An Introduction to the Enterprise-Participant (EP) Data Model

KEVIN XU, BHARAT BHARGAVA
Purdue University
YI SUN
The City College of New York

The relationships among partial computable functions, e.g., the query "If applying a function x to a function y yields a function z?", are undecidable in general, and therefore we do not have corresponding built-in operators in programming languages or database management systems. Limited to a finite set of finite functions, i.e., finitely many functions with finitely many ordered argument-value pairs, however, the relationships are decidable. The Enterprise-Participant (EP) data model is one that manages data as finite sets of finite functions and that has a set of built-in operators to express the relationships. In this article, we introduce the EP data model. Considering a data model as a recursive language, i.e., its executions always terminate, the EP data model is the most expressive among all contemporary data models including the relational data model and hierarchical data models (e.g., XML), and among all contemporary structured data representations including semi-structured data and graph-based data. Further, it mathematically reaches the limit in database management, i.e., we can use the EP data model to manage arbitrarily computable information. The aim is to consolidate heterogeneous data into a general data structure and to offer more powerful built-in operators for data analysis in cloud computing and big data initiatives.

Keywords: Data model, data presentation consistency and generality, query optimization, big data

1. INTRODUCTION

The relational data model (SQL) has an expensive operator - join, e.g., the time complexity is O(n ln(n)) in querying many-to-many relationships. It causes the response time unacceptably long when data volume becomes very large. Application data and logic in a database applicaton system are separately embedded in the database and in the source code of a programming language. It causes dependencies between database and source code. A data partitioning (sharding) strategy change due to data volume increase, for example, may require a source code change. NoSQL technologies, e.g., Google Bigtable, Apache HBase, and MongoDB [Ferguson et al. 2012, Hayes et al. 2010, Krzyzanowski et al. 2011, and Ward et al. 2013], resolve the time complexity issue in a price of losing some of SQL expressive power. NoSQL does not have a solution to the dependency issues. Even worse, it increases the dependencies because it loses some expressive power of SQL. Now a typical database system is added with another component NoSQL beyond the complexity of SQL and programming language. The complexity further isolates and buries originally connected information deeper into heterogeneous systems, which causes data analytics more difficult.

Lacking a mathematically sound data model that is able to consistently represent arbitrarily computable information is the root cause of the challenging issues substantially and inevitably faced by cloud computing, where big data is stored and data analytics is desired within a big data system or across multiple big data systems. Such a data model would have to generalize all existing data models and data structures. Additionally, we require the data model to have low time complexity in executing advanced built-in operations, including SQL join operations.

The Enterprise-Participant (EP) data model meets all the requirements. It has an underneath

Authors' addresses: Kevin Xu, Bharat Bhargava, Purdue University, Dept. of Computer Science, West Lafayette, IN, U.S.A.; Yi Sun, The City College of New York, Dept. of Electrical Engineering, New York, NY, U.S.A.

data structure that preserves all the relationships among managed objects (instead of shredding objects into separated tables) and therefore the time complexity is optimized for all kinds of queries, including SQL-equivalent queries, e.g., the linear time complexity of $O(n)$ in expressing SQL many-to-many relationship queries.

The EP data model was originally proposed to have a single data structure for both database management and data communication [Xu et al. 1996]. Later it was found that it is a method of semantically approximating the lambda calculus [Xu et al. 2014]. As a result, the EP data model can model all kinds of business data in a consistent manner, and therefore offer an opportunity to reduce the degree of data variety in the future big data management.

Here are a few examples that intuitively demonstrate how EP databases are constructed and will be referenced in the later discussion.

Example 1.1 The set $x\ x$ `:=` x is an EP database, where x can be viewed as a function that yields to itself when it is applied to itself, and yields to *null* (a special constant in EP) when it is applied to other arguments. Given the set $x\ x$ `:=`x, there are infinitely many expressions x, $x\ x$, $x\ x\ x$, $x\ x\ x\ x$, ... that are reducible to x. The letter x actually represents an approximation to all the functions that yield to themselves after being applied to themselves, e.g., the identity function $\lambda x.x$.

Example 1.2 A directed graph with connections from v_1 to v_2, from v_2 to v_1, and from v_2 to v_3 can be expressed in an EP database: $v_1\ v_2$ `:=`v_2; $v_2\ v_1$ `:=` v_1; $v_2\ v_3$ `:=` v_3, where v_1, v_2, and v_3 represent vertices, and an assignment represents a directed connection. In this example, the vertex v_1 is viewed as a function that yields to v_2 when it is applied to v_2, and yields to null when it is applied to others. The function v_2 yields to v_1 when it is applied to v_1, v_3 when it is applied to v_3, and *null* otherwise. The vertex v_3 is also a function, but a constant one for the time being. It may become a non-constant function later with an EP update operation. With this database, we can reduce infinitely many expressions $v_1\ v_2$, $v_1\ v_2\ v_1$, ..., $v_1\ v_2\ ...\ v_1$, ... to x_1. It could be thought of as if one walked along the circle x_1 to x_2 and x_2 to x_1 as many times as she liked and eventually stopped at the vertex x_1. The query "If is there a path from x_1 to x_3 ?" is simply expressed in EP as x_3 `(=+` x_1, where the binary operator `(=+` reflects a pre-ordering relation among functions, arguments, and values that exist in a database. The answer would be the truth value true.

By adding constants, such as integers, strings, and truth values, the EP data model can express common business data in software practice.

Example 1.3 A school administration can be expressed in the following database:
SSD.gov John SSN `:=` *'123456789'*;
SSD.gov John birth `:=` *'6/1/1990'*;
SSD.gov John photo.jpg `:=` ...; /* a binary stream for a photo*/
college.edu admin (*SSD.gov John*) *enroll* `:=` '9/1/2008';
college.edu admin (*SSD.gov John*) *Major* `:=` *college.edu CS*;
college.edu CS CS100 (*college.edu admin* (*SSD.gov John*)) *grade* `:=` "F";

The sampe database is alternatively expressed in a diagram in Figure 1. The immediately visible benefits of using the EP data model include: 1) The EP data model has more expressive power than any existing data models (or data structures) for data analytics. 2) Analogous to network file system (NFS), multiple EP database systems can be plugged together to provide a larger and consistent view as a way of application integration. As a matter of fact, more and more information can be accumulated in the system to reflect better and better the real world. This process is done purely in the EP data model without other technologies. Taking as a data exchange protocol, the EP data model will also find broad applications in large integrated systems such as cloud computing and knowledge management. 3) The EP data model is extended to a full programming language Froglingo which logically and physically treat business data and business logic equally. 4) Analogous to NFS, the EP data model has built-in security (data access control)

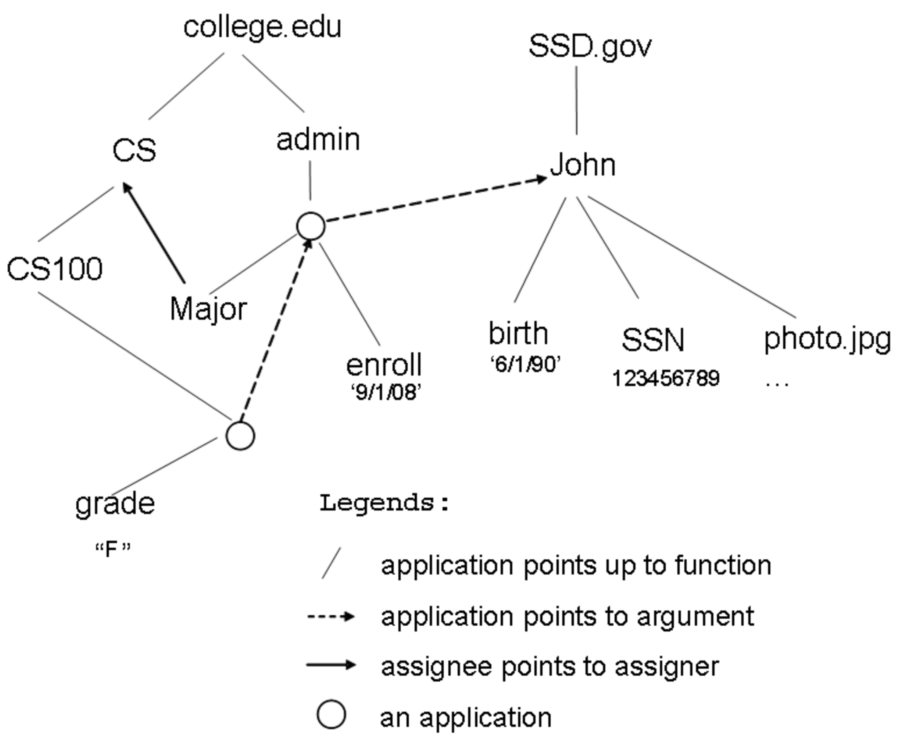


Figure 1: Figure. 1: A sample EP database

mechanism for protecting data privacy. 5) EP data model has a built-in ability for comparing objects in similarity that is often needed in knowledge management.

In this article, we introduce the EP data model itself as a language, from which we see how one constructs databases in the EP data model. We also discuss the decidable (dependent and pre-ordering) relations among the objects in EP databases. The relations are the mathematical underpining of a rich set of built-in operators that supersede the available database operations of the contemporary database technologies. The dependent relations are the mathematical underpining of the tree-structures in the implementation of the EP data model (The optimized performance of the EP data model comes from the tree-structures).

In Section 2, we give the core concepts of the EP data model: terms, assignments, database, normal form, and reduction. Separately in Sections 3 and 4, we discuss the built-in operators of the EP data model and their uses in set-oriented queries. (Note that the proofs of related lemmas and theorems can be found in [Xu et al. 2014 and 1999.]). In Section 5, we briefly review previous research work on data models. We also briefly mention Froglingo, a programming langugae expanded from the EP data model that helps in data security, web service, and data analytics.

2. THE EP DATA MODEL

Rather than defining functions in formulas or algorithms, the EP data model defines functions in ordered argument-value pairs. A collection of ordered pairs, in the form of assignments, is called a database. Starting from a set of identifiers, we construct a set of EP terms.

Definition 2.1 Given a set of constants $\boldsymbol{C}$ including a special constant *null* and a set of (functional) identifiers $\boldsymbol{F}$, the set of *EP terms*, denoted as $\boldsymbol{E}$, is defined inductively[1] as follows:
$x \in \boldsymbol{C} \cup \boldsymbol{F} \Rightarrow x \in \boldsymbol{E}$
$x, y \in \boldsymbol{E} \Rightarrow x\ y \in \boldsymbol{E}$

We adopt many notations from the lambda calculus, including *sub-term*[2] (SUB(t) for all sub-terms in t), *left-most* [3] sub-term (LMS (t) for all left-most sub-terms in t), and applications without surrounding parentheses, e.g., $a\ b$, instead of $(a\ b)$ when there is no ambiguity. Given a term $a\ b\ (c\ d)$, for example, SUB $(a\ b\ (c\ d))$ = $a\ b\ (c\ d)$, $a\ b$, $c\ d$, a, b, c, d, LMS $(a\ b\ (c\ d))$ = $a\ b\ (c\ d)$, $a\ b$, a. Given a term t, we further call a sub-term of t, except for t itself, a proper sub-term, and therefore we use SUB$^+(t)$ to denote all the proper sub-terms of t, i.e., SUB$^+$ (t) = SUB(t) - t.

A constant can be a string, an integer, a character, a time, an image file, or any binary stream data. So any objects required to be managed in big data are manageable in the EP data model.

An EP database (simply called a database in this article) is a finite set of assignments where assignees and assigners are terms. Some terms that have a constant as a left sub term, e.g., *3 t*, are not allowed to be in a database. We identify a sub set out of $\boldsymbol{E}$, denoted as $\boldsymbol{E}^0$, which is inductively defined as:

$x \in \boldsymbol{F} \Rightarrow x \in \boldsymbol{E}^0$
$x \in \boldsymbol{E}^0, (y \in \boldsymbol{C} \cup \boldsymbol{E}^0) \Rightarrow (x\ y) \in \boldsymbol{E}^0$

It means that if an application $a\ b$ is a term in $\boldsymbol{E}^0$, then the left sub term a cannot be a constant. An assignment is in the form of

$a := b$

where $a, b \in \boldsymbol{E}^0$ are called the assignee and the assigner respectively.

Definition 2.2 A database D is a finite set of assignments with the following constraints:

(1) Each assignee has only one assigner, i.e.,
$a := b_1$ and $a := b_2 \in D \Rightarrow b_1 \equiv b_2$
(2) A proper sub term of an assignee cannot be an assignee, i.e.,
$a := b \in D \Rightarrow \forall\ x \in \text{SUB}^+(a)\ [\forall\ c \in \boldsymbol{E}^0\ [x := c \notin \text{D}]]$
(3) An assigner is a non-*null* constant, an identifier, or a sub-term of an assignee, i.e.,
$a := b \in D \Rightarrow b \in \boldsymbol{C}$ - *null*, $b \in \boldsymbol{F}$, or $\exists\ c, d \in \boldsymbol{E}^0\ [c := d \in D$ *and* $b \in \text{SUB}(c)]$
(4) If there is a sequence of assignments $a_0 := a_1, a_1 := a_2, ..., a_{j-1} := a_n \in D$ for $n > 0$, then a_n must not be identical to a_0, i.e.,
$a_0 := a_1, a_1 := a_2, ..., a_{j-1} := a_n \in D \Rightarrow a_0 \neq a_n$

The system being discussed in this article was originally called the Enterprise-Participant data model [Xu et al. 1996], where given an application $a\ b \in \boldsymbol{E}$, a was meant an enterprise and b a participant. The notion term in Definition 2.1 says a lot about how objects in the real world interact to each others, such as a person participates in a party. The constraints in Definition 2.2 require that a participation has a outcome, i.e., applying a function to an argument yields a value. Put it differently, a database is a collection of functions, each of which has a finite set of triplets function-argument-value, i.e., $a\ b := c$ where a, b, c are respectively a function, an argument, and the corresponding value. An assignee is normally an application, and an assigner is a value. The first constraint ensures that each application yields only one value. The second constraint prevents two equivalent applications from being assigned with multiple values. Given

[1]When an inductive definition is given, it will always be understood that the class defined is the smallest set satisfying the conditions. It is applied to other inductive definitions introduced later in the article.
[2]Precisely, $t \in \Lambda \Rightarrow t \in \text{SUB}(t)$, and $m\ n \in \text{SUB}(t) \Rightarrow m,n \in \text{SUB}(t)$.
[3]Precisely, $t \in \Lambda \Rightarrow t \in \text{LMS}(t)$, and $m\ n \in \text{LMS}(t) \Rightarrow m \in \text{LMS}(t)$.

assignments a_0 ... a_i ... a_k := b, a_i := $c \in D$, and $b \neq e$, for example, we prevent the assignment a_0 ... c ... a_k := e from being in D.

The value of an application is a term which is in turn interpreted as a function as well. We could choose any term in $\boldsymbol{E}$ to be a value along with an adjustment to the definition 2.2. But we limit it by the third constraint in an ultimate form desired to represent a value. The ultimate form is called a normal form.

As a notation, we say that terms a and b are in database, denoted as $a, b \in D$, if a := $b \in D$. Similarly, we say that b is in database, denoted as $b \in \boldsymbol{D}$, when there is a term $a \in D$ and $b \in$ SUB(a).

Definition 2.3 Given a database D, a term $n \in \boldsymbol{E}$ is a normal form in D if it is a constant or a term in D except for an assignee in D, i.e., $n \in \boldsymbol{C}$, or $n \in D$ and $\forall\ b \in \boldsymbol{E}^0$ [n := $b \in D$].

In other words, a normal form is a constant, an identifier in D, or a proper sub-term of an assignee, i.e., $n \in \boldsymbol{C}$, $n \in \boldsymbol{F} \cap D$, or $\exists\ a, b \in \boldsymbol{E}^0$ [a := $b \in D$ and $n \in \mathrm{SUB}^+(a)$]. For example, the terms x, v_1, v_2, v_3, *John*, *SSD.gov John* are normal forms of the example databases in Section 1. We use NF^{D} to denote all normal forms with a database D.

Now, we want every term in $\boldsymbol{E}$ to have a value. This becomes possible by introducing a set of reduction rules.

Definition 2.4 Given a database D, we have one-step reduction rules, denoted as $\rightarrow$:

(1) An assignee is reduced to the assigner, i.e.,
a := $b \in D \Rightarrow a \rightarrow b$

(2) An identifier not in database is reduced to *null*, i.e.,
$a \in \boldsymbol{F}$, $a \notin D \Rightarrow a \rightarrow$ *null*

(3) If a and b are normal forms and $a\ b \notin D$, then $a\ b$ is reduced to *null*, i.e.,
$a, b \in NF^{\mathrm{D}}$, $a\ b \notin D \Rightarrow a\ b \rightarrow$ *null*

(4) $a \rightarrow$ a', $b \rightarrow$ b' $\Rightarrow a\ b \rightarrow$ a' b'.

Definition 2.5 Let $a \rightarrow a_0, ..., a_{n-1} \rightarrow a_n$ for a number $n \in \mathbf{N}$. Then we say that a is effectively, i.e., in finite steps, reduced to a_n, denoted as $a \rightarrow_{EP} a_n$.

Definition 2.6 A term a has a normal form b if b is a normal form and $a \rightarrow_{EP} b$.

If $a_1 \rightarrow_{EP} b$ and $a_2 \rightarrow_{EP} b$, then we say that b, a_1 and a_2 are equal, denoted as b == a_1 == a_2.

Here are a few sample equations under the databases of Section 1:
SSD.gov John SSN == *123456789*
college.edu admin (SSD.gov John) Major == *college.edu CS*
$v_1\ v_2\ v_1$ == v_1
$v_1\ v_2\ v_1\ v_2\ v_1\ v_1$ == v_1
$x\ x\ x$ == x

With the reduction rules, we want the value of an assignee to be the value of its assigner. Note that a reduction may not terminate if there is a set of circular assignments, i.e., there are a chain of assignment a_0 := a_1, a_1 := a_2, ..., a_{j-1} := $a_n \in D$ for a $n > 0$ such that a_0 := a_1, a_1 := a_2, ..., a_{j-1} := $a_n \in D$, and $a_0 \equiv a_n$. The constraint 2.2.4 ensures that all reductions terminate, which lead to the following conclusion: a term has one and only one normal form, i.e., it is strongly normalizing.

Lemma 2.7 An assignee has a unique normal form in a database.

Theorem 2.8 A term $a \in \boldsymbol{E}$ has one and only one normal form under a database D.

Our discussion has emphasized that under a database D, a term a has a normal form as its value, either a constant, an identifier in D, or a proper sub-term of an assignee. In comparing with the contemporary programming language and database management systems, it is easy to understand that a takes a constant as its value, or a takes a proper sub-term of an assignee which is not a itself as its value. One difference in the EP data model is that a proper sub term of an assignee, say a as a term in $\boldsymbol{E}$, takes itself as its normal form. It has been a misleading in this article so far to say that a takes a itself as it's own value. Precisely, we say that a is a name to a's value which is not explicitly given in D but implicitly derivable from D. For example, the term *SSD.gov John* is a normal form in the database of Example 1.3. Rather than saying that *SSD.gov John* has a value *SSD.gov John*, we should have said that it has a value of a function: {(*birth*, *'6/1/90'*), (*SSN, 123456789*), (*photo.jpg*, '...')}, here '...' stands for a binary stream as the content of a file named *photo.jpg*. Allowing a proper sub term of an assignee to take itself as its normal form and to reference an implicit value provides a syntactical mean to express meaningful applications without constants and to express self applications in the EP data model. That is the distinguishing feature making the EP data model untyped. With the discussion so far, we can see that the EP data model is a language that generalizes all the concepts appeared in the contemporary technologies. A key-value pair in NoSQL, for example, is simply expressed as an assignment in the EP data model, where the assignee is an identifier. The containment relationships in hierarchical data models are well embedded in EP databases where assignees give the containment relationships (See Section III below for the dependent relations that precisely capture the containment relationships).

The many-to-many relationships in the relational data model are "denormalized" in the EP data model without redundancy. For example, to express the many-to-many relationships between projects and employees where an employee is tasked with multiple projects and a project needs multiple employees to work on, we will have a sample EP database:

Example 2.9 *Emps 1 name* := "*John*";
Emps 2 name := "*Mary*";
Projs 10 (*Emps 1*) := *true*;
Projs 10 (*Emps 2*) := *true*;
Projs 20 (*Emps 1*) := *true*;

For the query "Print all the projects and for each project print the names of all employees who work for the project", which is a joint operation in the relation data model, the EP data mode has the corresponding expression: *select mterm p,e name where pe* == *true*, where *mterm* is a constant function that returns the right sub-term of an application.

To facilitate our further discussion in coming two sections, we give another sample EP database representing food recipes [Xu et al. 2010].

Example 2.10 A sample database representing two recipes:

(1) *grill* (*marinate* (*rub* (*beef short loin 8 oz*) *garlic salt*)) := *dish1*;

(2) *grill* (*plate* (*grill* (*rub* (*beef short loin 8 oz*) *garlic salt*)) (*sauce chile*) (*cheese cheddar*)) := *dish2*;

The first assignee embeds the preparation steps of a recipe: 1) Rub beef short loin, 8 ounces, with garlic and salt; 2) Marinate it; and 3) grill the short loin. The second term embeds the preparation steps of another recipe: 1) Rub beef short loin, 8 ounces, with garlic and salt; 2) grill the short loin; 3) place short loin in a plate and spread with chile sauce and cheddar cheese; and 4) place the plate back to grill.

The database above does not exactly reflect the recipes in reality. However, Froglingo allows a recipe to be represented as accurate as a software application desires by accumulating more attributes. It does not care if an attribute of a recipe is an ingredient, a piece of cooking equipment, or a process. All the attributes are indiscriminatingly embedded in high-order functions.

Another benefit of doing this is that all the recipes are stored together with the attributes shared among them. For example, the attribute "Rub beef short loin, 8 ounces, with garlic and salt", which appeared in both sample recipes given earlier, is stored once in the database as:

rub (*beef short loin 8 oz*) *garlic salt*;

Before ending this section, we show that given a database D, the EP data model defines a total function with a bounded support. A function f : $X \longrightarrow Y$, where X and Y are arbitrary sets of objects, has a finite support if and only if there exists a finite set $A \subset X$ and a member $a \in Y$ such that

$$f(x) = \begin{cases} b, \text{ where } b \in Y \text{ and } b \neq a & \text{if } x \in A \\ a & \text{if } x \notin A \end{cases}$$

A function $f : X \longrightarrow Y$, where X and Y are arbitrary sets of objects, has a bounded support, if and only if there exists a finite set A $\subset Y$ such that

$f(x) \in A$ for all $x \in X$.

In this article, we simply call a function finite if it has a finite support, and bounded if it has a bounded support. A finite function is bounded, but a bounded function may not be finite.

Lemma 2.11 Given a database D, the set of all non-constant normal forms, i.e., NF^{D} - C, is finite.

Given a database D and a term $m \in \boldsymbol{E}$, we use m^{nf} to denote m's normal form.

Theorem 2.12 Given a database D, there exists a computable function Y^{D}: $\boldsymbol{E} \longrightarrow \boldsymbol{E}$ such that

$Y^{\mathrm{D}}\ (m) = m^{\mathrm{nf}}$

for all m, $n \in \boldsymbol{E}$.

Theorem 2.13 Given a database D, there exists a finite function X^{D}: $D \longrightarrow NF^{\mathrm{D}}$ such that

$X^{\mathrm{D}}\ (m) = m^{\mathrm{nf}}$

for all $m \in \boldsymbol{E}$ and $m \in D$.

Theorem 2.14 If there exists a sequence of assignments a_0 := a_1, a_1 := a_2, ..., a_{j-1} := $a_n \in D$ for an $n > 0$ such that ${a_0}^{\mathrm{nf}} \neq null$ and $a_n \in$ SUB(a_0), then Y^{D} is not finite but bounded.

Y^{D} is the constant application operator we discussed in Section 1. In the next two sections, we discuss many more operators including <=+ mentioned in Example 1.2.

3. DEPENDENT RELATIONS

An application depends both on its function and on its argument; and thereafter it depends on the sub-terms of the function and the argument. This leads to the development of the dependent relations.

Given an application $m\ n$, we called m and n the left sub term and the right sub term earlier. To give the relevant operators in this section and the coming section, we also called m and n the plus-term and the minus-term in [20], in corresponding to the signs "+" and "-" that appear in the operators.

Definition 3.1 (1) Given an application $m\ n$ in a database D, the operators {+ and {- in the following expressions are defined such that the expressions are evaluated to be true:

$m\ n$ {+ m
$m\ n$ {- n

(2) Given a term m in a database D, let l, s, r are a left-most sub-term, a sub-term, and a right-most sub-term accordingly, then the operators {=+, {=-, and {= in the following expressions

are defined such that the expressions are evaluated to be true:
m {=+ *l*
m {=- *r*
m {= *s*

Here are the sample expressions with the value of *true*:

SSD John birth {+ *SSD John*;
SSD John birth {=+ *SSD*;
SSD John birth {=- *birth*;
College CS CS100 (College admin (SSD John)) {=- *John*;
birth {= *SSD John birth*;
John {= *SSD John birth.*

In a correspondence with the binary operator {+ and {-, there are two unary operators *pterm* and *mterm*, where the first letters '*p*' and '*m*' are the prefixes of "plus" and "minus". Given an application *m n*, we define that *pterm* ($m\ n$) $= m$ and *mterm* ($m\ n$) $= n$. In Section II, we have used the operator *mterm.*

We will define a type of ordering relations - dependent relations and show that the operators above are dependent relations.

Definition 3.2 Given a non-empty set A, a strict subset $B \subset A$, and x, $y \in A$, x is dependent on y in B, denoted as $x\ \rho\ y$, if and only if $x \in B$ implies that $y \in B$, i.e., $\rho = \{<x, y>|\ x, y \in A$; if $x \in B$, then $y \in B\}$. Here we say that ρ is a dependent relation in B.

Proposition 3.3 The relations {+, {-, {=+, {=-, and {= are dependent relations in a database D.

Notation 3.4 The operators {+, {- , {=+, {=-, and {= are called *functional dependency*, *argumentative dependency*, *recursively functional dependency*, *recursively argumentative dependency*, and *recursively neutral dependency* correspondingly.

The EP data model supports set-oriented operations in conjunctions with ordering relations. We give two examples to demonstrate it in the rest of the section.

Example 3.5 Select all the participations of John in the database of Example 2.5:

select $x *where* $x {=- *John*;

The output would be *John*, *SSD John*, *College admin (SSD John)*, *College CS CS100 (College admin (SSD John))*.

Note that a variable, e.g., $x in the example above, can appear in the where clause of a select operation. Syntactically, it is the same as those introduced in Section 6. Semantically, however, it plays no role at all to the construction of a database since it is local to the select expression.

Given the recipes representation in Example 2.10, the recursively neutral dependency, i.e., {=, is found useful in supporting the queries from the customers in a restaurant who know nothing about knowledge representation but remembering a few attributes about dishes. For example, she may express: "I want a marinated beef short loin. I hate cheese". A system having the EP data model may simply translate her query into a list of key words with possible negations: "beef, short, loin, marinate, no cheese". Further, the system is able to translate the query to an EP expression:

Example 3.6 *select dish where dish* {= *beef and dish* {= *short and dish* {= *loin and dish* {= *marinate and not* (*dish* {= *cheese*);

The system would include *dish1*, but not *dish2* of Example 2.10 as the result.

When a database stores thousands of recipes, the query above may bring up hundreds of satisfied answers. To narrow down to fewer answers that customers really want, the customers can describe more attributes as a part of queries.

Before ending this section, we give the relationships among the dependent relations.

Lemma 3.7 (1) m {+ $n \Rightarrow$ m {=+ n
(2) m {- $n \Rightarrow$ m {=- n
(3) m {=+ n, or m {=- $n \Rightarrow$ m {= n

The above lemma say that if a term is dependent on another, so is it recursively, and if a term is functionally or argumentatively dependent on another, so is it neutrally. In other words, {+ and {- are stronger than {=+ and {=-; and {=+ and {=- are stronger than {=.

Dependent relations imply partial orderings. The first four relations in Notation 3.4 are further tree-structured orderings. The operator {= is not tree-structured, but only partial ordering. For example, a term A B (A C) neutrally dependent on both A B and A C; and the terms A B and A C dependent on term A.

4. PRE-ORDERING RELATIONS

The normal form, resulting from an application, does not have to depend on the function and the argument of the application because it exists independently. However, the normal form is derivable from the application; and therefore it is derivable from the function, from the argument, and from the sub-terms of the function and the argument. This leads to the development of the pre-ordering relations.

Definition 4.1 (1) Let m, n, $q \in$ $\boldsymbol{E}$ and D a database, if m n == q, then the operators (+ and (- in the following expressions are defined such that the expressions are evaluated to be *true*:
q (+ m
q (- n
(2) Let m, q, l, s, r $\in$ $\boldsymbol{E}$, D a database, m == q, l is a left-most sub-term of m, s is a sub-term of m, and r is a right-most sub-term of m. Then the operators (=+,(=-, and (= in the following expressions are defined such that the expressions are evaluated to be true:
q (=+ l
q (=- r
q (= s

Example 4.2 Given the databases in Example 1.1 and 1.2, here are a few Boolean expressions with *true* values:
"*F*" (+ *College CS CS100 (College admin (SSD John))*;
"*F*" (=+ *College CS CS100*;
"*F*" (=- *SSD John*;
"*F*" (= *College admin*;
v_2 (=+ v_1; v_1 (=+ v_2;
v_1 (= v_2;

Notation 4.3 (1) A relation ρ in a set X is reflexive iff $x\ \rho\ x$ for each x in X. ρ is symmetric if $x\ \rho\ y$ implies $y\ \rho\ x$, and it is transitive iff $x\ \rho\ y$ and $y\ \rho\ z$ imply $x\ \rho\ z$.
(2) ρ is antisymmetric iff whenever $x\ \rho\ y$ and $y\ \rho\ x$ imply $x = y$.
(3) A relation ρ is called partial ordering in X iff ρ is reflexive, antisymmetric, and transitive.
(4) A relation ρ is called pre-ordering in X if ρ is reflexive and transitive.

Proposition 4.4 (=+, (=- and (= are pre-ordering.

The pre-ordering relations appear loose and irrelevant to the queries supported in traditional database technologies. However, they were found useful in the following examples.

Example 4.5 Given the data presentation in Example 1.2 for a directed graph,

(1) The query: if there is a path from v_1 to v_3 is expressed as: v_3 (=+ v_1. It is evaluated to be

true since v_3 == v_2 v_3 == $(v_1\ v_2)$ v_3.

(2) The query: if there is a circle between v_1 and v_2 is expressed as: v_1 (=+ v_2 and v_2 (=+ v_1. It is evaluated to be true because v_1 == v_2 v_1 and v_2 == v_1 v_2.

In the real life, an object can have many different names. For example New York steak is also named beef short loin. it can be represented as an assignment in the EP data model:

New York steak := beef short loin;

Then the query: "I want a marinated New York steak and I do not like cheese" would be translated to the following expression:

Example 4.6 *select dish where dish (= New and dish (= York and dish (= steak and dish (= marinate and not (dish (= cheese)*;

This expression may not exactly produce the result that Example 4.5 does, but it guarantees that *dish1* of Example 2.10 will be included as part of the outcome.

Notation 4.7 The operators (+, (- , (=+, (=-, and (= are called *functional derivative*, *argumentative derivative*, *recursively functional derivative*, *recursively argumentative derivative*, and *recursively neutral derivative* correspondingly.

Lemma 4.8 (1) m {=+ $n \Rightarrow$ m (=+ n
(2) m {+ $n \Rightarrow$ m (+ n
(3) m {=- $n \Rightarrow$ m (=- n
(4) m {- $n \Rightarrow$ m (- n
(5) m {= $n \Rightarrow$ m (= n
(6) m (=+ n, or m (=- $n \Rightarrow$ m (= n

The lemmas say that if a relation is dependent, it is also derivative. In other words, the dependent relations are stronger than derivative relations.

5. RELATED WORK

Developing a generic language system that is suitable to all possible software applications has been a main stream the fields of programming language and database management. Many database programming languages, i.e., descriptive languages over databases, were proposed between the late 1970s and the early 1990s. The work on Machiavelli [Ohori et al. 1989] was a typical example that used functional programming language over relations. Driven by the concept of the semantic web [Feigenbaum et al. 2007], many descriptive languages geared toward the management of the web-related data have been currently proposed. The examples are OWL (Web Ontology Language), a language taking ontology as the underneath data structure [OWL 2004], RuleML (Rule Marked Language), a family of Web rule language using XML as the underneath data structure [Boley et al. 2001, 2010, and Lee 2003], the Linked Data, a language to publish and to connect data available on the web by taking RDF as the underneath data structure [Bizer et al. 2009, Ferguson et al. 2012, and OWL 2004], and XQuery, a language taking XML as the underneath data structure [Boncz et al. 2006]. Obviously each approach has its unique features in terms of methodologies and the scopes of their applications. But the most fundamental difference relevant to Froglingo lies in the underneath data structures. Instead of discussing the languages themselves, we here focus on the data structures used by the different approaches in this section.

Table (or relation) in relational DBMSs has been the dominant data structure for database management systems in industry. However, the popularity of the relational DBMSs does not say that table is perfect. As soon as the relational data model was proposed in 1970s, the flat structure of table was recognized as a weakness in expressing richer semantics [Kent et al. 1979]. The weakness has become evident in the wake of big data initiatives. Hierarchical data, for example, can be folded (or decomposed) into a table in a relational DBMS, but its containment relationships cannot be captured by the relational data model.

The hierarchical data structure has been very successful in managing those business applications embedding containment relationships. For example, a file management system organizes files in hierarchies, and a LDAP (Lightweight Directory Access Protocol) system like a Microsoft Active Directory manages resources in a corporate environment in hierarchies. Another well-known example is the HTML and XML documents embedding text-based data in hierarchies, which constitute almost the entire information available in the Web. Unfortunately, not all the business applications can be well fit into hierarchical structures. The most typical example is the many-to-many relationships well managed in relational data model.

The network-based structure, i.e., semi-structured data and graph-oriented data, has been widely applied in programming languages, by which almost all of the software applications by far were constructed. In addition to being embedded in programming languages, the network-based structure now has been widely used in many NoSQL database systems such as Google Bigtable and MongoDB with which a set of built-in operators are supported. However, the network based structure has its own dilemma: while it is capable of capturing everything by incorporating the hierarchical structure, it found itself the inability of representing and querying cyclic data. The work presented in the article [Buneman et al. 2000] was an attempt to resolve the cyclic data issue for semi-structured data. However it does not work without introducing an extra concept "marker". In addition, the network based structure without the incorporation of hierarchical data structure cannot contribute to set-oriented operations except for the navigational walking one node at a time.

In this article, we have demonstrated that the EP data model can express any business data and queries that are expressible in the traditional data models and network-based data structures. The conclusion has become clear when the EP data model is said to approximate partial computable functions in [Xu et al. 2014]. In addition, the EP data model has much more functionality. The application function introduced in Section 2 and the pre-ordering relations are some examples of the advances of the EP data model. The query of "if there is a path between two vertices in a directed graph" in Example 1.2 is a case that the EP data model is more expressive.

Without addressing data security and system performance, the EP data model would not truly address the issues of big data. The EP data model has its built-in data security facilities. Heavily relying on the dependent relations discussed in Section 3, the EP data model organizes data in multiple tree structures and therefore the built-in data security facilities control data access according to data dependencies, as if the access control facilities of an operating system controlled file accesses according to the hierarchical structure of files.

The limitation of the relational data model in expressive power has been well known since it is proposed in the 1970s. But the most noticeable obstacle facing big data is its poor performance scalability, i.e., the response time increases much faster than the increase rate of relational database sizes. Separating dependent data into multiple tables by normalization is the root cause of expensive join operations. Similar to hierarchical data models and network-based data structures, the EP data model stores dependent data together and uses indexes. As a result, the system performance of the EP data model is minimized. For example, the query given in Example 2.9 has the complexity O(n) while the corresponding join operation in SQL is O(n log(n)) [Xu et al. 2006]. The experiment in the EP data model also demonstrates a high performance scalability over various data media including data, files, and HTML web pages, and over various data volumes.

Added with variables, the EP data model is expanded to a Turing-machine equivalent programming language, called Froglingo [Xu et al. 2013]. For example, the factorial function, $f(n) = 1$ if $n = 0$ otherwise $n * f(n\text{-}1)$, can be expressed in Froglingo as a database: *f 0* := *1*; *f $n* := (*$n * f* (*$n - 1*)), where all business logic are treated as data and are expressed in databases. Froglingo, with additional extension, is for data constraints, web service, and data analytics [Xu et al. 2010].

Beyond the considerations of big data, the EP data model is claimed to be a universal data exchange protocol. It is concluded in [Xu et al. 2014] that a database in the EP data model defines a finite set of finite functions that is an approximation to partial computable functions. Therefore the union of all databases and the union of all partial computable functions amount to the same object: the universe of all computable information. Put it differently, though the EP data model is not as expressive as a programming language, it can be as practically effective as a programming language in the sense that it produces all the outputs that a programming language can produce, as far as the limitation of finite time and space available to computers is concerned. Saying that the EP data model is as practically effective as Turing machines does not mean that a programming language is no longer needed in software development for a productive software development. However, it gives us a full confidence to choose the EP data model as a standard database management tool and a data exchange protocol. One can imagine that some day we have one database that accumulates more and more data to better and better approximate the real world. The accumulation process implies that all business data are stored as finite sets of finite functions and that no further tools are needed to translate or interpret data, in a contrast to today's practice in the contemporary technologies.

6. CONCLUSION

Lacking a higher expressive data type in programming languages for finite data is a reason to utilizing multiple data types such as lists, trees, graph-based structures, and relations imported from a relational database. As a result, the communications among the multiple data types complicate software development efforts. The EP data model eliminates the extra burden in expressing finite data in programming languages and in data communications.

Provided that all data desired to be managed in a database is bounded functions definable in the EP data model, the EP data model is a mathematical underpinning universally for arbitrary data management and data exchange. The desire of managing the bounded functions for business data is quite reasonable because all a computer can do is to produce finite approximations to partial computable functions. Even more, the desire is a little luxurious because bounded functions are more than finite approximations.

Froglingo has been implemented as a language system that unifies database management, programming languages, file systems, and web services. It is an all-in-one tool for general-purpose software development. A food recipe advisor in Froglingo was an award recipient in the ICCBR 2010 Computer Cooking Contest. Many commercial applications, including the website www.froglingo.com, a school registration website, a data format converter, and an on-line health advisor, have been implemented in Froglingo. Through the experiences, Froglingo demonstrated the anticipated results (in addition to a high performance scalability as discussed earlier). It allows developers to focus on business logic and barely worry about anything else beyond since it is a unified tool and does not have much facilitating concepts (constructors) such as classes or types. Given a software application written with 1000 lines of source code in traditional programming languages, for example, our experience shows that one only needs to write about 100 line of code in Froglingo. Also it hardly leaves security holes for cyber attacks since it is a highly abstracted - business focused - language.

REFERENCES

Bizer, B., Heath, T., Berners-Lee, T. Linked data - the story so far. In *Journal on semantic web and information systems, 2009*

Boley, H., Paschke, A., Shafiq, O. RuleML 1.0: The Overarching Specification of Web Rules. To appear in the Fourth International Conference of Rule ML, 2010 In *The fourth international conference of Rule ML, 2010*

Boley, H., Tabet, S., Wagner, G. Design Rationale of RuleML: A Markup Language for Semantic Web Rules. in *Proc. SWWS01, Stanford, July/August 2001.*

Boncz, P., Grust, T., Keulen, M.V., Manegold, S., Rittinger, J., Teubner, J. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *SIGMOD 2006, June 27-29, 2006, Chicago, Illinois, USA.*

BUNEMAN, P., FERNANDEZ, M., SUCIU, D. P. Buneman, M. Fernandez, D. Suciu. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. The VLDB Journal, 2000. In

CHANG, F., DEAN, J. Bigtable: A Distributed Storage System for Structured Data. in *Google Inc., IOSDI 2006.*

FEIGENBAUM, L., HERMAN, I., HONGSERMEIER, T., NEUMANN, E., STEPHENS, S. The Semantic Web in Action. In *Scientific American, 297(6), pp. 90-97, (December 2007)*

FERGUSON, M. Architecting A Big Data Platform for Analystics. In *IBM White Paper, Intelligent Business Strategies, October 2012.*

HEITMANN, B., HAYES, C. Enabling Case-Based Reasoning on the Web of Data. In *Workshop Proceedings of the Eighteenth International Conference on Case-Based Reasoning (ICCBR 2010), pp. 133 - 140.*

KENT, W. Limitations of Record-based Information Models. In *ACM Transactions on Database Systems, Vol. 4, No. 1, March 1979, Pages 107 - 1*

KRZYZANOWSKI, P. Bigtable, a NoSQL massively parallel table. In *November 2011, available at: http://www.cs.rutgers.edu/ pxk/417/notes/content/bigtable.html.*

LEE, J.K., SOHN, M.M. The eXtensible Rule Markup Language. In *Communication of the ACM, Volume 46, Issue 5, pp. 59-64, May 2003.*

OHORI, A., BUNEMAN, P., BREAZU-TANNEN, V. Database Programming in Machiavelli, a polymorphic language with static type inference. In *ACM SIGMOD, 1989, pp. 46 - 57.*

TRENTELMAN, K. Survey of Knowledge Representation and Reasoning Systems. In *Defence Science and Technology Organization, Australia, DSTO-TR-2324, July 2009.*

WARD NoSQL Databases in the Cloud: MongoDB on AWS. In *Amazon Web Services, March 2013.*

XU, K.H. A Notion of Database from the Lambda Calculus. In *The 2014 Computability in Europe Conference (CiE 2014), Budapest, Hungary, June 2014*

XU, K.H., ZHANG J., GAO S. Approximating Knowledge of Cooking in Higher-order Functions, a Case Study of Froglingo. In *Workshop Proceedings of the Eighteenth International Conference on Case-Based Reasoning (ICCBR 2010), pp. 219 - 228.*

XU, K.H. Froglingo, A Database Programming Language. In *A Talk at Microsoft Research Cambridge, May 2006.*

XU, K.H. A Users Guide to Froglingo, An Alternative to DBMS, Programming Language, File System, and Web Server. In *Available at the website: http://www.froglingo.com/FrogUserGuide10.pdf*

XU, K.H. Froglingo, an Monolithic alternative to DBMS, Programming Language, Web Server, and File System. In *The Fifth International Conference on Evaluation of Novel Approaches to Software Engineering, 2010.*

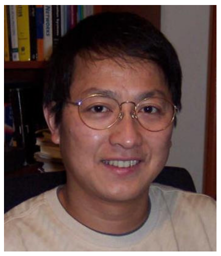
XU, K.H. EP Data Model, a Language for Higher-Order Functions. In *Manuscript unpublished, March 1999. http://www.froglingo.com/ep99.pdf.*

XU, K. H., BHARGAVA, B. 1996. An Introduction to Enterprise-Participant Data Model. In *The seventh international workshop on database and expert systems applications (September, 1996, Zurich, Switzerland), page 410-417*

OWL OWL Web Ontology Language Overview. In *W3C Recommendation, 10 February 2004, available at: http://www.w3.org/TR/owl-feature.*

Kevin Xu is a Ph.D. student in the Department of Computer Science. He is interested in database management and programming languages and is conducting a research to unify SQL, NOSQL, and programming languages for general database applications, including Big Data. He has work experience in IT security and business analysis in financial service and telecom industries.

Yi Sun, Ph.D., is an Associate Professor with the Department of Electrical Engineering at the City College of City University of New York. He received the B.S. and M.S. degrees from the Shanghai Jiao Tong University, Shanghai, China, and the Ph.D. from the University of Minnesota, Minneapolis, MN, all in electrical engineering. Dr. Sun's research interests have been in modeling, parameter detection and estimation, algorithm development, and performance analysis in various human-designed and biomedical systems by means of signal processing, image processing, probability and statistics, and information theory. His research previously focused on image processing, wireless communications, and robotic searching, and now focuses on biomedical nanoscopy and quantitative bioimaging.



Prof. Bharat Bhargava is currently a Professor of computer science at Purdue University. He received the BE degree from the Indian Institute of Science, and the MS and PhD degrees in electrical engineering from Purdue University, West Lafayette, IN. He His research involves mobile wireless networks, secure routing and dealing with malicious hosts, providing security in Service Oriented Architectures, adapting to attacks, and experimental studies. His name has been included in the Book of Great Teachers at Purdue University. Moreover, he was selected by the student chapter of ACM at Purdue University for the Best Teacher Award. He is a fellow of the IEEE.