Initial Value Computation

August 16, 2011

1 Defining domains

In this draft we want to explain the notion of domains in DBToaster calculus [1]. DBToaster uses query language AGCA(which is stands for AGgregation CAlculus). AGCA expressions are built from constants, variables, relational atoms, aggregate sums (Sum), conditions, and variable assignments ($\leftarrow$) using "+" and "$\cdot$". The abstract syntax can be given by the EBNF:

$$q ::- q \cdot q | q + q | v \leftarrow q | v_1 \theta v_2 | R(\vec{y}) | \text{c} | \text{v} | (M[\vec{x}][\vec{y}] ::- q) \tag{1}$$

The above definition can express all SQL statements. Here v denotes variables, $\vec{x}, \vec{y}$ tuples of variables, R relation names, c constants, and θ denotes comparison operations ($=, \neq, >, \geq, <,$ and $\leq$). "+" represents unions and "$\cdot$" represents joins. Assignment operator($\leftarrow$) takes an query and assigns its result to a variable(v). A map $M[\vec{x}][\vec{y}]$ is a subquery with some input($\vec{x}$) and output($\vec{y}$) variables. It can be seen as a nested query that for the arguments $\vec{x}$ produces the output $\vec{y}$, it is not defined in [1] but we added here for the purpose of this work.

The domain of a variable is the set of values that it can take. The domain of all the variables in a query expression can easily be computed recursively if some rules are respected. We will use through out the entire paper the notation of $\mathsf{dom}_{\vec{x}}(q)$ for the domain of a set of variables, where q is the given query and $\vec{x}$ is a vector representing the variables(not necessarily present in the expression q). We will start by saying the $\vec{x} = <x_1, x_2, x_3, \cdots, x_n>$ will be the schema of all the variables and that $\vec{c} = <c_1, c_2, c_3, \cdots, c_n>$ will be the vector of all constants, that will match the schema presented by $\vec{x}$. It is

not necessary that $\vec{x}$ has the same schema as the given expression. We will give the definition of $\mathsf{dom}_{\vec{x}}(R(\vec{y}))$:

$$\mathsf{dom}_{\vec{x}}(R(\vec{y})) = \left\{ \vec{c} \,\middle|\, \sigma_{\forall x_i \in (\vec{y}): x_i = c_i} R(\vec{y}) \neq \text{NULL} \right\} \tag{2}$$

Thus, we can evaluate $\mathsf{dom}_{\vec{x}}$ for a broader range of $\vec{x}$ and it is not restricted by the schema of the input query expression. In such cases the dom is infinite as the not presenting variables in the query can take any value.

For the comparison operator $(v_1 \theta v_2)$, where v_1 and v_2 are variables, we can compute the domain as follows:

$$\mathsf{dom}_{\vec{x}}(v_1 \theta v_2) = \left\{ \vec{c} \,\middle|\, \forall i, j : (v_1 = x_i \wedge v_2 = x_j) \Rightarrow c_i \theta c_j \right\} \tag{3}$$

The domain of a comparison is infinite.

For the join operator we can write:

$$\mathsf{dom}_{\vec{x}}(q_1 \cdot q_2) = \{\vec{c} | \vec{c} \in \mathsf{dom}_{\vec{x}}(q_1) \wedge \vec{c} \in \mathsf{dom}_{\vec{x}}(q_2)\} \tag{4}$$

while for the union operator the domain definition is very similar:

$$\mathsf{dom}_{\vec{x}}(q_1 + q_2) = \{\vec{c} | \vec{c} \in \mathsf{dom}_{\vec{x}}(q_1) \vee \vec{c} \in \mathsf{dom}_{\vec{x}}(q_2)\} \tag{5}$$

$$\mathsf{dom}_{\vec{x}}(constant) = \left\{ \vec{c} \right\} \tag{6}$$

$$\mathsf{dom}_{\vec{x}}(variable) = \left\{ \vec{c} \right\} \tag{7}$$

In (6) and (7) $\vec{c}$ stands for all possible tuples match schema of $\vec{x}$, so the domains in these two cases are infinite. Finally, we can give a formalism for expressing the domain of a variable that will participate in an assignment operation:

$$\mathsf{dom}_{\vec{x}}(v \leftarrow q_1) = \left\{ \vec{c} \,\middle|\, \vec{c} \in \mathsf{dom}_{\vec{x}}(q_1) \wedge \left(\forall i, j : (x_i = v = x_j) \Rightarrow (c_i = c_j = q_1) \right) \right\} \tag{8}$$

In fact using implication operator in the above definition allows us to extend the $\vec{x}$ to whatever vector we want, as we already said the schema of $\vec{x}$ is not necessarily the same as the schema of q. We can define the domain of a map(for map's definition refer to [1], [2]) as follow:

$$\mathsf{dom}_{\vec{w}}(map[\vec{x}][\vec{y}]) = \{\vec{c} | \vec{c} \in \mathsf{dom}_{\vec{w}}(\vec{x} \cup \vec{y})\} \tag{9}$$

We define function Ext to extend(shrink) the schema of a domain, a as input with schema $\vec{y}$:

$$\mathsf{Ext}_{\vec{x}}(a_{\vec{y}}) = \{\vec{c}|\sigma_{\forall i\, y_i \in \vec{x}:(y_i=c_i)} a \neq \text{NULL}\} \tag{10}$$

In (10) we can consider a as relation since it is a set and using the relational algebra on it. It is clear that if $\vec{x} \cap \vec{y} \neq \vec{x}$ it produces an infinite domain.

2 Defining arities

2.1 Definitions

Having the definitions for the domains, we will try to give some insight regarding to the notion of arity. We will start with an example relation:

$$q = R(a, b) \cdot S(b, c)$$

the schema for q will have three variables (a, b, c). The arity of a tuple is the number of its occurrences in the relation, in other words the order of multiplicity of that tuple. The arity of a tuple will increase or decrease if insertions or respectively deletions will be made to a relation. For example:

R	a	b	arity
	<1	2>	1
	<1	3>	2
	<3	4>	1

Table 1: Relation R

S	b	c	arity
	<1	1>	1
	<2	2>	2
	<3	5>	2

Table 2: Relation S

We need to maintain the maps for certain values as long as the arity of a tuple is greater then 0. If the arity of the tuple drops to 0 then the tuple

$R \cdot S$	a	b	c	arity
	<1	2	$2>$	2
	<1	3	$5>$	4
	<3	4	$*>$	0
	$<*$	1	$1>$	0

Table 3: Relation $R \cdot S$

will not be taken into consideration and therefore it can be eliminated from the domains of the maps.

We need to store the arities inside each map and also way to compute the arity of an AGCA expression. Since we substitute the subexpressions with maps, we can easily consider the relations as the maps without input variables. Also a map with input variables can be seen as a relation with a group-by clause. Input variable of a map bind some variables. Thus if we compute the map values by all different combinations of these variables, we will look up into these values and return the appropriate value according to the input variables.

We define the function *arit* which will be used to compute the arity of a tuple in a certain relation. The function will be defined on the relation and the tuple for which the multiplicity order is desired to be computed.

$$arit(\text{Relation } q, \text{Tuple } t) = \text{multiplicity order of } t \text{ in the } q = \pi_t(q)$$

where $\pi_t(q)$ means the projection of relation q for the tuple t. This function can be used for the computation of the arity of the expressions from the AGCA: q ::- $q \cdot q \mid q + q \mid q\theta t \mid t \leftarrow q \mid$ constant $\mid$ variable. However constants and variables can be eliminated from the computation because relations are of interest.

$$arit(q_1 \cdot q_2, t) = \sum_{\{t_1\} \bowtie \{t_2\} = t} arit(q_1, t_1) * arit(q_2, t_2) \tag{11}$$

$$arit(q_1 + q_2, t) = arit(q_1, t) + arit(q_2, t), \text{ where Schema}(q_1) \tag{12}$$

$$= \text{Schema}(q_2) \tag{13}$$

$$arit(v_1\ \theta\ v_2, t) = \begin{cases} 1, & \text{if } v_1 \theta v_2 \text{ is true and } v_1 \in t \wedge v_2 \in t \\ 0, & \text{otherwise is false} \end{cases} \tag{14}$$

$$arit(v \leftarrow q, t) = \begin{cases} 0, & \text{if } t \setminus \{v\} \notin \mathsf{dom}_{Schema(t)}(q) \\ 1, & \text{otherwise} \end{cases} \tag{15}$$

2.2 Extending the notion of domains to arities

In the first part of the document we have talked about the definition of domains. Now we are going to talk about a combination between domains of tuples and the arities of those tuples. We can define something like the following:

$$\mathsf{arity}_{\vec{x}}(R(\vec{y})) = \left\{ (\vec{c}, arit) \middle| \sigma_{\forall i : x_i \in \vec{y} \wedge x_i = c_i} R(\vec{y}) = arit \wedge arit \neq 0 \right\}$$

The notion arity will help us construct both domains for each variable, but also compute the correct order of multiplicity of a tuple within a relation. If we have, for example, a relation R with the following schema $R(\vec{y} = <x_1, x_2, \cdots, x_n>)$ and know that only a part of the variables defined by the schema are used for data flow to other relation $\vec{x} = <x_i, x_{(i+1)}, \cdots, x_{(i+j)}>$, then the $\mathsf{arity}_{\vec{x}}(R(\vec{y}))$ will be all the groups of unique tuples with their order of multiplicity. This would be the same with having a query on relation R with a group by using $\vec{x}$:

SELECT $\vec{x}$,COUNT(*) FROM R GROUP BY $\vec{x}$

The query will produce a table with all the tuples and their arities. The tuples will be unique.

For example:

R	a	b	c	d
	<1	2	4	3>
	<2	3	5	4>
	<7	2	5	2>
	<6	3	9	4>
	<11	2	4	2>
	<1	2	9	3>
	<3	3	5	5>
	<3	2	5	2>
	<3	2	5	3>

Table 4: Relation R

In table 1 we have the relation R with the schema $\vec{y} = <a, b, c, d>$. From that schema there are being used only two variables, $\vec{x} = <b, d>$ for the communication with other relation from a specific query. Applying the query presented in the previous paragraph we can compute all the unique tuples of $\vec{x}$ and their arities.

$\mathsf{arity}_{<b,d>}(R)$	b	d	arit
	<2	3>	3
	<3	4>	2
	<2	2>	3
	<3	5>	1

Table 5: Relation R

Taking into account the rule defined for the computation of the arity, we are going to have for relation R the following:

$$\mathsf{arity}_{<b,d>}(R(<a, b, c, d>)) = \{((2, 3), 3), ((3, 4), 2), ((2, 2), 3), ((3, 5), 1)\}$$

So far we have explained only one type of expression in the AGCA: the simple relation $R(\vec{y})$, however AGCA presents many more types. Therefore we have to explain each of other types of expression. We remind that in AGCA an expression q may be of the following form:

$$q ::\text{-}\ q \cdot q | q + q | v \leftarrow q | v_1 \theta v_2 | R(\vec{y}) | \text{c} | \text{v} | (M[\vec{x}][\vec{y}] :: -q) \tag{16}$$

Exactly as we did with the definitions for the domains, we can write the definitions for the arities just extending the definition of the domain. For

q::-$v_1 \theta v_2$:

$$\mathsf{arity}_{\vec{x}}(v_1 \theta v_2) = \left\{ (\vec{c}, arit) \,\middle|\, \forall i, j : (v_1 = x_i \wedge v_2 = x_j) \Rightarrow (c_i \theta c_j = arit) \right\} \quad (17)$$

For q::-$q_1 \cdot q_2$:

$$\begin{aligned} \mathsf{arity}_{\vec{x}}(q_1 \cdot q_2) = \big\{ (\vec{c}, arit) \,\big|\, & (\vec{c}, arit_1) \in \mathsf{arity}_{\vec{x}}(q_1) \\ & \wedge (\vec{c}, arit_2) \in \mathsf{arity}_{\vec{x}}(q_2) \wedge arit = arit_1 \times arit_2 \big\} \end{aligned} \quad (18)$$

For q::-$q_1 + q_2$:

$$\begin{aligned} \mathsf{arity}_{\vec{x}}(q_1 + q_2) = & \{ (\vec{c}, arit) \,|\, ((\vec{c}, arit_1) \in \mathsf{arity}_{\vec{x}}(q_1) \vee (\vec{c}, arit_2) \in \mathsf{arity}_{\vec{x}}(q_2)) \\ & \wedge arit = arit_1 + arit_2 \} \end{aligned} \quad (19)$$

For q::-$t \leftarrow q$:

$$\begin{aligned} \mathsf{arity}_{\vec{x}}(v \leftarrow q) = \Big\{ (\vec{c}, arit) \Big| \vec{c} \in \mathsf{dom}_{\vec{x}}(q) \wedge \Big(\Big(\big(\forall i : (x_i = v) \\ \Rightarrow (c_i = q) \big) \wedge (arit = 1) \Big) \vee (arit = 0) \Big) \Big\} \end{aligned} \quad (20)$$

3 Domain computation

We are trying to compute the domain of variables that appear in queries. This computation is performed on the parse tree. We can traverse the tree in post order, first visiting the leaves that are represented by some relations and afterwards visiting the parent nodes and combining the relations of the children nodes. Using this technique we are trying to compute the domains, but also the vector $\vec{x}$ of all the variables defined in that query.

The intuition behind the algorithm is simple. For computing the domain of a node we first compute the domain of its left child and according to the operator of the node itself we decide how to send the information from left child to right child, and then the domain of the right child.

The algorithm needs as inputs the root of the tree and a structure that must be previously defined and returns a structure that contains the vector of variables($\vec{x}$) and the domains for each variable defined in the vector(dom). This structur is called *NodeAttribute* and it looks like:

```
struct {
x: the vector of all the variables
dom: the domains of all the variables
}
```

3.1: $NodeAttribute$

In Algorithm 1 we define function computeDomain which helps us to compute the domains for variables within a given query. When invoking the function, we pass as arguments of the function, the root of the parse tree and a structure which will have the vector of variables and the domain of the variables as nil, computeDomain$(root, s)$.

In Algorithm 1 we have a variable *node* which represents a node in the parse tree. It has 2 children which can be accessed by fields $left, right$. Also each node has a type field, which can be accessed by *type*. A type of a node can be any of different types in definition (1). For handling map types we suppose that each node that represents a map has a child (accessible via filed *child*, line 11) which points to the map itself (i.e. Figure 1). In other words each map introduces an intermediate node in the tree structure, this node has two fileds $\vec{x}, \vec{y}$ to represent input and output domain of the map.

Algorithm 1 computeDomains($node$,s)

Input: $node$ as the root of the tree to be traversed, and s is of type $NodeAttribute$

Output: $result$ of type $NodeAttribute$ that contains vector of variables $\vec{x}$ and the domains of each variable $\mathsf{dom}_{\vec{x}}(query)$

1: **if** $node.type =$"+" **then**
2: $s_1 \leftarrow$ computeDomain$(node.left, s)$
3: $s_2 \leftarrow$ computeDomain$(node.right, s)$
4: $result.\vec{x} \leftarrow s_1.\vec{x} \cup s_2.\vec{x}$
5: $result.dom \leftarrow \mathsf{dom}_{result.\vec{x}}(s_1.dom) \cup \mathsf{dom}_{result.\vec{x}}(s_2.dom)$
6: **else if** $node.type =$"*" **then**
7: $s_1 \leftarrow$ computeDomain$(node.left, s)$
8: $result \leftarrow$ computeDomain$(node.right, s_1)$
9: **else if** $node.type =$"Map" **then**
10: $node.\vec{x} \leftarrow \mathsf{dom}_{s.\vec{x}}(s.dom)$
11: $node.\vec{y} \leftarrow$computeDomain$(node.child, s)$
12: $result.\vec{x} \leftarrow s.\vec{x}$
13: $result.dom \leftarrow node.\vec{y}$
14: **else**
15: $result.\vec{x} \leftarrow s.\vec{x} \cup \{\text{Var}(node)\}$
16: $result.dom \leftarrow \mathsf{Ext}_{result.\vec{x}}(s.dom) \cap \mathsf{dom}_{result.\vec{x}}(node)$
17: **end if**
18: **return** $result$

The order of Algorithm 1 is $O(n \cdot P)$ where n is the number of nodes in the parse tree and P is the time needed for any rule of dom's computation, according to previous section. Clearly this algorithm visits each node only one time.

Algorithm 1 all types of nodes: union nodes, join nodes, map nodes and all other type of nodes. In addition to nodes, we have three types of leaves: simple relations $R(\vec{y})$, inequalities $v_1 \theta v_2$ and assignment relations $v \leftarrow q$. A union node or a join node will always have two children, therefore the line 14 will be executed when a leaf is visited. When a map node is encountered, line 9, then the algorithm should produce the domain of input and output variables. In line 16 we use function Ext to extend the schema of the input domain to a broader schema of $result.\vec{x}$. We use this for extending a domain regarding to a new schema vector $result.\vec{x}$.

In line 15 we want to compute the set of variables in the node. Since the node is a leaf, it can represents a relation, an assignment or a comparison. We can compute the set of variables in a leaf with the following rules:

$$\mathrm{Var}(R(\vec{y})) = \{y_1, \cdots, y_n\} \tag{21}$$
$$\mathrm{Var}(v_1 \theta v_2) = \{v_1, v_2\} \tag{22}$$
$$\mathrm{Var}(v \leftarrow q_1) = \{v\} \cup \mathrm{Var}(q_1) \tag{23}$$

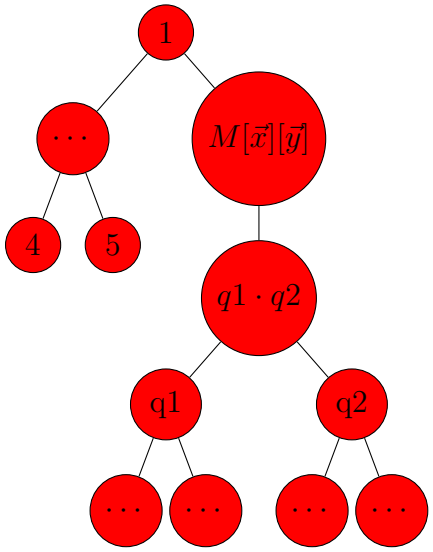

Figure 1: Tree representation

The algorithm starts with node 1, which represents the root of the parse tree. It recursively goes through each of the leaves, therefore constructs the variable vector and also the domain for each variable.

We have mentioned that besides map nodes, there are two more types of intermediate nodes: the union node and the join node. When a union node is met, then it is known that domains of variables will not pass over this operator and for that the variables and their domains, which were obtained in the parent, will be passed to each of this node's children. Thus it applies the distribution rule of the join operator over the union operator. For example if we have $q1 * (q2 * q3 + q4 * q5) \equiv q1 * q2 * q3 + q1 * q4 * q5$, the variables and

domain obtained for $q1$ will be passed both to $q2 * q3$ and $q3 * q4$, without modifying any of the domains, regarding that some domains may change on one branch, for example $q2 * q3$. For the other type of node, the join node, the passing of domains is allowed, and therefore a domain can be refined by the relation on the right side of the parent node.

Every time the algorithm encounters a map then it will create the domain for that specific map. Here we can make a difference between input and output variables, because the domain of input variables is dependent on the domains of output variables from the left side of the map's parent node, and the domain of the output variables are going to be produced by the children of the map's node.

A node which is represented by a map will have only one child, because as we mentioned in the definition of the AGCA expression, a map can be defined over an expression or expressions: $M[\vec{x}][\vec{y}]$::-q

Theorem 3.1. *Algorithm 1 computes the domain of each variables and maps.*

Proof. The proof is by induction on the height of the tree. The theorem is held for a tree consisting of only one node. Since this node is a simple relation and the Algorithm 1 goes through lines 14-16. As we defined, the domain of any empty set is infinite(all possible values) so the domain is correctly computed in line 16.

Now suppose that the Algorithm 1 works for all the trees of a height less than k. We want to prove that it will work as well for trees of height k. Let T be a tree with height k. If root represents a map, it has a child and we correctly computed its output domain, so the same is held for the map itself and the theorem follows in this case. Otherwise the root should have two children. The height of both of these children should be less than k. So we have correctly compute the domain of the left child and its input variable. Now we have two cases, if the root node is a union node or a join node. If this is a union node then we can compute the domain of the right child independently, since no information is passed over the "+" in DBToaster[1]. We compute the domain of the root according to definition (5). But if the root node is a join node, we have to pass the information from left child to the right child, since the right child may have some input variables which are defined in the left child. This is done in lines 6,7.

Thus in either of two cases we compute the domain of the root correctly and the theorem follows. □

4 Arity computation

We have defined some rules for the computation of the domains and the arities of each tuple. Having them, we can start writing an algorithm to compute the domains and the tuples' arities. We will start with the algorithm that will compute the domains and arities for each element. The algorithm will be similar to the algorithm that we have written for determining only the domains of variables that appear within a query.

As the algorithm presented for the computation of the domains, this one must have as well a structure which will help in the computation.

```
struct {
x: the vector of all the variables
arity: the domain and arities of all the variables
}
```

4.1: *nodeArityAttribute*

Algorithm 2 is similar to the Algorithm 1. The steps are the same, we traverse the parse tree, seeking all the nodes and test them to find out the type. The basic mechanism is the same when talking about reaching a node that will be a union node or a join node or even a map node. The difference will appear at the results presented, because in Algorithm 1 we will compute only domains of each tuple, whilst in Algorithm 2 we will compute the combination between the domains and the arity of each tuple inside the domains.

Algorithm 2 computeArities($node$,s)

Input: $node$ as the root of the tree to be traversed, and s which is of type $nodeArityAttribute$

Output: $result$ of type $nodeArityAttribute$ that contains vector of variables $\vec{x}$ and the domains of each variable with arities

1: **if** $node.type =$"+" **then**
2: $s_1 \leftarrow$ computeArities($node.left, s$)
3: $s_2 \leftarrow$ computeArities($node.right, s$)
4: $result.\vec{x} \leftarrow s_1.\vec{x} \cup s_2.\vec{x}$
5: $result.\mathsf{arity} \leftarrow \{(\vec{c}, arit) | (\vec{c_1}, arit_1) \in s_1.\mathsf{arity} \wedge (\vec{c_2}, arit_2) \in s_2.\mathsf{arity} \wedge (\vec{c_1} = \vec{c_2} \Rightarrow (\vec{c}, arit) = (\vec{c_1}, arit_1 + arit_2)) \wedge (\vec{c_1} \neq \vec{c_2} \Rightarrow (\vec{c}, arit) = (\vec{c_1}, arit_1) \vee (\vec{c}, arit) = (\vec{c_2}, arit_2))\}$
6: **else if** $node.type =$"*" **then**
7: $s_1 \leftarrow$ computeArities($node.left, s$)
8: $result \leftarrow$ computeArities($node.right, s_1$)
9: **else if** $node.type =$"Map" **then**
10: $node.\vec{x} \leftarrow s.\mathsf{arity}$
11: $s_1 \leftarrow$ computeArities($node.child, s$)
12: $node.\vec{y} \leftarrow s_1.\mathsf{arity}$
13: $result.\vec{x} \leftarrow s_1.\vec{x}$
14: $result.\mathsf{arity} \leftarrow node.\vec{y}$
15: **else**
16: $result.\vec{x} \leftarrow s.\vec{x} \cup \{\text{Var}(node)\}$
17: $aux \leftarrow extendAritiesToSchema_{result.\vec{x}}(s.\mathsf{arity})$
18: $result.\mathsf{arity} \leftarrow \{(\vec{c}, arit) | (\vec{c}, arit_1) \in aux \wedge (\vec{c}, arit_2) \in \mathsf{arity}_{result.\vec{x}}(node) \wedge (arit = arit_1 * arit_2)\}$
19: **end if**
20: **return** $result$

5 Maintaining the domains using arities

The main goal is to maintain the domains of the variables of each map that are presented in query expression. We give a solution of maintaining these domains, see how the domains will increase or decrease if tuples are being added or deleted from the relations.

Theorem 5.1. *The arity of a tuple in the domain of a simple relation will*

always be greater than 0.

Having this algorithm we can compute now the combination between the domain and the arities of the tuples that compose the domain. The algorithm should traverse the parse tree until it reaches the leaf that represents the relation that needs an update. Therefore a Depth First Algorithm can be used. First we shall talk about inserting a new tuple to an existing relation. When a leaf with the specified relation is encountered, we check if the tuple already exists in the set of arities. If the tuple exists, then we will retrieve the arity, and just for the fact that it exists in the domain of a relation the arity will be greater than 0. We increment the arity and repack everything and put it back in the set. If the tuple is not found in the domain of the relation that means that we will have a transition of the arity from $0 \rightarrow 1$, because we have to add the tuple to the existing domain, and therefore we have to recompute the domains for each and every map starting from that leaf.

When talking about making a deletion of a tuple inside a relation, most of the algorithm will remain the same. We should first find the leaf that represents the designated relation, find if the tuple is or is not inside the domain of the relation. If the tuple is not inside the domain then there appears an error because you can not delete something from the table of database without that tuple appearing inside the table. Therefore the tuple that is going to be deleted must appear in the domain of the relation. Having the group (domain,arity) we can test the arity. If the arity, after deleting the tuple, remains greater than 0 then the algorithm stops by making the modification to the arity, packing the results and putting it back into the set of the domain. Otherwise, if the arity drops to 0, then we will have a transition of $1 \rightarrow 0$ and thus we have to recompute each and every domain of the maps starting from the leaf that suffered a change.

To conclude the small description of the algorithm, we are interested only in transitions $0 \rightarrow 1$ or $1 \rightarrow 0$, because only then we will have to recompute every domain of the maps.

A sketch of the algorithm would look like as follows:

Algorithm 3 maintainDomains($node$,$(tuple, arity)$,$name_{relation}$)

Input: the root of the parse tree, the tuple that needs to be added with arity 0 (at first), the name of the relation

Output: the algorithm would not output nothing because it will produce the necessary changes along it goes

1:

5.1 Incrementally Maintaining For Insertion and Deletion

In this section we give an algorithm for computing the domains incrementally after each deletion or insertion. Let T be the parse tree as we had in the previous sections. We want to know how the domain of each node changes with any insertion or deletion from any relations. If the domain of any leaves of T changes, in the worst case we need to reevaluate domains of all the nodes in T. For an example, suppose the leftmost leaf changes and all other leaves depends on it, like $R(\vec{y}) \cdot \prod_i (y_i > a_i)$ where a_i are constants. Although in the worst case we need to reevaluate the domains of all nodes, but in some cases we can have some optimization in order to avoid recomputing the domains. We define function $\mathsf{InputVars}$ to a function that takes a node and returns its input variables. We can define a similar function $\mathsf{OutputVars}$ for output variables.

$$\mathsf{InputVars}(R(\vec{y})) = \{\} \tag{24}$$
$$\mathsf{InputVars}(v_1 \theta v_2) = \{v_1, v_2\} \tag{25}$$
$$\mathsf{InputVars}(q_1 + q_2) = \mathsf{InputVars}(q_1) \cup \mathsf{InputVars}(q_2) \tag{26}$$
$$\mathsf{InputVars}(q_1 \cdot q_2) = \mathsf{InputVars}(q_1) \cup \mathsf{InputVars}(q_2) - \mathsf{OutputVars}(q_1) \tag{27}$$
$$\mathsf{InputVars}(v \leftarrow q_1) = \mathsf{InputVars}(q_1) \tag{28}$$
$$\mathsf{OutputVars}(R(\vec{y})) = \vec{y} \tag{29}$$
$$\mathsf{OutputVars}(v_1 \theta v_2) = \{\} \tag{30}$$
$$\mathsf{OutputVars}(q_1 + q_2) = \mathsf{OutputVars}(q_1) \cap \mathsf{OutputVars}(q_2) \tag{31}$$
$$\mathsf{OutputVars}(q_1 \cdot q_2) = \mathsf{OutputVars}(q_1) \cup \mathsf{OutputVars}(q_2) \tag{32}$$
$$\mathsf{OutputVars}(v \leftarrow q_1) = \{v\} \tag{33}$$

According to the above definitions we can give an algorithm for computing the changes to each map for any insertion or deletionXXXX. For our purpose

we need a way to show the set of relations contains in each node, we show this by Rel and define it as follow:

$$\mathsf{Rel}(R(\vec{y})) = R \tag{34}$$
$$\mathsf{Rel}(v_1 \theta v_2) = \{\} \tag{35}$$
$$\mathsf{Rel}(q_1 + q_2) = \mathsf{Rel}(q_1) \cup \mathsf{Rel}(q_2) \tag{36}$$
$$\mathsf{Rel}(q_1 \cdot q_2) = \mathsf{Rel}(q_1) \cup \mathsf{Rel}(q_2) \tag{37}$$
$$\mathsf{Rel}(v \leftarrow q_1) = \mathsf{Rel}(q_1) \tag{38}$$

Algorithm 4 CheckRightNode($node, v_1, R, s$)

Input: *node* as the root of the tree to be traversed, v_1 a flag indicating if the left subtree has changed, R the relation to be modified and s as the structure
Output: A pair whose first element is a boolean to indicate if any domains in the subtree rooted at *node.right* has changed and the second element is the $\mathsf{dom}(node.right)$

1: **if** $R \in \mathsf{Rel}(node.right)$ **or** ($v_1 =$ **true and** $\mathsf{OutputVars}(node.left) \cap \mathsf{InputVars}(node.right) \neq \emptyset$) **then**
2: **return** MaintainDomains($node.right, R, s$)
3: **else**
4: **return** (**false**, $node.right.s$)
5: **end if**

The intuition of Algorithm 5 is simple. Suppose we want to modify relation R(insertion or deletion). In each node we first check whether the left subtree contains R or not. If so, we continue recursively in the left subtree(lines 1-5). We run the algorithm on the right subtree if and only if it contains R or the left subtree has changed due to the modification and the right subtree has some input variables which depend on the output variables of the left subtree. Deciding whether or not to run the algorithm on the right subtree is preformed by Algorithm 4. According to the type of nodes, as we had in Algorithm 1, we compute the domain of *node*. Here we suppose that each node has a left and right child as before and structure s as defined in 4.1. We suppose that before any executions of Algorithm 5, Algorithm 1 has been executed at least one time. So for each *node* we have associated a structure s. Thus with s we can access to the domain(dom) of each node and the set of its variables($\vec{x}$).

Algorithm 5 MaintainDomains($node, R, s$)

Input: *node* as the root of the tree to be traversed, R the relation to be modified and s as the structure

Output: A pair whose first element is a boolean to indicate if any domains in the subtree rooted at *node* has changed and the second element is the $\mathsf{dom}(node)$

1: **if** $R \in$Rel(node.left) **then**
2: $(v_1, s_1) \leftarrow$ MaintainDomains($node.left, R, s$)
3: **else**
4: $(v_1, s_1) \leftarrow$ (**false**, $node.left.s$)
5: **end if**
6: **if** $node.type =$"+" **then**
7: $(v_2, s_2) \leftarrow$CheckRightNode($node, v_1, s$)
8: $node.s.dom \leftarrow \mathsf{dom}_{node.s.\vec{x}}(s_1.dom) \cup \mathsf{dom}_{node.s.\vec{x}}(s_2.dom)$
9: **else if** $node.type =$"*" **then**
10: $(v_2, node.s.dom) \leftarrow$CheckRightNode($node, v_1, s_1$)
11: **else if** $node.type =$"Relation" **then**
12: Apply changes to the relation if it is $R(\vec{y})$ and set v_2 **true** if the domain changed
13: $node.s.dom \leftarrow \mathsf{dom}_{node.s.\vec{x}}(node)$
14: **else if** $node.type =$"Map" **then**
15: $node.\vec{x} \leftarrow \mathsf{dom}_{s.\vec{x}}(s.dom)$
16: $(v_2, node.\vec{y}) \leftarrow$MaintainDomains($node.child, R, s$)
17: $result.\vec{x} \leftarrow s.\vec{x}$
18: $result.dom \leftarrow node.\vec{y}$
19: **else**
20: $node.s.dom \leftarrow \mathsf{dom}_{node.s.\vec{x}}(s.dom) \cap \mathsf{dom}_{node.s.\vec{x}}(node)$
21: $v_2 \leftarrow$ **false**
22: **end if**
23: **return** $(v_1 \textbf{or}\, v_2, result)$

References

[1] C. Koch, *Incremental Query Evaluation in a Ring of Databases*, preprint (2011).

[2] O. Kennedy, Y. Ahmad, C. Koch. *DBToaster: Agile views for a dynamic data management system*. In CIDR, 2011.