Programming Efficient Joins

Fritz Henglein

Department of Computer Science, University of Copenhagen (DIKU)

Joint work with Changjun Li, Annie Liu, Mikkel Kragh Mathiesen, Mads Rehof

2026-03-13
BOB 2026
Berlin, Germany

Who am I?

Fritz Henglein

Professor of Programming Languages and Systems
University of Copenhagen

Current areas of interest

- Programming language technology
- Algebraic methods in algorithms, semantics, logic
  - High-performance computing
  - Quantum programming
  - Resource accounting
- Decentralized systems
  - Distributed ledger technology
  - Supply network modeling
  - Financial and commercial contract technology

Academic background, affiliations, guest positions

TUM · Rutgers, The State University of New Jersey · NYU · Universiteit Utrecht · IT University of Copenhagen · Cornell University · UNSW Sydney · University of Oxford

Startups (co-founder)

Hf Hafnium · IT University of Copenhagen · DEON DIGITAL

A very common programming problem

- You have list structured data in memory and/or from multiple (streaming) sources.
- You need to *join* them in some fashion:
  - Top-sales people of wool sweaters in Iceland.
  - Friend triangles: $\{(x, y, z) \mid R(x, y) \wedge R(y, z) \wedge R(z, x)\}$ where R is a list of friend pairs.
- What to do?
  - Employ a library for filtering (selecting), projecting, joining pairs of lists (LINQ).
  - Formulate the query in SQL and dispatch it to an embedded SQL query interpreter or compiler.
  - Code it yourself. How hard can it be?

First some dry stuff

What, exactly, is a “joining”?

Querying, logically

- A *(relational) query* is en expression of the form $\{(x_1, \ldots, x_k) \mid \Phi\}$ where Φ is a purely relational first-order logic formula.
- A query maps input relations to an output relation.
- A *conjunctive query* is a query where Φ is built from relation symbols, conjunction ($\wedge$) and existential quantifier ($\exists$) only and $FV(\Phi) = \{x_1, \ldots, x_k\}$.
- A *join query* is a conjunctive query without quantifiers.

Join queries: Examples

- Represent directed graph by a binary relation R on universe $U = \mathbb{N}$.
- *Paths of (vertex) length 3*:

$$P_3(R) \quad = \quad \{(x_1, x_2, x_3) \mid R(x_1, x_2) \wedge R(x_2, x_3)\}$$

- *Triangles* (3-cycles): Triangles are paths of length 3 where the last node points to the first node.

$$T(R) \quad = \quad \{(x_1, x_2, x_3) \mid R(x_1, x_2) \wedge R(x_2, x_3) \wedge R(x_3, x_1)\}$$

- In SQL:

```
SELECT R1.x AS x1, R2.x AS x2, R3.x AS x3
FROM R AS R1, R AS R2, R AS R3
WHERE
     R1.y = R2.x
AND R2.y = R3.x
AND R3.y = R1.x
```

Triangles, naively

```
def triangles_naive(rel):
      result = []
      for (x, y) in rel:
          for (y2, z) in rel:
              if y == y2:
                  for (z2, x2) in rel:
                      if z == z2 and x == x2:
                          result.append((x, y, z))
```

Alice and her friends

- Parameterized data set: $A_n = \{(1, j) \mid 1 \leq j \leq n\} \cup \{(i, 1) \mid \leq < i \leq n\}$
- "Alice likes everybody, and everybody likes Alice."
- A_n has $3n - 2$ triangles.

Triangles, naively

- Platform: Apple MacBook Air M1 (2020), 16 GB Ram, MacOS 26.3.1.

n	$\lvert A(n)\rvert$	Time (s)	$\lvert$Triangles$\rvert$
100	199	0.06	298
200	399	0.47	598
400	799	4.51	1198
800	1599	42.65	2398

Table: Naive triangle enumeration on $A(n)$

- Scales with $\Theta(n^3)$.
- Beautiful, simple code. Hopelessly slow.
- Conventional wisdom: That's why we need clever data structures, algorithms, query optimization and more.

Dictionary-based join (hash join)

```
def triangles_dict(rel):
    d = {}                  -- empty dictionary
    for (x, y) in rel:
        if x not in d:      -- if x not already a key of d
            d[x] = set()    --    add x as key and initialize d[x] := {}
        d[x].add(y)         -- add y to the set of neighbors of x
    result = []
    for x in d:                          -- for each key x of d
        for y in d[x]:                     -- for each key y of d[x]
            for z in d.get(y, set()):        -- for each z in d[y]
                if x in d.get(z, set()):       -- if x is in d[z]
                    result.append((x, y, z))   --    add (x, y, z)
    return result
```

Dictionary-based join

n	$\lvert A(n)\rvert$	Time (s)	$\lvert$Triangles$\rvert$
100	199	0.0022	298
200	399	0.0088	598
400	799	0.0387	1,198
800	1,599	0.1455	2,398
1600	3,199	0.5355	4,798
3200	6,399	2.1354	9,598
6400	12,799	8.4900	19,198
12800	25,599	34.3772	38,398

Table: Dictionary-based triangle enumeration on $A(n)$

- Scales with $\Theta(n^2)$.
- As good as it gets, or is it?

Triangles, using Postgres

n	$\lvert A(n)\rvert$	Time (s)	$\lvert$Triangles$\rvert$
100	199	0.0184	298
200	399	0.0259	598
400	799	0.0783	1,198
800	1,599	0.2523	2,398
1600	3,199	1.2993	4,798
3200	6,399	4.4188	9,598
6400	12,799	21.1838	19,198
12800	25,599	99.1208	38,398

Table: Triangle enumeration on $A(n)$ using PostgreSQL

- Scales with $\Theta(n^2)$.
- Uses Postgres server running in a separate process.
- Maybe try an in-process query engine?

Triangles, using SQLite

n	$\lvert A(n)\rvert$	Time (s)	$\lvert$Triangles$\rvert$
100	199	0.0056	298
200	399	0.0185	598
400	799	0.0631	1,198
800	1,599	0.2561	2,398
1600	3,199	1.0250	4,798
3200	6,399	3.9669	9,598
6400	12,799	15.0921	19,198
12800	25,599	64.5064	38,398

Table: Triangle enumeration on $A(n)$ using SQLite

- In-process SQL query engine.
- Scales with $\Theta(n^2)$.
- Almost as good as handwritten dictionary-based Python code.
- A bit disappointing, though, isn't it?

Triangles, using DuckDB

n	$\lvert A(n)\rvert$	Time (s)	$\lvert$Triangles$\rvert$
100	199	0.0661	298
200	399	0.0922	598
400	799	0.1838	1,198
800	1,599	0.3558	2,398
1600	3,199	0.7115	4,798
3200	6,399	1.5080	9,598
6400	12,799	3.2905	19,198
12800	25,599	7.6738	38,398
25600	51,199	19.4697	76,798
51200	102,399	52.4164	153,598

Table: Triangle enumeration on $A(n)$ using DuckDB

- Scales with $\Theta(n^{1.5})$.
- Wow! Asymptotically beats best-possible classical query optimization. Tricky (CIDR 2025).

Triangles, using Simple Join

In Python:

```
def triangles(rel):
    # Compute triangles
    r1 = dict2(rel)                          -- dict. for 1st R
    r2 = r1                                  -- dict. for 2nd R
    r3 = dict2([(b, a) for a, b in rel]) -- dict. for 3rd R
    result = []
    for x in minset(r1, r3):                 -- for x in smallest ..
        for y in minset(lookup(r1,x), r2):  -- for y in smallest ..
            if x in r1 and y in r1[x]:          -- for z in smallest ..
               for z in minset(lookup(r2,y), lookup(r3,x)):
                   if y in r2 and z in r3[x]:
                      result.append((x, y, z))
    return result
```

- Looks like the dictionary-based join before.
- Simple and quickly coded (by hand or generated by a tool).
- Is it efficient?

Triangles, using Simple Join

With *all* auxiliary functions:

```
def dict2(rel):
    # Converts (key1, key2) pairs into a nested dictionary, O(n)
    nested_dict = dict()
    for (key1, key2) in rel:
        if key1 not in nested_dict:
            nested_dict[key1] = dict()
        nested_dict[key1][key2] = True
    return nested_dict

def lookup(d,k):
    return d[k] if k in d else set()

def minset(s1,s2):
    return s1 if len(s1) <= len(s2) else s2
```

Triangles, using Simple Join

n	$\lvert A(n)\rvert$	Time (s)	$\lvert$Triangles$\rvert$
100	199	0.000160	298
200	399	0.000290	598
400	799	0.000657	1,198
800	1,599	0.001324	2,398
1600	3,199	0.003088	4,798
3200	6,399	0.005265	9,598
6400	12,799	0.010330	19,198
12800	25,599	0.023742	38,398
25600	51,199	0.054950	76,798
51200	102,399	0.128696	153,598

Table: Triangle enumeration on $A(n)$ using `triangles.py`

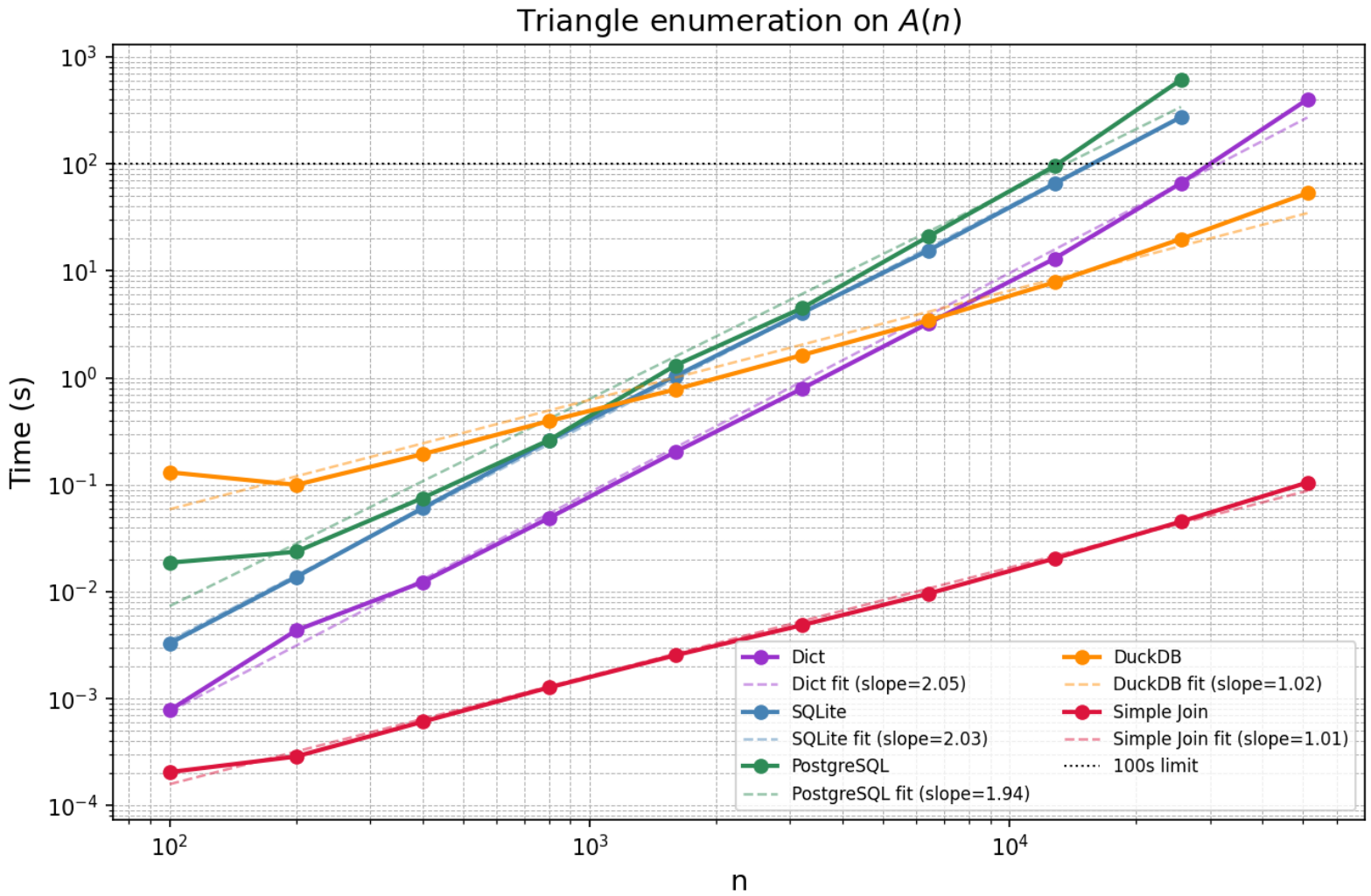
- Scales with $\Theta(n)$ (!).

Triangles, combined

n	$\lvert A(n)\rvert$	Dict	SQLite	PostgreSQL	DuckDB	Simple Join
100	199	0.0008	0.0033	0.0187	0.1310	0.0002
200	399	0.0044	0.0137	0.0237	0.0999	0.0003
400	799	0.0123	0.0604	0.0752	0.1933	0.0006
800	1,599	0.0489	0.2594	0.2615	0.3945	0.0013
1600	3,199	0.2029	1.0385	1.3076	0.7800	0.0025
3200	6,399	0.7909	4.0342	4.4762	1.6296	0.0049
6400	12,799	3.2332	15.508	20.910	3.4358	0.0096
12800	25,599	13.047	65.533	95.709	7.8268	0.0205
25600	51,199	66.235	–	–	19.901	0.0453
51200	102,399	–	–	–	53.397	0.1050

Table: Triangle enumeration on $A(n)$: run times in seconds. Entries marked – exceed the 100 s cutoff.

Triangles, all methods



Wat?

More dry stuff: Worst-case optimal join algorithms

- An algorithm for computing join queries is **worst-case optimal** if its run time is asymptotically linear in the sum of
  - the size (sum of cardinalities) of the input relations;
  - the size (cardinality) of the largest possible output relation for input relations of the same size as the given input relations.
- There is **no select-project-join query plan** that yields a worst-case optimal algorithm for *cyclic* join queries (Ngo, Porat, Ré, Rudr (2012)).
  - Textbook-style query optimizers are inherently asymptotically suboptimal.
- Do worst-case optimal join algorithms even exist?

Largest possible outputs: Examples

Let N be the size of R, the number of *edges* in the graph.

$$\begin{aligned} P_3 &= \{(x_1, x_2, x_3) \mid R(x_1, x_2) \wedge R(x_2, x_3)\} \\ T &= \{(x_1, x_2, x_3) \mid R(x_1, x_2) \wedge R(x_2, x_3) \wedge R(x_3, x_1)\} \end{aligned}$$

- Standard join algorithm:
  - P_3: For each $x_1 \in dom(R), x_2 \in R\{x_1\}, x_3 \in R\{x_2\}$, output (x_1, x_2, x_3).
  - T: Compute P_3. Then filter those (x_1, x_2, x_3) away where $(x_3, x_1) \notin R$.
  - Runs in time $\Theta(N^2)$ for both P_3 and T.
- Largest possible output sizes:
  - P_3: $\Theta(N^2)$.
  - T: $\Theta(N^{\frac{3}{2}})$. (Follows from *Atserias-Grohe-Marx fractional edge covering* bound.)
- P_3: Algorithm is worst-case optimal.
- T: Algorithm is **not** worst-case optimal.
  - Problem: Algorithm may compute many intermediate tuples, almost all of which are eventually thrown away.

Worst-case optimal join algorithms

- Worst-case optimal algorithms for cyclic queries discovered in 2012.
  - Atserias-Grohe-Marx fractional edge cover based algorithm: Carefully count cardinalities of intermediate joins as they are being built and switch between building them.
  - Leap-frog trie join: Perform repeated sort-merge join steps, but "leap" over runs of elements that are not in both tries.
- Not implemented in mainstream database systems (MySQL, Postgres, MS SQLServer, SQLite, ...).
- Leap-frog trie implemented in LogicBlox's proprietary query engine
  - Originally submitted for patenting. (Abandoned in 2018.)
  - Now in updated form (*Generic Join*) in Relational AI's engine (commercial).
- Implementing worst-case optimal joins must be very difficult.
  - Or is it?

Simple Join

- Implementing worst-case optimal join queries is *simple*.
- Ingredients:
  - Nested dictionary with *constant-time* length (size), iteration, lookup functions and linear-time bulk building.
    - No support for element deletion, addition, merging or other operations required.
  - Iterate always over the smallest set when intersecting sets.
  - Nested iteration over variables in a query *in any order*, *without any preprocessing*.
    - Eliminate the *variables* one at a time, not the *conjuncts*.
- Proof of optimality is constructive and general:
  - Amortized complexity analysis by allocating each constant-time computation step to a constant fraction of outputs eventually produced.
  - Input padding (ghost data) to attach computation steps to.

Simple Join: Algorithm

Informally:

- Let $Q = \{(x_1, x_2 \ldots, x_k) \mid C_1(\ldots) \wedge \ldots \wedge C_l(\ldots)\}$ be a join query where, w.l.o.g., the variables that are arguments in each C_i occur *at most once* and *in the same order*.
- Build nested dictionaries $\bar{C}_i : U \to \ldots \to U \to \mathbb{B}$ such that $\bar{C}_i(x_{i1})(x_{i2})\ldots(x_{ir_i}) = \mathit{true}$ iff $(x_{i1}, x_{i2}, \ldots, x_{ir_i}) \in C_i$.
- Consider all C_i whose first argument is x_1. (The others don't have x_1 amongst its arguments.) Choose the $\bar{C}_i$ *with the smallest domain*.
- For each $x_1 \in \mathit{dom}(\bar{C}_i)$ do:
  - Compute $D_i = C_i(x_1)$.
  - Keep $D_j = C_j$ if C_j does not contain x_1.
  - Compute $Q(x_1) = \{(x_2, \ldots, x_k) \mid D_1(\ldots) \wedge \ldots \wedge D_l(\ldots)\}$.
  - For $(x_2, \ldots, x_k) \in Q(x)$ do:
    - Return $(x_1, x_2, \ldots, x_n)$.

Simple Join: Triangles

$$
\begin{aligned}
T &= \{(x_1, x_2, x_3) \mid R(x_1, x_2) \wedge R(x_2, x_3) \wedge R(x_3, x_1)\} \\
T' &= \{(x_1, x_2, x_3) \mid R(\mathbf{x_1}, x_2) \wedge R(x_2, x_3) \wedge R^t(\mathbf{x_1}, x_3)\} \\
T'_{x_1} &= \{(x_2, x_3) \mid R_{x_1}(\mathbf{x_2}) \wedge R(\mathbf{x_2}, x_3) \wedge R^t_{x_1}(x_3)\} \\
T'_{x_1 x_2} &= \{x_3 \mid R_{x_1 x_2} \wedge R_{x_2}(\mathbf{x_3}) \wedge R^t_{x_1}(\mathbf{x_3})\} \\
T'_{x_1 x_2 x_3} &= R_{x_2 x_3} \wedge R^t_{x_1 x_3}
\end{aligned}
$$

```
triangles rel = [ (x1, x2, x3) |
     let s = dict rel,              -- map x to {y | (x,y) in rel}
     let sTransp =                  -- map y to {x | (x,y) in rel}
         dict [(y, x) | (x, y) <- rel]
     x1 <- minDom s sTransp,        -- iterate over smallest domain
     let t1 = apply s x1,           -- {y | (x1,y) in rel}
     let t3 = apply sTransp x1,     -- {z | (z, x1) in rel}
     x2 <- minDom t1 s,             -- iterate over smallest of t1 or s
     contains t1 x2,                -- check whether x2 is in t1
     let u2 = apply s x2,           -- {y' | (x2,y) in rel}
     x3 <- minDom u2 t3,            -- iterate over smallest of u2 or t3
     contains u2 x3,                -- check whether x3 is in u2
     contains t3 x3                 -- check whether x3 is in t3
     ]                              -- add (x1, x2, x3) to the solution
```

Optimality

Theorem

Simple Join is worst-case optimal.

Proof.

(Sketch)

- Take dictionaries with constant-time lookup, length and iteration; e.g. radix trees.
- Construct dictionary for each conjunct in query in linear time.
- Assume that a constant fraction of the elements in $dom(\bar{C}_i)$ occur in all the domains of the other $\bar{C}_j$. (!)
- Allocate the cost of each constant-time step in the algorithm to a constant fraction of the number of tuples eventually produced in the output.
- Pad input relations with additional tuples (“ghost data”) such that above assumption is satisfied, at most doubling the input.
- The algorithm then runs in time linear in the size of the output, including ghost data.
- Use that for all join queries Q there exists k such that its worst-case size i $\Theta(|\mathcal{M}|^k)$ by AGM.
- Conclude that the algorithm runs in time asymptotically bounded by the size (sum of cardinalities) of the input plus the worst-case size of the output.

Simple Join: Observations

- *Worst-case output size* for triangle query in general: $O(N^{\frac{3}{2}})$.
- *Output size for triangle query on A_n*: $O(N)$.
  - Note: N is number of edges. $N = O(n)$ for A_n(sparse graph).
- *Run time of Simple Join*: $O(N)$ on A_n (since A_n is already *padded*).

Simple versus Generic Join

- Generic Join (Ngo, Re, Rudra, 2014): Worst-case optimal join algorithm generalizing Leapfrog Trie (2013) and size counting algorithm (2013).

Algorithm 3 Generic-Join($\bowtie_{F \in \mathcal{E}} R_F$)

Input: Query Q, hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$

1: $Q \leftarrow \emptyset$
2: **If** $|\mathcal{V}| = 1$ **then**
3: return $\bigcap_{F \in \mathcal{E}} R_F$
4: Pick I arbitrarily such that $1 \leqslant |I| < |\mathcal{V}|$
5: $L \leftarrow$ Generic-Join($\bowtie_{F \in \mathcal{E}_I} \pi_I(R_F)$)
6: **For** every $\mathbf{t}_I \in L$ **do**
7: $Q[\mathbf{t}_I] \leftarrow$ Generic-Join($\bowtie_{F \in \mathcal{E}_J} \pi_J(R_F \ltimes \mathbf{t}_I)$)
8: $Q \leftarrow Q \cup \{\mathbf{t}_I\} \times Q[\mathbf{t}_I]$
9: **Return** Q

- Simple Join: Generic Join without Step 5. The intersections of the dictionaries involved are not computed.
  - The dictionary with smallest cardinality is chosen in constant time instead.
  - Generic Join intuitively performs the intersection twice, in Step 5 and later when looking up values of keys.
- New worst-case optimality proof by amortization with ghost data.

More speed

- Employ *acyclic join techniques* after variable elimination (within a loop over that variable).
- Use *symbolic Cartesian products* to represent relations. E.g.
  - $P_3(R) = \bigcup_{x \in dom(R) \cap ran(R)} (\{R^t\{x\}\} \times \{x\} \times R\{x\})$
  - Facilitates *computing* the cardinality of $P_3(R)$ in $O(N)$ for $|R| = N$ even though the output cardinality of $|P_3(R)| = \Theta(N^2)$.
- Exploit *low-rank* symbolic Cartesian product representation of input and intermediate data. E.g.
  - Alice data set has *tensor rank* 2:
    $A(n) = \{1\} \times \{j \mid 1 \leq j \leq n\} \ \cup \ \{i \mid 2 \leq i \leq n\} \times \{1\}$
  - Compute triangles by computing 8 intersections.
- *Parallelize*: Construct dictionaries in parallel, execute for-loops in parallel, map to GPUs.
- Use *specialized data structures* instead of general dictionaries (Cai et al, WG2.1 Working Conference, 1991).

The secret sauce

- Background:
  - finitary relational calculus, an extension to domain relational calculus over finite-cofinite relations and without any *safety* restrictions on queries (Henglein, Li, Mikkelsen, Rehof, DBPL 2025);
  - algebraic structures (algebras over $\mathbb{B}, \mathbb{N}, \mathbb{Z}, \mathbb{Z}[X]$, etc; Henglein, Mikkelsen, Sales, MSFP 2022);
  - analytic structures (vector spaces over $\mathbb{R}, \mathbb{C}, \ldots$).
- Recipe:
  - embed given structures (here: finite relations) into rich mathematical spaces and their algebras;
  - exploit algebraic equalities for efficient symbolic computation at run time;
  - map back to original structure;
  - hide what you've done (avoid words like "algebra", "spaces", "linear operators", "tensor", "homomorphism");
  - show only performance numbers.
- Simple Join is an instance of a more general algorithm—it was discovered by "shaking off" its generality.

Programming joins: Dos and don’ts

- Do:
  - Build a nested dictionary for each input relation at the beginning.
    - Same variable order in each dictionary.
    - Must be built fast and return its size in $O(1)$.
    - Python’s standard dictionaries satisfy the asymptotic complexity requirements!
  - Intersect sets by iterating over the *smallest* of them and checking membership in the others.
  - Write nested loop in chosen variable order.
  - Your join algorithm will be worst-case optimal!
- Don’t:
  - Use merge-sort join.
  - Use only binary joins.
  - Intersect sets without choosing the smallest one.
  - Any one of these things will destroy worst-case optimality.

Thank you!

Contact: henglein@diku.dk or fritz@henglein.com