TETRA CA PBC

TECHNICAL WHITEPAPER

GRAPH DATABASE

TAKING THE RED PILL

Can the Matrix Replace Tables?

A graph database with no indexes, no cache, and no buffer pool — built end to end out of matrix representations, and encrypted before it is stored.

Preliminary results · LDBC SNB Interactive Short · Scale factor 1 · Single node

TETRA-CA.COM

SUMMARY

RESULTS AT A GLANCE

Measured against an industry-leading graph database provider on the same box, back to back, within the same hour.

317.7 MB	**2.65** GB	**43,740** ops/s	**5.16** ms
Database on disk	RAM under load	Throughput @ 64 clients	p99 latency @ 64 clients
`5.1× smaller`	`38% less`	`31% more`	`29% lower`

ABOUT THESE NUMBERS

Preliminary results. LDBC SNB Interactive Short, scale factor 1, seven queries, single node. Interactive Complex and larger scale factors are still in progress. TETRA data is encrypted end to end; there is no unencrypted mode, so **every TETRA number in this paper was produced on encrypted data.**

	TETRA	INDUSTRY-LEADING PROVIDER
Database on disk	**317.7 MB**	1,636 MB
RAM under load	**2.65 GB**	4.30 GB
Throughput @ 64 clients	**43,740 ops/s**	33,279 ops/s
p99 latency @ 64 clients	**5.16 ms**	7.22 ms
Allocation per query	**~100 KB**	—
Result sets diffed, mismatches	**175 / 0**	—
Encrypted at rest	**yes**	no

TETRA vs. an industry-leading graph database provider — same input, same host, same hour.

Source CSV for the same graph: 1,007.6 MB.

SECTION 01

HOW DO LLMS STORE THEIR DATA?

Large Language Models, the tech behind “AI,” surprised us all with their precipitous rise. We hear often about the huge data center requirements to build and train, then serve and leverage, models trained with massive amounts of information. The model itself, as you probably know, is a matrix of numbers.

I became very curious about how the model stores and retrieves information, and ultimately how it compresses data while making it queryable. Having read many papers on LLM technology, I felt like none of them sufficiently explained the *mechanism* — HOW do LLMs store their data?

Let me share what I mean. I can go to Hugging Face, download a Qwen3 8b parameter model. 20 GB or so. And I can run it with `llama.cpp` — or my own software that took me not-that-long-to-write, with LLMs — I can run it and prompt it and get a lot of information from this model.

> 20 GB. No index. No tables. No joins. No query planner. Nothing got looked up. Every answer got **calculated**.

Yes, sometimes it hallucinates. Sometimes it follows the wrong “track” across the information, but that only increases my curiosity — why does THAT happen, too? As you operate and learn more about these things, the questions only proliferate.

You can take a 20 GB model of a certain accuracy of numbers; you can round the numbers and reduce the size to store and query the model. And do you know what happens? It retains **~80% of its original accuracy**.

What gets cut in half and survives by 80%?

Nothing you have ever put in a database. Drop half the bytes out of a B-tree and see what your query gives you back.

SECTION 02

OR DOES IT?

Alright, so, this has nothing to do with databases. Or does it?

Because here's the thing I couldn't let go of. Round the numbers and the model still works — so whatever the model knows, it isn't keeping it in the digits.

The information is in the arrangement. The numbers encode structure within the empty space: what sits near what, what points where, how far apart things are. Chop the precision off every one of them and the structure is still standing, **because the structure was never in the precision.** That's the answer to *what gets cut in half and survives by 80%* — something that keeps its meaning in how it is arranged rather than in how exactly it is written down.

And that is a filing system, not a party trick.

So: what if the math and mechanisms we have learned for building and operating LLM models can be cross-tuned and re-appropriated to build a **hallucination-free database**? Nothing gets trained. Everything gets declared. And queries? They don't just store and retrieve. They embed and they calculate.

Well why in the Agent Smith would I want that?

We built a data system that only uses matrix representations. End to end. It works with the open-source query language, openCypher, over the Bolt wire protocol — so nothing in your stack finds out anything unusual happened.

Here's what that bought us. The LDBC social network benchmark at scale factor 1 — 3.18 million nodes, 17.4 million edges, 20 properties, 14 labels — becomes a **317.7 MB file**. Not an export you load into something. A file you query in place. The raw CSV it came out of is 1,007.6 MB, so the queryable, encrypted version is a third the size of the source data. The industry-leading provider builds 1,636 MB out of that same input. At SF10 our format produces a 3.4 GB file.

It sits at **2.65 GB of RAM** under load where the industry-leading provider sits at 4.30 — and serves **31% more throughput** while it's doing it.

> **WHAT IS NOT IN THERE**
>
> There are no indices. None. No caches. No buffer pool. Nothing stored that we could just calculate. A query costs about **100 KB**, and that number does not care how big your graph is. And all of it is encrypted — every byte, end to end. That isn't a mode we turned on for the demo; it's how the system works.

This paper is about how.

SECTION 03

THE SPOON THAT BENDS

All of the data systems we have grown accustomed to go back to a single idea from 1970.

A guy at IBM named Ted Codd wrote a paper that year, and the argument in it was simple: you should be able to ask for information by *what it is*, not by where somebody happened to file it. Before him, you got at your data by following pointers — a chain of "this record knows where that record lives" that somebody had to lay down by hand and everybody else had to walk. Codd thought that was insane. He was right.

So he gave us relations. Tables. Ask for what you want, let the machine find it.

But look at what he had to work with. 1970. A computer that did one thing at a time, at a speed we'd now call sedate. The only way to deliver "ask for what it is" on that hardware was to line everything up in rows, walk the rows, and build extra structures on the side so the walking didn't take all day. Those extra structures are your indexes. **Fifty-six years later you're still buying RAM for them.**

Here's the spoon.

> Tables were never the idea. Tables were the idea **compiled for a machine that could only do one thing at a time.**

We don’t have that machine anymore. We have machines that do millions of things at once, and we’ve spent the last decade building the math to drive them — enormous coordinate spaces, arithmetic run flat across all of it. It’s what a GPU is for. It’s what every LLM is made of.

So ask the question: what does Codd’s idea look like if you compile it for *this* hardware instead? You don’t get tables. You get an address space where the position of a thing is what the thing means, and finding it is arithmetic.

Now, none of the ingredients here are ours. Graphs and matrices have been the same object for a long time and serious people have been using that fact for years. But they use it to make certain operations faster — an adjacency multiply here, a traversal kernel there — while underneath, the database is still a database. Tables. Indexes. A buffer pool. A planner.

> We don’t use the math to speed things up. We use it to **be** the system.

One matrix. The data, the relationships between the data, and the plan for getting at it — all in the same thing, compressed, and queryable while it’s still compressed. Nothing gets unpacked on the way to an answer.

And here’s what makes it work in practice: your business database is not all of human knowledge. An LLM has to swallow the entire internet, which is a hard problem and an approximate one. You have customers, orders, accounts, events. Bounded. Structured. Yours. That’s a far easier thing to compress, and you can do it **without giving up a single true answer.**

SECTION 04

THE FOLDED MAP

We use less RAM to do more work, because we spend CPU on calculating instead of RAM on storing.

Numbers pack tighter than things. You can fit vastly more of them in the same space than you can long blobs of text — and every “unstructured” field in your database is a blob somebody gave up on.

So we represent everything as numbers. Then we build a map out of them — a tiled one, where position carries meaning — and we fold it up.

When a query comes in, we don't unfold the map. We open the panel we're standing on, see where we need to go next, and open that one. We never open the whole thing. Most of the map is never touched at all, and the parts we do touch, we touch once.

That's the entire trick. That's how you do more with less, consistently.

Conventional engines work the other way around. To answer a question they pull pages into memory to find out whether those pages contain the answer. They keep indices so they can guess earlier, caches so they don't have to guess twice, buffer pools to hold the guesses, and eviction policies to decide which guesses to forget.

> Most of your RAM bill, most of your I/O, and most of the energy is spent **discovering where the data isn't.**

We know where it is. The address *is* the meaning. So we read the span that holds the answer and leave the rest of the file untouched — on disk, unread, costing nothing.

The substrate is mapped and it does not move. The garbage collector never sees it. Under sustained load TETRA spends **0.375% of CPU in garbage collection**, because the only thing there is to collect is the row on its way out the door.

SECTION 05

INDEXING ISN'T SOMETHING YOU DO. IT'S SOMETHING THAT IS.

Here's where it gets strange.

An index isn't something you give the data. It's something the data tells you, through its relationships. In matrix land a relationship is a question of coordinates: which square are you in, and which squares are you connected to? You don't look that up. You *are* that.

Think about what an index actually is: a second copy of data you already have, arranged differently, so that a question you anticipated can be answered faster. **You pay for it three times.**

1. **In space** — holding the copy.
2. **On every write** — keeping the copy honest.
3. **In human judgment** — somebody has to guess which questions are coming, and somebody has to notice when the guess goes stale.

That third cost never appears on the invoice, and it's usually the largest. Index selection reviews. `EXPLAIN` archaeology. The quarterly audit of which forty indices are actually used. The bulk load that finishes in an hour and then rebuilds for six. Somebody on your team knows exactly which query falls off a cliff at month-end, and **that knowledge is load-bearing infrastructure that appears on no architecture diagram.**

We don't have any of that, because there's nothing to derive. The arrangement isn't a copy of the storage — it **is** the storage. Being in a place is already being indexed.

> **THE RECEIPTS ARE ON DISK**
>
> In the industry-leading provider's SF1 store directory, the schema folder — the indexes — is **251 MB**. That is by itself **79% of TETRA's entire database file**, which contains the whole graph, encrypted, with nothing left out. No state. No cache. No index. Nothing extra.

> There is no index to choose, because **there is no index to choose from.**

SECTION 06

PROPERTIES, OR: WHAT I LEARNED IN FRENCH CLASS

Properties are fun. Properties are really just relationships that you have to yourself.

So, instead of me *being* an age, I **have** an age. In English you *are* thirty years old, like it's something you're made of. In French you say *j'ai trente ans* — I have thirty years. It's something you're holding.

I learned that in French class when I was 12.

Neo learned it when they pulled his body out of an aquarium and pulled a metal straw out of his occipital lobe.

So we don't store a node, store its properties somewhere else, and stitch them together on the way out. A property is an edge from a thing to a value. Same coordinate space, same structure, same read.

> There is exactly one kind of thing in this database: **a relationship.** Nodes relate to nodes. Nodes relate to their values. It's all one substrate.

And here's what falls out of that.

In a conventional database, filtering on a property means crossing from where the topology lives to where the properties live — **a join you never wrote**, that doesn't appear in your query, that has a cost model and an optimizer pass and a cardinality estimator built entirely to guess around it. It's the reason your fast traversal got slow the moment you added a `WHERE` clause.

The industry-leading provider keeps that split on disk, plainly: a node store at 64 MB, a property store at 576 MB, a separate string store at 160 MB, and a relationship store at 576 MB. Four files, and every property read is a trip between them.

We don't cross. There's nowhere to cross *to*. Filtering isn't a step that happens after traversal. It's the same step, in the same square, in the same read.

The same logic applies to edge types. A type isn't a scalar you compare against on every edge — it's an axis. Typed filtering is row enumeration, not a per-edge test. When we removed the last per-edge type check from the hop walk, a single *length* call that had been running once per edge turned out to be **1.2% of a query on its own**, and the type gate as a whole was **17.9%**.

And absence gets the same treatment. `null` is the structural zero. A missing value isn't a sentinel, a nullability bit, or a column of mostly-nothing. It's simply not in the sparsity pattern.

> Absence is free — not cheap, **free.** It has no representation at all.

SECTION 07

MEMORY SCALES WITH THE ANSWER, NOT THE DATA

Every database's memory story is the same story: how much of the dataset can you keep hot. Indices, buffer pools, caches, eviction policies — machinery for holding a fraction of your data close, all of it scaling with the data.

We don't have a fraction. We have a frontier.

A traversal touches the nodes it touches. Memory follows the wavefront, not the corpus, so the working set is a function of the answer's shape rather than the dataset's size.

Here is what that looks like measured. These are `runtime.MemStats` deltas across a ten-second run, divided by queries served — not estimates:

QUERY	BYTES / QUERY	ALLOCATIONS / QUERY
IS1 — person profile + city	75,746	434
IS2 — 10 newest messages → root post → author	259,663	963
IS3 — friends + edge property, ordered	140,969	783
IS4 — message content + date	**33,229**	**252**
IS5 — message → creator	62,847	364
IS6 — message → post → forum → moderator	58,538	391
IS7 — replies + author + `KNOWS` back-check	105,532	476
Mean	**105,218**	**523**

LDBC SNB Interactive Short — allocation per query, SF1 (3,181,724 nodes · 17,436,661 edges).

A query costs about **100 KB and 500 objects**. Ask for a message and its properties and you're out 33 KB.

Now go look at what those queries are running against. 3,181,724 nodes. 17,436,661 edges. And the greediest query in the set spends a quarter of a megabyte.

> Nothing on that table is proportional to the graph. That's the whole thing, and it's not an argument — **it's a `MemStats` delta.**

And when nobody's asking

Conventional engines are always warm. A buffer pool is a bet about the next query. A cache is a bet you already placed. An index is a bet you placed at write time and have paid for on every write since. An idle tenant is a bet still costing you money — memory held, structures maintained, all of it hedging against a question that may never come.

We hold nothing warm. There's no cache to keep, no pool to fill, no index resident in memory. When a question arrives we open the panels we need, answer it, and let go. Between questions there is nothing to maintain.

> **PUT NUMBERS ON IT**
>
> 2.65 GB against their 4.30 GB means a 64 GB host holds **24 of our tenants where it holds 14 of theirs.** Same box, ten more customers on it, 38% less memory per tenant.

Which means bursty, many-tenant, mostly-idle — the actual shape of nearly everything anybody ships — stops being the hard case. A thousand tenants asking a question a minute don't need a thousand warm working sets. They need answers, and only for as long as somebody's asking.

SECTION 08

KEEPING AGENT SMITH IN LINE

LLMs have learned to reach for APIs and, increasingly, for databases. That's no surprise. Models need context, and context is data — yours.

But an LLM doesn't know about the schema Tom on your data team lovingly and meticulously submitted to Janine, who did her level best to give constructive feedback on the actual access patterns of the business software. It doesn't know that `events` is fine to scan but `events_archive` will take the box down. It doesn't know the composite index only helps if the predicates arrive in the right order, because that's something Tom learned in 2023 and told three people.

Sorry, Tom. Sorry, Janine.

LLMs have an idea and then they do it. Remember the early days of agents? Some serious hardware got wiped. Still does, now and then.

So how do we stop Agent Smith from giving you déjà vu at 3 AM, on-call, watching the same incident play out differently because something rewrote the query it ran yesterday?

Here's the thing about a runaway query: the damage isn't that it's wrong. It's that it's **unbounded**, and nobody knew until it was already running. A missing filter, an unbounded traversal, a pattern that fans out — the engine accepts all of it, pulls pages until the pool is full, evicts everything everyone else was using, and takes the instance down with it. One bad question, and every tenant on the box feels it.

Two things change here.

1. **A question has a knowable price before you answer it.** The plan is a matrix; the frontier is countable. Cost isn't something that emerges under load and surprises you — it's something the shape of the query tells you up front. You can refuse a question, bound it, or bill it before a single byte moves.
2. **There is nothing to poison.** No buffer pool to flood, no cache to evict, no plan cache to corrupt with one pathological shape. A greedy query is expensive for itself and invisible to everybody else, because there's no shared warm state for it to ruin.

> Agent Smith can multiply all he wants. **Every copy is on his own.**

SECTION 09

ENCRYPTED, END TO END

Everything is encrypted before it’s stored. Not the disk — the data.

LAYER	ALGORITHM
Section payload	AES-256-GCM, authenticated, per-section key and nonce
Key envelope	ML-KEM-768 — post-quantum KEM, FIPS 203
Passphrase → keypair	Argon2id (t=3, m=64 MiB, p=4)
Key wrap	AES key wrap, 8-byte recipient fingerprint

The cryptographic stack, in full.

Because position carries meaning, we only ever decrypt the spans we actually read. A query opens a few panels of the folded map, decrypts those, and never touches the rest. Bytes an agent isn’t authorized to see are bytes that **never became plaintext anywhere, in any buffer, at any point.** Structure stays legible; values do not — the shape of the graph is what we navigate, and the shape is not the secret.

Want to see it? Grep both files for anything that reads like English. Printable strings, eight characters or longer, first four megabytes:

FILE	PRINTABLE STRINGS IN FIRST 4 MB
`sf1v1.tetra`	**0**
Industry-leading provider, string store	**35,619**

Thirty-five thousand. Names, biographies, message bodies, sitting on the disk in plain English waiting for somebody to `cat` them.

So: a snapshot, a backup tape, a laptop somebody left somewhere, a mount that got configured wrong on a Friday. **Whoever finds it reads your graph. Finds ours, reads noise.**

And the bill for all of that? **0.9% to 1.8% of query CPU.** That’s it. That’s the whole price of at-rest encryption on this workload.

Which brings the agent problem home. You can price a question before you answer it. There's no shared warm state to poison. And whatever gets through can only decrypt what it was permitted to open. Blast radius isn't a policy enforced at the edge — it's a property of the substrate.

Every number in this paper came out of an engine doing that. The 317.7 MB file is encrypted. The 2.65 GB resident figure is encrypted. The throughput and the latency numbers are encrypted.

> There is no unencrypted mode. **This is the system.**

SECTION 10

WHY WE'RE A PUBLIC BENEFIT CORPORATION

We are told, constantly and expensively, that the future requires more data centers. More power. More water. More land, more grid, more concrete — that the buildout is simply what intelligence costs, and the only question left is who pays for it.

We think a great deal of that is inefficiency wearing a lab coat.

Our aim is to make compute **ten times** more efficient. Not ten percent — ten times. By refusing to move data that doesn't need moving. By refusing to keep things warm against questions nobody asked. By refusing to store what can be calculated.

Do that across enough of the stack and the arithmetic of the buildout changes completely. The infrastructure that already exists starts to look less like a constraint and more like something we have barely learned to use.

That isn't a smaller future. It's a much larger one, running on what's already built.

You spend less. We build a business. And a great deal of hardware that was going to be poured in concrete never has to be.

You win. We win. The planet wins.

IN CLOSING

WHAT HAPPENS NEXT

We’re going to hang up the phone, and show you a world without them.

- Without a cloud bill that decides how much of your own data you can afford to ask about.
- Without a rack of expensive machines kept warm against questions nobody asked.
- Without index selection reviews, cache warmups, buffer pool tuning, and the quarterly audit of which forty indices are actually used.
- Without the 3 AM page because something rewrote a query and nobody could see the price of it until it was already running.

And without your data — or your customers’ data — ever sitting in the clear. Encrypted before it’s stored, decrypted only in the span being read, on cloud or on prem, the same either way.

Tom finally gets to sleep through the night.

We are **TETRA CA PBC**. We’re here to make data and compute simpler, cheaper, and considerably less painful than what you’re doing now.

EARLY ACCESS

You've already bent the spoon. Ready to go down the rabbit hole?

Our early-access program is a smaller group — teams who want to run TETRA against their own data, their own query shapes, their own bad Monday-morning workload, and tell us what breaks. Two minutes, three questions: your scale, your current engine, your worst query.

TELL US ABOUT YOUR WORKLOAD

tetra-ca.com

inquiries@tetra-ca.com

Full benchmark report — methodology, per-query breakdown, test rig, and reproduction commands — available on request.

TETRA CA PBC