`:- the, life, and, death, of, a, fact.`

A Datalog story

Peter Goodman

I came to understand Datalog by thinking about parsers

My mental models of logic languages were steeped in recursive descent, PEGs, and Earley.

My programming history went from web DSLs to decompilation

01	**Before university**	PHP/MySQL bulletin boards; first compilers: HTML-embedded DSLs → PHP
02	**University**	Formal languages, parsers (Earley, PEGs, Pratt), and compilers
03	**Grad coursework**	Optimizing compilers, transformations, and dataflow analyses
04	**Research + later**	JIT machine-code → machine-code binary translation
05	**First industry job**	Static binary → compiler IR lifting
06	**Later**	Decompilation back to source; Hex-Rays (IDA Pro)
07	**Present day**	AI-assisted analysis; LLMs can generate lots of Datalog

The work moved from compiling source to recovering it from binaries.

Logic programming kept resurfacing along the way

01	**Before university**	PHP/MySQL; HTML-embedded DSLs → PHP	
02	**University**	Formal languages, parsers, compilers	**02 · Prolog** My first encounter with logic programming
03	**Grad coursework**	Optimizing compilers, dataflow analysis	**between 03 and 04 · Datalog** bddbddb; Doop
04	**Research + later**	JIT binary translation	
05	**First industry job**	Static binary → compiler IR	**05 · Prolog + Datalog** OOAnalyzer, DDisasm; my own engines
06	**Later**	Decompilation; Hex-Rays / IDA Pro	
07	**Present day**	LLMs as program-analysis authors	**07 · AI + Datalog** LLMs write rules; Lemmalog, Minigraf

The logic-programming thread ran alongside the compiler and decompiler work.

A cut makes Prolog's ordered clauses behave like PEG choice

PEG: CONSUME THE INPUT

```
choice <- "a" "b"
        / "a" "c"
        / "a" "c" "d"

input:  "acd"
cursor: ac|d
```

1. "ab" fails; rewind.
2. "ac" succeeds and commits.
3. "acd" is never tried.

```
whole <- choice !.       FAILS
split <- choice "d" !.   SUCCEEDS
```

PROLOG: MATCH AN INPUT LIST

```
choice([a,b|R], R) :- !.
choice([a,c|R], R) :- !.
choice([a,c,d|R], R) :- !.

input: [a,c,d]  =>  R=[d]


whole(L) :- choice(L,R), R=[].
split(L) :- choice(L,R), R=[d].



whole([a,c,d])  FAILS
split([a,c,d])  SUCCEEDS
```

Prolog ! commits a successful choice; PEG !. tests end-of-input. Without cuts, Prolog can try the third clause.

Earley completion and new Datalog facts both wake waiting rules

EARLEY: THE DOT MOVES LEFT-TO-RIGHT

```
S → A B        A → "a"
```

```
S → • A B
```

recognize A by scanning "a"

```
A → "a" •
```

```
S → A • B
```

A completed constituent advances S.

DATALOG: ANY BODY OCCURRENCE

```
new head fact:  tc(2,3)
```

```
tc(F,T) : • tc(F,M), tc(M,T).
```

```
tc(F,T) : tc(F,M), • tc(M,T).
```

Dots mark activation sites, not syntax.

```
Then probe the residual:
tc(3,T)  or  tc(F,2)
```

Earley consumes grammar symbols in input order. Push Datalog can anchor a join at any matching atom, then solve the rest.

The push method jumps from proved heads into waiting rule bodies

```
tc(F,T) : tc(F,M), tc(M,T).
```

Source: *Stefan Brass & Heike Stephan, “Pipelined Bottom-Up Evaluation of Datalog Programs: The Push Method” (2018).*

1. PRODUCE

a new head tuple
tc(2,3)

2. ACTIVATE

each matching tc(...)
in another rule body

3. FINISH

index scans solve
the residual conjuncts

4. PRODUCE AGAIN

derive head facts,
then activate again

The left-corner intuition: recognition rises to a waiting rule; execution descends through indexes to finish its tail.

An activation point may be any matching body atom, modulo binding dependencies or explicit execution barriers.

A non-linear recursive rule has two places where a new tuple may enter

ACTIVATION SITE 1 / LEFT CONJUNCT

```
tc(F,T) : tc(F,M), tc(M,T).
```

ACTIVATION SITE 2 / RIGHT CONJUNCT

```
tc(F,T) : tc(F,M), tc(M,T).
```

New tuple matches the first atom

```
# new tc(F, M)
for T in TC_GIVEN_FROM[M]:
    tc(F, T)
```

New tuple matches the second atom

```
# new tc(M, T)
for F in TC_GIVEN_TO[M]:
    tc(F, T)
```

Both access paths require an index, even though the Datalog program has one relation.

Positive Datalog can compile into a set, two indexes, and recursion

```
from collections import defaultdict
TC = set()
BY_FROM = defaultdict(list)
BY_TO = defaultdict(list)

def edge(src, dst): tc(src, dst)

def tc(src, dst):
    if (src, dst) in TC: return
    TC.add((src, dst))
    BY_FROM[src].append(dst)
    BY_TO[dst].append(src)
    i = 0
    while i < len(BY_FROM[dst]):
        nxt = BY_FROM[dst][i]; i += 1
        tc(src, nxt)
    i = 0
    while i < len(BY_TO[src]):
        prev = BY_TO[src][i]; i += 1
        tc(prev, dst)
```

What is actually here?

- One set of facts
- Two append-only indexes
- Two join cases
- Live scans; no snapshots

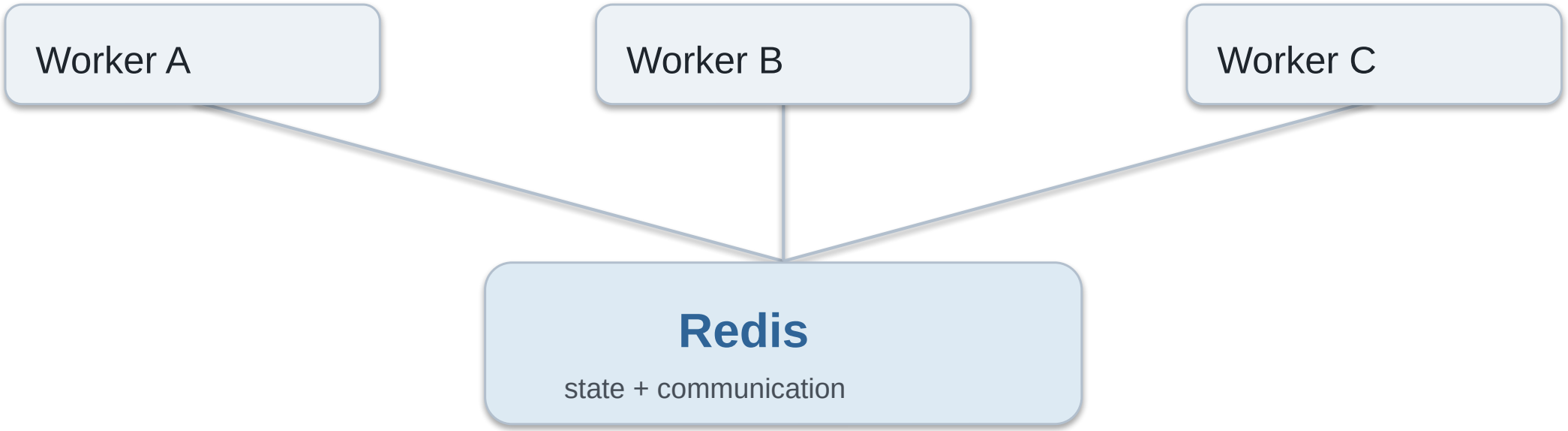
Both indexes update before recursion; scans observe later appends. Single-threaded, insertions only.

I started seeing Datalog-shaped problems everywhere

And writing an engine kept changing what I understood about data, messages, and execution.

Redis was our database and our message bus

Cyber Grand Challenge: a hub-and-spoke cyber reasoning system



- Services read and wrote shared state, and passed messages through Redis.
- Orchestration checks and local caches kept reappearing in different services.

If Redis had the data, why could the data not drive communication?

PROCEDURAL: READ, WRITE, PUBLISH

```
a = redis.exists(f"binary:{job}")
b = redis.exists(f"symbols:{job}")
if a and b:
    redis.set(f"state:{job}", "queued")
    redis.publish("analyze", job)
```

`read → read → set → publish   (network gaps)`

DATALOG: DEPENDENCY → DERIVED FACT

```
analyze(Job) :
    binary(Job), symbols(Job).
```

`facts arrive → rule derives a fact`

- Redis reads, writes, and pub/sub sends were separate remote operations; state could change between them.
- Stateless, retry-safe orchestration demanded discipline not enforced by the language.
- The Datalog rule removes procedural glue—but its head, analyze(Job), is still data.

The head was still a fact. What if I did not store it?

Same rule, two possible lifetimes

```
analyze(Job) : binary(Job), symbols(Job).
```

PERSISTENT HEAD

```
analyze(42)
```

Keep the row for later joins or queries

EPHEMERAL HEAD

```
analyze(42)
```

Notify consumers, then forget the row

The logic did not change; only the storage policy did.

That was the bridge from push-style fact activation to message passing.

Message streams grow the EDB; positive rules stay monotone

t_0 : binary(42) arrives

```
known: binary(42)

no analyze(42) yet
```

t_1 : symbols(42) arrives

```
known: binary(42), symbols(42)

emit: analyze(42)
```

- Input messages can add facts to the EDB at different times; stored facts allow later arrivals to join.

- With positive rules, adding EDB facts never invalidates old conclusions: monotonicity.

Dr. Lojekyll made messages first-class—and I wanted AND NOT

A second, more ambitious engine

```
#database pipeline.
#consume got_binary(u32 J).
#consume got_symbols(u32 J).
#consume got_blocked(u32 J).
#local binary(u32 J) : got_binary(J).
#local symbols(u32 J) : got_symbols(J).
#local blocked(u32 J) : got_blocked(J).
#produce analyze(u32 J) @differential :
    binary(J), symbols(J), !blocked(J).
```

- `#consume` receives inputs; `#produce` emits messages.
- `!blocked(J)` can withdraw an earlier `analyze(J)` output.

AND NOT breaks monotonicity even when inputs only grow

```
analyze(J) : binary(J), symbols(J), !blocked(J).
```

Before: eligible to analyze

```
binary(42). symbols(42).

analyze(42) = true
```

After: blocked input arrives

```
+ blocked(42).

analyze(42) = false
```

An input insertion can require a retraction of an earlier derived fact or output message.

Recursive push could test NOT before lower proofs were complete

```
reachable(J,F) : entry(J,F).
reachable(J,G) : reachable(J,F), calls(J,F,G).
blocked(J)     : reachable(J,0xBAD).
analyze(J)     : binary(J), symbols(J), !blocked(J).
```

- blocked(J) may depend on an arbitrarily long path in the binary.
- Recursive push leaves unfinished reachable proofs suspended in stack frames.
- Testing !blocked(J) too early can publish analyze(J) prematurely.

An absence check needs a scheduling boundary, not just an index lookup.

Stratification supplies the ordering that negation depends on

Stratum 0

binary, symbols, entry, calls

input relations

Stratum 1

reachable, blocked

compute its fixed point

Stratum 2

analyze

evaluate !blocked

Finish the positive call-path closure before concluding that a job is NOT blocked.

Retraction makes a tuple UNKNOWN before it is absent

Dr. Lojekyll: tri-state truth maintenance

PRESENT → **UNKNOWN** → **PRESENT** or **ABSENT**

- On losing a proof, mark affected conclusions UNKNOWN—not immediately ABSENT.
- Search alternative derivations through top-down table/index scans.
- Re-proof must not circularly rely on the very tuple being checked.

Example: losing one call path need not make blocked(J) false if another survives.

Avoiding hidden global state made database instances disposable

```
// BuDDy: hidden process-global BDD state
// unrelated BDD users could not run concurrently

class AnalysisDB {
    // instance-owned tables, indexes, temporary state
};

AnalysisDB first;
AnalysisDB second;   // independent instances
```

- I was integrating Datalog into existing C++ code, so I wanted state owned by each database.
- The surprise: if I can instantiate two, I can make one temporary—and discard it.

The first database instance was an algorithm I could throw away

```
// Schematic embedding inside an ordinary C++ algorithm
{
    AnalysisDB db;              // temporary facts and tables
    db.add_inputs(...);
    auto answers = db.results();
    consume(answers);
}   // discard the database and its intermediate state
```

- The larger program owns the instance lifetime.
- Datalog is a convenient way to sketch a local algorithm, not necessarily the whole application.

Repeated prefix variables suggested another use for database instances

```
#database global.
#consume edge(u32 Project, u32 Func, u32 From, u32 To).
#local reach(u32 Project, u32 Func, u32 From, u32 To).
reach(P,F,A,B) : edge(P,F,A,B).
reach(P,F,A,C) : reach(P,F,A,B), reach(P,F,B,C).
#query paths(bound u32 P, bound u32 F,
             free u32 A, free u32 B) : reach(P,F,A,B).
```

Project and Func identify an enclosing scope, but they are repeated in every local relation and eventually in the exported query.

A repeated tuple prefix could be carved out as a sub-database

- Tries and BDDs can share long tuple prefixes without changing the language.
- But the repeated prefix often identifies a higher-level container or analysis scope.

- What if a sub-database were carved out, with the prefix removed from its relations?

- This suggested local storage and lifetime, rather than just a compressed index.

Related language mechanisms (not equivalent): Soufflé components and CodeQL classes / member predicates.

A foreign functor hid a per-function Datalog computation

```
#database analysis.
#consume analyze_func(u32 Project, u32 Func).
#functor run_reach(bound u32 Project, bound u32 Func,
                   free u32 From, free u32 To) @range(*).
#produce paths(u32 Project, u32 Func, u32 From, u32 To)
  : analyze_func(Project, Func),
    run_reach(Project, Func, From, To).
```

- For each function, `run_reach`(P,F) loads the edges, runs a disposable database, and yields (From,To).

- But the Datalog compiler cannot see the nested rules, input mapping, fixed point, or lifetime hidden inside the native functor.

A nested database makes Project and Func constructor parameters

```
#database per_func(u32 Project, u32 Func).
#functor source_edge(free u32 From, free u32 To) @range(*).


#local reach(u32 From, u32 To).
reach(A,B) : source_edge(A,B).
reach(A,C) : reach(A,B), reach(B,C).

#produce paths(u32 From, u32 To) : reach(From, To).
```

After carving out the prefix: `reach(P,F,A,B)` becomes `reach(A,B)` inside `per_func(P,F)`.

`source_edge` uses the per-database Project/Func context; `#produce` is the result root. Parent/child message wiring is not implicit.

A slow per-function computation can return its results later

```
#database analysis.
#consume analyze_func(u32 Project, u32 Func).
#functor run_reach(bound u32 Project, bound u32 Func,
                   free u32 From, free u32 To) @async.
#produce paths(u32 Project, u32 Func, u32 From, u32 To)
  : analyze_func(Project, Func),
    run_reach(Project, Func, From, To).
```

- Adding `@async` turns the invocation into request and later response messages.

- A child's fixed point—or a Z3 call—might take arbitrarily long. Other independent work need not wait on the same call stack.

Messages made the carve-out boundary a unit of parallelism

The same boundary does three jobs

Storage	`reach`(From,To) lives inside `per_func`(`Project`,`Func`)
Composition	a child owns its fixed point, temporary state, and results
Messages	Requests start work; responses return results asynchronously

`@async` runs an ordinary functor in an implicit child; messages carry its results back.

Composition became the ball and chain

Why I stopped choosing Datalog for real programs

The rules were easy to combine. The resulting program was not easy to live with.

A missing tuple can hide a missing fact or a broken rule

- Why did a tuple appear? Follow the supporting facts and rules.
- Why did the expected tuple not appear? Find the failed join, filter, or missing input.
- `!validated(Call)` means absent from *this* fact database—not impossible in the program.
- Missing CFG edges, exceptions, or dynamic calls can hide real witnesses—or make negation succeed spuriously.

An analysis is only as faithful as its inputs: omissions can cause both false negatives and false positives.

Effortless composition made the program sprawl—and joins expensive

- A few hundred lines could describe a huge graph of dependencies.
- Rule changes had surprising effects on distant results, intermediate tuples, and indices.
- Relational joins became the one broadly available hammer—even for local state, order, or destructive updates.
- A small procedural fixed-point loop was often clearer and cheaper.

The lexical economy concealed a maintenance and runtime tax.

A declarative language makes you accept its computational ideology

- In a procedural language, I can pick the representation and control flow I want.

- In Datalog, the language makes certain patterns easy and others unexpectedly expensive.

- Ascent and egglog do impressive things; I respect the research behind them.

- I still found the complexity of restoring everyday procedural flexibility hard to justify.

Sometimes it felt a bit like the emperor’s new clothes.

Eventually, I stopped trying to justify Datalog to my users

- They wanted tools whose costs and blind spots they could understand.
- I could not sell them a big opaque rule engine merely because the rules were elegant.
- Writing fixed-point loops by hand was often not that difficult anyway.
- So I left Datalog alone for a few years.

I still liked the idea. I just did not like maintaining the programs.

AI changed the programmer and the audience

The work is delegated; the expectations of the user changed, too.

People increasingly supervise results from fallible tools, rather than prescribing each step.

Users will tolerate opaque discovery—but still need checkable results

- Prompts increasingly act as programs: people supervise objectives and outcomes, not every intermediate step.
- A model may take an unpredictable path to a finding; that is tolerable if there is a concrete witness, path, or reproducer to verify.
- A false-negative is different: an empty result cannot establish completeness.
- Users still dislike discovering that an LLM made a silly implementation mistake.

The discovery process may be a black box; the evidence should not be.

Datalog leaves fewer implementation choices to the LLM

- A model can botch loops, invalidation, caching, threading, and glue code.
- With Datalog, much of that machinery is the engine’s responsibility.
- The narrower vocabulary limits the ways an implementation can go wrong.
- Bad rules, pathological joins, and missing facts remain very real failures.

The computational ideology I resented becomes a guardrail for generated analysis.

Writing and debugging Datalog no longer has to be my job

- Turn a question into typed facts, rules, and an observable `#query`.
- Use errors and counterexamples to revise the analysis—not just the prose answer.
- If the expected answer is known, debug the rule; otherwise, produce leads to check.
- Discard a temporary rule set when the investigation moves on.

Writing one-off Datalog can be cheaper than owning it forever.

Slow agents change what performance tradeoffs are reasonable

- A model may spend substantial time planning, making tool calls, and interpreting results.

- For those workflows, extra engine overhead may be less important than simpler rule generation.

- That is not permission to materialize catastrophic joins or ignore memory.

- If models get faster, the balance changes again.

The right benchmark is the whole investigation, not just one query.

A program-analysis conversation keeps changing its mind

- An initial hypothesis leads to a query; results suggest a new hypothesis.
- Counterexamples require retracting conclusions, not merely adding more facts.
- Source revisions, missing edges, and corrected assumptions change the evidence.
- Rules evolve too: the analysis is a moving target, not one batch input.

This looks like an evolving knowledge base rather than a single report.

A hypothesis has two histories: when it held and when we believed it

Valid time (when true)

- When the behavior was true in the target program.
- Across revisions, configurations, or execution intervals.

Transaction time (when recorded)

- When the investigation accepted a hypothesis.
- When new evidence caused correction or retraction.

Bitemporal systems retain past versions: “when was this fact alive?” becomes a query. Minigraf and Lemmalog are examples.

Differential updates handle retractions; bitemporal history and provenance need explicit design.

The economics of Datalog changed, and so did its audience

- Effortless composition drew me in, then became a maintenance and trust problem.
- An LLM can write and discard rules I would not want to own forever.
- Users tolerate exploratory black boxes when results can be independently checked.
- A constrained language and a real runtime reduce the need for improvised machinery.

“My data does the work to make my data.” This time I may not have to write it all.