Joins via Polynomial Multiplication

Scott Kovach
https://cutfree.net

January 9, 2024

[draft]

1 Introduction

This post is on the subject of how to see operations on relations (such as natural join, projection, filtering) and operations on (sparse) numeric arrays as instances of "ordinary" addition and multiplication.

Let's talk about single-variable polynomials over some (semi)ring K. We can write a polynomial like so: $f = \sum_{i=0}^{\deg f} f_i x^i$, where $f_i \in K$.

There are usually two ways of describing polynomial multiplication. First, there's the explicit formula for coefficients:

$$(f \cdot g)_i = \sum_{0 \le j \le i} f_j g_{i-j}.$$

Second, there's the appeal to linearity: the key fact is that $x^i \cdot x^j = x^{i+j}$, and everything else follows from "multiplying things out". Multiplication by $f_i x^i$ is linear, that is, $f_i x^i (a + b) = f_i x^i a + f_i x^i b$, so

$$fg = \left(\sum_{i=0}^{\deg f} f_i x^i \right) \left(\sum_{j=0}^{\deg g} g_j x^j \right) = \sum_{i,j} (f_i x^i)(g_j x^j) = \sum_{i,j} f_i g_j x^{i+j}.$$

The first formula is more direct and explicit, but maybe the second is more intuitive. The core thing is just the rule for how powers of x multiply, and everything else is a consequence of the ring axioms.

Often relational joins are explained in terms of operations on sets, since relations are naturally thought of as sets of tuples anyway. The result is usually an *explicit formula* for the join of some relations. There are many perspectives (Jaffray, 2023), however, and in this note we'd like to call attention to the last entry of the linked post, "join is a ring product". This approach relies on the appeal to linearity to give a very straightforward definition of the join operation using polynomial arithmetic. Everything boils down to a rule that is just slightly different from $x^i x^j = x^{i+j}$.

We'll go beyond the post to see how this technique applies to tensors, graphs, and more. Finally, we'll talk about an implementation technique for actually efficient calculations, *indexed streams*, and how they naturally arise from thinking about polynomial multiplication.

2 Join from a semiring

Starting with a certain polynomial semiring, we'll construct a quotient so that multiplication in the resulting semiring directly encodes natural join. If you're unfamiliar with this jargon, there are two ideas to keep in mind:

- A polynomial is a sum of products of some pre-selected formal variables. "Formal" means that these variables don't intrinsically *mean* anything; they are merely opaque entities that can be added and multiplied. Given a coefficient semiring K and a set of variables $\mathbf{X}$, there is a corresponding polynomial semiring $K[\mathbf{X}]$.
- A quotient imposes certain equations; for instance, quotienting by $x^2 = y$ means that anytime x^2 appears in a polynomial, it can be replaced by y (or vice-versa). We will impose some equations so that every polynomial has a *normal form* with certain desirable properties.

2.1 The Variables

The first ingredient is a *schema*. Usually a schema tells us something about the tables, columns, and value types you find in a database. For us, it only specifies the possible columns, a set A, which we call *attributes*. For simplicity, we assume there is one value type U (for universe); you can imagine it as the disjoint union of whatever other types you might want.

We start with the simplest building block: (attribute, value) pairs. The idea is to make up a formal variable for every possible pair $(a, v) \in A \times U$. We use a mnemonic notation for these variables: for all $(a, v) \in A \times U$, we write "$[a \mapsto v]$" to denote a new formal variable. We choose this notation because a pair is supposed to represent a being mapped to v. For the set of all of them, we'll use the notation

$$\{A \to U\} := \{[a \mapsto v] \mid a \in A, v \in U\}.$$

We assume these variables commute with each other. The first polynomial semiring of interest to us is

$$\mathrm{Bool}[\{A \to U\}].$$

Here, $\mathrm{Bool} := \{0, 1\}$, with addition given by logical-or (i.e. $1 + 1 = 1$) and multiplication logical-and.

This semiring is all you need to represent relations:

- A tuple is a product of variables (one for each column).
- A relation is a sum of tuples.

Example: Say we have $A = \{a, b\}$ and $U = \{1, 2, 3\}$. Then The tuple $\{a := 1, b := 3\}$ would be represented by the product $[a \mapsto 1][b \mapsto 3]$. Note that the tuple components are labeled by name, not by their order, so it is fine that the variables commute, and the "order" in which the variables are multiplied doesn't matter.

To represent a collection of tuples, just add them together! A relation is a sum of products of these variables, or in other words, a polynomial.

2.2 The Equations

What if we multiply $[a \mapsto 1]$ by $[a \mapsto 2]$? What does $[c \mapsto 3]^2$ encode? Although it's clear that any relation can be represented as a polynomial in $\text{Bool}[\{A \to U\}]$, there are other things that *do not* correspond to normal relations living inside this set, like $1 + [c \mapsto 3]^2 + [a \mapsto 1][a \mapsto 2]$. Moreover, we claimed that we would recover join as multiplication, but multiplication will generate terms like those above that don't represent tuples.

The solution we take is to impose two equations on our semiring:

- (I_1) For all a, v: $[a \mapsto v]^2 = [a \mapsto v]$.
- (I_2) For all $a, v_1 \neq v_2$: $[a \mapsto v_1][a \mapsto v_2] = 0$.

It's easy to see that this guarantees that any $f \in K[\{A \to U\}]$ is equivalent to one where

- No exponent greater than one appears, and
- no two variables with the same attribute appear in any monomial,

which is sufficient to ensure that every monomial represents a tuple. We still entertain objects like $[a \mapsto 2] + [b \mapsto 3]$ which do not correspond to a relation whose tuples have the same attribute set.

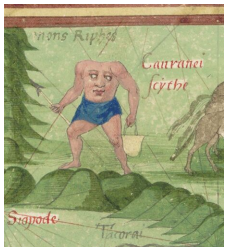
For the rest of this note, when discussing polynomials, we assume they belong to $K[U^A] := K[\{A \to U\}]/(I_1, I_2)$, the semiring with variables $\{A \to U\}$ modulo the identities we just defined.

The power of this definition is that *(equi)join is just product*:

Fact If $r_1, r_2 \in K[U^a]$ represent the relations R_1, R_2, then $r_1 r_2$ represents $R_1 \bowtie R_2$.

Example: Suppose we have these relations storing a subset of wikipedia:

$$\begin{aligned}
\textbf{pages} &= [\text{id} \mapsto 1][\text{title} \mapsto \text{“Headless men”}][\text{url} \mapsto \text{link}] \\
&\quad + [\text{id} \mapsto 2][\text{title} \mapsto \text{“Yōkai”}][\text{url} \mapsto \text{link}] \\
\textbf{main-image} &= [\text{id} \mapsto 1] \begin{bmatrix} \text{image} \mapsto \end{bmatrix} \\
&\quad + [\text{id} \mapsto 2] \begin{bmatrix} \text{image} \mapsto \end{bmatrix}
\end{aligned}$$



Note that both relations share the id attribute. The conflicting value rule means that when we multiply these together, only tuples with matching id values contribute to the product:

$$\begin{aligned}
&(\textbf{pages}) \cdot (\textbf{main-image}) \\
&= ([\text{id} \mapsto 1][\text{title} \mapsto \text{“Headless men”}][\text{url} \mapsto \text{link}] + [\text{id} \mapsto 2][\text{title} \mapsto \text{“Yōkai”}][\text{url} \mapsto \text{link}]) \cdot \\
&\left([\text{id} \mapsto 1] \begin{bmatrix} \text{image} \mapsto \end{bmatrix} + [\text{id} \mapsto 2] \begin{bmatrix} \text{image} \mapsto \end{bmatrix} \right) \\
&= [\text{id} \mapsto 1][\text{title} \mapsto \text{“Headless men”}][\text{url} \mapsto \text{link}] \begin{bmatrix} \text{image} \mapsto \end{bmatrix} \\
&\quad + [\text{id} \mapsto 2][\text{title} \mapsto \text{“Yōkai”}][\text{url} \mapsto \text{link}] \begin{bmatrix} \text{image} \mapsto \end{bmatrix}
\end{aligned}$$

Example: Suppose we have vector spaces U, V, W with bases $\{u_i\}, \{v_j\}, \{w_k\}$ and two matrices $A : U \to V$ and $B : V \to W$ written according to these bases.

We encode the matrices like so:

$$A = \sum_{i,j} A_{i,j}[U \mapsto u_i][V \mapsto v_j],$$
$$B = \sum_{j,k} B_{j,k}[V \mapsto v_j][W \mapsto w_k].$$

Note that for the sake of encoding a matrix as a polynomial, the attribute names U, V, W are unimportant; what is important is that we use a consistent encoding for both A and B. Now, what is AB?

$$\begin{aligned} AB &= \left(\sum_{i,j} A_{i,j}[U \mapsto u_i][V \mapsto v_j]\right)\left(\sum_{j,k} B_{j,k}[V \mapsto v_j][W \mapsto w_k]\right) \\ &= \sum_{i,j_1,j_2,k} (A_{i,j_1}[U \mapsto u_i][V \mapsto v_{j_1}])(B_{j_2,k}[V \mapsto v_{j_2}][W \mapsto w_k]) \\ &= \sum_{i,j,k} A_{i,j}B_{j,k}[U \mapsto u_i][V \mapsto v_j][W \mapsto w_k] \end{aligned}$$

We used the second equation to cancel terms where $j_1 \neq j_2$ and the first equation to simplify those terms that remain. What you see is *quite similar* to matrix multiplication, and all you need to do to close the gap is substitute 1 for all the $[V \mapsto -]$ variables (see Section 4).

3 Choice of Coefficients

Relational vs. Bag/Multiset Semantics There are at least two useful ways of representing relations in practice: relational and multiset semantics. The difference with the latter is that tuples may be stored more than once. Although the relational model is what Codd intended, the bag semantics might increase the efficiency of certain operations, since they don't need to check for duplicates when constructing their output. If we want bag semantics polynomials, we can simply choose a different coefficient semiring: instead of Bool, take $\mathbb{N}$. Other choices can be used to encode various other things:

- As in the matrix example above, we can choose $\mathbb{R}$ and assign a real value to each tuple; then relations may encode tensors.
- We might assign probabilities and then regard a polynomial as a probabilistic database, recording the likelihood of a given tuple.
- The tropical semiring uses $\mathbb{R}$ as its carrier set, min as its addition operation, and addition (of real numbers) as its product. With a universe consisting of vertices, polynomials over two attributes encode weighted graphs.

4 General Relational Algebra

Join is not the only operation of relational algebra. Here are some others, along with their encoding as polynomial operations:

union (merge together relations)

implementation: addition.

projection (eliminate an attribute from a table; denoted $\pi_a f$)

implementation: to project attribute a, substitute 1 for each $[a \mapsto v]$ variable. This is polynomial evaluation.

selection (Also known as *filter*; denoted $\sigma_P(f)$ by Codd) We want to filter out tuples that don't satisfy a predicate P: $\sigma_P(f) := \{t \in f \mid P(t)\}$.

implementation: suppose $p(t) = 1$ if t satisfies P and $p(t) = 0$ otherwise. Then encode the extension of P as a power series: $P := \sum_t p(t)t$. Now filtering is simply multiplication: $\sigma_P(f) = P \cdot f$, which is a polynomial.

rename (trivial but important; replace an attribute with another: $f_{a \mapsto a'}$)

implementation: substitute $[a' \mapsto v]$ for $[a \mapsto v]$, for all v.

Example: Suppose we want to talk about a matrix $A : V \to V$ and compute A^2. This is awkward but possible using our approach: we need attributes V', V'', then represent A using $K[V, V']$ (to distinguish the domain from the codomain), and then the matrix product is

$$(\pi_{V'} A(A_{V \mapsto V', V' \mapsto V''}))_{V'' \mapsto V'} \in K[V, V'].$$

5 Computations

Suppose we have a number of such polynomials stored in memory. Can we efficiently calculate product, union, projection, selection, in practice? In the remainder of the note, we'll focus on the case of product.

We know that two polynomials f, g can be multiplied by considering each pair of their monomials, multiplying them, and adding those up:

```
for t1 in f:
  for t2 in g:
    output += t1 * t2
```

However, a lot of cancellation occurs because of our reduction rules, so ideally we would avoid considering the large set of intermediate results that arise from considering all pairs.

The first bit of structure we're going to use is the attribute set. Say $A = \{a_1, a_2, \ldots, a_{k-1}, a_k\}$. We can equivalently write

$$K[U^A] \simeq K[U^{a_1}][U^{a_2}] \cdots [U^{a_{k-1}}][U^{a_k}].$$

The point is to think of an element $f \in K[U^A]$ as a polynomial over just the variables $\{\{a_k\} \to U\}$ with coefficients that are *also polynomials* living in $K[U^{a_1}][U^{a_2}] \cdots [U^{a_{k-1}}]$. If we can efficiently multiply elements of R and also polynomials over one attribute, then by induction we can multiply polynomials over several attributes.

5.1 Multiplying polynomials with one attribute

This is simple: because of equation I_2, the multiplication algorithm is equivalent to computing an intersection. For each $v \in U$ in the intersection, multiply the corresponding coefficients.

Example: Say we have $f = [a \mapsto 1] + 2[a \mapsto 2] + [a \mapsto 4] + [a \mapsto 8]$ and $g = 4[a \mapsto 1] + 3[a \mapsto 2] + [a \mapsto 3] + [a \mapsto 4]$. Then $fg = 4[a \mapsto 1] + 6[a \mapsto 2] + [a \mapsto 4]$.

Suppose we have a set data structure that supports constant time membership and iteration in time $O(|S|)$. Merging several sets $S_1, S_2, \ldots, S_k$ can be done in time $O(k \cdot \min_i |S_i|)$. The same is possible for maps which associate a coefficient to each key.

5.2 The general case

Example: Now suppose we have

$$f = ([b \mapsto 6] + [b \mapsto 7])[a \mapsto 1] + ([b \mapsto 2] + [b \mapsto 3] + [b \mapsto 4])[a \mapsto 2]$$

and

$$g = ([b \mapsto 3] + [b \mapsto 5])[a \mapsto 2] + [b \mapsto 2][a \mapsto 3]$$

Notice that for $[a \mapsto 2]$, $f_{[a \mapsto 2]} g_{[a \mapsto 2]} = [b \mapsto 3]$, but for other $x \in \{\{a\} \to U\}$, $f_x g_x = 0$. It follows that

$$fg = [b \mapsto 3][a \mapsto 2].$$

This generalizes in a straightforward way to any number of polynomials over arbitrary subsets of A. One way to compute this operation efficiently is to store the input polynomials using tries (with one level per attribute, according to the attribute order), then generate a nest of loops, with one per attribute. Each loop performs the required merge, and for each value in its intersection the next loop is run. The innermost loop multiplies K-values and stores them in an output data structure.

For more details, please see our paper, Kovach et al. (2023). We give a more complete characterization of state machines that suffice to compute these products efficiently and describe our compiler that implements the model for relations and sparse tensor arithmetic.

References

Scott Kovach, Praneeth Kolichala, Tiancheng Gu, and Fredrik Kjolstad. 2023. Indexed Streams: A Formal Intermediate Representation for Fused Contraction Programs *(PLDI 2023)*. Association for Computing Machinery. https://doi.org/10.1145/3591268