Map Initializations

August 24, 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:

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

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.

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 "·". The abstract syntax can be given by the EBNF:

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

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

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

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:

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

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.

We can express every expression of AGCA in a parse tree with EBNF 2. 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.

3 Equijoins

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

$$\begin{aligned} \text{SELECT sum}(\cdots)\ \text{FROM}\ R_1, R_2, \cdots, R_n\ \text{WHERE}\ R_i.x_{ik} = R_j.x_{jt} \\ (\forall i, j \in \{1..n\} \wedge i \neq j \wedge (x_{ik} \in Sch(R_i)) \\ \wedge (x_{jt} \in Sch(R_j)) \end{aligned} \tag{11}$$

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 by these relations.

For a the given query 11, the compilation algorithm will replace it with a map $q[][]$. When computing the delta regarding to a relation R_k, 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.

$$\begin{aligned}\text{SELECT} \cdots \text{ FROM } & R_1, R_2, \cdots, R_{k-1}, R_{k+1}, \cdots, R_n \\ & \text{WHERE } R_l.x_{lt} = R_m.x_{ms} \\ & (\forall l, m \in \{1 \cdots n\} - \{k\} \wedge l \neq m) \\ & \text{GROUP BY } x_i, \cdots, x_j\end{aligned} \tag{12}$$

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}, exactly like 12, 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.

Taking into account the definitions of the AGCA expressions in 2, the following theorem will work with a subset of those definitions, namely with:

$$q ::\text{-}\ q \cdot q | q + q | R(\vec{y}) | (M[\vec{x}][\vec{y}] ::\text{-} q)$$

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}:

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

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.

Theorem 1. *The value of a map for a specific tuple, which is not in the domain of the map, will always be 0.*

$$\vec{y} \notin \mathsf{dom}_{\vec{x}}(q) \Rightarrow m_q[][\vec{y}] = 0$$

Proof. We will give a proof based on induction on the parse tree.

We presume that the map will be defined over a simple relation: $m_R[][\vec{y}] :: R(\vec{y})$. The vector $\vec{y}$ from the output variables of the maps will correspond to the $\vec{y}$ from the simple relation. Therefore the following statement is true: $\mathsf{dom}_{\vec{x}}(m_R[][\vec{y}]) = \mathsf{dom}_{\vec{x}}(R(\vec{y}))$.

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

1. $\vec{y_1} \in \mathsf{dom}_{\vec{x}}(R(\vec{y}))$, therefore the domain of the relation will remain the same and the map is already instantiated

2. $\vec{y_1} \notin \mathsf{dom}_{\vec{x}}(R(\vec{y}))$. If the tuple is not in the domain then:

   $$\mathsf{dom}_{\vec{x}}(R(\vec{y})) \cup = \vec{y_1} \text{ and } m_R[][\vec{y_1}] = 0$$

   because the order of multiplicity of that tuple in R was zero and thus any operation (sum, count) will produce a NULL result.

$\vec{y_1} \notin \mathsf{dom}_{\vec{x}}(R(\vec{y})) \Rightarrow m_R[][\vec{y_1}] = 0$ and therefore the base case is true.

First we will talk about join relations and maps defined over a join of expressions: $m_q[][\vec{y}] :: q_1 \cdot q_2$. Definition 5 says that the domain of a join expression is the intersection between the domains of q_1 and q_2. If $\vec{y_1} \notin \mathsf{dom}_{\vec{x}}(q_1 \cdot q_2)$ then we will have the following cases:

1. $\vec{y_1} \notin \mathsf{dom}_{\vec{x}}(q_1)$ which means that the map defined over the relation q_1 will be 0 for that tuple.

   $$m_{q_1}[][\vec{y_1}] = 0 \wedge m_q[][\vec{y_1}] : m_{q_1}[][\vec{y_1}] * m_{q_2}[][\vec{y_1}] \Rightarrow m_q[][\vec{y_1}] = 0 * m_{q_2}[][\vec{y_1}] = 0$$

   where $m_{q_1}[][\vec{y_1}] :: q_1$ and $m_{q_2}[][\vec{y_1}] :: q_2$.

2. $\vec{y} \notin \mathsf{dom}_{\vec{x}}(q_2)$ the same proof as in the first case, but now for the expression q_2.

Secondly we will talk about union expressions and maps defined over union expressions: $m_q[][\vec{y}] :: q_1 + q_2$. Definition 6 says that the domain of a union expression is the union between the domains of q_1 and q_2. Therefore, if $\vec{y} \notin \mathsf{dom}_{\vec{x}}(q_1 + q_2)$ then: $\vec{y} \notin \mathsf{dom}_{\vec{x}}(q_1) \wedge \vec{y} \notin \mathsf{dom}_{\vec{x}}(q_2)$.

$$m_{q_1}[][\vec{y_1}] = 0 \wedge m_{q_2}[][\vec{y_1}] = 0 \wedge m_q[][\vec{y_1}] : m_{q_1}[][\vec{y_1}] + m_{q_2}[][\vec{y_1}] \Rightarrow m_q[][\vec{y_1}] = 0$$

where $m_{q_1}[][\vec{y_1}] :: q_1$ and $m_{q_2}[][\vec{y_1}] :: q_2$. $\square$

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.

4 Simple inequalities

Joins between relations can be easily made also by using inequalities between the variables of those relations. For example, if we have the following query:

$$\begin{aligned} \text{SELECT } & sum(a * d) \\ \text{FROM } & R(a,b), S(c,d) \\ \text{WHERE } & b<c \end{aligned}$$

the result will depend on the evaluation of the inequality $b<c$, where b comes from relation R and c comes from relation S.

The delta regarding to the relation R will be:

$$\begin{aligned} a * \text{SELECT } & sum(d) \\ \text{FROM } & S(c,d) \\ \text{WHERE } & b<c \end{aligned}$$

where the new query will be replaced by a map which will have an input variable $m_R[b][]$. The domain of the map be will given by the values offered by relation R, however the result of the map will be influenced by the value of c from the relation S.

The initial value of the map $m_R[b][]$ will be influenced by the relation S. Therefore we will have the following situations:

1. the relation S is empty and therefore no value of c can be produced and thus the initialization of the map $m_R[b][]$ will always be 0, because b cannot be compared with any value of c

2. if relation S is not empty then every update to relation R will need a check with every value c from S. Therefore we can say that the initial value of map $m_R[b][] = \sum_{\forall c>b} S(c,d) * d$

When relation S is not empty, the initialization of map $m_R[b][]$ can easily be maintained incrementally, because knowing a value of the map for a specific b, the other one can be easily deduced. We will have $sum = m_R[b_1][]$, where b_1 is a value that we had to compute the result of the map from scratch. When adding a value b_2 to the relation R, we will have the following cases:

1. $b_1 > b_2$ then $m_R[b_2][] = sum + \sum_{\forall \text{c where } b_2<c\leq b_1} S(c,d) * d$

2. $b_1 < b_2$ then $m_R[b_2][] = sum - \sum_{\forall \text{c where } b_1\leq c<b_2} S(c,d) * d$

Starting from this example we will offer a generalization and a proof that the initial value of maps, when talking about joins done by inequalities, are going to be exactly like in the example.

5 A method for inequalities

In this section we are going to propose an algorithm for computing the initial value of a map for an input value which is not present in the domain. What we have presented in Theorem 1 is not true any more for queries with inequalities. As an example XXX

We will introduce the notion of graph dependency between the variables and afterwards present the algorithm for the value computation. Suppose we are given a map $m[\cdot][\cdot]$ and for it we want to know the value of $m[\vec{x'}][\cdot]$. In other words, we want to know the initial value of a new input value $\vec{x'}$ for the specified map. In any expression which is represented by a map we will have 3 types of variables: input variables(IV), output variables(OV) and intermediate variables(ITO). The intermediate variables are the variables which are not input nor output variables. We will not have any comparisons between input and output variables, except for equality. Thus, in this section we don't consider the equality operator as a comparison operator. We use x_i, y_j to show the input and intermediate variables. Now, suppose we have a comparison operator $v_1 \theta v_2$. There are 3 cases for v_1, v_2:

- $v_1 = x_i, v_2 = x_j$, in other words, both of the variables in the comparison are from input variables. We don't need to worry about this case and we can easily ignore it, since we can enforce it by applying it to the input value.

- $v_1 = y_i, v_2 = y_j$, in other words, both of variables in the comparison are from intermediate variables. We don't need to consider this case either.

- $v_1 = x_i, v_2 = y_j$. When we have a comparison operator between an input and intermediate variable. This is the only case which we need to consider.

We can model the comparisons with bipartite graph $G(\mathcal{A} \cup \mathcal{B}, E)$ as follows. $\mathcal{A}$ is the set of all input variables and $\mathcal{B}$ is the set of all intermediate variables. For each comparison which contains an input variable and an intermediate variable, we add an edge between the corresponding vertices. This bipartite graph may contain some cycles.

5.1 Method

Before starting this subsection we have to stipulate that the following arguments and paragraphs are true as long as we don't have any "$\neq$" operators. In this section we are going to propose an efficient way to evaluate the initial value of a map for a new input value.

We build a Range Tree(or Kd-Tree) data structure (DS) over the intermediate variables. Let $\vec{x'}$ to be the new input value for which we want to initialize the map. Since the comparison operators contain the input variables and the value of input variables are fixed, thus, these values($\vec{x'}$) define a volume in the search space of the intermediate variables, in other words, the search space of the DS.

We just need to consider all the tuples in this space for answering the query which initialize m for $\vec{x'}$. This space may drastically reduce the search space, specially for the initial value of zero when there is no tuple in the search space.

Lemma 1. *We can compute the initial value of a map for a new value in* $O(\log^{d-1} n + k)$ *where* n *is the number of tuples in the whole search space and* k *is the number of tuples induced by the input variables.* d *is the number of intermediate variables.*

Proof. As be said before, we build a Rang Tree on the intermediate variables. Any region on this data structure can be identify in $O(\log^{d-1} n + k)$. $\square$

We can also use KD-tree data structure whose order is $O(n^{(1-s/d)} + k)$ where $s < d$ is the number of input variables.

5.2 Another Method

In this part we propose another method which in some situations is more efficient that the previous one. The previous method is based on building a data structure on the intermediate variables and with which we bind the search space. But this method is based on the fact that all queries are aggregates and the results of all of them are integers.

The core idea behind this method is exploiting the fact that the queries are aggregate. Suppose we want to evaluate a query over a set of tuples. We can incrementally introduce the tuples one by one and then verify the query over it. Here we can use a memoization method to reduce the cost of reevaluating the tuples.

We build a range tree(or again a Kd-tree) over the input variables. In the other words the DS is defined over the input of the map. Suppose we want to know the initial value of $\vec{x'}$. $\vec{x'}$ represents a point in the space of DS. In order to evaluate the map with the new point we can use the fact that each point in this space is evaluated over a volume of space and if we find another point which has already evaluated, we just need to evaluate the difference between the volumes.

6 Map initialization

The maps used for the materialized view of the delta queries, are functions which have two types of arguments: input variables and output variables. Input variables are those variables that have the domain dependent on the relations that are not present in the underlying expression of the map. Whereas, output variables are variables that are bounded to the relations that appear in the underlying expression of the map.

For example:

$$m[\vec{x}][\vec{y}] :: q$$

is a map defined over the expression q. Expression q is defined by AGCA expressions in 2. Vector $\vec{x}$ will represent the vector of input variables of map m, and these variables will depend on the domains of other relations. Vector $\vec{y}$ will represent the vector of output variables of map m, an these variables will depend on the domains of the relations inside expression q.

We will start by giving a short example. If we have the following query and Δ regarding to R, written in DBToaster Calculus [1]:

$$q[][] = R(a,b) \cdot S(b,c) \cdot T(c,d) \cdot (a < d)$$

$$\Delta_R(a',b') = S(b',c) \cdot T(c,d) \cdot (a' < d)$$

then $\Delta_R(a',b')$ will be replaced by a map which will have variable a as an input variables and variable b as an output variable: $m_R[a][b]$. This map will appear in the trigger of onR regarding to an update which is done to relation R

When an insertion is made to relation R, the map should be initialized with a value. However if the value for variable b is not in the domain $\mathsf{dom}_b(m[a][b])$, the map will be initialized by 0, regardless of the fact that the map has an input variable.

Theorem 2. *A map with output variables will always be initialized with 0, when adding a tuple to the underlying expression of the map and the subtuple of the added tuple coresponding to the output variables is not in the domain of the map's output variables.*

$$m[\vec{x}][\vec{y}] :: q$$

$$\vec{z1} =< \vec{x1}, \vec{y1} > \text{ the tuple that will be added to the map's expression } q$$

where $\vec{x1}$ corresponds to the input variables of the map m and $\vec{y1}$ corresponds to the output variables of the map m

$$\vec{y1} \notin \mathsf{dom}_{\vec{y}}(m[\vec{x}][\vec{y}]) \Rightarrow m[\vec{x1}][\vec{y1}] = 0$$

Proof. The proof is based on the theorem 1, because equijoins will have only output variables. The property from the equijoins will be kept to maps with input and output variables.

Domains of output variables will always depend on relation that appear in the map's underlying expression q and therefore the domain can be easily computed. Exactly as we said in theorem 1, if the tuple is not in the domain then the initial value of the map will always be 0. □

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.

[3] Mark de Burg, Otfrid Cheong, *Computational Geometry Algorithms and Applications, 3rd Edition* Springer, 2008.