Building New SQL

Those who afraid of simplicity
do not get along with real importance.

Eugene Panferov, e@bestmx.net

Abstract

In the following article we criticize SQL for not following the relational algebra; describe the problem of the multitude of different relation representations in SQL; analyze in detail the disadvantages of the notions “TABLE” and “FOREIGN KEY”; and propose a complex solution which includes brand new query language and relatively small yet very significant alteration of the data storage concept.

In the proposed language we will try to overcome the following apparent flaws of SQL:

- human language mimicking
  - which leads to inconsistent, completely inextensible, and needlessly complicated syntax
- mixing relational operations together and with the output
- imperative DDL
- impossibility of relations between relations
  - which leads to non-homogenous representation and undermines the relation abstraction itself, lowering the level of programming

by abandoning the notion of table and by creating:

- frugal, extensible, functional-style syntax, utilizing “set-theory properties” of relations
- implicit and independent notions of relational operations and I/O operations
- declarative DDL
- single unified representation for relations capable of adopting another relation as a domain

To support the last and the most important novelty we propose certain storage means for relations between relations.

Intro

SQL is a huge, unbelievable success. It is the most successful non-imperative algorithmic language ever existed. It is the only non-imperative language commonly accepted by the ignorant majority of which the only way of thinking is dumb straight imperative. Yet it is not perfect.

As environment is constantly changing, imposing new requirements and proposing new grounds, dissatisfying predictions and creating opportunities, SQL remains basically unchanged, keeping initial imperfections and adopting only minor (mostly controversial) “features”.

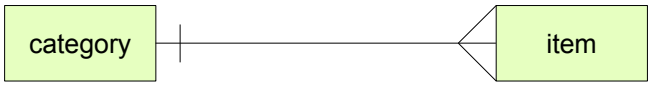
So, nowadays, in the era of immensely powerful beyond the human comprehension computers, the Founding Fathers of computer science could not even dream of, when high level of programming fantasies can finally come true, we are facing not at all as high level language as SQL could and was meant to be, we are still tinkering with “how many bytes to reserve for a surrogate key”. It is a shame!

This situation motivates hordes of uneducated no-SQL proponents who simply did not get the idea of high level programming at all, and they are trying to abolish everything that SQL has achieved, they are pushing us down to the level of imperative (general purpose) languages where a programmer is obliged to take care of every byte and the most advanced data management tool is a garbage collector.

Moreover, SQL has been betrayed by its own proponents – ISO sanctioned XML contamination, which effectively creates a rivaling alternative to SQL within SQL – the apparent sign of decay, not even mentioning the utmost inherent failure of XML itself.

We propose you an opposite way, the way up, to the higher than SQL level of programming.

What Do We Have



A very typical approach to database design is Entity-Relation diagrams. It is very intuitive and informative way of visualizing data structure.

I the example above we have two entities and one relation between them. Let us map them into a relational database. The entities will be mapped as tables, and the relation will be mapped as a foreign key.

But tables are supposed to represent relations! From the perspective of relational algebra the *Item* and *Category* entities are relations. So, we have three relations: *Category, Item,* and the relation between them.

Why do we represent two of them properly (as it meant to be) and at the same time simulate the third one by low level programming of foreign keys? Is there a real need for a non-homogenous representation? Is the relation “item belongs to category” any worse or better than relations *item* or *category?*

The price of foreign keys

For this particular disturbance of the representation we pay by lowering the level of programming dramatically. Foreign keys are very low-level thing relatively to tables.

Compare a table representing a relation versus a foreign key representing a relation:

```
CREATE TABLE category (
  Id   INT,
  name TEXT );
```

Here you just declare: these two attributes constitutes a relation. You do not care how the association between them will be built and maintained. Ever since you put this declaration you can freely manipulate pairs (id,name) and apply whatever relational operations to the relation category.

```
CREATE TABLE item (
  Id   INT,
  category INT REFERENCES category(id),
  name TEXT );
```

Here you have created a fake attribute *category* for the relation *item*, you have defined it's type which have no correspondence to any domain knowledge entity (remember? Domain knowledge is “item belong to category” – it says nothing about integers), and then you command to your RDBMS to check your input in order to keep obvious garbage out (in other words, this “references” directive defines a subset of the integer, making the type of this fake attribute more relevant).

You have created a relation manually! You are involved in the very internals of the representation.

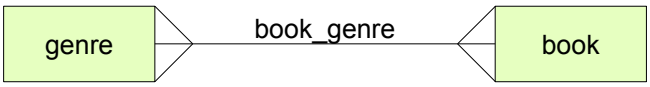
The maintenance is up to you. The interpretation is up to you too. The system does not recognize this representation as a relation. You can not apply relational operations to it.

So, we have +1 agent that undermines the status of an RDBMS, as we can see, it is not as relational as it is nominated.

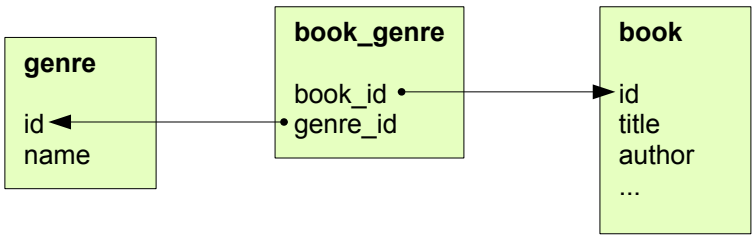
The price of link-tables

Because foreign keys are not capable of representing many-to-many relations, there is another alternative representation for relations: link-tables. It is also a low-level simulation of relations, it also compromises the idea of RDBMS by rivaling tables. And it is available for extra price.

Given the simplest case of a relation many-to-many:



The most recommended mainstream way to represent it in terms of tables is:



Let us select genres of a book X:

```
SELECT genre.name FROM book_genre, genre, book
  WHERE book.title = X
    AND book_id  = book.id
    AND genre_id = genre.id
```

Nothing unusual, all three tables properly joined according to the foreign keys provided. But, look, *book_genre* is a relation on the cartesian product of *book* and *genre*. All information we want to retrieve is located inside this product. In other words we need to perform a search of the dimension (book, genre). What do we do in the select above? We produce a cartesian product of *book*, *genre*, and *book_genre* itself! And then we perform a search of the dimension twice bigger than needed.

Easy to see that an RDBMS appears now even less relational.

The diversity of rivals

Please, do not confuse these two representations, they have nothing in common and are not related at all. A foreign key (as a representation for relations) is not a part of a link-table representation for relation since in this case a foreign key itself DOES NOT represent any particular relation. However, this diversity has its price too: all foreign keys in a database come in two varieties: (a) representing a relation (b) not representing a relation, so that you have a set of absolutely indistinguishable objects with opposite semantic.

Moreover, there are more than these two alternative representations of a relation in SQL. Some RDBMSes introduce subclasses and inheritance. Needless to say that a subclass a relation on a class, therefore, we have another rival representation.

Some RDBMSes introduce complex types, arrays, collections which are sort of relation representations too. Let us take a look at oracle documentation:

```
---- a collection is defined as:

     CREATE OR REPLACE TYPE emp AS OBJECT (
       e_name VARCHAR2(53),
       s_sec VARCHAR2(11),
       addr VARCHAR2(113) );

---- You can create a table with this object
```

This is a relation, no more and no less, yet another representation!

Which representation to choose for a particular relation? Is there any method to choose representations form this multitude? Why so many representations after all?

The price of tables

This representation zoo is the price for the ugliness of the notion of table.

There are two useful definitions of a relation in math.

A relation L over the sets X1, …, Xk is a subset of their Cartesian product,
written $L \subseteq X1 \times \ldots \times Xk$.

A relation L over the sets X1, …, Xk is a tuple L=(X1, …, Xk, G(L)),
where G(L) is a subset of the Cartesian product $X1 \times \ldots \times Xk$.
G(L) is called the graph of L.

A relation is a set.
A relation's domain is a set.
Ergo: a relation is capable of being another relation's domain.

SQL does not reflect this very basic fact.

Any domain knowledge is not a plain set of relations, it is whole hierarchy of relations. And not surprisingly only a minority of relations are leaves of this hierarchy, typically a majority of relations adopt other relations as domains.

We have a representation: relation → table; domain → attribute.

But a table can not adopt another table as an attribute! Therefore, we can not represent our domain knowledge in terms of tables. There simply is no room for relations between relations.

Though, we can represent SOME relations.

Once we represent any relation with a table we prevent table representations for all relations that includes the current one and all that are included by the current one. The representation of the former takes link-tables and foreign-keys. The representation of the latter takes complex types and collections.

This is why we are stockpiling alternative representations. And it not a solution at all. An RDBMS operates with tables, which are involved in mere part of a data model, so that we have not have our data model properly controlled – our data model situates partially (and mostly) outside a system – on what premises you still call it “relational”?

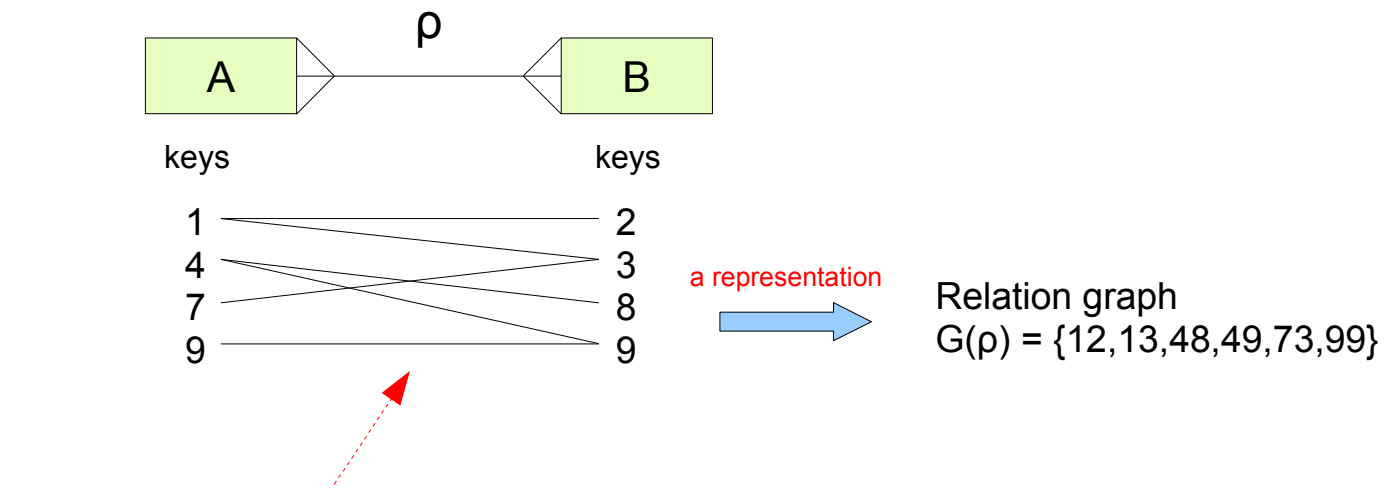
What Do We Have To Have

First of all we must make possible relations between relations. In order to achieve this, we only have to make a closure:

```
relation ::= domains graph
domains  ::= domain
domains  ::= domains domain
domain   ::= relation
domain   ::= scalar_type
```

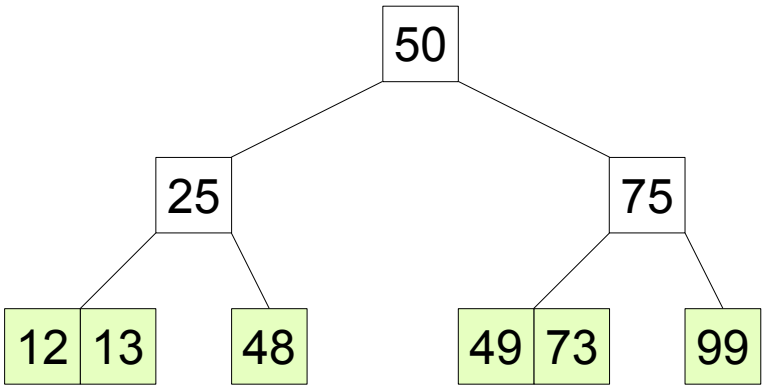
Everything is plain and clear in this definition except for the mysterious “graph” that is yet to be defined.

Let's say we have a binary relation **ρ** between two sets *A* and *B*:



Assuming we already have the domains A and B somehow represented, we only have to represent the legs of the graph. Therefore we may treat a relation graph as a set. Finite set. And every finite set can be represented as a set of integers. In this particular example we have the relation graph represented as as a set of six integers.

Let us just create an index on this set:



This index (by-design) represents the graph of the relation **ρ**

The primary purpose of a relation graph is to answer the question: “whether a tuple is a member of the relation”. This is the question an index is supposed to answer too.

A graph and an index share the purpose and share representation.

Because of this, from our perspective: a relation graph IS an index.

What is missed on this index is not a member of **ρ**. What is on this index contains a primary key reference to the element of the domain. Furthermore, nothing prevents this index from holding storage node references.

Did we assume the domains *A* and *B* are of any particular types?

No, we did not impose any restrictions on them. It means that domains of the relation **ρ** can be relations.

We have just created a relation between relations and it is not out of ordinary. It is merely an index. The only novelty is that attributes of this index come from different tables. It is significant novelty? But it does not create any wrong about the index itself. Contemporary software (with very little improvement) can handle such indexes.

Also we can treat it as usual relation, as we just did few paragraphs before and succeeded. Strictly speaking an index is a very special relation – a relation with linear order? But it is still relation anyway, and it contains all information we needed.

Thus, introduction of multitable indexes will allow us to store relations between relations and treat them as other relations. So that we would stick with single unified representation for relations, that leads to a homogenous (rather to say self-similar) scalable representation of a domain knowledge.

Paradigm Shift

As I demonstrated, tables are incapable of representing relations. But tables already replaced the very idea of relations. Everybody think of relations as “geeky euphemism” for tables. It is gross, but it is very strong public opinion, and despite it is unexpressed, it is probably the principal cause of SQL's waning.

Tables aren't relations! – Indexes are!

Also, multitable indexes put joins out of the job.

A join creates a relation from existing relations. At first glance, it looks a reasonably useful operation, but practice reveals that a resulting relation does already exist... always.

Let us select something from the first example “item-->category”:

```
SELECT category.name, item.name
 FROM category, item
 WHERE category.id = item.category_id
```

The relation highlighted with yellow, is already in our database, but it is hidden behind the foreign key shown in red. And it is not “new” information created inside the database, this information is put inside the database on purpose. Every pair of tables you ever join them you by yourself MADE JOINABLE!

A significant part of a programmer's labor is to make all joins precisely predictable. A programmer must make sure all joins will result in a set of relations that are meant to be stored. No join will reveal any new information.

Joins do merely convert relation representations, and they do this meaningless job each time we are accessing information.

Moreover, a programmer is forced to codify this representation conversion routine for every relation between relations.

Do we need such a resource consuming operation of which the result we already know? Whereas to avoid this embarrassingly needless labor we only need to change the representation of relations.

With multitable indexes we can just STORE relations between relations instead of recalculating them repeatedly.

NO MORE JOINS, NO MORE TABLES, NO MORE REPRESENTATION ZOO.

The Language Proposal

We need a language to be pure, simple, and coherent. Notation must be clear, unambiguous, and intuitively human readable (but not alike a human language, in fact resemblance of a human language does only complicate understanding). Ideally, similar objects must be described by the similar sentences of the language, while dissimilar ones by the easily distinguishable sentences. Also, I want to keep a number of keywords and unique syntax constructs to the bare minimum.

As everything is already invented, we will try to stick with s-expressions and follow the functional style.

Separate relational operations. We want an explicit notation for projection and pure relational selection (without “order by”, “limit” etc).

Separate output operation and make it explicit. It allows us to further purify scripting, and at the same time enrich output formatting.

Split context. In SQL we had to deal with certain limitations on SELECT depending on its context. In some context certain clauses are disallowed. We want to get rid of this complication by introducing two distinguishable contexts (relational (where any selection is possible) and non-relational (where no selection possible)) and a predefined (fixed) set of operations that cause context shift (e.g. “order by” takes relational object and returns non-relational object).

Declutter the notation. We will keep the notation free from meaningless variety of separators – space is enough. For example, if we want to construct a triple, we have to provide triple members (and optionally their order), like this: (x1, x2, x3). The question is what information the comma symbol adds to this notation? The only right answer is: void. Because of this, we simply discard the garbage, so we got: (x1 x2 x3). This seemingly superficial change in fact is a very significant improvement to the syntax. It effectively removes the whole parasite idea of “in between”, which ordinarily causes whole series of tiny annoying problems (particularly nasty in machine generated scripting, you know) starting with “duplicate separator”.

Make basic types and relations interchangeable. This is the pivot point of the language. It makes the language capable of expressing relations between relations.

Introduce variables and assignments. SQL does not provide a room for assignments, they are totally alien to the SQL's structure, yet strongly demanded (at least nowadays). Our variables will be IMMUTABLE, will have a single transaction lifespan and visibility, will be interchangeable with relations in every context except for data definition, and will represent only sets of tuples (subsets of arbitrary relations, practically). Assignments have no need to be calculated immediately.

Make DML returning value. Since we have explicit output operations, and separated relational operations, and we have assignments, then we can spare the whole “RETURNING” clause by making DML return affected rows by default. And since we may utilize a return value only explicitly, then we can just discard it by not utilizing it.

Keep types as few as possible. The epoch of counting bytes has passed. There is no need in keeping several different types for integers, also we do not see any high level application for bitwise operations and related stuff. We now want computers to count their bytes (if they are concerned). On the other hand we provide a useful tool for constructing complex types of arbitrary complexity, namely relations – we do not need to anticipate all possible user's wishes by maintaining a library of fancy peculiar types which will be rarely known and never used (because user's wishes always prove themselves more peculiar than our anticipation).

Respect the fact that a relation is a function of its primary key. Indeed we can treat them as functions all the way long and that gives us an opportunity to create a procedural language later on.

Typographic Convention:

To describe syntax we will use BNF.
We will type terminals in **`bold`** font, keeping their literal value, if this value is predefined.
We will type terminals plain `UPPERCASE`, if their value is variable.
We will give lowercase `identifiers` for non-terminals.
We will highlight rules of particular significance with red sign `::=`

NOTE: These BNF snippets are not the actual grammar used in the prototype software. These are designed for better understanding, using broader set of tokens and redundant rules, these describe the language from the perspective of a user. The actual grammar differs to these BNF snippets since it is designed for different purpose (see file: “parser.y” in the source code).

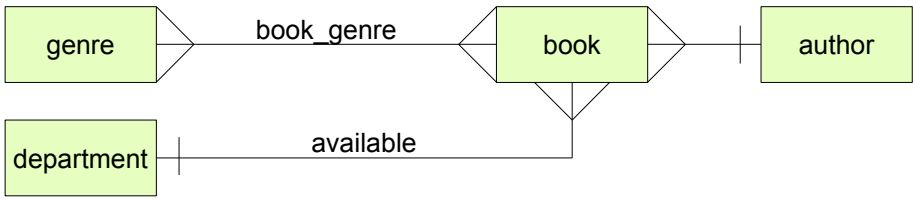
Data Definition

```
definition ::= relation ( NAME domains )
definition ::= domain   ( NAME domains )
definition ::= function ( NAME domains ) expression
domains    ::= domain
domains    ::= domains domain
domain     ::= type
domain     ::= ( NAME type )
type       ::= TYPE_NAME
type       ::= RELATION_NAME
// we will define expression later
```

There are three classes of relations (they are all relations in any sense):
simple relation – a relation which tuples are defined by a user,
domain – a relation which always contains all possible tuples,
function – a relation which tuples are formally defined (can be calculated and cannot be altered).

Examples:

Let there be a tiny public library:



Assuming hereafter that we have predefined scalar types: *timestamp, text, int, real*

The definition of this library will look like:

```
relation (author (name text) (birthdate timestamp))
relation (book author (title text) timestamp)
relation (genre text)
relation (book_genre book genre)
relation (department text)
relation (available book department)
```

Now let us illustrate a domain definition:

```
domain (point2d real real)
```

```
domain (circle (radius real) (center point2d))
relation (my_circle circle)
```

In the example above circle is a set of all possible circles. While my_circle is a set of user defined circles – user himself controls which tuples do belong to the latter set.

Domains play a role of complex types.

Of course we can select from them too, if we can determine finiteness of a set prior to output, then why not. Of course we are assuming laziness. Laziness is so natural in a transactional context – the entire workflow is already cut by checkpoints.

Arithmetic

We like sex.

```
expression ::= ( OPERATOR list )
expression ::= ( member OPERATOR list )
expression ::= ( TYPE_NAME expression ) // this is a typecast
list ::= member
list ::= list member
member ::= CONST
member ::= NAME
member ::= selection
member ::= expression
// we will define selection later
```

To keep some operators (namely comparison) easily readable for everyone we decided to allow alternative placement of an operator within an expression, nevertheless we prefer prefix notation.

Examples:

```
(+ 1 2 3 4 5)
(& (> (-17) (* 1 2 3 (-5))) (“xcf” < “fgh”))
function (avg2 (a real) (b real)) (/ (+ a b) 2)
(+ (int “123”) 4)
// operator type is defined by the type of the first operand
// however, we will be glad to get rid of operator overloading
```

Selection

What information do we need to provide to a system in order to select all tuples of a relation?
The relation name – and nothing more!
So that selection operation basically looks like:

```
(author)
(book)
```

```
selection ::= ( NAME )
selection ::= ( NAME list )
selection ::= ( NAME : expression )
```

```
list ::= member
list ::= list member
member ::= selection
member ::= expression
member ::= CONST
member ::= . // this is a shorthand for a non-captured domain
// and probably this one too
selection ::= ( NAME list : expression )
```

Here the list acts like a positional reference to the relation's domains, so that selection may be treated as function call. The expression after "**:**" acts like a "where" clause, it is a filter of the relation's tuples. These both are equivalent in any sense.

Examples:

```
(author "Dawkins" "1941")
(author :(name ~ "A.*"))
(book (author :(name ~ "A.*")) (text) (timestamp))
// here "meaningless" selections (text) and (timestamp) stand for
// non-captured arguments
// the two last domains of a book may be of any value.
// Surely we must create a shorthand for this.
(book (author :(name ~ "A.*")) . .)
// all books of all authors with the name's initial "A"
```

Tuple Constructors And Set Constructors

Quite naturally, for tuple construction we will employ Cartesian product, for set construction we will employ union operation, and we will create the following notation for them:

```
product ::= { list }
union   ::= ( list )
list    ::= member  // we do not see a practical reason for explicit construction of empty sets
list    ::= list member
member  ::= CONST
member  ::= expression
member  ::= selection
```

Examples:

```
{1 2 "txt"}   // a triple of two integers and one text
(1 2 3)       // a set of integers
({1 2} {3 4}) // a set of two pairs of integers
{(1 2) 3}     // also a set of two pairs: {1 3} and {2 3}
```

Of course we can do all the same with functions and selections: put selections into their arguments, and put function calls into selections and arbitrary combine them with constructors. All functions accept SETS instead of each argument and they act as if they are "MAPPED" by each argument (see, for example, "map" function definition in Haskell).

Examples:

```
{(author) “he is author”} // we extend each selected tuple
((genre) “bore”)          // we extend a selected set
(avg2 (1 2 3) 3)          // == ((avg2 1 3) (avg2 2 3) (avg2 3 3))
```

Join-like Operation

Since we have all relations between relations explicitly defined, the data scheme is a graph. So that a pathfinding problem can be formalized. It gives us very powerful operation: finding a connection between a relation and another relation's subset.

```
connection ::= { NAME selection }
```

Example:

```
// select “all genres of the given author”

{genre (author 'Dawkins' ?)}
```

The difference to the “JOIN” is that we do not have to specify the exact connection between relations (are being searched). On the contrary! We command a computer to find this connection and utilize. Please, refer to the “Data Definition” example, a path from the *genre* to the *book* is clearly visible there. However, it's semantic is not specified, we (as we designed this data scheme) know what it means.

We think this operation must return a Cartesian product of the two relations involved in the search.

We are not sure about a return value in case there is no connection between given relations.

Projection Operation

As we are tampering with the parenthesis notation, we have no option but to employ this notation again (not because of the style, but because of the resolving of expressions).

Please, note: since domains themselves could be relations you may apply projection operation to them as well.

```
projection ::= [ selection list ]
list ::= member
list ::= list member
member ::= DOMAIN_NAME
member ::= [ DOMAIN_NAME list ]

// the closure, in order to utilize projection in a relational context
selection ::= projection
```

Examples:

```
[(book) author title]
[(book_genre . (genre 'sci-fi')) [book author]]
```

```
[(book_genre . (genre 'sci-fi')) [book [author name] title]]
```

Data Management

The most difficult and even controversial matter is the UPDATE operation.

In the perspective of a pure relational system it is rather doubtful if this operation have even a right to exist.

It is no-brainer in a context of old-school tables keeping spreadsheets of junk fields. But in our system, every field matters (remember we put all insignificant fields aside), so that changing a field value is in fact creating an entirely different tuple. Where by “different” I mean a tuple semantic. What is a reason of changing “Shakespeare” record into “Dawkins” record? If you want to, I strongly suggest “add / remove” procedure.

The most sensible apology for update operation are probable data input mistakes. So that keeping them in mind I propose the following syntax:

```
command ::= add RELATION_NAME set
command ::= remove RELATION_NAME set
command ::= update RELATION_NAME set ( list )
set ::= product
set ::= union
set ::= selection
list ::= member
list ::= list member
member ::= DOMAIN_NAME expression
member ::= DOMAIN_NAME CONST

// plus “remove cascade” written in a single word
command ::= abolish RELATION_NAME set
```

Examples:

```
add genre {“bore”}
add author ({“Dawkins” “1941”} {“Homer” “800 BC”})

remove book (book . “War And Piece” .)

// removal with respect to another relation
remove book (genre “bore”)

update author (author) (name (capitalize name))
update author (book:(title ~ “A.*”)
       (
         name      (capitalize name)
         birthdate “1910”
       )
```

Thus, we achieve significant improvement in notation for DML operations referring multiple relations, such as notoriously ugly “DELETE FROM FROM” and “UPDATE FROM”

We are not sure about positional “field” assignments, but they may be possible too.

Assignments, Laziness, Transactions

We have quite a structure in our database. Putting a tuple into a relation may require several tuples in other relations.

Let's assume we do not have “Homer” and we try to add “Ulysses”.

```
add book ((author “Homer”) “Ulysses” “750 BC”)
```

it will cause data integrity error. So that we do:

```
add author {“Homer” “800 BC”}
add book {(author “Homer”) “Ulysses” “750 BC”}
commit
```

Here we have a point of optimization. Common sense tells us that we do not need do explicitly select a tuple we just added to a relation, we may maintain a sort of a pointer left over the DML operation. Perhaps this particular imperfection would be algorithmically detectible, and possible for a machine to optimize, but pointers will also improve the outfit of the notation (if there are more than one of them, of course):

```
Homer = add author ('Homer' '800 BC')
add book {(Homer) 'Ulysses' '750 BC'}
commit
```

Theoretically (taking transaction qualities in account) we may naturally introduce laziness into DML. Everything we did before commit we actually did not, we only did put plans for the operation we need to perform. It looks like laziness perfectly fits this application. Also we can defer error reporting until a commit point, allowing a user to temporarily “violate” some constraints. Because it is just a plan that have no immediate effect on a database, you can even think about it as an unfinished sentence.

This is a good problem to think about. Do we really need the immense multitude of these old-school options affecting the behavior of transactions? Can we, in our new environment, put this cultural pluralism to an end by creating a single (or few?) firmly defined behavior that always makes sense?

Also assignments will be useful to break complicated selections.

```
Dawkins = ('Dawkins' '1941-03-26')
His_Books = (book (Dawkins) . .)
(book_genre (His_Books) .)
```

Note: we do not demand assignments to be materialized, we also appreciate them as nominators. This is another ground for laziness.

In order to make this fun with variables as less messy as possible we constitute the following principles:

- variables are immutable
- the visibility and the lifespan is a single transaction
- the type is always is “set of tuples”
- variables are interchangeable with relations
  (variable can and only can substitute relations in every context)
  Note: the parenthesis around variables, it is a selection, hence NAME terminal can not be an expression, but selection can.

The definition of assignment is:

```
assignment ::= NAME = command
assignment ::= NAME = selection
```

while variable usage is already defined above, you can put variable's name in any context where you can put relation's name (except for the DML statements... but, wait, why not?).

Thus, a variable's simplest usage comes wrapped in parenthesis, as shown in examples above. By the way it is an unconditional selection, and we may do conditional selections (from variables) as well.

```
A = (author:(name ~ "A.*"))
B = (A:(birthdate > "1940"))
C = (A . "1940")
```

Output

```
output ::= output selection
output ::= output formatting selection
```

Here we define WHERE we want TO LOCALAZE all nasty output formatting stuff that breaks relational model and makes SQL astray.

The formatting features themselves are out of the scope of this article.

Strict isolation of the output formatting allows us to define much more sophisticated and diverse format methods, including those the most impossible for SQL, you probably very familiar of.

We are free to do literally anything with our selection result, since in the context of the “output” operation it is no longer in the relational context and we are not limited to relational operations. We can even introduce cycles!

PL

All in all we inevitably will need procedural language. However, it is out of the article's scope, easy to observe that we can turn our transactions into procedures – stored parameterized transactions. Also we already introduced pure functions. Also the proposed language seems to be capable of seamless incorporation in a functional workflow.

Plausibility Of The Example

There you could have noticed damn few “attributes” in the relations I have pictured, which is very unlike “real” 100-fields tables. There is a reason, besides an educational purpose, to keep this example data structure so seemingly oversimplified.

All those 100-field tables (being results of “optimization”) ARE NOT RELATIONS.

Most of their numerous fields are not their domains (usually only few of the fields are, while the rest represent all sorts of META-information or information somehow linked to relation tuples and should be stored separately).

Style Questions

- case sensitivity
- quotation marks (SOLVED: double-quotes only; opening and closing symbols are equal)
- assignment syntax (there is an opportunity for several alternatives, will they be useful?)
- filter symbol (“:” doesn't look good, while usual set-comprehension “|” looks even worse)
- arithmetic notation (prefix or infix) (SOLVED: an operator is allowed to take second position)
- product and union parenthesis (there is a reasonable opinion (opposing to the current proposal) to use “{}” for enumerated sets or unions and to use “()” for cartesian products)