Join Sizes

Sometimes, the size of a join result can be exponential in the size of the input relations, even if the join is acyclic.

Example 1

Consider $A_1A_2 \bowtie A_2A_3 \bowtie \cdots \bowtie A_{n-1}A_n$.

- Let each A_i have domain $\{1, 2, 3, 4\}$.
- Let each relation consist of the eight tuples such that one component is odd, the other even.
- Then the join result consists of all tuples over $A_1A_2 \cdots A_n$ with alternating odd and even components, a total of 2^{n+1} tuples.
- Yet the sum of the sizes of the $n-1$ input relations is only $8(n-1)$.

Complexity of Acyclic Joins

- The appropriate measure of "input" to the problem of computing a join is the sum of the sizes of the input relations and the result.
- Desirable: algorithm that is polynomial in this "size."
- If the join is acyclic, we can always compute the join in polynomial time.
  - Start with a full reducer, which is polynomial in the sizes of the input relations.
  - Then, join in any order.
  - Since there are no globally dangling tuples, the join can only increase in size at each step, so each intermediate result is polynomial in the size of the *output*.

Complexity of Cyclic Joins

The bad news is that the computation of an acyclic join can be exponential in the sum of the input and output sizes.

Example 2

Consider the attributes and relations of Example 1, but include an additional relation A_nA_1 in the join, as:

$$A_1A_2 \bowtie A_2A_3 \bowtie \cdots \bowtie A_{n-1}A_n \bowtie A_nA_1$$

- This join is clearly cyclic.

- If n is odd, its result is empty, because the join of the first $n-1$ relations allows only sequences of odd (o) and even (e) numbers of the forms $oeoe \cdots oeo$ and $eoeo \cdots eoe$, while the last term $A_n A_1$ requires A_n and A_1 to have values of different parity.

- The full reduction doesn't change the relations, because every attribute of every relation has $\{1, 2, 3, 4\}$ in its column.

- No matter how we group the relations for a join, before the final join there will be a relation formed from at least $(n+1)/2$ of the given relations.

- This intermediate relation has at least $2^{(n+5)/2}$ tuples, so it surely takes time exponential in n to compute.

Computing the Projection of a Join

Things only get worse when what we want is not the join of relations, but some projection of that join, e.g., $\pi_{ACE}(AB \bowtie BCD \bowtie DE)$.

- This form appears commonly in queries, but there is usually enough selection applied to the relations before joining that there is an efficient query plan.

- Not so, when the "query" is really the definition of a materialized view. Then the joined relations are often entire base tables, and the exponentiality of the problem is real.

Projections of Acyclic Joins: Yannakakis' Algorithm

The key idea is to use the "parse tree" implicit in a GYO reduction to guide the order of joins.

- First step is to fully reduce the input relations.

- During the join phase, project out all unnecssary attributes (those not in the final projection and not needed in any future join) after each join step.

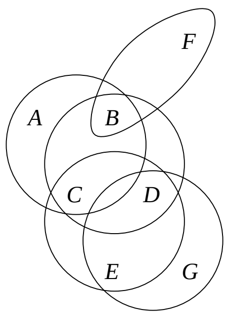
- Intermediate relations are no larger than the product of the input and output sizes.

Example 3

Consider the acyclic join-projection:

$$\pi_{AG}(ABC \bowtie BF \bowtie BCD \bowtie CDE \bowtie DEG)$$

Here is its acyclic hypergraph:



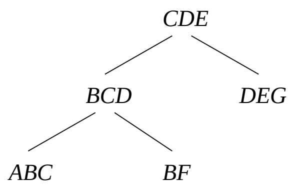
Parse Trees

When we perform a GYO reduction, we may construct a parse tree as follows:

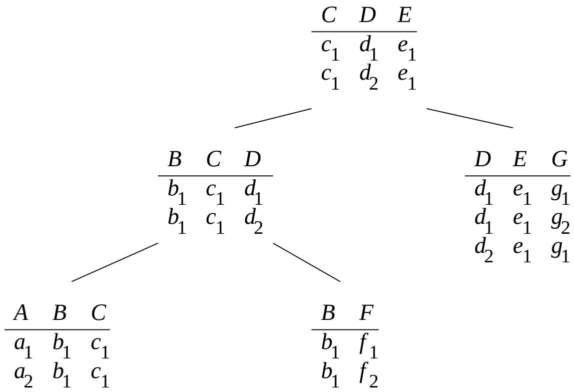
- Tree nodes correspond to hyperedges.
- The children of tree node H are all those hyperedges consumed by H.
- We choose as a join order one in which each node is joined with its parent, in some bottom up order (i.e., do not join a node into its parent, until all its children have been joined into it).
- After each join into a relation R, project the result onto the set of attributes that are either in the schema of R or on the projection list.

Example 4

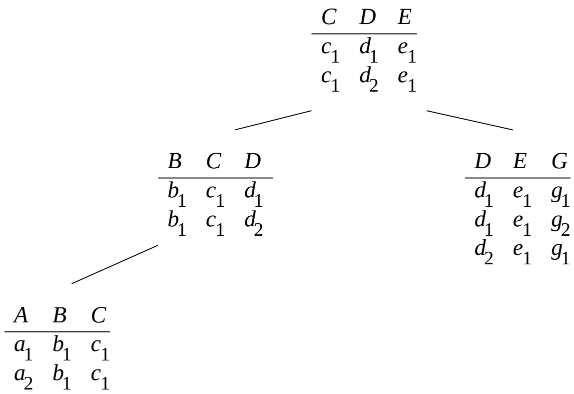
Here is one possible parse tree for the join hypergraph of Example 3:



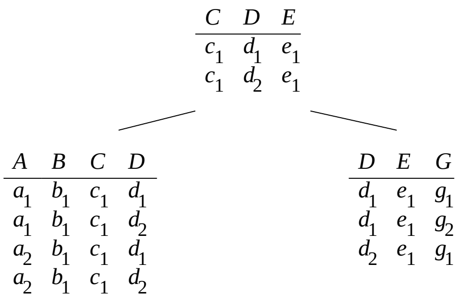
Here are example relation instances, which are already fully reduced:



We may join in any bottom-up order. Suppose we first join BF into BCD. We get a relation with 4 tuples. However, F is not in the projection list (A, G), so we project this relation onto BCD again, leaving the same relation for BCD. The result is that BF has been eliminated, with no other changes.



We next choose to join ABC into BCD. Since A appears in the projection list, it is retained at the node for BCD, which now has schema $ABCD$, as:



Suppose we next join $ABCD$ into CDE. We must project out B, since it is neither an attribute of CDE nor on the project list. However, A remains because it is on the project list:

A	C	D	E
a_1	c_1	d_1	e_1
a_1	c_1	d_2	e_1
a_2	c_1	d_1	e_1
a_2	c_1	d_2	e_1

D	E	G
d_1	e_1	g_1
d_1	e_1	g_2
d_2	e_1	g_1

Last, we join DEG into $ACDE$. Attribute G remains, because it is in the project list:

A	C	D	E	G
a_1	c_1	d_1	e_1	g_1
a_1	c_1	d_1	e_1	g_2
a_1	c_1	d_2	e_1	g_1
a_2	c_1	d_1	e_1	g_1
a_2	c_1	d_1	e_1	g_2
a_2	c_1	d_2	e_1	g_1

Our final step is to project the resulting relation $ACDEG$ onto AG, which gives a final result consisting of the four tuples $\{a_1g_1, a_1g_2, a_2g_1, a_2g_2\}$.

Why Yannakakis' Algorithm is Polynomial

Consider a relation R at some node, which at some time during the algorithm has been replaced by

$$\pi_{R \cup Y}(R \bowtie S_1 \bowtie S_2 \bowtie \cdots \bowtie S_k)$$

where:

1. $S_1, \ldots, S_k$ are some of the relations descending from R in the parse tree.

2. Y is the set attributes on the project list that are not in R but in at least one of $S_1, \ldots, S_k$.

** Tricky point: No attribute other than R and output attributes ever need to be in the schema of R. The proof depends on a number of ideas we haven't had:

  - Before there was GYO reduction, the full-reducer theorem was proven using a different definition of acyclicity. A hypergraph was said to be acylic if its hyperedges could be mapped to nodes, and the nodes placed in a parse tree, so that for each attribute A, the nodes with A in their schema formed a subtree (not necessarily at the root).

- One can prove that this definition of "acyclic" is equivalent to the GYO-based definition we use today.
- Thus, if A is an attribute at a child of R, but not at R, A cannot appear at any ancestor of R and is thus not needed after we join that child with R.

In what follows, we count the "size" of a relation instance as the number of its tuples. Technically, we need to consider the number of components of tuples as well, but since the set of relations and their schemas may be considered fixed, we are ignoring constant factors only.

- Let $T = R \bowtie S_1 \bowtie S_2 \bowtie \cdots \bowtie S_k$.
- Then $\pi_{R \cup Y}(T) \subseteq \pi_R(T) \times \pi_Y(T)$. To see why, notice that R and Y are disjoint sets of attributes. (As always, we're using R as both an instance and a schema, where appropriate.)
- Because the relations are fully reduced, joins only increase in size, and no tuple in an intermediate join can be dangling. Thus:
  1. $\pi_Y(T)$ is no larger than $\pi_L(T)$, where L is the entire project list for the query.
  2. $\pi_L(T)$ is no larger than the output, since every tuple of T extends to at least one tuple in the join of all relations.
  3. Putting (1) and (2) together: $\pi_Y(T)$ is no larger than the output!
- $\pi_R(T) = R$ (again, because the relations are fully reduced), so $\pi_R(T)$ is surely no bigger than the input.
- We conclude that $\pi_{R \cup Y}(T)$ is no bigger than the product of the input and output, i.e., polynomial in the input + output sizes.
- Final step: the number of computations of polynoimal-sized relations is a constant, depending only on the schemas and not on the instances.