Initial Value Computation

August 4, 2011

1 Definitions

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 $\mathsf{dom}_{\vec{x}}(R(\vec{y}))$:

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

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),(7) and (9) $\vec{c}$ 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. For the sake of completeness we define the domain of empty set as infinite:

$$\mathsf{dom}_{\vec{x}}(\text{NULL}) = \{ \vec{c} \} \tag{9}$$

2 Computing the arities of the AGCA expressions

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

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

Table 3: Relation $R \cdot S$

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

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 *arity* 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.

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

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.

$$arity(q_1 \cdot q_2, t) = \sum_{\{t_1\} \bowtie \{t_2\} = t} arity(q_1, t_1) * arity(q_2, t_2) \tag{10}$$

$$arity(q_1 + q_2, t) = arity(q_1, t) + arity(q_2, t), \text{ where Schema}(q_1) \tag{11}$$

$$= \text{Schema}(q_2) \tag{12}$$

$$arity(v_1\ \theta\ v_2, t) = \begin{cases} 1, & \text{if } v_1 \theta v_2 \text{ is true and } v_1 \in t \wedge v_2 \in t \\ 0, & \text{otherwise is false} \end{cases} \tag{13}$$

$$arity(v \leftarrow q, t) = \begin{cases} 0, & \text{if } t \setminus \{v\} \notin \mathsf{dom}_{Schema(t)}(q) \\ 1, & \text{otherwise} \end{cases} \tag{14}$$

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). The structure will look like:

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

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). Each node has boolean field($isMap$) which indicates if it is representing a map or not. For representing the maps we need another global structure for storing the properties of each map. Here we use $Maps$ as an associated array whose keys are the nodes and the values are the domain of the input parameters to the map(accessible via field $\vec{x}$) and the domain of its output variables(accessible via $\vec{y}$).

Algorithm 1 computeDomains($node$,s)

Input: $node$ as the root of the tree to be traversed, and s as the structure
Output: a structure $result$ that will contain the 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: $result.\vec{x} \leftarrow s_1.\vec{x} \cup s_2.\vec{x}$ // this will compute the vector for all the variables, both from the right and left node
5: $result.dom \leftarrow \mathsf{dom}_{result.\vec{x}}(node.left + node.right)$
6: **else if** $node.type =$"*" **then**
7: $s_1 \leftarrow$ computeDomain$(node.left, s)$
8: $result \leftarrow$ computeDomain$(node.right, s_1)$
9: **else**
10: $result.\vec{x} \leftarrow s.\vec{x} \cup \{$all the variables of the $node\}$
11: $result.dom \leftarrow \mathsf{dom}_{result.\vec{x}}(s.dom) \cap \mathsf{dom}_{result.\vec{x}}(node)$
12: **end if**
13: **if** $node.isMap =$ **true then**
14: $Maps[node].\vec{x} \leftarrow \mathsf{dom}_{\vec{x}}(s.dom)$
15: $Maps[node].\vec{y} \leftarrow result$
16: **end if**
17: **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.

In Algorithm 1 we suppose that there are three types of nodes: union nodes that represent $q + q$ expressions, join nodes that represent $q * q$ expressions and the other remaining node types. If a node represents a map we show it like $M[\vec{x}][\vec{y}]$. 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 9 will be executed when a leaf is visited. When a node that is represented by a map, is encountered then the algorithm should produce a set of variables and respectively the domains of each variable. Also in line 11 we use the definition of dom to extend its schema to a broader range according to what

was said in the previous section, definition (2).

In line 11 we take the domain of a domain. We use this for extending the domain regarding to a new schema vector $\vec{x}$. In other words we add some other variables to the vector and therefore compute the domain for each variable within that vector. Without loss of generality, we can consider a domain as a set with schema or a relation. So we can easily extend the domain of a set to a broader set of variables with the definition of the domain or a relation in (2).

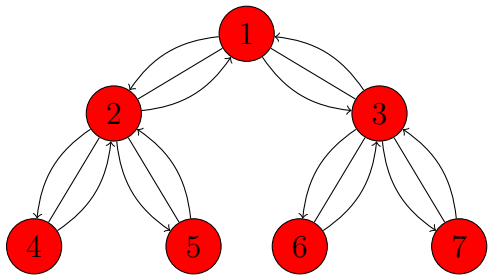

Figure 1: Tree traversal

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.

It is known that the left most leaf will always be a relation that produces only output variables. Otherwise, if this condition is not sustained then it will contradict the rules imposed by the DBToaster calculus. Therefore node 4 will always be a simple relation that will give some output variables, which may help in future nodes if those nodes have input variables. If, for example node 5 will have input variables, then those variables will depend only on the output variables that node 4 is giving.

We have mentioned that besides map nodes, there are two more types of 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 applying the distribution rule of 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. All this information will be stored in a global variable which can be easily accessed.

Theorem 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 9-11. As we defined, the domain of any empty set is infinite(all possible values) so the domain is correctly computed in line 11.

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. The root should have two children since the construction of the tree only creates nodes with leaves or two children [1]. 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 Computing and maintaining the domains using the arities

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

References

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