Database Theory

William Schultz

November 8, 2022

Joins

At a high level, database (i.e. SQL) tables can be viewed as n-ary relations (or, more plainly, as "spreadsheets"). For example, consider the following relation P

Name	**Age**	**HouseId**
Alice	31	6
Bob	32	3
Jane	25	4

and another relation H

Id	**Year**
6	1904
4	1965

We can consider the *cross product* of these two relations $P \times H$, which is simply the Cartesian product of all rows (i.e. tuples) in P with all rows in H, giving relation $P \times H$ as

Name	**Age**	**HouseId**	**Id**	**Year**
Alice	31	6	6	1904
Alice	31	6	4	1965
Bob	32	3	6	1904
Bob	32	3	4	1965
Jane	25	4	6	1904
Jane	25	4	4	1965

with a total tuple count of $|P \times H| = |P| \times |H| = 6$.

On its own, the full cross product of two tables may not be very useful, but a commonly useful operation to apply after doing this cross product is the *join*, which essentially just applies some filter (e.g. predicate) to the tuples that are generated as a result of this cross product operation. If we filter the result $P \times H$ based on the predicate $HouseId == Id$, then we say we're "joining" the two tables, P and H, on $HouseId == Id$, which gives as a result:

Name	**Age**	**HouseId**	**Id**	**Year**
Alice	31	6	6	1904
Jane	25	4	4	1965

which is basically the set of people in P associated with the house in H they own. More compactly, we typically notate a join on predicate p between two relations A and B as

$$A \bowtie_p B$$

Again, we can think of this as simply a composition of cross product and filtering operations i.e.

$$A \bowtie_p B = \sigma_p(A \times B)$$

where σ_p represents the filtering operation for a given predicate p on tuples.

Note that joins are *commutative* i.e. $A \bowtie_p B = B \bowtie_p A$. This can be easily seen from examining the decomposed form of joins in terms of cross products and filtering i.e. since $A \times B = B \times A$ (if we ignore ordering of columns in the output tuples). Note also that we can view a sequence of joins as a big cross product followed by a filtering operation at the end i.e.

$$(A \bowtie_p B) \bowtie_q C = \sigma_q(\sigma_p(A \times B) \times C) = \sigma_{p \wedge q}(A \times B \times C)$$

Furthermore, joins are *associative*. This means that we can re-order joins as we please (since they are both *associative* and *commutative*), so there may be many different join executing orderings that produce the same final result (TODO: query optimization). That is,

$$\begin{gathered}(A \bowtie_p B) \bowtie_q C \\ A \bowtie_p (B \bowtie_q C) \\ A \bowtie_p (C \bowtie_q B) \\ (A \bowtie_p C) \bowtie_q B \\ (C \bowtie_p A) \bowtie_q B\end{gathered}$$

are all equivalent. Some orderings may, however, be much more efficient to execute. More generally, we can think of a particular join ordering as a binary tree, that essentially maps to the syntactic parse tree of the above expressions.