

Map Initializations

August 16, 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 :

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

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, :

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

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

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. The main goal of this section is to study how two maps $m_R[b]$ and $m_S[b]$ will be initialized when an insertion is triggered either from $+onR(a, b)$ and also $+onS(b, c)$.

2 Definitions

DBToaster uses query language AGCA(which is stands for AGgregation CAculus). AGCA expressions are built from constants, variables, relational atoms, aggregate sums (Sum), conditions, and variable assignments (\leftarrow) using “+” and “.”. The abstract syntax can be given by the EBNF:

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

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 ($=, \neq, >, \geq, <, \text{ and } \leq$). “+” represents unions and “.” 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 $\text{dom}_{\vec{x}}(q)$ 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 $\vec{x} = \langle x_1, x_2, x_3, \dots, x_n \rangle$ will be the schema of all the variables and that $\vec{c} = \langle c_1, c_2, c_3, \dots, c_n \rangle$ 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 $\text{dom}_{\vec{x}}(R(\vec{y}))$:

$$\text{dom}_{\vec{x}}(R(\vec{y})) = \left\{ \vec{c} \mid \sigma_{\forall x_i \in (\vec{y}) : x_i = c_i} R(\vec{y}) \neq \text{NULL} \right\} \quad (3)$$

Thus, we can evaluate $\text{dom}_{\vec{x}}$ 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:

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

The domain of a comparison is infinite.

For the join operator we can write:

$$\text{dom}_{\vec{x}}(q_1 \cdot q_2) = \{ \vec{c} \mid \vec{c} \in \text{dom}_{\vec{x}}(q_1) \wedge \vec{c} \in \text{dom}_{\vec{x}}(q_2) \} \quad (5)$$

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

$$\text{dom}_{\vec{x}}(q_1 + q_2) = \{ \vec{c} \mid \vec{c} \in \text{dom}_{\vec{x}}(q_1) \vee \vec{c} \in \text{dom}_{\vec{x}}(q_2) \} \quad (6)$$

$$\text{dom}_{\vec{x}}(\text{constant}) = \{ \vec{c} \} \quad (7)$$

$$\text{dom}_{\vec{x}}(\text{variable}) = \{ \vec{c} \} \quad (8)$$

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:

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

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:

$$\text{dom}_{\vec{w}}(\text{map}[\vec{x}][\vec{y}]) = \{ \vec{c} \mid \vec{c} \in \text{dom}_{\vec{w}}(\vec{x} \cup \vec{y}) \} \quad (10)$$

We can say that domains can be grouped into two categories, Complete and Incomplete. These definitions form the complete domains.

We show the incomplete domains with $D_{incomplete}$ and complete domains by $D_{complete}$ and we can say that every time a value is added to a complete

domain, it will remain complete. Incomplete domains are subsets of complete domains. In other words some elements are not covered in the incomplete domains. If we modify the relations, the domains of all expressions will be changed. If we follow the definitions as we had in this section we will have the complete domains again. Thus, the complete domains are closed under the modifications of the relations. Complete domains will always have a straight forward connection with the underlying relations.

Theorem 2.1. *If we have a complete domain of a map it will remain complete with any modifications of its underlying relations.*

Proof. Suppose we have a map m and its complete domain as D . If we modify a relation, there are two cases:

1. This modification does not modify the domain of m . So The domain of m remains complete after the modification.
2. The modification does change the domain of the relation and it will propagate upward on the parse tree, therefore provoking a change to the domain of the map. The change done to a relation will update all the other relations before reaching the domain of the map and therefore maintaining the completeness of each domain.

□

When talking about incomplete domains, the theorem presented for the complete domains will not hold. Incomplete domains will not have a direct mapping to all the values from the relations. We will not want to use the entire domain, we want to create it as we go. This will imply that every time a new value is added to an incomplete domain, then that value should be tested if it appears in the complete domain, that we did not want to compute. We will have the following relation between complete and incomplete domains of the same variable:

$$D_{incomplete} \subset D_{complete}$$

Every time a new variable x is added to the incomplete domain we are going to have three situations:

1. $x \in D_{incomplete}$
2. $(x \notin D_{incomplete}) \wedge (x \in D_{complete})$

$$3. (x \notin D_{incomplete}) \wedge (x \notin D_{complete})$$

The first case is exactly as with complete domains, therefore the new value is in the domain and nothing will be changed, the second case means that the new value is not in the incomplete domain however there is such a value in its underlying relations, and therefore computation is necessary for determining the correct value for the newly added one, and in the third case the new value is quite new and therefore all the initialization for that one should be zero because nothing is dependent on it.

After giving these definitions, we can say that we can easily construct a complete domain starting from an incomplete domain, by adding values to the incomplete one.

3 Equijoins

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

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

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

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

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 replaced by a map $m_{R_1}[\] [x_i, \dots, x_j]$, where x_i, \dots, x_j represent the communication variables between R_1 and the rest of the expression.

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

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

In a general case when computing the Δ_{R_k} , we are going to have a map assigned to the delta. $m_{R_k}[[x_i, \dots, x_j]]$, where x_i, \dots, 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.

```

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

```

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: $Sch(R_k) = x_1, x_2, x_3, \dots, x_n$, but only some variables are going to be used for the communication with the other relations: x_i, \dots, x_j . We will have the trigger $+onR_k(x_1, \dots, x_n)$, and when such an event appears we will test if the variables x_i, \dots, 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.

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

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

```

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

```

will produce a table, that will have a record with the specified tuple, therefore the tuple will be in the domain of the map, $\{x_i, \dots, x_j \in domain(m_{R_k})\}$.

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 that 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. \square

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.