Map Initializations

August 17, 2011

1 Introduction

In this section we would want to discuss the matter of map initialization. From [1] we know that the compilation algorithm takes an aggregate query and defines a map for it, which represents the materialized view of the query. The algorithm creates a trigger for each possible update on an event, that will specify how to update the main query. To this operation, the algorithm computes the delta query, and creates a new map that will represent the materialized view of the delta. For example, if we have the following query q:

$$\text{SELECT sum}(a \cdot c) \text{ FROM R(a,b), S(b,c) WHERE } R.b = S.b \qquad (1)$$

The compilation algorithm will take this query and replaces it with a map $q[][]$ and computes its deltas. We will have two events: onR, onS. For simplicity we will take for now only the inserting operation, the trigger code will look like, :

$$\begin{aligned} &+onR(a,b): \\ &\qquad q[][] += a * m_R[][b] \\ &\qquad m_S[][b] += a \end{aligned}$$

$$\begin{aligned} &+onS(b,c): \\ &\qquad q[][] += c * m_S[][b] \\ &\qquad m_R[][b] += c \end{aligned}$$

In [1] a map is defined by as function which takes input values and produces output values. With this definition, we can consider the initialization as a process of computing the function's output values for some new input values without computing the function body.

2 Definitions

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:

$$q ::\text{-}\ q \cdot q | q + q | v \leftarrow q | v_1 \theta v_2 | R(\vec{y}) | \text{c} | \text{v} | (M[\vec{x}][\vec{y}] ::\text{-} q) \tag{2}$$

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

$$\mathsf{dom}_{\vec{x}}(R(\vec{y})) = \left\{ \vec{c} \,\middle|\, \sigma_{\forall x_i \in (\vec{y}) : x_i = c_i} R(\vec{y}) \neq \text{NULL} \right\} \tag{3}$$

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:

$$\mathsf{dom}_{\vec{x}}(v_1 \theta v_2) = \left\{ \vec{c} \,\middle|\, \forall i, j : (v_1 = x_i \wedge v_2 = x_j) \Rightarrow c_i \theta c_j \right\} \tag{4}$$

The domain of a comparison is infinite.

For the join operator we can write:

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

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

$$\mathsf{dom}_{\vec{x}}(q_1 + q_2) = \{\vec{c} | \vec{c} \in \mathsf{dom}_{\vec{x}}(q_1) \vee \vec{c} \in \mathsf{dom}_{\vec{x}}(q_2)\} \tag{6}$$

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

$$\mathsf{dom}_{\vec{x}}(variable) = \left\{ \vec{c} \right\} \tag{8}$$

In (7) and (8) $\vec{c}$ 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:

$$\mathsf{dom}_{\vec{x}}(v \leftarrow q_1) = \left\{ \vec{c} \,\middle|\, \vec{c} \in \mathsf{dom}_{\vec{x}}(q_1) \wedge \left(\forall i, j : (x_i = v = x_j) \Rightarrow (c_i = c_j = q_1) \right) \right\} \tag{9}$$

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:

$$\mathsf{dom}_{\vec{w}}(map[\vec{x}][\vec{y}]) = \{\vec{c} | \vec{c} \in \mathsf{dom}_{\vec{w}}(\vec{x} \cup \vec{y})\} \tag{10}$$

As presented in the papers [1] and [2], maps are functions that are defined on a set of values and that will produce a result for each value of that set. The set of values will represent the domains, which were computed using the definitions presented so far. We can make a distinction between a complete map and an incomplete map. A complete map will be characterized by the fact that each value of its domain will have assigned a result, whilst

an incomplete map will be a map that will not have all the values of the domain and therefore neither the result for those values, on each insertion the incomplete map must compute the exact value of the tuple added to the domain.

In other words we can consider a complete map as a total function and an incomplete one as a partial function. A total function is a function that assigns a value to every element of its domain. But a partial function has some elements in its domain which have not been assigned to any value in its codomain.

According to 2 we can express every expression of AGCA in a parse tree. The root of parse tree represents the whole expression and its leaves are relations or comparisons. Each node can be regarded as a map and thus it has a domain. Any modification to the relations, the leaves are modified and this modification should be propagated upward through the parse tree. During the propagation process the domains of intermediate nodes may be changed. In Theorem 2.1 we prove that if all maps in the parse tree are complete, then with any modification the maps will remain complete as well.

Theorem 2.1. *A complete map will remain complete, after provoking a modification to the underlying expression of the map.*

Proof. We will give a proof based on induction.

In the first step we presume that the map will be defined over a simple relation:

$$m[][\vec{y}] :: R(\vec{y})$$

The vector $\vec{y}$ from the output variables of the maps will correspond to the $\vec{y}$ from the simple relation. Therefore there will be a direct connection between the domain of the map and the values offered by the relation. Therefore we will have:

$$\mathsf{dom}_{\vec{x}}(m[][\vec{y}]) = \mathsf{dom}_{\vec{x}}(R(\vec{y}))$$

If we add tuple $<y_1, y_2, \cdots, y_n>$ to the relation R, we are going to have to situations:

1. $<y_1, y_2, \cdots, y_n> \in \mathsf{dom}_{\vec{x}}(R(\vec{y}))$ and therefore the map remains complete
2. $<y_1, y_2, \cdots, y_n> \notin \mathsf{dom}_{\vec{x}}(R(\vec{y}))$ and therefore the tuple should be added to domain of the relation, however due to the fact that there is a direct

connection between the domain of the relation and the domain of the map, the domain of the map will change and the map will remain complete

Secondly if we have a map defined on a join expression:

$$m[][\vec{y1}] :: R(\vec{y2}) \cdot q$$

The domain of the map will be determined by the domain of the join expression:

$$\mathsf{dom}_{\vec{x}}(m[][\vec{y1}]) = \mathsf{dom}_{\vec{x}}(R(\vec{y2}) \cdot q)$$

and therefore using the definition for the join expression the map will be a complete map.

When adding a new tuple into relation R we will have exactly the same possibilities as in the previous case, however in the second case there must be added something: the domain of the map will be influenced by the join operator, therefore taking only the common tuples between $R(\vec{y})$ and q. If the tuple that we are adding is also in the domain of expression q, then it will be added to the maps domain, otherwise no. However in both cases the map completion will stand.

Thirdly if we have a map defined on a union expression:

$$m[][\vec{y1}] :: R(\vec{y2}) + q$$

Exactly like in the previous case the domain of the map will be given by the union relation:

$$\mathsf{dom}_{\vec{x}}(m[][\vec{y1}]) = \mathsf{dom}_{\vec{x}}(R(\vec{y2}) + q)$$

When talking about union expressions,we will not have communication over between this operator, furthermore the domain of a such expression will be the union of the two domains. Therefore if we change add a tuple $<y_1, y_2, \cdots, y_n> \in \mathsf{dom}_{\vec{x}}(R(\vec{y}))$ to relation R, the cases from the first step will remain, regardless if the tuple is or not in the expression q.

Finally if we have a map defined on a join or union expression

$$m[][\vec{y1}] :: q1 \alpha q$$

where $\alpha = \{\cdot, +\}$, the relation will hold for q1 and for q2 due to the induction steps and therefore it will also hold for the $q1 \alpha q2$. □

3 Equijoins

We will start talking about map initialization in the simplest of cases: queries represented by relations that are joined only by equalities.

$$\text{SELECT sum}(\cdots) \text{ FROM } R_1, R_2, \cdots, R_n \text{ WHERE } R_i.a = R_j.a \\ (\forall i, j \in \{1..n\} \wedge i \neq j)$$

When having only equality joins, then the maps defined over expressions consisting of simple relations will have only output variables. Every variable will be bounded to the relations and therefore their domains will depend on the values provided from the relations. If we have

$$\text{SELECT } \cdots \text{ FROM } R_1, R_2, \cdots, R_n \text{ WHERE } R_i.a = R_j.a$$

the compiler will assign a map to the main query $q[][]$ and each delta query will be replaced by the materialized view. If we have Δ_{R_1}, the relation R_1 will disappear from the main query and the new expression will be replace by a map $m_{R_1}[][x_i, \cdots, x_j]$, where $x_i, \cdots, x_j$ represent the communication variables between R_1 and the rest of the expression.

The map $m_{R_1}[][x_i, \cdots, x_j]$ can also be defined using a simpler query with only $n-1$ relations:

$$\begin{aligned} &\text{SELECT} \cdots \text{ FROM } R_2, \cdots, R_n \\ &\quad \text{WHERE } R_l.a = R_m.a \\ &\quad (\forall l, m \in \{2 \cdots, n\} \wedge l \neq m) \\ &\quad \text{GROUP BY } x_i, \cdots, x_j \end{aligned}$$

In a general case when computing the Δ_{R_k}, we are going to have a map assigned to the delta. $m_{R_k}[][x_i, \cdots, x_j]$, where $x_i, \cdots, x_j$ will be exactly as mentioned up, the variables that will have to be replaced by values when an event onR_k will appear. The map can easily be compared with a query, which will be simpler and will also have a group by clause.

$$\begin{aligned} &\text{SELECT} \cdots \text{ FROM } R_1, R_2, \cdots, R_{k-1}, R_{k+1}, \cdots, R_n \\ &\quad \text{WHERE } R_l.a = R_m.a \\ &\quad (\forall l, m \in \{1 \cdots n\} - \{k\} \wedge l \neq m) \\ &\quad \text{GROUP BY } x_i, \cdots, x_j \end{aligned}$$

Theorem 3.1. *The initial value of a map for a specific tuple, which is not in the domain of the map, will always be 0.*

Proof. If the tuple that has been added is in the complete domain then the map has already been initialized.

We will apply reduction to the absurd to prove that whenever we add a new tuple that is not in the domain of a map, then that map should be initialized by 0.

Relation R_k will have the following schema: $Sch(R_k) = x_1, x_2, x_3, \cdots, x_n$, but only some variables are going to be used for the communication with the other relations: $x_i, \cdots, x_j$. We will have the trigger $+onR_k(x_1, \cdots, x_n)$, and when such an event appears we will test if the variables $x_i, \cdots, x_j$ from the arguments of $+onR$ are in the domain of the map or not.

We assume that the tuple is not in the domain of the map m_{R_k} and the initial value for the map regarding to that tuple will be different from 0.

$$m_{R_k}[][x_i, \cdots, x_j] \neq 0$$

If this is true then the query that can be generated for the map m_{R_k}:

$$\begin{aligned} &\text{SELECT} \cdots \text{FROM } R_1, R_2, \cdots, R_{k-1}, R_{k+1}, \cdots, R_n \\ &\quad \text{WHERE } R_l.a = R_m.a \\ &\quad (\forall l, m \in \{1 \cdots n\} - \{k\} \wedge l \neq m) \\ &\quad \text{GROUP BY } x_i, \cdots, x_j \end{aligned}$$

will produce a table, that will have a record with the specified tuple, therefore the tuple will be in the domain of the map, $\{x_i, \cdots, x_j \in domain(m_{R_k})\}$. This contradicts the sentence we assumed at first. Furthermore, when invoking the query for the given values the result of the query will not be $NULL$, and therefore contradicting the fact the that tuple is not defined in the table and the query result should be $NULL$. And therefore any time, a new tuple that is not in the domain will only provoke a zero initialization of the map for that tuple. □

The theorem will stand only if we are talking about having a complete domain. Otherwise, if the domain is incomplete, then for every tuple that we add with a trigger, we would have to check if that tuple exists in the table (the complete domain). If the domain is incomplete, then we could have a

different value for the initial value of a map, this value would be produced from the tables performing a query, even a simpler one.

Another problem of initial value computation, besides the problem of with which value should a map be initialized, is the problem of how fast to do the initialization. We have two different sort of initializations: an eager one and a lazy one. The eager one will initialize the right side of a trigger expression, when the left side of the expression will be initialized. The right side will be initialized if and only if it needs initialization. And the lazy one is based on the fact that only the left side will be initialized, and the right side no, leaving the rest side of the initialization to be done when the appropriate trigger is called.

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.