally incorporated into a document formation. Among the many connections between databases and documents, in this paper, we will examine data insertion and exchange between documents and databases (in both directions). For this, we build on the information system modeling based on the [21, 20, 22, 23] document-centric principle. We show the generalization of the easy-to-use "Merge" function for relational tables in the case of extended relational tables, which form the central part of the paper.

2. History and background

In the 70-year history of database management systems, the emergence of new data models marked important milestones. The main task of data models is to ensure the management of large amounts of data with a predefined structure, supporting user requirements and efficient implementation.

The first database models in the 60-es were IBM's hierarchical data model, and the CODASIL network data model. Codd in his seminal paper [14] employed a novel tool (the mathematical notion of a "relation") to address some of the inadequacies of the prevailing database models [29].

The 1975 ANSI-SPARC[4] report generated "the big debate" on data models that finally brought in the dominance of the relational data model. The success of the relational model was partly due to the simple mathematical model and the query languages that can be built on it. It soon became dominant, and standard SQL remains a core database technology. The design theory of the relational data model, the analysis of its query languages, and the extensions of the model to complex value and object-oriented directions motivated the research of the next twenty years. A very complete summary of these theoretical research achievements can be found in the book [2].

The rapid extension of the Internet and the World Wide Web produced a large amount of data that cannot be easily managed in a relational or object-oriented database. The semistructured data model was the database community's attempt to apply traditional database management techniques to such data, see Suciu[27]. After the release in 1998 of the XML 1.0 Recommendation by the World Wide Web Consortium [30], XML has evolved to become the de-facto standard format for data exchange over the World Wide Web. XML was originally developed to describe and present individual documents, it has also been used to build databases. See the book [1].

In our paper, we use elements of these two models to build novel data models. Therefore, we do not mention here Big Data technologies, semantic data models, the emergence of Data Science, and the AI-based technologies evolving during the last two decades.

We use the tuple type constructor from the relational model. We will handle collections of tuples, however, the tuple type can be chosen from several types over a finite set of attributes. Using a similar technique to the type definition of the XML DTD, the allowed types are specified by a formal language.

The central part of the article is the introduction of the extensions of the relational model and the analysis of the new models. In this, we intensively use the graph representations of formal languages. More details on this can be found in the Appendix (see Section 9).

In the history of the digitization of information systems, in addition to the initial processing based on batched, sequential data files, the management of information was based on the flow of paper-based documents in the systems. With the advent of direct access storage devices, sequential data files have been replaced by increasingly complex databases, but paper-based documents have not yet been replaced by digital documents. On-line transaction management increasingly enabled the direct integration of database services in the filling out and electronic management of user and administrative documents. This also includes documents that can participate in an information event in a visualization suitable for human perception (screen, print). The development of technology, along with the digitization of client-server systems, web services, ERP systems, BPS and everyday administrative, financial and other services in general, supplemented by mobile technologies, made a very wide range of digital documents available and accessible. The production of final, displayable documents can be the result of complex processes, during which documents are also transferred between computers, which does not constitute an information event. The world of the documents outlined above and their situation are the focus of our paper [22] dealing with the document-centric modeling of information systems.

This approach is also valid in today's revolutionary developing data processing world, where natural language processing, large language models (LLM) and generative artificial intelligence tools appear during the production and processing of documents.

3. Formal definition of extended relations

In the ELEMENT declaration of the XML Document Type Definition, the Element type can be specified by a regular expression. We use this for defining a system of tuple types over a finite set of attribute names. Over the finite set U of attribute names, we assign tuple type $<A_1{:}x_1, \ldots, A_m{:}x_m>$ to the finite list of attribute names, $w = A_1, \ldots, A_m$. An attribute name can appear in the list more than once, which means that the order of the $A_i{:}x_i$ pairs becomes fixed. In the instance of data collection, we allow several types, with the constraint that the list w is an element of a formal language L over a predefined U. In this way, we introduced the extended relational model, in which an extended relational table can contain several types of tuples, with the restriction that the attribute list specifying the tuple type must be a sentence of a given formal language L. In this paper, we introduce the models based on our previous papers [28] and [9] for regular languages in Section 4. Then, as a new result and the extension of our paper [8], the specifications are given for context-free languages in Sections 5 and 6.

We can also consider our model as identifying the occurrences of several relational tables with a single name, and using this level for defining the operations. In the paper [18] we found a very early version of querying multi-databases, which we did not cite in our former papers. The operations of the relational algebra are defined at the level of the extended relations and functional dependencies are also investigated. Our main technique is based on graph representation of formal languages. For readers, not familiar with formal languages we recommend Révész's book [26]. Some explanations and further use of graph representations are given in the Appendix (see Section 9).

In the standard relational data model, attribute names are considered as sets in schemas. When specifying the row type, the order of the attributes does not matter, the order of the columns in the table representation form can be reordered. Therefore, the rows of the table can also be considered as functions, where the attribute names are the arguments of the function.

The type constructor of the relational model is the tuple constructor. Formally, over the finite set of attribute names $U = \{A_1, \dots, A_n\}$ the set of pairs $\{A_i{:}x_i \mid i = 1, \dots, n\}$ gives the tuple type. A tuple instance is given by the valuation $x_i = v_i$, $i = 1, \dots, n$.

In our data model, the first deviation from this is that we allow multiple occurrences of an attribute name in specifying the row type. With this, the order of the attributes becomes fixed, the type is given by the word $w = A_1 \dots A_k$, where $A_i \in U$, $i = 1, \dots, k$. We extend this option to nested relations as well, so that the schema of the nested relation can also be a sequence of attribute names.

After the informal introduction here we give a formal specification of sorts and their interpretations using the styles of Complex Values Section from [2].

Let $\boldsymbol{R}$ be the set of relation names, $\boldsymbol{U}$ be the set of elementary attributes, $\boldsymbol{V}$ be the set of nested attributes. The set of elementary values is denoted by $\boldsymbol{dom}$. Sorts, denoted by τ, will be defined by tuple and set constructors as follows:

$\tau = \boldsymbol{dom} | \mathit{tuple} | \mathit{set}$,

$\mathit{tuple} = <\mathrm{A}_1\colon \tau_1, \dots, \mathrm{A}_\mathrm{k}\colon \tau_\mathrm{k}>$,

$\mathit{set} = \{\mathit{tuple}_1, \dots, \mathit{tuple}_\mathrm{n}\}$, where $k \geq 0$, $n \geq 0$, $\{A_1, \ldots, A_k\}$ are *not necessary different* attribute names. The sequence $w = A_1, \ldots, A_k$ is the type of the tuple. For $A_i \in \boldsymbol{U}$, τ_i is of sort **dom**, for $A_i \in \boldsymbol{V}$, τ_j is of sort *set.*

The set of values of sort τ (i.e., the interpretation of τ), denoted $[\tau]$, is defined by

1. $[\boldsymbol{dom}] = \boldsymbol{dom}$,

2. $[\{tuple_1, \ldots, tuple_j\}] = \{\{v_1, \ldots, v_j\} \mid j \geq 0, v_i \in [tuple_i], i \in (1, j)\}$, and

3. $[<B_1 : \tau_1, \ldots, B_k : \tau_k>] =$
$= \{<B_1 : v_1, \ldots, B_k : v_{k\text{-}}> \mid v_j \in [\tau_j], j \in (1, k)\}$.

A relation schema of arity k is defined the same way as in the standard model: $R(A_1 \colon \tau_1, \ldots, A_k \colon \tau_k)$. The main difference is that an attribute may have more than one occurrence.

Finite collections of rows of any type can be handled similarly to the semi-structured data model see [1]. Each line must contain its own description and structure.

Occurrences of a given attribute A in $w = A_1 \ldots A_k$ can be characterized by positional numbers $j_1, \ldots, j_k$. If we consider the pairs $<A_j, j_l>$ as new attribute names, we can get back the original relational tuple constructor. However, the second component of the name, the positional number, can be used to produce the serialization corresponding to the word w.

The XML standard supplemented the semi-structured data model with the feature that the permitted cases of structures can be defined by formal language specifications using the XML DTD and XSLT. In the construction of the next level of our data model, we follow this path.

The semantics of the relational row type can be based on the natural meaning of the attribute names for the user. (This is not essential for the machine). With the previously introduced word-based tuple constructor, the “linguistic” meaning of the word can be used to assign structural semantics to the tuple type. A next level provides a possibility for this, we specify a language over $\boldsymbol{U} \cup \boldsymbol{V}$, and words belonging to it are the allowed tuple types. Let $L(G)$ be a language defined by grammar G over the attribute set $\boldsymbol{U} \cup \boldsymbol{V}$. This makes it possible to consider a set of rows consisting of occurrences of any of the row types defined by several words $w_1, w_2, \ldots, w_N$ as a single extended relation. This is the second level of the model. At this level, it is also possible to define, use, and implement features similar to traditional relational operations and constraints. Naturally, these options strongly depend on the type of the chosen grammar G. In our paper, we will build and analyze this into regular and context-free grammars. The construction is based on the labeled graphs assigned to the grammars.

Formal languages are specified with generative grammars by a quadruple $G(N, \Sigma, P, S)$, where N is a finite set of linguistic (non-terminal) symbols, Σ is a finite set of terminal symbols, P is a finite set of derivation rules of the form $\alpha \Rightarrow \beta$, $\alpha \in (N \cup \Sigma)^*$, $\beta \in (N \cup \Sigma)^*$, and finally $S \in N$ is the initial or sentence symbol. Applying the rule $\alpha \Rightarrow \beta$ to a word $x\alpha y \in (\Sigma)^*$ results in the word $x\ \beta\ y$. The word y can be derived from x, in notation $x \Rightarrow^* y$, if y can be obtained from x by applying a finite number of rules. The language $L(G)$ defined by the grammar G is the set of words derivable from S. These words are called sentences of the language $L(G)$. Formally: $L(G) = \{w \mid S \Rightarrow^* w\}$. For those interested in formal languages, we recommend the book by Révész [26].

We continue the formal specification of *tuple* and *set* types with the formal specification of extended relations. The scheme of the relation $R \in \boldsymbol{R}$ over the finite attribute sets $U \in \boldsymbol{U}$ and $V \in \boldsymbol{V}$ is defined by the formal languages L_R and $\{L_v | \ v \in \boldsymbol{V}\}$ over the terminal symbols $\Sigma = \boldsymbol{U} \cup \boldsymbol{V}$

The occurrence *I(R)* of the relation R is given in two steps. The first step consists of a finite number of relation schemas: $R\{w_1, \ldots, w_n | w_i \in L_R$, $i=1,\ldots,n\}$. For an embedded attribute $B \in V$ occurring in the scheme w, the selectable types are specified by the language L_B. Each occurrence of B has its own finite set of schemas. In the next step, each scheme $w \in R$ has a finite number of tuple instances of the tuple type specified by w.

In our paper, we will use regular languages in the case of non-nested attributes, while we use context-free grammar for nested attributes. In both cases, we build the model on two kinds of graph representations: the graph of finite-state automata and the graph representation of regular expressions. The traversals of the graphs give the schemes. We use this method of schema specification throughout the discussion. We assign extended relation schemes to finite subgraphs and define operations and dependencies at the graph representation level.

4. Extended relations defined by regular languages

4.1. First approach: extended relational schemas and instances specified by regular grammars

This subsection is a brief summary of our first approach from [28]. We start with the observation that Instances of a particular type of XML Element can be considered a data collection. The declaration of a DTD Element consisting of simple values can be considered as a general relational schema definition. An instance from the extended schema is a tuple type from the language given by the regular expression. Occurrences of an Element corresponding to a tuple type of a relation make a relation instance. Such an extension is included in [1]. Our aim was to find a representation of the extended relation that can be used to define and analyze functional dependencies. We have introduced the dual language of regular grammars, which can also be specified by traversing the vertices of the graph of the finite state automaton belonging to the grammar. Tuple types are given by the traversals of the graph. Functional dependencies are analyzed on this graph.

All of these started with a seemingly silly "What if?" question: What if we looked at the rows of a relational table as if we had obtained them with generative grammar?

Let us show it with an example:
Let $R(A, B, C, D, E)$ be a relational schema and $<A : a, B : b, C : c, D : d, E : e>$ be a tuple. Consider the following regular grammar (set of production rules) $S \Rightarrow a\ A,\ A \Rightarrow b\ B,\ B \Rightarrow c\ C,\ C \Rightarrow d\ D,\ D \Rightarrow e\ E,\ E \Rightarrow \varepsilon$. It generates a regular language and produces the tuples of the relation R. The produced word is *abcde*, and the relational scheme is given by the sequence of the language symbols $ABCDE$ as they are used in the production. This way we received two words: a regular language word and a dual language word. Appendix (see Section 9) contains some additional discussion on dual languages.

As a generalization, let $G(N,T,S,P)$ be a regular grammar, where N is the set of non-terminals, T is the set of terminals and S is the start symbol.
P: $\{A \Rightarrow bB \mid \varepsilon,\ A,B \in N,\ b \in Dom_B \subseteq T\}$ is the set of production rules. G generates the regular language $L(G)$.

The derivation: $S \Rightarrow a_1A_1,\ A_1 \Rightarrow a_2A_2, \ldots, A_{k-1} \Rightarrow a_kA_k,\ A_k \Rightarrow \varepsilon$ Can be represented as tuple: $<A_1 : a_1,\ A_2 : a_2, \ldots, A_k : a_k>$. The associated two words are $\omega = a_1a_2\ldots a_k \in L(G)$ and $\omega' = A_1A_2\ldots A_k \in L(G')$, where $G'(N', N, S, P')$ is the dual grammar of G. The dual sentence $\omega' \in L(G')$ is an extended tuple type. A tuple instance of sort ω' is called an extended tuple. The extended relational schema is a finite set of the sentences of the dual language. An instances of the extended relation contains a finite set of extended tuples.

Functional dependencies are given by a pair $X \subseteq Y$ of attribute subsets of a relational schema. A table instance satisfies the given functional dependency if it contains no two tuples having the same values on X and different values on Y. In order to extend the notion of functional dependencies, we need a way to specify ordered substrings from dual sequences. For specifying substrings, we use the graph of the *finite-state automaton* (FSA) that can be given directly from a regular grammar.

Let $L(G)$ be regular language and let $M(G)$ be the graph representation of its FSA. The node labels along traversals on $M(G)$ (from START to END) build up the dual language $L(G')$ associated to $L(G)$.

To define regular functional dependencies on $L(G)$ we can select two subsequences as lists of nodes along a path in M(G), as the left and right side of the dependency, let we state this as the syntactic specification for the dependency. The details of the selection will be given in Section 5. An extended instance having the scheme of sentences $R \subseteq L(G)$ satisfies this dependency, when there exist no two extended tuples in R so, that they are identical in all the attributes along the left side subsequences, but on the subsequences selected for the right side, they differ on at least one position. We proved in [28] that the decision problem of implication for extended functional dependencies is decidable. The proof is based on the Chase technique visualized on two colored version of the *M(G)* graph.

4.2. Second approach: extended relational schemas and instances specified by regular expressions

This section is based on our paper [9]. Our previous model in Section 3 could be effectively used for handling functional dependencies (FD). In the relational model FDs offer the basis for normalization (e.g. Boyce-Codd Normal Form, 3$^{\text{rd}}$ Normal Form), to build a non-redundant, well-defined database schema. However, the FSA based model cannot handle the join operation among instances (what is used to define lossless join decomposition) because the projection of a schema according to a set of nodes or two joined schemas would not necessarily lead to a new, valid schema. We need an improved model for regular databases. The main difference is that we use regular languages to define sequences of attributes as tuple types directly. This approach is closely related to XML declaration of an Element consisting of a regular expression over Elements with simple values. Our actual model based upon a graph representation for regular expressions. With the help of the graph constructed for the regular expression, in addition to functional dependencies, the operations of relational algebra can also be defined, which makes normalization definable. We shall call the selected version of graph representation the schema graph. This graph is more redundant than the FSA graph, but it supports subsequence selection and by this, it is capable for handling database schema normalization. Let us start with the definition of extended relations given by a regular language.

Definition 1. (*Extended Relation for Regular Types, shortly XRelation.*) Let L be a regular language over the set of attribute names U. Let $w = w_1 \dots w_n \in L$ be a sentence, then we say that w is a regular tuple type over U. Let dom be sets of data values, then $\{w_1 : a_1, \dots, w_n : a_n \mid a_i \in dom\}$ is the set of possible tuples of type w. A finite subset of these tuples is an instance of the regular relation. We say that the set of the tuple types for all $w \in L$ compose the schema of a regular relation based on L. The tuple types of all tuples in an instance compose the schema for the instance.

If the regular language is given by a regular expression, then there exist a great number of algorithms for the efficient construction of a finite automaton from a given regular expression. There are two main types of them according to the working-method of the resulting state machines: non-deterministic (NFA, e.g. Glushkov-automaton) and deterministic (DFA, e.g. Brzozowski's construction). The classical algorithm of Berry and Sethi [10, 3] constructs efficiently a DFA from a regular expression when all symbols are distinct. We use here another algorithm to construct a graph representation from a regular expression ([9]). We call the graph representation of L the schema graph.

A graph representation of a regular language is an edge-, or node-labeled directed graph with one entry point and one exit point. Routes from the point of entry to the point of exit are called traversals. The series of labels in the traversals make up the language.

Definition 2. (*Regular Expression Syntax.*) Let U be a finite set of symbols (alphabet), then a regular expression RE over U (denoted by RE_U, or simply RE, if U is understood from the context) is recursively defined as follows:

$$\mathrm{RE} := 0|1|\alpha|(RE)|\mathrm{RE}+\mathrm{RE}|\mathrm{RE}^\circ\mathrm{RE}|\mathrm{RE}^* \mid \mathrm{RE}^?, \cdots \text{ where } \alpha \in U$$

The regular expression RE generates the regular language $L(RE)$. $L(0)$ is the empty language, $L(1)$ is the language consisting of the empty string ε alone. Note that 0 and 1 are not symbols from the alphabet U, 0 represents the empty regular expression, 1 represents the empty string ε.

We need a construction for the graph representation of regular expressions. We will construct a graph from vertices picked from a suitably large symbol set Γ. We assume that $\{IN, OUT\} \subseteq \Gamma$ and by picking a vertex $v \in \Gamma$ we remove it from Γ. The vertices IN and OUT get the labels IN and OUT, respectively.

Algorithm 1. Construction of the Graph-Representation for a regular expression according to Definition 2.

Input: regular expression RE (built from the alphabet U),

Output: vertex labeled digraph $G(RE) = (V, E)$ representing RE.

1. if $RE = 0$, then $V = \varnothing$ and $E = \varnothing$.;
2. if $RE = 1$, then $V = \{IN, OUT\}$ and E $= \{(IN, OUT)$;
3. if $RE = A$, $A \in U$, then we pick a node $v \in \Gamma$, set $V = \{IN, OUT, v\}$, and $E = \{(IN, v), (v, OUT)\}$. We label the node v with A.
4. if RE_1 and RE_2 are regular expressions, then $G(RE_1 + RE_2)$ will be formed by uniting the IN and OUT nodes of $G(RE_1)$ and $G(RE_2)$, respectively.
5. if RE_1 and RE_2 are regular expressions, then in order to build the graph $G(RE_1 \circ RE_2)$ we first rename the OUT node of $G(RE_1)$ and the IN node of $G(RE_2)$ to $JOIN$, then unite them using the $JOIN$ node as a connecting switch in order to get a more compact graph (Figure 1).
6. if RE is a regular expression and $G(RE) = (V, E)$, then $G(RE^?) =$ $= (V, E \cup (IN, OUT))$.

7. if *RE* is a regular expression, then in order to build the graph *G(RE*)* we first pick a node $v \in \Gamma$, then we create the graph $G^*(RE) = G(RE) \cup \{v\}$ (It means that $V^* = V \cup \{v\}$, the node v gets the special label *STAR*). Let us denote $\{a_1, \ldots, a_n\}$ the nodes with ingoing edge from *IN* and $\{z_1, \ldots, z_n\}$ the nodes with outgoing edge to *OUT*, respectively. Let us create the graph $G_{IN}(RE, STAR) = \bigcup_1^n (v, a_i)$ and the graph $\mathrm{G_{OUT}(RE, STAR)} = \bigcup_1^n (z_i, v)$, respectively. Then $G(RE^*) = G^*(RE) \cap G_{IN}(RE, STAR) \cup G_{OUT}(RE, STAR) \cup (IN, STAR) \cup (STAR, OUT)$. (Figure 2)

8. Output = $G(RE_U)$.

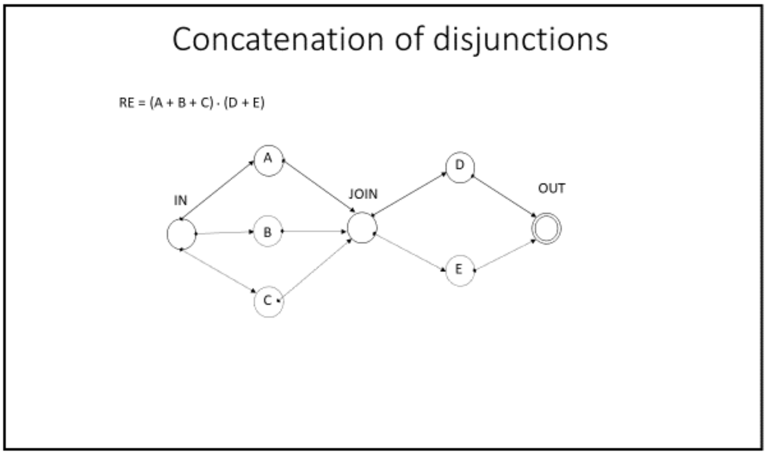


Figure 1. Concatenation and disjunction

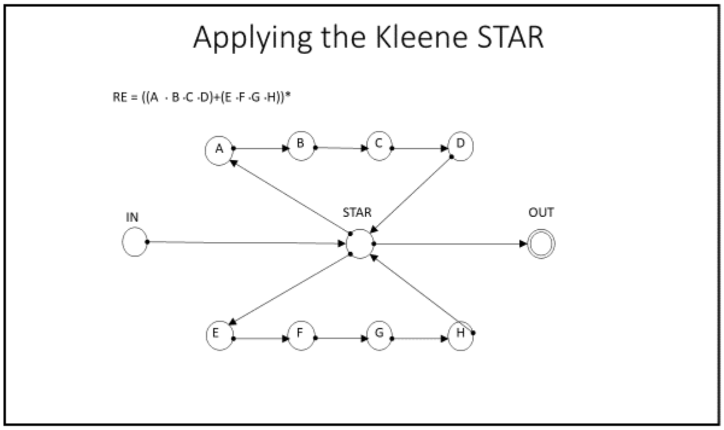


Figure 2. The Kleene-star

We say that an *(IN,...,OUT)* walk on *G(RE)* is a traversal on *G(RE)*. Let T be the set of traversals on *G(RE)*. Denoting the sequence of node labels from U along the traversal t by *w(t)*, we can easily prove that $L(G)=\{w(t)|t\in T\}$.

We shall use the following notations: E, E_i denotes regular expressions. The corresponding schema graphs are *G(E)* and $G(E_i)$, the extended relations are XR and XR_i. We omit the explicit notation of the attribute set U. The nodes *IN, OUT, JOIN* and *STAR* are auxiliary nodes, the other called ordinary node.

The definition of set operations on XRelation is straightforward, since XRelation instances are sets of tuples. The schema of the instances is the set of the occurrence of tuple types. It is also possible to define set operations between XRelations defined by different regular languages. It is based on the fact that regular languages are closed under set operations.

The benefit of introducing *G(E)* is that we can use the traversals to select schemas, or sets of schemas and define subsequence selections. We use subsequence selection in defining projections, join attributes, functional dependencies and join dependencies.

We should delete nodes from the graph *G(E)* by shortcutting two nodes on the same path. Shortcutting is allowed only between ordinary nodes. After a number of shortcutting, the remaining *G(SE)* graph is a projection graph on *G(E)*. Each traversal w on *G(E)* defines a traversal on *G(SE)*, which gives a subsequence *S(w)* of the original attribute sequence.

The projection onto *S(w)* of the relation instance belonging to relation schema w is defined in standard way. The projection of a tuple t of sort w onto *S(w)* is denoted by *t[S(w)]*. Note, that the complement of a selection graph is a selection graph too. The union / intersection of selections can be defined at the traversal level. From a traversal w on *G(E)* we select a node if it is on one / both of the selection paths. In the case of union, all the nodes participating in any of the selection graphs will be a node in the united selection graph. The shortcuts are defined in consecutive order along traversals. Selection graphs are closed under union, complement, and intersect.

Definition 3. (*Join – equijoin by coinciding substrings.*) Let XR_1 and XR_2 be *XRelations*, and S_1E_1, S_2E_2 selection expressions on E_1 and E_2 respectively with $G(S_1E_1) = G(S_2E_2)$. Suppose that for two traversals $w_1 \in G(E_1)$ and $w_2 \in G(E_2)$ the condition $S_1(w_1) = S_2(w_2)$ holds. The following construction specifies the join-schema defined by SE_1 and SE_2:

For two consecutive nodes A and B in $S_1(w_1) = S_2(w_2)$, AxB is the path in $G(SE_1)$, and AyB is the path in $G(SE_2)$, than the conjunctive joined path between A and B will be $AxyB$. The result of conjunctive join of tuples t of sort w_1, s of sort w_2 and with $t[S_1(w_1)] = s[S_2(w_2)]$ is:
$t \bowtie s[A\ x\ y\ B] = t[A\ x]^o\ s[y\ B]$.

In the disjunctive version, the join graph between A and B will be defined by $A(x+y)B$. See the difference in Figure 3.

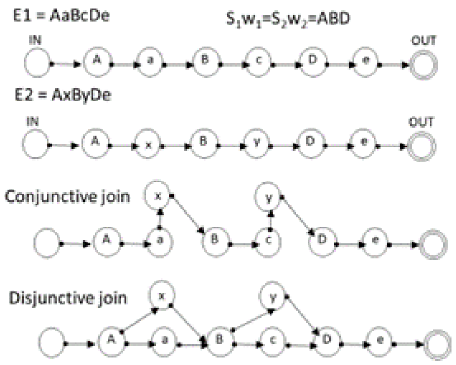


Figure 3. Conjunctive and disjunctive join

The definition of functional dependencies on Xrelation needs two consecutive selection graphs. The first one defines the right hand side of the dependency, the second the left side.

Definition 4. (*Functional dependency on Xrelations.*) We should pick up two sets of nodes from the graph $G(E)$: one set for the right side, specified by $G(SE)$, and its schema graph is denoted by $G(Y)$, another one for the left side, specified on $G(Y)$ by a selection graph C and denoted by $G(X)$. The instance $I(XR)$ satisfies $G(X) \rightarrow G(Y)$ if for any $t \in I(XR)$ of sort $w1$, $s \in I(XR)$ of sort $w2$, and $S(w1) = S(w2)$ and $t[X] = s[X]$ then $t[Y] = s[Y]$. Here $t[Y]$ and $t[X]$ denote the projection defined by SE and SY respectively.

Implication problem of extended functional dependencies is decidable by the use of a special version of the Chase procedure (see [2, 19]).

We don't go into the discussion of general join dependencies. Closing this part, we only show here lossless join decomposition property of functional dependencies. The functional dependency $G(X) \rightarrow G(Y)$ as denoted in Definition 4. defines the projection onto $G(Y)$ and onto the union of selections $G(X)$ and $G(Z)$, where $G(Z)$ denotes the complement selection of $G(Y)$.

Claim 1. *If an instance $I(XR)$ of the XR defined by $G(E)$ satisfies the functional dependency $G(X) \rightarrow G(Y)$ then the equijoin by $G(X)$ of the projections of $I(XR)$ onto $G(Y)$ and $G(X) \cup G(Z)$ gives back the instance $I(XR)$.*

5. Extended relations defined by context free languages

The first version of the extended relation schemes given by context-free languages was published in the IDEAS conference paper [8]. In this section of the paper, we discuss it in more detail. The new model can be matched to the joint declaration of several XML Elements, which can be considered context-free grammar. In the regular case, defining with a dual language (FSA graph) and defining with a direct language (regular expression graph) proved to be equally suitable. For context-free languages, a dual language generated by a grammar would be obtained, for example, by using the traversal of internal nodes of derivation trees. However, there would be many traversals to choose from, and we would get difficult-to-interpret types. A good graphical representation could not be found either. We were looking for a choice that is close to the XML DTD element declaration when defining elements together at multiple levels. In our solution, we use both language symbols and terminal symbols to define a schema. Terminals will represent an attribute with an ordinary value, non-terminal (linguistic) symbols will represent an attribute with a complex value type. For this, the rules of the context-free language must be given in a special form. For each non-terminal symbol, a regular grammar or a regular expression generating a language over terminals and non-terminals is specified.

For each grammar, we assign a finite state automaton. Edges with a non-terminal label cannot be traveled, but their endpoints can be connected by the current version of the finite state automaton belonging to the label of the edge. We build the extended scheme graphs on the unlimitedly expandable graph defined in this way. Same way as in Subsection 4.2, we can introduce and examine functional dependencies using the schema graphs. In the construction of the schema, a terminal symbol represents an ordinary attribute, and a nonterminal represents an attribute with complex value.

In the case of using regular expressions in rule specifications, the graph representation is based on the graph representation of regular expressions in Subsection 4.2. The extension of the graph uses subgraph substitutions for nonterminal symbols. Subsection 5.2 contains the details.

5.1. Graph representation for context-free languages by RSM extension

In the case of context-free languages, specifying a graph representation is a complex task. It is not sufficient to specify a single graph but we need to give possibly several graphs that call each other recursively. Such a representation can be find in [25], the graphs of the so-called Recursive Finite State Machine. RSM behaves as a set of finite state machines (or FSM). Each FSM is called a box or a component state machine. Edges in the FSM of a box may be labeled by a box name as well. A box works almost the same as a classical FSM, but it also handles additional recursive calls while traveling through an edge with a box label and employs an implicit call stack to call one component from another and then return execution flow back. RSMs are equivalent to context-free languages. This kind of computational machine extends the definition of finite state machines and increases the computational capabilities of this formalism.

We introduce recursive state machines (RSM) from [25].

Definition 5. A recursive state machine R over a finite alphabet Σ is defined as a tuple of elements $\left(M, m, \{C_i\}_{i\in M}\right)$ where:

- $\{M\}$ is a finite set of labels of boxes,
- $m \in M$ is an initial box label,
- set of component state machines or boxes,
  where $C_i = \left(\Sigma{\cup}M, Q_i, q_i^0, F_i, \delta_i\right)$,
- $\Sigma{\cup}M$ *is a set of symbols,* $\Sigma{\cap}M = \emptyset$,
- Q_i is a finite set of states, where $Q_i{\cap}Q_j = \emptyset,\ \forall i{\neq}j$,
- q_i^0 is an initial state for the component state machine C_i,
- F_i is a set of final states for C_i, where $F_i{\subseteq}Q_i$,
- δ_i is a transition function for C_i, where $\delta_i : Q_i \times (\Sigma{\cup}M) \to Q_i$.

A version of the RSM construction will be given in the following. To do this, we start by introducing a special form of context-free grammars. Each language symbol is assigned a single regular grammar whose terminal symbols are the terminal and linguistic symbols of the original grammar.

Definition 6. (*Extended Context-Free Grammar (ECFG).*) $G = (N, T, S, P)$ is a context-free grammar given by regular languages, where T is the set of terminal, N is the finite set of non-terminal symbols, P is the set of production/derivation rules, and S is the sentence symbol. The right side of production rules are given by regular languages: for all p in P, $p = \{A => w \mid w \in$ $\in$ *L(*G_A*),* $A \in N$, $G_A = \{N \cup T,\ N_A,\ P_A,\ S_A\}$ *is a regular grammar*$\}$

This special form of CFG rules gives the possibility to construct an infinitely expanding graph for each nonterminal. Similarly, to the schema graph introduced in Section 4.1, finite versions of the graph can be used as schema graphs.

During the construction of graph representations for each language symbol, we shall build the finite state automata by expansion iteration. First, for each $A \in N$, we construct a finite-state automaton for its rules, the graph $BoxA$, with $Start_A$ and End_A states of entering and leaving respectively. M_A is the extended graph for nonterminal A. During the expansion we will get graphs that have edges labeled by elements of T and N. The restriction of graph F on the terminals is denoted by ${}^{T}F$. The restricted graph is an ordinary FSA and so it defines its regular language, $L({}^{T}F)$. During graph extension we receive a sequence of growing regular languages for each nonterminal.

Construction 1. Expanding the FSA graphs for the non-terminals of the grammar G in Definition 6.

Let's start with the graph $M_S = BoxS$.

The first sub-language, $L({}^T M_S)$, is obtained by erasing all edges with non-terminal labels and taking the language generated by the remaining automaton. If this language is not empty, then each sentence can be directly produced from S, so it is an element of the language $L(G)$.

Iteration step: We have constructed the F_i graph and the corresponding $L({}^T F_i)$ language. Choose an edge (p, q) in F_i with an $A \in N$ label. Insert the graph $p \to BoxA \to q$ parallel with the A edge. Do this for all occurrences of edges with label A. This will be the F_{i+1} graph. The new language will be $L({}^T F_{i+1})$. Obviously $L({}^T F_{i+1})$ contains $L({}^T F_i)$. The vertex labels of graphs are defined by the non-terminal symbols of regular grammars in the rules. Repeated use of vertex labels is not a problem with pastes. For example, the language $L(G)$ can be produced as an expanding series of regular languages generated by extending the graph M_S The procedure is not deterministic, but it can be made deterministic by always inserting a $BoxA$ graphs in a cyclic order of N.

Example 1. – two boxes, A and B

Production rules: P: $\{A \Rightarrow ap_1\ ,\ p_1 \Rightarrow Bp_2\ |b,\ \ p_2 \Rightarrow b,$

$B \Rightarrow bq_1\ ,\ q_1 \Rightarrow Aq_2\ |b,\ \ q_2 \Rightarrow a\ \}$

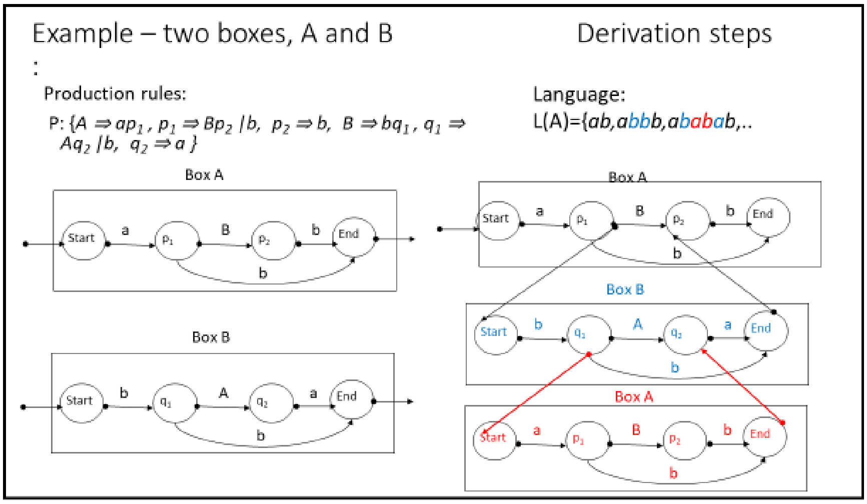

Example – two boxes, A and B
:
Production rules:
P: {A ⇒ ap1 , p1 ⇒ Bp2 |b, p2 ⇒ b, B ⇒ bq1 , q1 ⇒ Aq2 |b, q2 ⇒ a }
Box A
Start
a
p1
B
p2
b
End
b
Box B
Start
b
q1
A
q2
a
End
b
Derivation steps
Language:
L(A)={ab,abbb,abababab,..
Box A
Box B
Box A

Figure 4. Example – two boxes, A and B,
Production rules: P: {A ⇒ ap$_1$, p$_1$ ⇒ Bp$_2$ | | b, $p_2 \Rightarrow b$, $B \Rightarrow bq_1$, $q_1 \Rightarrow Aq_2 \mid b, q_2 \Rightarrow a\}$

Stopping the extensions, we get a given final version FSA for each nonterminal. Each FSA defines one language over terminals and another over $N \cup T$. Following the approach in Section 4.1 we can use the FSA-s as extended relational schema graphs.

We can get the simplest case, the schema graph defined by edges with terminal labels if we delete all the edges labeled with nonterminal. The result is an edge-labeled schema graph. It is possible to use it in the same way as the node-labeled schema graph is used in Section 4.1.

Using the notation in Construction 1 , let M_A for $A \in N$ denote from now on the actual version of the graphs in the construction. We call a subgraph from these graphs an extended schema graph if it satisfies the following restrictions:

Let $w = w_1 \dots w_n$ be a sequence of edge labels along a traversal in M_A from $Start_A$ to End_A We call it an admissible traversal if it is an ordinary sequence where all labels are terminal or, if a nonterminal B occurs in the sequence, than there is at least one admissible traversal from the $Start_B$ node to the End_B node of the inserted version of M_B.

Claim 2. *An admissible sequence can be extended to an ordinary sequence by a series of substituting non-terminals by admissible traversals.* (*This process is equivalent to a derivation tree of the final ordinary sequence.*)

The definition of tuple types starts with a selection of an admissible sequence. The terminal labels specify simple attributes with elementary value types, and a nonterminal label specifies possible complex types. Then for each nonterminal, an admissible traversal should be selected as a tuple type. At the end, an ordinary sequence gives the flat tuple type. During the substitution of a nonterminal, we have to decide if a simple tuple is to be included in the outer tuple, or a set of tuples, as a nested relation value is the choice.

Definition 7. (*Shema graphs for Extended relations defined by context-free languages.*) M_S is the schema graph of the extended relation defined by CFG G, and the graphs M_A for $A \in N$ are the schema graphs for nested attributes. Schemas are given by tuple types. A tuple type is given by an admissible sequence.

Example 2. From Figure 4 a sequence of a schema selection is: $w = a\ B\ b > a\ B[b\ A\ a]\ b\ > a\ B[b\ A[a\ b]\ a]\ b$.

From this selection, the final ordinary sequence is a $b\ a\ b\ a\ b$. The nonterminal B may be an attribute with complex vale of tuple-type $b\ A\ a$. Replacing A in this type by the ordinary sequence $a\ b$, we get the schema $a\ B[b\ a\ b\ a]\ b$.

Functional dependencies for extended relations.

The verification of a functional dependency does not depend on the sort of values. So in the selection of defining attribute sets for an FD, we don't take care of the elementary or complex attribute types. The next definitions are similar to regular functional dependencies that are presented in [15, 16]. They are syntactically defined on the graph for the accepting FSA of a regular language, and semantics were given for them on sentences of the language. We extend this definition to $ECFG$ with nested attributes so that the syntax of the FDs will be defined on the graphs from Construction 1. We don't need to make distinction between attribute types since only the equality of two values play role in checking satisfaction of FD-s. This new extended functional dependency is defined on scopes. The scope is related to a nonterminal, $A \in N$, and defined on the extended graph M_A of grammar $G_A = \{NUT, N_A, P_A, S_A\}$. The following definitions specifies the selection of a sequence of edges for each traversal of the M_A. The edge labels give the sequence of attributes.

Definition 8. (*Assignment*) Let $A \in N$ be a non-terminal symbol, $G_A = \{N \cup T, N_A, P_A, S_A\}$ is the regular grammar and let $M_A = (V, E)$ be the schema graph of G_A (Definition 7). We call A the scope of selection. We say that the tuple $Y = (Y1, Y2)$, where $Y1 \subseteq E$ and $Y2$ is a sub graph of the transitive closure of M_A is an assignment on M_A. $Y1$ is taken from non-recurred part of M_A and from the first traversals of cycles, $Y2$ refers to nodes and edges whose are (could be) repeatedly visited during a traversing.

Let Y be an assignment, Y selects a unique subsequence from a given sentence $w = \{v_1, v_2, \ldots, v_n\} \in L(G_A)$ as follows:

Definition 9. (*Selection on Scope A*). Let $Y = (Y_1, Y_2)$ be an assignment on M_A for the scope A and let w be a traversing on M_A. The edges in Y_1 will be selected in order of their exploration (when visited). For each edge $e \in Y_2$ when (may be several times) the edge will be closed between its endpoints during the traversing on w, the first and the last edge of the connecting path will be selected in their succession order (when visited at all). It may occur that a labeled edge has been selected already, this case we don't select duplicates. The edges in Y_2 will give selection by each visiting (if any) during the traversing on walk (w). The selection will be processed for all edges in Y_2 autonomously. By the end of the edge-selection the labels of the selected edges build up the (possibly empty) sequence $w(Y) = (v_{i_1}, \ldots, v_{i_k})$ $(1 \leq i_1 < i_2 < \cdots < i_k \leq n)$ $(k \gg 0)$.

Note: It is possible that more than one edges from Y_2 close at the same time. In this case the first edges are selected in the traversal order; the last edges are the same.

Let $\omega = (v_1, v_2, \ldots, v_n)$, be a tuple type defined by the traversing w on M_A. Let $t =< v_1{:}x_1,\ w_2{:}x_2, \ldots, v_n{:}x_n >$ be a tuple of type ω. We interpret the selected $w(Y) = (v_{i_1}, \ldots, v_{i_k})$ sequence of symbols as list of "attributes" and the projection of the tuple t to $w(Y)$ is $t[Y] =< v_{i_1}{:}x_{i_1}, \ldots, v_{i_k}{:}x_{i_k} >$?. If $w(Y) = \{\}$, then $t[Y] = \{\}$ as well.

Concerning the regular language $L(G_A)$ we can define functional dependency over M_A, considering the non-terminal A as the scope for the functional dependency (Definition 9).

Definition 10. (*Scoped Functional Dependency*). Let A be a scope in the $ECFG$ G and let M_A be the corresponding graph representation. Let $X = (X_1, X_2)$ and $Y = (Y_1, Y_2)$ be two assignments (Definition 8) over M_A. A functional dependency defined over M_A (FD_A) is an expression of the form $X \to Y$. The R (finite) database instance of A satisfies the $X \to Y$ functional dependency (denoted by $R \mid = X \to Y$), if for any two tuples $t_1, t_2 \in R$ of type w_1 and w_2 and $w_1(X) = w_2(X)$, $w_1(Y) = w_2(Y)$ and $t_1[X] = t_2[X]$ then $t_1[Y] = t_2[Y]$ also comes true. We call the case $Y = M_A$ key dependency.

It is possible to extend the scoped functional dependency with the scoped FD on a nonterminal attribute B. It is applicable when B is used as un-nested, simple tuple insertion. For a formal definition, we insert the edge labeled by B into $Y1$ with additional specification of a scoped selection $B(Z_1, Z_2)$ on B. The new assignment requests the insertion of the selected sequence in place of B.

In the case of scope S FD_S specifies the functional dependency on the extended relation, and the in case of other non-terminals FD_A specifies constraints for the values of the embedded relation values for nested attribute A.

The implication problem for scoped FD-s is decidable based on a version of the Chase algorithm using two colored versions of the schema graph. The proof is the same as in the case of regular extended relations in Section 4.1, see in [28].

5.2. Graph representation for context-free languages given by regular expressions

This section presents the model based on the second graph representation. The starting point is the choice of a special form of context-free grammar. For each non-terminal, we define a regular expression over terminals and non-terminals as the set of possible substitution rules. The graph representation uses the construction described in Section 3. Nonterminal nodes are closed between an IN and OUT node. The IN and OUT nodes of the non-terminal node can also be connected here with the current graph belonging to the node-label. A finite subgraph of an infinitely expandable graph is considered a schema graph. For this, we define specific nested relational schemas.

We can replace the regular grammars in the Definition 6 of $ECFG$ by equivalent regular expressions. In this section, regular expressions shall be used to define the rules for extended context-free languages. The following definition rephrases Definition 6.

Definition 11. (*Extended Context-Free Grammar* ($ECFG$)). $G = (N, T, S, P)$ is a context-free grammar given by regular languages, where T is the set of terminal symbols, N is the finite set of non-terminal symbols, P is the set of production/derivation rules, and S is the sentence symbol. The right side of production rules are given by regular expressions: for all p in P, $p = \{A \Rightarrow w \mid w \in L(E_A(N \cup T)), A \in N, E_A$ is a regular expression over $N \cup T\}$.

This special form of CFG rules gives the possibility to construct an infinitely expanding graph for each nonterminal. Similarly, to the schema graph introduced in Section 5.1, finite versions of the graph can be used as schema graphs.

The advantage of using regular expressions instead of regular grammar is the more convenient and frequent use in data modeling. In particular, it is the basic formal tool in XML DTD declarations. The following special DTD structure demonstrates the use of the ECFG. Let select a set T of ELEMENT names for simple structure, and a second set N for complex structures. The form of the DTD:

For each $x \in T$ the declaration is:

<!ELEMENT x (#PCDATA)>

For each $X \in N$ the declaration is:

<!ELEMENT X ($\mathrm{E_X}$(N ∪ T)>, where $\mathrm{E_X}$(N ∪ T) is a regular expression over N ∪ T.

Example 3. Let G be the following $ECFG$:

G={R => (a b (A + B)*), A => (c B*), B => (d A*)},

An associated XML DTD fragment:

```
<!ELEMENT TABLE(R*)>
<!ELEMENT R (a,b,(A|B)*)>
<!ELEMENT A (c,B*)>
<!ELEMENT B (d, A*)>
<!ELEMENT a (#PCDATA )>
<!ELEMENT b (#PCDATA )>
<!ELEMENT c (#PCDATA )>
<!ELEMENT d (#PCDATA )>
```

In Construction 2 we shall use the graph constructed by Algorithm 1 for regular expressions. Similarly, as in the RSM case, we shall extend the graphs by adding subgraphs for nonterminal nodes. The technical solution is the *elementary extension step.*

You can insert the appropriate expression graph in place of a non-terminal vertex, so you can get newer, longer traversals. All the finite traversals through the infinite graph obtained by infinitely extending the graph of the sentence symbol give the sentences of the language.

For each regular expression $E_A(N \cup T)$ defining the production rule for A, the language $L_A \subseteq (N \cup T)^*$ denotes the generated language and the corresponding graph-representation can be constructed according to Algorithm 1 using $U = N \cup T$ as input alphabet. During derivation by the grammar G we substitute the non-terminal A by a sentence from L_A, so the vertex-labels set by Algorithm 1 will be picked from either N or T. In order to demonstrate the path that leads to the ordinary attributes of a schema we use the dotted list of non-terminal symbols that will be used during the derivation process leading to the given terminal symbol. In the first step we use the start symbol S as the beginning. In order to create a graph for the extended context-free language *L(G)* generated by the grammar G we should repeat the construction for all vertices, labeled with a symbol $X \in N$. A nonterminal node is considered as a regular expression in brackets, so it has its own **IN** and **OUT** nodes. In the elementary extension step for node A we add an expression graph connecting the **IN** and **OUT** of A.

Construction 2. (*Construction of the infinite graph for an extended context-free language.*) The result is denoted by $G^*(E_A)$ graph for each $A \in N$. $G^*(E_A)$ is the *full language graph* of nonterminal A.

Let G be an *ECFG* as it is given in Definition 3. We start with the construction of graphs $G(E_A)$ from $E_A(N \cup T)$ using Algorithm 1 for each $A \in N$ to create the starting expression graphs. Note, that for a nonterminal node A we use the $\boldsymbol{IN} \rightarrow A \rightarrow \boldsymbol{OUT}$ form.

1. **Elementary extension step**: Suppose that an $\boldsymbol{IN} \rightarrow A \rightarrow \boldsymbol{OUT}$ nonterminal node that occurs in the current version of an extended graph F has no more paths from $\boldsymbol{IN}$ to $\boldsymbol{OUT}$. We call this occurrence of A *non-extended* nonterminal node. Take a copy of the $G(E_A)$ graph and unite the two $\boldsymbol{IN}$ node and $\boldsymbol{OUT}$ node. See Figure 5.

2. **Infinite iteration step**: (deterministic version) in a cyclic order on N, for a nonterminal A perform all the possible elementary extension steps. Follow for the next nonterminal.

3. **Suspend step**: in order to have a finite subgraph, the construction after a cycle of iteration can be suspended. The result after k cycles is denoted by $G^k(E_A)$ and k is the level of the language graph.

Example 4. Let *G({R, A, B},{a, b, c, d}, R, P)* be an extended context-free grammar where *P = {R => (a b (A + B)*), A => (c B*), B => (d A*)}*.

Figure 6 (left) shows the starting graphs for each nonterminal and Figure 6 (right) shows the first extension step for $G(E_R)$.

Instead of using the small $G(E_A)$ graph we can use the current version of the extended graph for nonterminal A in an equivalent way to create the infinite graph. Our aim is to create a language graph as a representation form for extended relations and tuple-types.

Definition 12. *Schema graph for ECFG extended relations.* Let us fix a version $G^k(E_A)$ and delete all non-extended nodes with non-terminal label. Then, delete all nodes with no outgoing edge repeatedly. The remaining graph $SCH^k(E_A)$ is a schema graph for scope A.

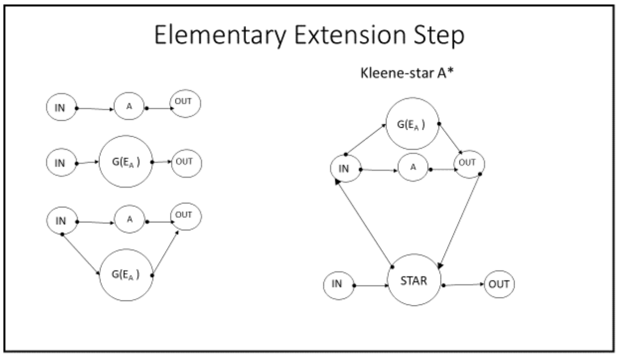
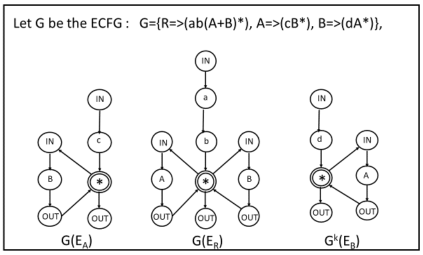


Figure 5. Elementary extension steps



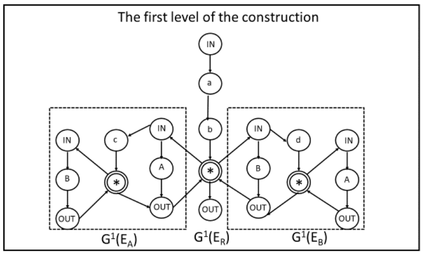


Figure 6. Starting expression graphs (left); The first level construction for $G(E_R)$ (right)

The most important property of the schema graph is that any traversal from its IN and OUT node can be extended to a traversal containing only nodes with terminal labels. The proof is based on the fact that for any remaining nonterminal B for the **IN**→B→**OUT** path then the extension with *SCH*^{l}*(*E_B*)*, must be a schema graph of lower level. The proof can be finished by induction on the levels of extensions. In the following we omit to use the upper index denoting the level of a schema graph unless it has some role.

Any traversal of *SCH(*E_A*)* defines a tuple-type containing possible non-terminal symbols. Each non-terminal B symbol has its own *SCH(*E_B*)* schema graph as it is given in *SCH(*E_A*)* connecting the IN and OUT nodes of B. The next step is to select a traversal on this *SCH(*E_B*)* as a tuple type for B. Still a data modeling decision remains: one option is to accept the tuple-type for B as one tuple included in the tuple generated for the outer path; another option is to define B as a nested table-type.

In the first case, we include each attribute x in the $B.x$ form into the outer tuple schema as a dotted notational form, in the second case the list of the attribute is closed in brackets after **B**, like **B** (list of attributes).

A more general option for assigning data-type to a non-terminal node **B** is to use its given schema graph *SCH(*E_B*)* and the domain is the set of the instance of the extended relation. We omit the discussion of this option.

Definition 13. (*Schema graph with nested attributes for ECFG extended relations.*) Let $SCH(E_A)$ be the selected schema graph for the scope $A \in N$ obtained for *ECFG G* by *Construction* 2. Each occurrence of nodes with label B may represent either a sequence of attributes formed by a traversal of its extension $SCH(E_B)$, or a nested relation type, defined by its extension $SCH(E_B)$ schema graph. In the letter case the bold face version of the nonterminal will be used. A traversal of $SCH(E_B)$ gives a tuple type. Denote the sequence of node labels before reaching the IN node of B by x, and after leaving the OUT node by y. If B is of simple version and z is a traversal from its IN node to OUT node, then the resulting traversal will be $y\ z\ x$. In the nested case, we get the traversal $x\ B\ y$, and the attribute B is a nested relation of schema z, or, considering a more general possibility, the value type for B is an *ECFG* extended relation defined by $SCH(E_B)$.

Example 5. Using the grammar G form Example 2 the sequence of Figures 6 and 7 is an example how the schema graph $SCH(E_R)$ obtained from $G^2(E_R)$. Deleting the nodes in red circles result in $SCH^2(E_R)$ Denoting both occurrences of B as nested attribute, we can get the nested schema graph by deleting nodes in blue circles.

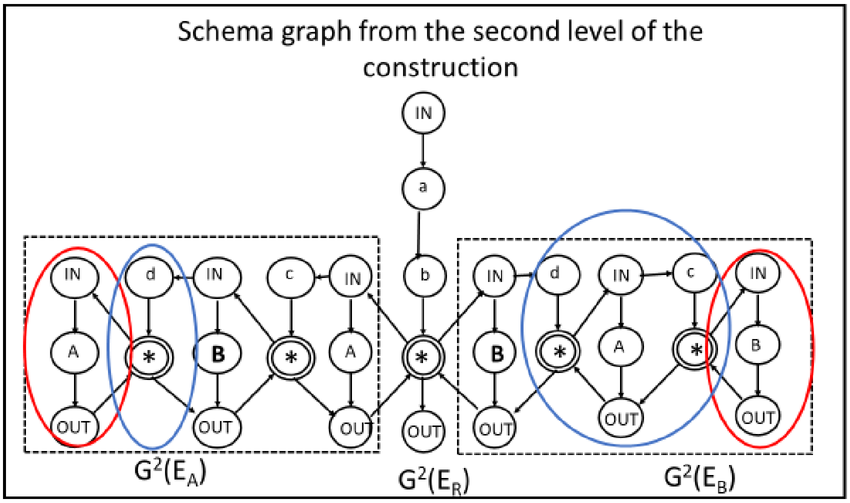


Figure 7. A possible schema graph for G({R, A, B}, {a, b, c ,d}, R, P) , where P= {R => (a b (A + B)*), A => (c B*), B => (d A*)}

Each time of the selection of one or more cycles for a Kleene-star it is possible to select different traversals for the non-terminals.

After the selection of the schema graphs and the nested attributes is done, the graph became a simple graph of a complex regular expression. So all the definitions used in Part 2. are applicable.

The representation of a legal traveling of the schema graph is a sequence of terminal and non-terminal symbols. For a nested attribute B the traversal in $SCH(B)$ is given in brackets [ ]. We use the prefix dot-notation for the labels of the traversals of non-nested attribute B, like $B.x$.

There are the following possible generated (ordinary and nested) traversals from $G^2(E_R)$:

$R(a.b)$, $R(a\ b\ A.c)$, $R(a\ bA.c\ A.\mathbf{B}[d])$, and using the Kleene-star of A : $R(a\ b\ A.c\ A.\mathbf{B}[d]A.\mathbf{B}[d]), \ldots$

$R(a\ b\ \mathbf{B}[d])$, $R(a\ b\ \mathbf{B}[dA.c])$, and using the Kleene-star of B : $R(a\ b\ \mathbf{B}[d\ A.c\ A.c])$, $R(a\ b\ \mathbf{B}[d\ A.c\ A.c\ A.c]), \ldots$

and finally using the Kleene-star of R any sequence of the former traversals is an accepted traversal.

Our aim is the construction of relational schemas with attribute names of possible having some meaning. So the above notation of the traversals needs the separation by comma of the node-labels. For the nested attribute the brackets { } notation expresses the set type of the domain of the attribute.

Some rewritten form: $R(a, b, A.c, A.\mathbf{B}\{d\}, A.\mathbf{B}\{d\})$, $R(a, b, \mathbf{B}[d, A.c, A.c, A.c])$

Functional dependency for ECFG extended relations.

Before defining functional dependency for ECFG extended relations it is important to make all the type selections for nonterminal nodes. If B has a simple tuple type, then the list of attributes should be included in the attribute sequence used to specify a functional dependency. In the case of **B** having nested relational type, the attribute **B** should be considered as an ordinary member of the attribute list.

Let $SCH(E_A)$ be the schema graph for scope A and for all nodes with nonterminal labels the selection of a simple or bold version is given. After deleting the schema graphs for bold attributes, the remaining graph is a simple graph representation of a regular expression in the form given in Algorithm 1. This way we reduced the definition of functional dependencies to the case in Section 4.2, Definition 2–4.

6. Instances of extended relations, operations, implementations

6.1. ECFG extended relation instances

Instances of relational databases with complex values consist of a finite number of complex relational schemas and a finite set of values of the sort given by each schema. (See [2] Chapter 20.) Following this, we define the instance of a nested $ECFG$ extended relation.

For the given $ECFG$ we can suppose, without any loss of generality, that the language $L(A)$ associated with any non-terminal symbol A is not empty. Using this assumption, any traversal generated by Construction 1 and 2 can be extended to a legal schema instance by a finite number of construction steps. Using the notation in Definition 12 legal schema instances for a scope A are traversals of the selected schema graphs $SCH^k(E_A)$. Using the level k of the schema graph requires that the schema selection remains inside to the current graph of the construction. See in Figure 7.

Let $SCH(G)$ denote the set of schemas given by language $L(G)$.

A schema instance from $SCH(G)$ specifies a tuple-type. For a non-terminal symbol B in this sort a tuple-type is given from $SCH(B)$. The associated value of B in a relation instance can be a simple tuple (un-nested case) or a table as a set of tuples from this type (nested case). Attributes with terminal names can take only simply values. Using the dotted prefix notation for denoting the nonterminal of the current sub-traversal proved to be helpful in defining nested operations.

Definition 14. Relation instance of the context-free schema $SCH(G)$ is given by the pair $(\boldsymbol{R},\boldsymbol{I})$, where $\boldsymbol{R}$ is a finite subset of $SCH(G)$, and $\boldsymbol{I}$ is the set of complex-valued relation instances for each element of $\boldsymbol{R}$. For $r \in \boldsymbol{R}$ the corresponding instance is denoted by $\boldsymbol{I}(r)$.

6.2. Operations on complex relations

Set operations are defined over two instances $(\boldsymbol{R_1}, \boldsymbol{I_1})$, and $(\boldsymbol{R_2}, \boldsymbol{I_2})$. of the context-free schema *SCH(G)*. The operations have to be defined on the relation instance level as ordinary set operation. The schema of the result depends on the results of the operations on pairs of relation instances.

The **Union** operation: $(\boldsymbol{R_1}, \boldsymbol{I_1}) \cup (\boldsymbol{R}\ , \boldsymbol{I_2}) = (\boldsymbol{R}, \boldsymbol{I})$ where the new extended schema $\boldsymbol{R} = \boldsymbol{R_1} \cup \boldsymbol{R_2}$ is the union two subset of schema instances from *SCH(G)*. Then for each schema r $\in$ $\boldsymbol{R_1} \cap \boldsymbol{R_2}$ the relation instance is $\boldsymbol{I(r)} = \boldsymbol{I_1(r)} \cup \boldsymbol{I_2(r)}$. For r $\in \boldsymbol{R_1} - \boldsymbol{R_2}$ the relation instance $\boldsymbol{I(r)} = \boldsymbol{I_1(r)}$ and for r $\in \boldsymbol{R_2} - \boldsymbol{R_1}$ the relation instance $\boldsymbol{I(r)} = \boldsymbol{I_2(r)}$.

The **Intersect** operation: $(\boldsymbol{R_1}, \boldsymbol{I_1}) \cap (\boldsymbol{R_2}, \boldsymbol{I_2}) = (\boldsymbol{R}, \boldsymbol{I})$, where for each schema r $\in$ $\boldsymbol{R_1} \cap \boldsymbol{R_2}$ the relation instance is $\boldsymbol{I(r)} = \boldsymbol{I_1(r)} \cap \boldsymbol{I_2(r)}$. The schema of the result is $\boldsymbol{R} = (\boldsymbol{R_1} \cap \boldsymbol{R_2}) - \{r | \boldsymbol{I_1(r)} \cap \boldsymbol{I_2(r)} = \emptyset\}$.

The **Minus** operation: $(\boldsymbol{R_1}, \boldsymbol{I_1}) - (\boldsymbol{R_2}, \boldsymbol{I_2}) = (\boldsymbol{R}, \boldsymbol{I})$ where for each r $\in$ $\boldsymbol{R_1} \cap \boldsymbol{R_2}$ the relation instance is $\boldsymbol{I(r)} = \boldsymbol{I_1(r)} - \boldsymbol{I_2(r)}$ and for r $\in$ $\boldsymbol{R_1} - \boldsymbol{R_2}$ $\boldsymbol{I(r)} = \boldsymbol{I_1(r)}$.
The schema of the result is $\boldsymbol{R} = (\boldsymbol{R_1} - \boldsymbol{R_2}) \cup \{r | \boldsymbol{I_1(r)} - \boldsymbol{I_2(r)} \neq \emptyset\}$.

Cross-product can be defined over instances from two schemas generated by context-free grammars G_1 and G_2. Using the standard notation *SCH*(G_1) $\times$ *SCH*(G_2)= *SCH*(G_1G_2), where G_1G_2 is the grammar of the concatenated languages. It is important to require that for the sets of non-terminals $\boldsymbol{N_1} \cap \boldsymbol{N_2} = \emptyset$. At the instance level, for ($\boldsymbol{R_1}$, $\boldsymbol{I_1}$) , and ($\boldsymbol{R_2}$, $\boldsymbol{I_2}$). form *SCH(G₁) and SCH(G₂)* respectively the cross product is taken for each pair $r_1 \in \boldsymbol{R_1}$ and $r_2 \in \boldsymbol{R_2}$.

Remark 1. The common grammar for the cross product is the union of the two set of production rules and an additional rule $\boldsymbol{R} \Rightarrow \boldsymbol{R_1},\boldsymbol{R_2}$. The self-cross product $\boldsymbol{R} \times \boldsymbol{R}$ needs a renaming of the non-terminals.

The definition of the **projection** and **join** operation needs the subsequence selection method from Section 4.2.

We should delete nodes from the actual schema graph $G(E_R)$ by shortcutting two nodes on the same path. Shortcutting is allowed only between nodes with terminal or non-terminal labels. After a number of shortcutting, the remaining $GS(E_R)$ graph is a projection graph on $G(E_R)$. Each traversal w on $G(E_R)$ defines a traversal on $GS(E_R)$, which gives a subsequence $S(w)$ of the original attribute sequence.

The **projection** operation: The projection of ($\boldsymbol{R}$, $\boldsymbol{I}$) onto the subsequence selection $GS(E_R)$ projects each tuple from a relation instance of sort w to $S(w)$. The schema graph of the projection is $GS(E_R)$.

The **join** operation: Let $\boldsymbol{R1}$ and $\boldsymbol{R2}$ be extended relations defined by schema graphs $G(E_{R1})$ and $G(E_{R2})$. The two subsequence selection on the schema graphs are $GS_1(E_{R1})$ and $GS_2(E_{R2})$ selection expressions on E_{R1} and E_{R2} respectively with $GS_1(E_{R1}) = GS_2(E_{R2})$ Suppose that for two traversals $w_1 \epsilon\ G(E_{R1})$ and $w_2 \epsilon\ G(E_{R2})$the condition $S_1(w_1) = S_2(w_2)$ holds. The following construction specifies the join-schema defined by SE_1 and SE_2:

For two consecutive nodes α and β in $S_1(w_1) = S_2(w_2)$, α x β is the path in $G(SE1)$, and α y β is the path in $G(SE2)$, than the conjunctive joined path between α and β will be α x y β. The result of conjunctive join of tuples t of sort w_1, s of sort w_2 and with $t[S_1(w_1)] = s[S_2(w_2)]$ is: $t \bowtie s[\alpha\ x\ y\ \beta] = t[\alpha\ x]\ s[y\ \beta]$.

In the disjunctive version the join graph between α and β will be defined by $\alpha\ (x + y)\ \beta$.

The ECFG expression of the join is still an open problem.

The **self-join** operation: in the special case of $\boldsymbol{R1}$ and $\boldsymbol{R2}$ having the same $G(E_R)$ schema graph and one subsequence selection S we further require that the $S(w_1) = S(w_2)$ is fulfilled when they are the same subgraph of $G(E_R)$. In this case, the types of result of the disjunctive join remain in the types of $G(E_R)$. Both sub-paths x and y from α to β is a legal sub-path of $G(E_R)$. Depending on the length n of $S_1(w_1) = S_2(w_2)$, the number of possible tuple types of the join is 2^{n+1}. So, the size of the result of the join of two matching tuples might be as large as 2^{n+1}. The sub-paths to the first node and from the last node of $S_1(w_1)$ and $S_2(w_2)$, might be different in w_1 and w_2.

Remark 2. The lossless-join property of decompositions can be defined by the self-join operation. The lossless-join property of a decomposition according to an extended functional dependency holds.

The **Nest** and **Un-nest** operation: The distinction of a nested and an un-nested version of a given nonterminal in the schema graph give the possibility to define these operations. These operations are defined syntactically on two instances $r_1 \in R$ and $r_2 \in R$. The traversals on the schema graph are the same for r_1 and r_2 but the choice of nested version or simple version of traversing the graph of the same node labeled non-terminal B is different; B in r_1 and $\mathbf{B}$ in r_2.

The result of Nest(***I(r₁)***) is of sort r_2, and for tuples of ***I(r₁)*** having the same values on all the attributes except the attributes defined by B, the result is only one tuple of the common part, and a new attribute **B** is inserted in place of B.[attribute list]. (We use here the dotted prefix attribute notation.) The value for **B** is a nested table consisting of the sub-tuples of B.[attribute list]. (We use here the dotted prefix attribute notation.)

In the opposite direction, the result of Un-nest(***I(r₂)***) is of sort r$_1$. The result consists for each tuple $\boldsymbol{t} \in$ ***I(r₂)***) as many new tuples of sort r$_1$ as the number of the tuples in the nested table for **B**.

See the exact definition of these operations in [2] Chapter 20.2.

Remark 3. A more general nested value of $\boldsymbol{B}$ is the extended relation of sort $G(E_B)$. In this case we need de-grouping them to a set of ordinary tables. Then for each table the Un-nest operation is to be performed. In the opposite direction first comes the Nest operation for each relation type from the actual $G(E_B)$ and then the grouping of the nested tables finishes the nesting. Let the nested value of B is given by the pair ($\boldsymbol{R}$, $\boldsymbol{I}$), where $\boldsymbol{R}$ is a finite subset of SCH(G$_B$), and $\boldsymbol{I}$ is the set of complex valued relation instances for each element of $\boldsymbol{R}$. For r $\in \boldsymbol{R}$ *the corresponding instance is denoted by* $\boldsymbol{I}(r)$. *Let the structure of a tuple with nested value* $\boldsymbol{I}$ *be* x $\boldsymbol{I}$ $\boldsymbol{y}$. *The group-destroy will be the set of nested tuples* $\{x\ \boldsymbol{I}(r)\ y \mid r \in \boldsymbol{R}\ \}$. *Finally, the full un-nested result will be the union of Un-nest(*$x\ \boldsymbol{I}(r)\ y$*) for all* r $\in \boldsymbol{R}$.

6.3. Implementation issues

In the following, we outline how ECFG extended relations can be implemented based on standard SQL.

For each schema instance r $\in R$ an SQL table has to be created. Operation specified on the extended relation has to be performed on each created table. The dotted prefix notation of the nodes uniquely identifies a traversal like the traversal R(a, b, A.c, A.B{d}, A.B{d}) from Example 4.

The sketch of a possible solution:

Creation of extended relation:

```
  CREATE E-RELATION <e-relation-name> (
grammar <ECFG>,
level <n>,
TABLES <e-tablename>);
```

where ECFG is an extended context-free grammar, n is the level of the schema graph construction. The TABLES refer to the name of auxiliary relation that stores the traversals defining the actual schema-instance:

```
  CREATE TABLE <e-tablename> (
    schema-id INT PRIMARY KEY,
    schema-specification VARCAR (256));
```

The generation of the SQL table instances uses the E-schema table and generates for each schema-specification a TABLE named by the schema-id and uses the dotted prefix notation to list the attributes in the corresponding CREATE TABLE instruction. The value type of a nested attribute should be defined by the User Defined Data option of SQL. Operations on e-relations are implemented by a special procedure that creates the corresponding SQL statement for each SQL table related to each row of the E-schemas table. For set operations, the following statement according to the standard SQL must be created for each pair of tables with the same schema:

```
(SELECT * FROM r1)
[UNION| INTERSECT| MINUS]
(SELECT * FROM r2);
```

The values associated to the individual occurrences of an attribute name compose a list of values. This makes an option to store tuples in standard SQL tables with specific values. Let for an attribute b the positions of the occurrences be $n_1, \ldots, n_k$, and the corresponding values be $x_1, \ldots, x_k$. The specific value of attribute b is $<x_1 : n_1, \ldots, x_k : n_k>$. Using this form of values for each attribute name, the original serial form of the tuple can be obtained. The schema of the tuple uniquely defines this value type for each attribute name.

7. Documents and X-relations

7.1. X-relations in XML documents

Let us start with an observation on embedded X-relations in special large XML documents.

From large XML document the occurrences of a special ELEMENT may form an instance of an extended relation. As an example, the special DTD as given in Example 6. can be translated to an ECFG grammar. It is possible to create an E-RELATION schema and load the data to the tables from the XML document. An ECFG extended relation is a collection of relations each having a different schema from the given schema graph. It is a generalization of the multi-relational model [18]. Here we concentrate on the formal structure of the tuple-types whilst the aim of the multi-relational model was to find some common meaning and operations of similar schemas.

Next example is an XML DTD and a document fragment to show the equivalent representation of the extended relation given by the ECFG G from Example 2.

Example 6. Let G be the ECFG from Example 2.

G={R => (a b (A+B)*), A => (c B*), B => (d A*)},

An associated XML DTD fragment:

```
<!ELEMENT TABLE(R*)>
<!ELEMENT R (a,b,(A|B)*)>
<!ELEMENT A (c,B*))>
<!ELEMENT B (d, A*)>
<!ELEMENT a (#PCDATA )>
<!ELEMENT b (#PCDATA )>
<!ELEMENT c (#PCDATA )>
<!ELEMENT d (#PCDATA )>
```

A fragment of XML document instance representing a long schema R(a b A[c B[d] B[d A[c] A]c]]). It generates the ordinary schema R(a, b, c, d, d, c, c), in dotted form R(a, b, A.c, A.B.d, A.B.d, A.B.A.c, A.B.A.c).

In case of no nested type, the corresponding XML fragment:

```
<R> <a>1</a> <b>2</b>
  <A> <c>3</c>
<B> <d>4</d> </B>
<B> <d>5</d> <A> <c>6</c></A><A> <c>7</c></A> </B>
 </A>
</R>
```

In case of a nested type; R(a b A[c **B**[d] B[d c c]), the bold B specifies nested relation. The change in the DTD for ELEMENT B is :

<!ELEMENT **B** (d, A*)|(d, A*)*>

We use here the bold face only to visualize the difference between the selection of a single tuple and the selection of a set of tuple of the type (d, A*). The corresponding XML fragment:

<R> <a>1</a> <b>2</b>

  <A> <c>3</c>

<**B**> <d>4</d> <d>8</d> <d>9</d> </**B**>

<B> <d>5</d> <A> <c>6</c> </A> <A> <c>7</c> </A> </B>

```
    </A>
  </R>
```

7.2. X-relations and free documents

End-user document formats typically contain names and texts that can be interpreted by end users, which ensures the understanding of individual occurrences. In traditional IS modeling the information service is supported by a separate database by loading or extracting the data into or from end-user documents. The papers [22] and [23] introduce the free documents, which have an included structure of specific fields for accepting the actual data. The specific fields have associated variable names. So a set of free documents are joined together via common field variables. This gives a hypergraph join-structure of documents. Free documents are similar to templates in mail-merge systems. Using an extended relation for merging data with a set of free documents is a feasible future application of our model.

The term "free documents," similar to the concept of "free tuples" in tableau queries, can be seen as documents containing free variables. As these documents undergo completion, more and more variables get values assigned, ultimately reaching a state where no variables remain that are not valuated. We refer to them as "ground documents". A ground document does not contain free variables, all placeholders are bound to constants in the ground domain.

From one system role's perspective, some parts of the documents may be considered finalized, while other parts may still contain free variables requiring further processing by other system roles. To attain a stable state for an instance of a larger business process within an Information System, all documents involved must become ground documents.

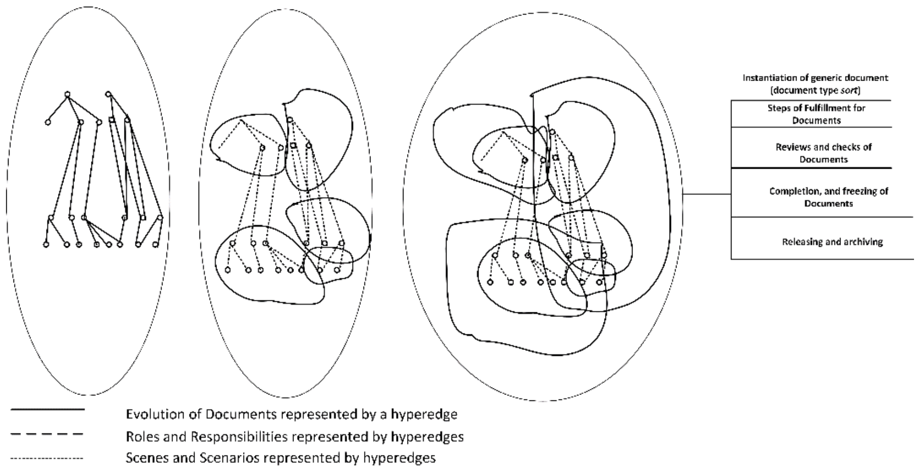

Instantiation of generic document (document type *sort*)
Steps of Fulfillment for Documents
Reviews and checks of Documents
Completion, and freezing of Documents
Releasing and archiving
Evolution of Documents represented by a hyperedge
Roles and Responsibilities represented by hyperedges
Scenes and Scenarios represented by hyperedges

Figure 8. The development or evolution of documents during processing represented in hypergraph

In addition to defining the logical processes for retrieving and modifying data, the document model should also encompass descriptions of the sequences of interactions between documents. Furthermore, it should handle collections of documents.

Distinctions can be made between documents, categorizing them as either static or dynamic. The structure of a dynamic document has the potential to change based on the system's response or indications from system roles. This response can generate instances of a general dynamic document, leading to a sequence of free documents. These free documents gradually transform into ground documents, progressing from generic to intentional forms and ultimately reaching finalization. See on Figure 8. Ground documents exclude free variables, allowing the names of variables within them to be integrated into the database's namespace.

The proposed approach of using the extended relations on document modeling of Information Systems makes it possible for the processed documents and the activities that process them to use a more complex data structure for exchanging data with the database.

7.3. Merge structures and X-relations

Mail merge systems are typical examples of the use of free documents. Mail merge lets you create a batch of documents that are personalized for each recipient. For example, a form letter might be personalized to address each recipient by name. A data source, like a list, spreadsheet, or database, is associated with the document.

A mail merge system, like MS Word, MS Excel, and a mailer such as Outlook supports the creation a special free document and later merge it with a list or table. During the creation of the main document you can insert merge fields as placeholders in the document Each merge field has associated match field from the table. That way, the system matches the merge fields in the current document to the fields in the data source.

In a similar way, we can create a free document template and connect it to a relational table, and perform the merge process tuple by tuple. The names of the merge fields can be chosen to be the same as the attribute names of the relational table, for example in the form ≪*attribute name*≫. The positions of the merge fields are irrelevant since the matching is performed by attribute names.

Next, we extend the document merge structure for X-relations.

In a specific free document, the merge fields are inserted using the form ≪*attributename*≫. A merge field variable can be inserted several times. The field names form a string in the serial form of the document. (If the document is two- or multi-dimensional, the fields must be numbered, for example in the form ≪n, *attribute name*≫..) This string has to be matched to an X-relational tuple type according to the sequence of attribute names.

The proposed free document is constructed using a formal language, similar to the XML DTD. The basic lexical elements of the language should be <text1≪*attribute name*≫text2>. The grammar only uses the attribute name. The context is represented by the text field in the document. The placeholder is ≪*attribute name*≫. The surrounding text is associated to the attribute.name. The grammar specifying the X-relation is the same as the grammar specifying the accepted forms of the document for the attribute name values. Thus, the X-tuple can be extracted from any filled-in (ground document) document, and an x-tuple can be inserted into a free document.

Formal definition of the X–merge structure

Let the G = (N, T, S, P) context-free grammar given by regular expressions, where T is the set of terminal symbols, N is the finite set of non-terminal symbols, P is the set of production/derivation rules, and S is the sentence symbol. The right side of production rules are given by regular expressions: for all p in P, $p = \{A => w \mid w \in L(EA(N \cup T)), A \in N, EA$ is a regular expression over $N \cup T\}$. Further, let $SCH(E_S)$ be a finite schema graph for the scope S.

The extended merge-schema is related to the finite schema graph $SCH(E_S)$. A traversal of $SCH(E_S)$ gives (a tuple type as a sequence of terminal and non-terminal labels $w= x_1 \ldots x_n$. Using the notations from Definitions 12 and 13, the bold-face version of a nonterminal represent a nested relation with a given tuple type. For each occurrences of a nonterminal a unique tuple type has been given by the traversal. The dot notation and the brackets are only introduced for highlight the structure.

Before starting the creation of a matching document to w, we have to select a surrounding text for each terminal and nonterminal symbols using the form <text1≪*symbol*≫text2>. Bold face nonterminal may have specific surrounding. Let *d(x)=<x-text1≪x≫x-text2>* denote the selected surrounding of label x. The merge- document associated to w is defined by $d(w) = d(x_1) \ldots d(x_n)$.

The result of merging of the free document *d(w)* with a tuple $t= < x_1{:}v_1, \ldots x_n{:}v_n>$ will be the sequence of texts of form

$<x_i$-*text1* $\boldsymbol{v_i}$ x_i-*text2*$>$ for terminal x_i

and $<$ x_i-*text1* {sequence of the matching of the ***subtuple*** of type x_i} x_i-*text2*$>$ for non-bold nonterminal,

and finally for a bold nonterminal $\mathbf{x_i}$ the resulting serial text will be $<\boldsymbol{x_i}$-*text1* {the serial form of the list of matching tuples of table $\boldsymbol{v_i}$ } $\boldsymbol{x_i}$-*text2*$>$.

The use of free documents in the X-merge system enables human recipients to perceive the loaded data in an understandable context and layout. This makes it possible to associate this formation with the corresponding meaning during the information event. During the flow and filling of free documents within the computers, no information events occur, only pre-programmed mutual transformation of formations take place. Here too, the X-merge option can be used in cooperation with the X-relational database. Function calls and expressions can be entered in the context of each merge variable.

Remark 4. The *text1* or *text2* may contain active parts that compute results from the inserted values. For example, in $\boldsymbol{x_i}$-*text2* we may use aggregate functions on the inserted table instance $\boldsymbol{v_i}$.

8. Discussion

Summarizing, in our paper, we dealt with the extension of the relational model in a direction where the goal was to formally specify the family of relational schemas. In the relational model, the relation/table as a collection consists of data occurrences of the same type. We deviated from this in that, according to some rule, a collection can consist of several types of data. We broke with the uniqueness of the attribute names within the schema, but due to the repeated attribute names, the attribute sequence defining the schema became ordered. The family of relational schemas was specified using formal languages over a set of predefined attribute names. The resulting families are called extended relations. The definition of the schemas and the operations that can be specified over the families were given using the graph representation of the formal languages.

We reviewed our models based on the use of regular languages and then gave a more detailed description on context-free languages. The new contribution, which we started to develop in the paper [8] at the IDEAS conference, is based on the specific use of context-free grammars. While in the case of regular languages we could symmetrically use terminal and non-terminal symbols to specify a schema, in the case of context-free languages we use both at the same time, and embedded relational schemas can be assigned to non-terminal symbols. The detailed specification of the context-free schema graphs, formal specification of extended schemas and instances and detailed definitions of algebraic operators are the main novel parts of the paper.

It is an important requirement for the use of representational forms of information that support the presentation suitable for human perception and understanding, and at the same time enable the specification of processing algorithms using the structural elements of the representational form. For example, the table names and attribute names of the relational model only play identification and reference role for the machine, the operation of the entire database system is invariant to the one-to-one replacement of these names. For the users, however, these names provide the possibility of interpretation for the displayed structures. This property can also be used for extended relations, with the addition that the grammars can be interpreted by the machine and have meaning for the users as well.

As another typical example, end-user document formats also contain names and texts that can be interpreted by end users, which ensures the understanding of individual occurrences. In traditional IS modeling the information service is supported by a separate database by loading or extracting the data into or from end user documents. The papers [22, 23] introduce the free documents, which has an included structure of specific fields for accepting the actual data. The specific fields have associated variable names. So a set of free documents are joined together via common field variables. This gives a hypergraph join-structure of documents. Free documents are similar to templates in mail-merge systems. As an example of data exchange between free documents of information systems and an X-relational database, we presented in detail the X-merge solution as a generalization of mail-merge systems using tables or relational databases. We emphasize that specifying the attribute names of X-relations can in itself promote human understanding in the display of rows with attribute names, thus the possibility of a meaningful information event. This is further confirmed by the textual contexts providing the interpretation in the X-merge document. The novelty of the solution is that the common generative grammar makes it possible to match the free merge-document and the X-relation tuple to any tuple type belonging to the given grammar.

Closing the discussion, we try to put our model to a wider context – we turn back to the info-sphere. Our ability to rearrange matter and the cognitive capabilities of handling information developed in cooperation and led to the development of humanity's info-sphere. See in Benczúr [6]. The set of information carriers that physically exists at any given time and the set of systems and instruments that ensure its use make up the info-sphere of humanity. In the possibilities of rearranging the material, we have now reached the point where – although we cannot make the material do thinking, but - we can make it perform computation. Rearranging matter and thinking come together: some activities of thinking (computing) can be done by the matter itself. The special feature of computer (computing machinery) is that the material it transforms is "only" the carriers of information. This also means that it becomes useful and becomes information only in the case of the appropriate meaning assigned to the carrier. The solution of the assignment is the challenge and task of informatics.

In the socio-cultural evolution, the emergence of language made a major transition as a tool that stabilized the transfer of culture between generations Given the rapid development in technologies to store, process, and distribute information, and recently the explosion of the data-sphere and the new AI-based technologies, all these might involvesimilar major cultural evolutionary effect on humanity as the language did. See in Carmel [13] and Floridi [16].

The eras of the info-sphere's history are characterized by the spread of new forms of information representation and the use of new technologies for information collection. Within this, structures play an important role, which relate both to the forms of representation and to the information that can be obtained about real phenomena. In the series of efforts to define the concept of information and provide a unified theory of information, the approach of Burgin's General Theory of Information [11, 12] is based on the central role of structures. According to his theory, the information of reality is carried by structures, and the information obtained by perceiving the structures is represented by artificially created structures.

Digital technologies have also brought rapid advancement in the use of structures. It was made possible by handling complex, large-scale structures managed with formal specifications and algorithms. Our models provide a new structural level above the traditional and basic relational model, and with this we have introduced new forms of information representation, and by this giving a small contribution to this progress.

Funding: This research was supported the Thematic Excellence Programme TKP2021-NVA-29 (National Challenges Subprogram) funding scheme, and by the COST Action CA19130 - "Fintech and Artificial Intelligence in Finance Towards a transparent financial industry" (FinAI)

Author Contributions: Conceptualization A.B., B.M. and G.S.; formal analysis A.B., B.M. and G.S.; writing – original draft preparation A.B and B.M.; writing – review and editing A.B., B.M. All authors have read and agreed to the published version of the manuscript.

9. Appendix

The next short digression is about the graphs that characterize formal languages and their dual languages. The initial idea of the authors of the article for the use of schema graphs started from the observation that when applying the derivation rules of a regular grammar, a step is always characterized by a pair $(B,\ b)$, where B is non-terminal and b is terminal symbol. Thus, an accepted sentence is a series of such pairs, which, given in the form $< B_1{:}b_1, \ldots, B_n{:}b_n >$, gives a relational tuple. Non-terminal symbols as attributes give the tuple-type. At the same time, the sequence $B_1, \ldots, B_n$ can be considered as a sentence of a language above non-terminals. This is the dual language. The dual language is obtained from the vertex-labeled FSA graph of the grammar as a sequence of vertex labels of accepting traversals. By exchanging the edges and vertices of the FSA graph, we can get the FSA graph accepting the dual language.

In the case of context-free languages, we cannot directly adopt this method to define the dual language and the graph representation of the language. A left-first traversal of the derivation tree could be used as a dual language, but this did not lead to a sufficiently usable schema-graph. The RSM representation option was suitable for representing the language with a graph. It would be possible to consider the order of the box calls as a dual language with the help of RSM boxes. We did not choose this, rather we have chosen graph constructions that use both terminal and non-terminal symbols, which provide greater expression power. For this purpose, the specification of the context-free grammar was given in a special form, with one regular language for each non-terminal over the combined set of non-terminals and terminals. We called this form of specification an extended context-free grammar, ECFG. In Sections 5 and 6 we built the schema-graphs upon this.

The ECFG specification does not expand the class of context-free languages

For each regular grammar $\mathrm{G_A}$, we can build the extensions of the RSM graph to produce the corresponding language over terminals *N*. The notation of these languages is $L(M_A)$.

We can generate regular expressions equivalent to G_A grammar and we denote them by $E_A(N \cup T)$. By writing the language $L(M_A)$ in the place of A in a regular expression, we get regular expression above the languages. Formally, by using the notation *f(A)*: A is replaced by $L(M_A)$, and extended it to N as *f(N)* denotes the replacement of A for each $A \in N$ by $L(M_A)$ in a regular expression. The expression $E(f(N) \cup T)$ is a regular expression over the languages $\{L(M_A) \mid A \in N\}$ obtained from the expression $E(N \cup T)$. The notation *f(A)* is interpreted as meaning that any element of the language $L(M_A)$ can be substituted in place of the occurrence of A in the regular expression.

Theorem 1. *The languages* $L(G)= L(M_S)$ *and* $\{L(M_A)|\ A \in N\}$ *satisfy the following system of equations:*

$$L(M_A) = E_A(f(N) \cup T), \text{ for all } A \in N.$$

Proof. Let we take a specific rule $A \Rightarrow w \in L(G_A)$, where $w = v\ B\ z$, and suppose that exists $x \in L(M_B)$ such that $v\ x\ z \in L(M_A)$. There is a traversal of the graph M_A according to w, including the B edge with non-terminal label. Any graph can be written in place of the edge B, which can be obtained during the extension of the graph M_B. This means that any sentence of the language $L(M_B)$ can be inserted here as a path, i.e. the whole language $L(M_B)$ can be inserted in place of B. ■

Turing-machines

In the case of both regular and context-free languages, we used the graphs of the machines that generate the languages. The question arises whether for Turing-machines could be given a dual language or some graph-based machine. The transition between the states of T-machines can be viewed as an edge-labeled directed graph based on the transition table, where the label of the outgoing edge for the next vertex is given by the input signal. The execution of the T-machine is shown by a series of vertex labels, which could be considered as the dual language of the given machine. The operation of the machine is not specified by the graph given in this way, the walk back and forth on the tape and the preservation of the cell contents are missing. This is solved by an architecture that can also be physically implemented with the following specification of communicating agents.

Consider a deterministic Turing machine $T = \{Q, [0,\ 1,\ \epsilon], \delta = Q \times [0,\ 1,\ \epsilon] \rightarrow$ $\rightarrow [0,\ 1] \times [1,\ -1] \times Q,\ q_0,\ F\}$, using a binary tape, where Q is the set of states, δ is the set of transitions, q_0 is the initial state, and F is the accepting or final state. Assign an agent to each state from Q. The agent receives the transition that apply to it. We can assume that the states are given by serial numbers, which are also the communication call number of the agent associated to the state. A step on the graph is a call between the two agents, which may include data transfer. Instead of the work tape of the original machine, the cells are represented by tokens with the corresponding current value. Agents store the tokens they have and their associated values. The steps of the Turing machine are implemented by the agents according to the following protocol:

The protocol for an agent's action:

Agent p receives a token ID t from its caller when it receives the control. It starts to broadcast a request on the network to obtain the t token and its associated value. Whoever has the t token sends its value via a direct connection and deletes it from his token list. It is an exceptional case if no one has the token. This means reading of the empty value and the creation of a new token by p is requested. Based on its motion table, p assigns a new value to token t and stores it in its token list. Let the value of the received token be x, and the corresponding transition of p: $(x,\ y\ ,k,\ q)$. So p stores the token $(t,\ y)$ with the new value it wrote, passes control to q, and sends the token ID $= t+k$. Obviously, the agent system performs the same computation on the tokens as the original T-machine. The result can be obtained by retrieving the tokens in sequence at the end.

To perform a calculation, the agents must first be loaded with the appropriate transition-table element. After that, n tokens must be created representing the *n-long* input. In the end, the tokens should be distributed arbitrarily among the agents. Agent q_0 receives the first token and starts the process.

The efficiency of the implementation depends on the memory management of the agents. It is advisable to store tokens in an ordered chained list. From the request broadcast message, everyone can see the ID number of the current token, and can move a pointer or counter up or down between the two token numbers in its chain. Communication can also be solved so that only the up-down value has to be communicated in the broadcast message. If the pointer reaches the serial number of the stored token, the token is sent. A limitation - as is usually the case with Turing-machines - is the size of the available memory per agent. This can be partially resolved by adding more physical component that implements an agents and by organizing the distribution of the token list.

References

[1] **Abiteboul, S., P. Buneman and D. Suciu,** *Data on the Web: From Relations to Semistructured Data and Xml*, Morgan Kaufmann, 2000.

[2] **Abiteboul, S., R. Hull and V. Vianu,** Foundations of Databases, Addison-Wesley, Vol. 8, 1995.

[3] **Albert, J., D. Giammarresi and D. Wood,** Normal form algorithms for extended context-free grammars, *Theoretical Computer Science*, **267(1-2)** (2001), 35–47.
https://doi.org/10.1016/S0304-3975(00)00294-2

[4] **Ansi/x3/sparc, Study Group on Data Base Management Systems,** Interim Report: ANSI/X3/SPARC Study Group on DBMSs 75-02-08. In: *ACM SIG on Management of Data*, **7(2)**, 1975.

[5] **Benczúr, A. and B. Molnár,** On the Notion of Information – InfoSphere, the World of Formations. In: *9th IEEE International Conference on Cognitive Infocommunications: CogInfoCom 2018: Proceedings*, Piscataway (NJ), USA: IEEE Computational Intelligence Society, Aug. 2018, pp. 33–38.
https://doi.org/10.1109/CogInfoCom.2018.8639904

[6] **Benczúr, A.**, On the carriers of information (formations, infospher, computation). In: E. Csuhaj-Varjú and P. Sziklai (Eds), *Conference on Developments in Computer Science: Budapest, Hungary, June 17-19, 2021*, Proceedings, Budapest, Hungary, Eötvös Loránd University, Faculty of Informatics, pp. 11–15.

[7] **Benczúr, A.,** From the appearance of ”The Computer and the Brain” to the revolution of the infosphere. In: *2023 IEEE 23rd International Symposium on Computational Intelligence and Informatics (CINTI)*, Ed. by **IEEE**. USA: IEEE, Nov. 2023, pp. 159–164.
https://doi.org/10.1109/cinti59972.2023.10382029

[8] **Benczúr, A. and Gy. I. Szabó,** On extended nested relational schemas generated by context-free grammars. In: *International Database Engineered Applications Symposium (IDEAS'22)*, August 22–24, 2022, Budapest, Hungary. IDEAS'22. New York, NY, USA, ACM, Aug. 2022, pp. 84-–93.
https://doi.org/10.1145/3548785.3548795

[9] **Benczúr, A. and Gy. I. Szabó** Towards a normal form and a query language for extended relations defined by regular expressions. In: *Journal of Database Management*, **27(2)** (2016), pp. 27–48.
https://doi.org/10.4018/jdm.2016040102

[10] **Berry, G. and R. Sethi,** From regular expressions to deterministic automata, *Theoretical Computer Science*, **48** (1986), 117–126.
https://doi.org/10.1016/0304-3975(86)90088-5

[11] **Burgin, M.,** *Theory of Information: Fundamentality, Diversity, and Unification*, World Scientific: New York, NY, USA; London, UK; Singapore, 2010.
https://doi.org/10.1142/7048

[12] **Burgin, M.,** The General Theory of Information as a Unifying Factor for Information Studies: The Noble Eight-Fold Path. Presented at the IS4SI 2017 Summit DIGITALISATION FOR A SUSTAINABLE SOCIETY, Gothenburg, Sweden, 12–16 June 2017.
https://doi.org/10.3390/is4si-2017-04044

[13] **Carmel, Yohay et al.,** Human socio-cultural evolution in light of evolutionary transitions: introduction to the theme issue. In: *Philosophical Transactions of the Royal Society B: Biological Sciences*, **378(1872)** (2023).
https://doi.org/10.1098/rstb.2021.0397

[14] **Codd, E. F.,** A relational model of data for large shared data banks. In: *Software Pioneers*, Vol. 13. 6., Springer Berlin Heidelberg, 2002, pp. 377–387.
https://doi.org/10.1007/978-3-642-59412-0_16

[15] **Denning, P. J. and T. Bell,** The information paradox, *American Scientist*, **100(6)** (2012), 470-–477.
https://doi.org/10.1511/2012.99.470

[16] **Floridi, L.,** A look into the future impact of ICT on our lives, *The Information Society*, **23(1)** (2007), 59–64.
https://doi.org/10.1080/01972240601059094

[17] **Floridi, L.,** *The Fourth Revolution: How the infosphere is reshaping human reality. How the infosphere is reshaping human reality.* First published in paperback. Oxford: Oxford University Press, 2016. 248 pp.

[18] **Grant, J. et al.,** Query languages for relational multidatabases, *VLDB Journal*, **2** (1993), 153–171.
https://doi.org/10.1007/bf01232185

[19] **Hector, G.-M., J. D. Ullman and J. Widom,** *Database Systems: The complete book. The complete book.* Ed. by J. D. Ullman and J. Widom. Second edition, Pearson new international Second edition, Pearson new international edition. Always learning. Harlow: Prentice-Hall, 2014. 11133 pp.

[20] **Molnár, B., A. Benczúr and A. Béleczki,** A model for analysis and design of information systems based on a document centric approach. In: *Intelligent Information and Database Systems IIDS*, Berlin: Springer-Verlag, 2016, pp. 290–299.

[21] **Molnár, B.,** Applications of hypergraphs in informatics: a survey and opportunities for research, *Annales Univ. Sci. Budapest., Sect. Comp.*, **42** (2014), 261–282.

[22] **Molnár, B. and A. Benczúr,** Document centric modeling of information systems, *Procedia Computer Science*, **64(10)** (2015), 369–378.
https://doi.org/10.1016/j.procs.2015.08.501

[23] **Molnár, B. and A. Benczúr,** The application of directed hyper-graphs for analysis of models of information systems, *Mathematics*, **10(5)** (2022), p. 759.
https://doi.org/10.3390/math10050759

[24] **Neumann, John von,** *The Computer and the Brain*, New Haven: Yale University Press, 2012.
https://doi.org/10.12987/9780300188080

[25] **Orachev, E. et al.,** Context-free path querying by Kronecker product. In: *Advances in Databases and Information Systems*, Springer International Publishing, 2020, pp. 49–59.
https://doi.org/10.1007/978-3-030-54832-2_6

[26] **Révész, Gy. E.,** *Introduction to Formal Languages*, Dover Books on Mathematics. Description based upon print version of record. Newburyport: Dover Publications, 2013. 437 pp.

[27] **Suciu, D.,** Semistructured data and XML. In: *Handbook of Massive Data Sets*, Vol. 4.
https://doi.org/10.1007/978-1-4615-1379-7_2

[28] **Szabó, Gy. I. and A. Benczúr,** Functional dependencies on extended relations defined by regular languages. In: *Lecture Notes in Computer Science*, Vol. 7153. Springer, Berlin, Heidelberg 2012, pp. 384-–403.
https://doi.org/10.1007/978-3-642-28472-4_22

[29] **TwoBitHistory,** *Important Papers: Codd and the Relational Model.* [Online; accessed 18. Nov. 2024]. Oct. 21, 2024.
https://twobithistory.org/2017/12/29/codd-relational-model.html (visited on 11/18/2024).

[30] **W3C,** *Extensible Markup Language (XML).* [Online; accessed 18. Nov. 2024]. Oct. 11, 2016.
https://www.w3.org/XML (visited on 11/18/2024).

András Benczúr
https://orcid.org/0000-0002-8678-3346
ELTE Eötvös Loránd University,
Faculty of Informatics
Budapest
Hungary
abenczur@inf.elte.hu

Bálint Molnár
https://orcid.org/0000-0001-5015-8883
ELTE Eötvös Loránd University,
Faculty of Informatics
Budapest
Hungary
molnarba@inf.elte.hu

Gyula I. Szabó
ELTE Eötvös Loránd University,
Faculty of Informatics
Budapest
Hungary
gyula@szaboo.de