Draft

1 Join Tree

Throughout these sections, for the sake of simplicity, we are dealing only with Boolean Conjunctive Query (BCQ) and we call them query. These result can be extended to other queries. Let fix a given query as Q. Suppose for Q we have its join tree $JT_Q < V >$ which $V, |V| = n$ represents the set of its nodes (i.e. Figure 1). This join tree has several nice properties. The most important property of join trees is that we can compute their results in linear time ???. The other important property of join trees is *connectedness property*. By the connectedness we mean that if we select a variable which occurs in the given query, ???.

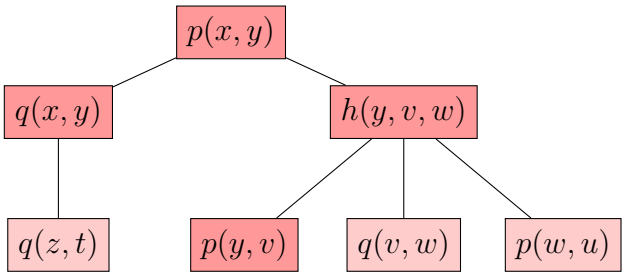

Figure 1: A sample join tree of a query

1.1 Number of different nodes is polynomial

Let fix k as the tree width of given JT_Q. Here we consider the case when k is constant (we have an upper bound for that). When we apply the Δ-operator to a join tree its nodes change also the structure. Here we prove

that the number of all nodes in all different trees that emerge from all different sequence of applying the Δ-operator is polynomial in term of n for a fixed k.

The nodes of JT_Q are all in form of $R_{i_1} \bowtie \cdots \bowtie R_{i_m}$ which $m \leq k$, m relations join together. Lets fix one particular node as l with m relations and the set of its indices is I. If we apply $\Delta_{\pm R_{i_p}}$ when $i_p \notin I$ this node is $\emptyset$. For the other case we will have node $R_{i_1} \bowtie \cdots R_{i_{p-1}} \bowtie R_{i_{p+1}} \cdots R_{i_m}$, in the other words relation R_{i_p} was deleted from the join operation. Thus, for counting the number of all nodes in all JT_Q for all sequences of applying Δ-operator, we should count all nodes which contain $1, 2, \cdots, k$ different relations. This number is $\sum_{i=1}^{k} \binom{n}{i} < (n+1)^k$ which is polynomial for a fixed k.

1.2 Cost of Evaluation

According to the last section the number of nodes in all delta trees is polynomial for a fixed k. Now we want to prove that the structures of all these trees are polynomial for a fixed k.

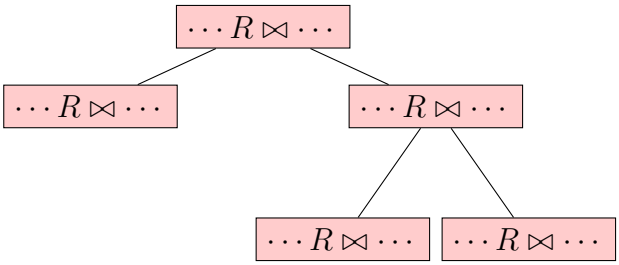

Figure 2: ??

When Δ_R-operator applies to a join tree it doesn't change the nodes that do not contain R relation. Thus, without loss of generality we can assume that R has appeared in all nodes, since we have connectedness property.

When we apply Δ_R-operator the such a tree, each node splits into 3 different nodes($\Delta(R \bowtie S) = \Delta(R) \bowtie S + R \bowtie \Delta(S) + \Delta(R) \bowtie \Delta(S)$). If we consider each $+, \bowtie$ as one operation, evaluation of $\Delta(R \bowtie S)$ needs

at most 5 operations. Thus, the cost of evaluation of all Δ trees is $O(n) *$ (Cost of join operator).

1.3 Hypertree Decomposition

Gottlob et al. have shown that for each Hypertree there exists a join tree with the same tree structure(Lemma 4.6). Thus, we have these results for hypertrees too.

2 Algorithms

In this section we give some algorithms for converting the JT_Q into the proper data structures which are used in DBToaster. Suppose we want to compute $\Delta_{\pm R} JT_Q$. As shown in 1.2 we can assume that every node contains relation R (i.e. 3).

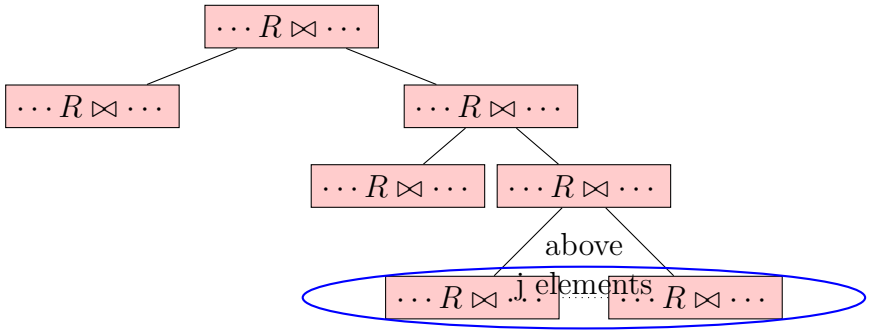

Figure 3: ??

Algorithm 1 Computing $\Delta_{\pm R}$

Input: node v as the root of JT_Q and relation R as the Δ variable.
Output: Calculus expression for Δ_R for node v

1: $J \leftarrow \emptyset$
2: **for all** Child i of node v **do**
3: **if** the subtree rooted at i has any node labeled by R **then**
4: Computes the Δ_R of i recursively and store it in $\Delta[i]$ if it is not already computed.
5: add i to J
6: **end if**
7: **end for**
8: $value \leftarrow ComputeChain(J)$
9: **return** $value.expr$

Algorithm 2 ComputeChain

Input: A set of sibling nodes of JT_Q as J and relation R as the Δ variable.
Output: Calculus expression for Δ_R for input nodes

1: **if** $|J| = 1$ **then**
2: $expr \leftarrow \Delta_R(J_1)$ $\{J_1$ is the first element of $J\}$
3: $join \leftarrow J_1$
4: **else**
5: $rest \leftarrow ComputeChain(J \setminus J_1)$
6: $join \leftarrow J_1 \bowtie rest.join$
7: $expr \leftarrow rest.expr \bowtie \Delta(J_1) + \Delta(J_1) \bowtie rest.join + J_1 \bowtie rest.expr$
8: **end if**
9: **return** $(exp, join)$

Obviously this algorithm needs to compute the delta of each element just one time. If we want to compute the Δ of a node with branching factor j it needs $2^j - 1$ different joint operation($\bowtie$) which is exponential in the worst case. But we can use a simple trick here. For the node we can first consider Δ of $j-1$ children and store it somewhere then with this value we can merge the jth one. With Algorithm. 2 we can compute the Δ in $O(n)$.

3 Nested queries

Suppose we have a nested query with 2 layers. Lets call the first layer as Q_1 and the second as Q_2. In alpha5 model we have the assignment operator with which we can make an temporary map as t_1 and compute the Q_2 as t_1 and then join it with Q_1 and other relations. For example we are given this query:

$$Q : \text{SELECT COUNT(*) FROM r WHERE r.b} \\ < \text{(SELECT SUM(s.c) FROM s WHERE s.a=r.a)}$$

For this query we can write it with alpha5 notations like: $\sum(r \cdot (t_1 \leftarrow \sum[(s(a) = r(a)) \cdot s(c))] \cdot (r(b) < t1))$ For each nested query we need a map and an assignment operator. So all of these queries can be expressed in alpha5 language.

This model is compatible with HyperTree Decomposition as we had in the previous sections. For that we need to have a node which contains the left hand and the right hand of the comparison operator which joins the nested queries. If the hypertree decomposition follow this condition then we can consider the hypertree as an ordinary one and use the previous algorithms on it for computation of delta queries. In the other words we have to split on the comparison operator during the tree decomposition process. A sample query:

select count(∗) **from** R, S **where** R.b=S.b **and** (
(**select count**(∗) **from** T **where** T.a=R.a)
<(**select count**(∗) **from** U **where** U.c=S.c))

And its representation in alpha5 notation: $\sum(R \cdot S \cdot (R.b = S.b) \cdot (t_1 \leftarrow \sum(T \cdot (T.a = R.a))) \cdot (t_2 \leftarrow \sum(U \cdot (U.c = S.c))) \cdot (t_1 < t_2))$

Consider we are given the hyper graph model of a query. Each comparison operator is a hyper edge consisting just 2 nodes. So if we split the hyper graph through these comparisons hyper edges(in fact contracting them since they are simple links), an introduce a new variable as the join of two endpoints, we can use the resulting hyper graph as the one we had in the previous section. But the cost of this splitting may affect the hyper tree width. In the worst case we may increase the hyper tree width by the number of nested

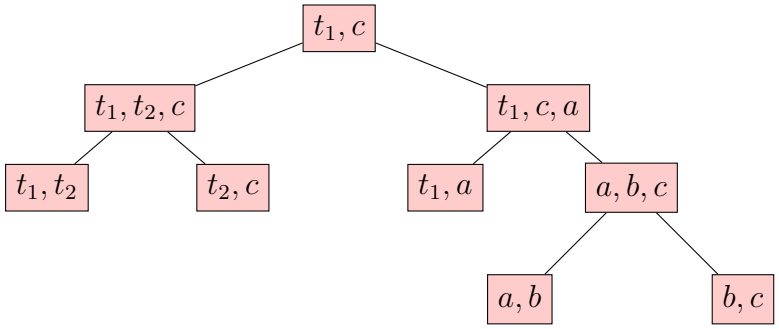

Figure 4: A sample of hyper tree decomposition of the query

levels. (??? can we make it better?)

In this way we just join the two variables of the comparisons edges, of somehow we have a node in the hyper tree which contains all of these new joined nodes we increased the hyper tree width by the number of these nodes(number of nested levels in the query).