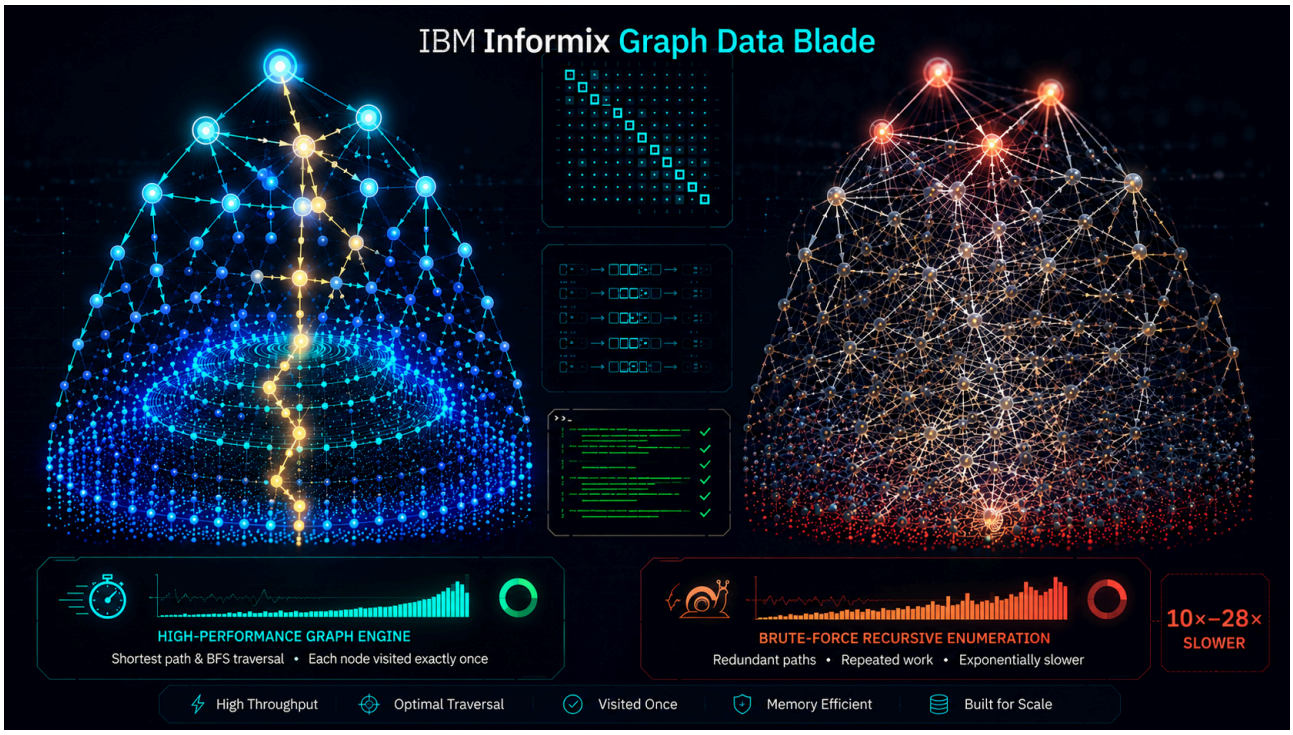
ifxtools
Document Version: 1.0 – 2026-08-13

Informix Graph DataBlade

Airtool

Content



This document describes an enterprise **graph-traversal DataBlade** for IBM Informix: a set of C UDRs that build an in-memory **CSR** (compressed-sparse-row) snapshot of an edge table and answer reachability, where-used and weighted shortest-path queries with a single visit-once traversal. It explains the class of problem these UDRs solve that **plain SQL cannot express and recursive SQL cannot execute economically**, and benchmarks the blade against two references: PostgreSQL **recursive CTEs** and the **Apache AGE** openCypher property-graph engine.

Every result in this document is produced by a reproducible, **three-way verified** harness — a Java reference (textbook BFS / Dijkstra over the same seeded data) is asserted equal to the Informix blade and to each comparison engine before any timing is accepted. The benchmark code is the source of truth for the tables shown here.

> **Headline result.** On a real x86 Informix server over a 46,750-node Bill-of-Materials DAG, the blade answers a depth-5 explosion in **26 ms** where the equivalent recursive CTE must enumerate **111,110 paths** to reach the same 27,736 parts (PostgreSQL 48 ms, Informix CTE 559 ms), and a **weighted** shortest-path in **5.7 ms** versus 39–440 ms for enumerate-and-MIN. Against Apache AGE, the blade matches openCypher exactly on neighbours, degree, reachability and connectivity, and provides a **weighted shortest path that openCypher has no native operator for**.

1. Motivation — Graphs Hiding Inside Relational Data

A great deal of enterprise data is a graph wearing a relational costume. A **Bill of Materials** is a directed acyclic graph of assemblies and components; an **org chart**, a **chart of accounts**, a **network topology**, a **document/permission hierarchy**, a **supply chain** and a **transaction graph** (for fraud-ring detection) are all edges between rows. The recurring questions are traversal questions:

- **Explosion** — given a finished product, what is the complete set of components it contains, to any depth? (procurement, costing, recall scope)
- **Where-used** — given a raw part, which finished products are affected if it changes or is recalled? (impact analysis — the reverse traversal)
- **Reachability / connectivity** — can A reach B at all, and within how many hops?
- **Cheapest path** — the minimum-cost route through a weighted network (sourcing, routing, settlement).

Keeping these traversals **inside the database**, next to the business data, avoids exporting the graph to a separate engine on every query, preserves transactional consistency, and lets a single SQL statement combine a traversal with ordinary relational filters (tenant, effective date, status).

1.1. System architecture

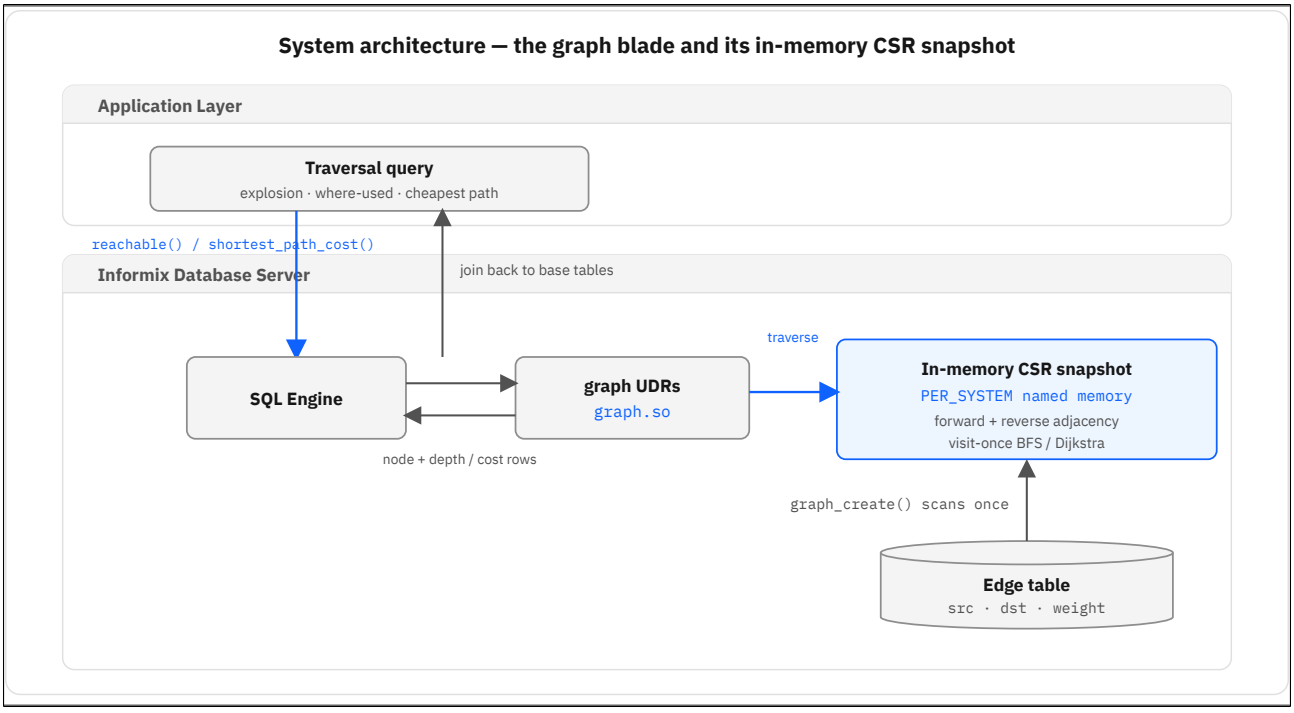


Figure 1 — System architecture — the graph blade and its in-memory CSR snapshot

2. The Problem Plain and Recursive SQL Cannot Solve Economically

2.1. Plain SQL cannot express transitive closure

A single `JOIN` reaches one level. Reaching "all components at any depth" requires **transitive closure**, which relational algebra cannot express without iteration. Before recursive CTEs, applications resolved this with a client-side loop — one round-trip per level — which is both slow and pushes graph logic out of the database.

2.2. Recursive SQL can express it, but enumerates paths — not nodes

A recursive CTE (`WITH` on Informix, `WITH RECURSIVE` on PostgreSQL) does express transitive closure. But its execution model **extends every partial path independently**: it has no global "visited" set, so a node reachable by `k` distinct paths is expanded `k` times. On a graph with sharing — exactly what a BOM is, where many assemblies reuse the same fastener — the work grows as **fan-out^{depth}**, not as the number of nodes.

> **Measured blow-up (46,750-node BOM, out-degree 10).** Reaching the same set of parts, the recursive CTE enumerates ever more redundant paths as depth grows:
>
> - depth 3 → **720 parts** reached via **1,110 paths**
> - depth 4 → **4,265 parts** via **11,110 paths**
> - depth 5 → **27,736 parts** via **111,110 paths** (≈ 4× redundant work, and rising)
>
> A real 20–40-level BOM makes path enumeration explode super-linearly while the part count is bounded by the catalogue. The blade visits each part **once** — work is bounded by `nodes + edges` regardless of how many paths reach them.

2.3. Weighted shortest path is worse in SQL — and absent in openCypher

A recursive CTE has no priority queue, so the cheapest weighted path can only be found by **enumerating every path and taking the MIN of the summed weights** — the explosion above, plus an aggregation. Apache AGE's openCypher supports `unweighted` shortest paths but, in the tested release, has **no native weighted shortest-path** construct (its `reduce()` path-folding is unsupported). The blade runs a textbook **Dijkstra** over the CSR snapshot — a single bounded traversal with a binary heap.

3. The Graph DataBlade — CSR Snapshot and Visit-Once Traversal

`graph_create()` scans an ordinary edge table once and materialises a **compressed-sparse-row (CSR)** adjacency structure in `PER_SYSTEM` named shared memory, keyed by a graph name. The CSR layout stores all neighbours contiguously with a per-node offset index — cache-friendly and allocation-free at query time. Traversal UDRs then operate purely in memory:

- `reachable(graph, start, maxDepth)` — breadth-first search with a visited bitmap; returns each reachable node once with its depth.
- `shortest_path_cost(graph, a, b)` — Dijkstra with a binary heap over the edge weights; returns the minimum total cost.
- `neighbors`, `degree`, `is_reachable`, `shortest_path` — direct CSR lookups and traversals.

A second snapshot built on the reversed edge list (`graph_create(..., 'dst','src',...)`) answers **where-used** queries as a forward traversal on the reverse graph.

3.1. Function reference

Table 1 — Graph DataBlade UDRs

Function	Signature	Returns	Purpose
`graph_create`	(name, table, srcCol, dstCol, wCol, directed)	integer (edges)	Build the CSR snapshot from an edge table.
`graph_refresh`	(name)	integer	Rebuild the snapshot after the edge table changes.
`graph_drop`	(name)	integer	Free the snapshot's named memory.
`graph_stats`	(name)	row	Node/edge counts and memory footprint.
`neighbors`	(name, node)	set of (node, weight)	Direct out-neighbours.
`degree`	(name, node)	integer	Out-degree.
`reachable`	(name, start, maxDepth)	set of (node, depth)	Depth-bounded reachable set (BFS, visit-once). maxDepth 0 = unbounded.
`is_reachable`	(name, a, b)	boolean	Connectivity test.
`shortest_path`	(name, a, b)	set of (seq, node, cost)	The cheapest weighted path, node by node.
`shortest_path_cost`	(name, a, b)	smallfloat	Minimum total weighted cost (Dijkstra).

4. Graph Analytics — In-Engine Algorithm Catalogue

Traversal answers "what is connected to what"; **graph analytics** answers the questions that carry the business value — `who is influential`, `what are the natural communities`, `who is structurally central`, `which entities are alike`. This is exactly the layer that **Neo4j GDS** and **Oracle Graph (PGX)** monetise, because these iterative algorithms are painful or impossible to express in SQL. The blade ships a first catalogue of them as set-returning UDRs that run **directly against the same CSR projection** built by `graph_create` — no separate analytics engine, no ETL to a graph database.

> **Same projection, new algorithms.** The hard part — turning relational edge rows into a compact in-memory adjacency structure — is already paid for by the traversal layer. Analytics kernels are pure iteration over that CSR, so the blade behaves like a miniature in-engine GDS. Two result row types are added: `graph_score_t (node BIGINT, score FLOAT)` and `graph_community_t (node BIGINT, community BIGINT)`.

4.1. Function reference

Table 2 — Graph analytics UDRs (worked in full in the Examples section)

Function	Signature	Returns	Purpose
`pagerank`	(name, damping, iters)	set of (node, score)	PageRank influence; damping NULL→0.85, iters≤0→20; scores sum to 1.
`eigenvector`	(name, iters)	set of (node, score)	Eigenvector centrality (power iteration; top = 1.0); iters≤0→100.
`closeness`	(name)	set of (node, score)	Closeness centrality (higher = more central).
`betweenness`	(name)	set of (node, score)	Brandes betweenness (unweighted) — brokerage / bottlenecks.
`wcc`	(name)	set of (node, community)	Weakly connected components (label = smallest member id; direction-agnostic).
`louvain`	(name, resolution)	set of (node, community)	Louvain modularity communities; resolution NULL/≤0→1.0. Splits a connected graph.
`label_propagation`	(name, iters)	set of (node, community)	Label-Propagation communities (iters≤0→10; deterministic per snapshot).
`sssp`	(name, src)	set of (node, score)	Single-source shortest distances (`score` = weighted distance).
`node_similarity`	(name, node, topk)	set of (node, score)	Top-k Jaccard-similar nodes (`score` # [0,1]).

Function	Signature	Returns	Purpose
node_similarity_weighted	(name, node, topk)	set of (node, score)	Top-k cosine similarity over weighted neighbour vectors.

Community and centrality functions assume an **undirected** projection (graph_create(..., 'f')). Result row types: graph_score_t (node BIGINT, score FLOAT) and graph_community_t (node BIGINT, community BIGINT) .

Scope and cost. Like GDS/PGX in-memory mode, analytics run on the resident CSR — sized for millions of edges, not billions — and on a single VP. PageRank/WCC/SSSP/Louvain are near-linear per pass; closeness and betweenness are O(V·(V+E)) (a traversal per node), intended for moderate graphs. Community and centrality kernels (Louvain, Label Propagation, betweenness, eigenvector) assume an **undirected** projection. Betweenness is computed over unweighted (hop-count) shortest paths.

5. Examples

Three complete, self-contained enterprise cases — each with its full dataset, the questions it answers, and the actual engine output. Every result below is reproduced by a JUnit test that loads the same data, so the numbers are real, not illustrative.

5.1. Bill of materials — manufacturing (DAG explosion & where-used)

A classic enterprise hierarchy: an e-bike (part 100) and the parts it is built from. The edge table `bom_edges` holds parent→child links with a per-edge quantity; the same table, registered in reverse, answers "where is this part used?". Every result below is reproduced by the project's automated test suite on a live engine.

```
CREATE TABLE parts (part_id BIGINT PRIMARY KEY, part_code VARCHAR(32));
INSERT INTO parts VALUES (100, 'EBIKE-X1');
INSERT INTO parts VALUES (110, 'FRAME-ASSY');
INSERT INTO parts VALUES (111, 'FRAME-TUBE-SET');
INSERT INTO parts VALUES (112, 'PAINT-KIT');
INSERT INTO parts VALUES (120, 'DRIVE-ASSY');
INSERT INTO parts VALUES (121, 'MOTOR-250W');
INSERT INTO parts VALUES (122, 'CONTROLLER');
INSERT INTO parts VALUES (123, 'BATTERY-PACK');
INSERT INTO parts VALUES (124, 'CELL-21700');
INSERT INTO parts VALUES (130, 'WHEEL-ASSY');
INSERT INTO parts VALUES (131, 'RIM-27.5');
INSERT INTO parts VALUES (132, 'SPOKE-SET');
INSERT INTO parts VALUES (133, 'HUB');
INSERT INTO parts VALUES (900, 'BOLT-M5');

CREATE TABLE bom_edges (src BIGINT, dst BIGINT, qty SMALLFLOAT, PRIMARY KEY (src, dst));
INSERT INTO bom_edges VALUES (100, 110,  1);
INSERT INTO bom_edges VALUES (100, 120,  1);
INSERT INTO bom_edges VALUES (100, 130,  2);
INSERT INTO bom_edges VALUES (110, 111,  1);
INSERT INTO bom_edges VALUES (110, 112,  1);
INSERT INTO bom_edges VALUES (110, 900, 12);
INSERT INTO bom_edges VALUES (120, 121,  1);
INSERT INTO bom_edges VALUES (120, 122,  1);
INSERT INTO bom_edges VALUES (120, 123,  1);
INSERT INTO bom_edges VALUES (120, 900,  8);
INSERT INTO bom_edges VALUES (121, 900,  4);
INSERT INTO bom_edges VALUES (123, 124, 40);
INSERT INTO bom_edges VALUES (130, 131,  1);
INSERT INTO bom_edges VALUES (130, 132, 36);
INSERT INTO bom_edges VALUES (130, 133,  1);
INSERT INTO bom_edges VALUES (133, 900,  6);

-- forward graph (explosion) and reverse graph (where-used), built once
EXECUTE  FUNCTION  graph_create('bom',           'bom_edges',  'src',  'dst',  'qty',
 't'::BOOLEAN);
EXECUTE   FUNCTION   graph_create('bom_used',   'bom_edges',   'dst',   'src',   'qty',
 't'::BOOLEAN);
```

```
┌──────────────┐
│ (expression) │
├──────────────┤
│           16 │
└──────────────┘

16 edges in each direction.
```

5.1.1. Explosion — every component of the product

```
SELECT p.part_id, p.part_code, t.r.depth
  FROM TABLE(FUNCTION reachable('bom', 100, 0)) t(r), parts p
 WHERE p.part_id = t.r.node AND t.r.depth > 0
 ORDER BY t.r.depth, p.part_id;
```

```
 part_id  part_code          depth
-------------------------------------
     110  FRAME-ASSY             1
     120  DRIVE-ASSY             1
     130  WHEEL-ASSY             1
     111  FRAME-TUBE-SET         2
     112  PAINT-KIT              2
     121  MOTOR-250W             2
     122  CONTROLLER             2
     123  BATTERY-PACK           2
     131  RIM-27.5               2
     132  SPOKE-SET              2
     133  HUB                    2
     900  BOLT-M5                2
     124  CELL-21700             3

 13 distinct components (Bolt M5 appears once, at its minimum depth 2).
```

5.1.2. Where-used — which assemblies contain a part (impact analysis)

```
SELECT p.part_id, p.part_code, t.r.depth
  FROM TABLE(FUNCTION reachable('bom_used', 900, 0)) t(r), parts p
 WHERE p.part_id = t.r.node AND t.r.depth > 0
 ORDER BY t.r.depth, p.part_id;
```

```
 part_id  part_code    depth
-----------------------------
     110  FRAME-ASSY       1
     120  DRIVE-ASSY       1
     121  MOTOR-250W       1
     133  HUB              1
     100  EBIKE-X1         2
     130  WHEEL-ASSY       2

 Bolt M5 is used by 6 assemblies — the recall/impact set.
```

5.1.3. Depth-bounded explosion and degree

```
-- direct sub-assemblies only (one level down)
SELECT p.part_id, p.part_code
  FROM TABLE(FUNCTION reachable('bom', 100, 1)) t(r), parts p
 WHERE p.part_id = t.r.node AND t.r.depth > 0
 ORDER BY p.part_id;
```

```
┌─────────┬─────────────┐
│ part_id │ part_code   │
├─────────┼─────────────┤
│     110 │ FRAME-ASSY  │
│     120 │ DRIVE-ASSY  │
│     130 │ WHEEL-ASSY  │
└─────────┴─────────────┘
```

```
EXECUTE FUNCTION degree('bom', 100);       -- the e-bike has 3 direct sub-assemblies
```

```
┌─────────────┐
│ (expression)│
├─────────────┤
│           3 │
└─────────────┘
```

```
EXECUTE FUNCTION degree('bom_used', 900);   -- Bolt M5 is used by 4 assemblies directly
```

```
┌─────────────┐
│ (expression)│
├─────────────┤
│           4 │
└─────────────┘
```

5.2. Logistics network — weighted shortest-path routing

A European distribution network: plants, hubs and DCs (nodes) joined by directed lanes whose weight is the shipping cost. The blade's in-memory Dijkstra answers the cheapest-route question that a recursive CTE can only approximate by enumerating every path. Every result below is reproduced by the project's automated test suite on a live engine.

```
CREATE TABLE locations (loc_id BIGINT PRIMARY KEY, name VARCHAR(16));
INSERT INTO locations VALUES (1, 'BCN-PLANT');
INSERT INTO locations VALUES (2, 'ZAZ-HUB');
INSERT INTO locations VALUES (3, 'LYO-HUB');
INSERT INTO locations VALUES (4, 'PAR-HUB');
INSERT INTO locations VALUES (5, 'FRA-HUB');
INSERT INTO locations VALUES (6, 'MUC-DC');
INSERT INTO locations VALUES (7, 'BER-DC');
INSERT INTO locations VALUES (8, 'MIL-HUB');

CREATE TABLE routes (src BIGINT, dst BIGINT, cost SMALLFLOAT, PRIMARY KEY (src, dst));
INSERT INTO routes VALUES (1, 2, 120);
INSERT INTO routes VALUES (1, 3, 300);
INSERT INTO routes VALUES (1, 8, 260);
INSERT INTO routes VALUES (2, 3, 140);
INSERT INTO routes VALUES (3, 4, 170);
INSERT INTO routes VALUES (3, 5, 210);
INSERT INTO routes VALUES (4, 7, 240);
INSERT INTO routes VALUES (5, 6, 110);
INSERT INTO routes VALUES (5, 7, 190);
INSERT INTO routes VALUES (6, 7, 150);
INSERT INTO routes VALUES (8, 5, 290);
INSERT INTO routes VALUES (8, 6, 320);

-- directed, weighted (lane cost in euros)
EXECUTE FUNCTION graph_create('routes', 'routes', 'src', 'dst', 'cost', 't'::BOOLEAN);
```

```
 (expression)
--------------
           12

12 directed lanes.
```

5.2.1. Cheapest route from the plant to Berlin

```
SELECT t.r.seq, p.loc_id, p.name, t.r.cost AS cum_cost
  FROM TABLE(FUNCTION shortest_path('routes', 1, 7)) t(r), locations p
 WHERE p.loc_id = t.r.node
 ORDER BY t.r.seq;
```

```
 seq | loc_id | name      | cum_cost
-----+--------+-----------+----------
   0 |      1 | BCN-PLANT |        0
   1 |      2 | ZAZ-HUB   |      120
   2 |      3 | LYO-HUB   |      260
   3 |      5 | FRA-HUB   |      470
   4 |      7 | BER-DC    |      660

cheapest route BCN→BER = €660 (the last cumulative cost).
```

```
EXECUTE FUNCTION shortest_path_cost('routes', 1, 7);   -- the scalar cost
```

```
 (expression)
--------------
       660.00
```

5.2.2. Reachability — how many sites the plant can supply

```
SELECT COUNT(*)
  FROM locations l
 WHERE l.loc_id <> 1 AND is_reachable('routes', 1, l.loc_id);
```

```
 (expression)
--------------
            7
```

5.3. Organisational collaboration network — graph analytics

The analytics catalogue applied in full to one network: who is influential, who is a bottleneck, what the natural teams are, and who works with similar people. Every result below is reproduced by the project's automated test suite on a live engine.

5.3.1. The example dataset — an organizational collaboration network

All examples below run against one small, self-contained graph: a company's **collaboration network**. Two teams of four — Team Alpha = {Ana 1, Ben 2, Cara 3, Dan 4} and Team Beta = {Eve 5, Finn 6, Gus 7, Hugo 8} — collaborate densely inside each team (3 hours/week between every pair), and a single thin tie links the two teams: Dan (4) and Eve (5) work together 1 hour/week. The edge weight is hours/week; the graph is undirected.

```
CREATE TABLE collab (a BIGINT NOT NULL, b BIGINT NOT NULL, hours SMALLFLOAT NOT NULL,
                     PRIMARY KEY (a, b));

-- Team Alpha {Ana 1, Ben 2, Cara 3, Dan 4} — every pair collaborates 3 h/week
INSERT INTO collab VALUES (1, 2, 3);
INSERT INTO collab VALUES (1, 3, 3);
INSERT INTO collab VALUES (1, 4, 3);
INSERT INTO collab VALUES (2, 3, 3);
INSERT INTO collab VALUES (2, 4, 3);
INSERT INTO collab VALUES (3, 4, 3);
-- Team Beta {Eve 5, Finn 6, Gus 7, Hugo 8} — every pair collaborates 3 h/week
INSERT INTO collab VALUES (5, 6, 3);
INSERT INTO collab VALUES (5, 7, 3);
INSERT INTO collab VALUES (5, 8, 3);
INSERT INTO collab VALUES (6, 7, 3);
INSERT INTO collab VALUES (6, 8, 3);
INSERT INTO collab VALUES (7, 8, 3);
-- the only cross-team tie: Dan (4) <-> Eve (5), 1 h/week
INSERT INTO collab VALUES (4, 5, 1);

-- Build the undirected, weighted in-memory projection once
EXECUTE FUNCTION graph_create('org', 'collab', 'a', 'b', 'hours', 'f'::BOOLEAN);
```

(expression)
13

13 collaboration edges loaded into the CSR snapshot.

5.3.2. Influence — PageRank and eigenvector centrality

Who matters most in the network? Both liaisons (Dan 4, Eve 5) carry the extra cross-team edge, so they rank highest; their six teammates are interchangeable.

```
SELECT t.r.node, t.r.score
  FROM TABLE(FUNCTION pagerank('org', 0.85, 100)) t(r)
 ORDER BY 2 DESC, 1;
```

node	name	score
4	Dan	0.150
5	Eve	0.150
1	Ana	0.117
2	Ben	0.117
3	Cara	0.117
6	Finn	0.117
7	Gus	0.117
8	Hugo	0.117

scores sum to 1.0; Dan(4) and Eve(5), the liaisons, rank highest.

```
SELECT t.r.node, t.r.score
  FROM TABLE(FUNCTION eigenvector('org', 200)) t(r)
 ORDER BY 2 DESC, 1;
```

node	name	score
4	Dan	1.000
5	Eve	1.000
1	Ana	0.918
2	Ben	0.918
3	Cara	0.918
6	Finn	0.918
7	Gus	0.918
8	Hugo	0.918

top score normalised to 1.0.

5.3.3. Centrality — who is the bottleneck

Closeness measures how near a node is to everyone else; **betweenness** counts the shortest paths that pass through it. Both single out the two liaisons — the people whose absence would most disconnect the company.

```
SELECT t.r.node, t.r.score
  FROM TABLE(FUNCTION closeness('org')) t(r)
 ORDER BY 2 DESC, 1;
```

node	name	score
4	Dan	0.318
5	Eve	0.318
1	Ana	0.206
2	Ben	0.206
3	Cara	0.206
6	Finn	0.206
7	Gus	0.206
8	Hugo	0.206

```
SELECT t.r.node, t.r.score
  FROM TABLE(FUNCTION betweenness('org')) t(r)
 ORDER BY 2 DESC, 1;
```

node	name	score
4	Dan	12.0
5	Eve	12.0
1	Ana	0.0
2	Ben	0.0
3	Cara	0.0
6	Finn	0.0
7	Gus	0.0
8	Hugo	0.0

Dan and Eve sit on every cross-team shortest path — the single points of failure.

5.3.4. Communities — connectivity, Louvain, and why Label Propagation differs

The graph is one connected component (the bridge links the teams), so `wcc` returns a single group. The interesting question is the `community` structure inside that component. **Louvain** recovers the two teams; plain **Label Propagation**, run on the same graph, collapses them into one — the precise reason Louvain is worth its extra cost.

```
-- Weakly connected components: one component, labelled by its smallest member id
SELECT t.r.community, COUNT(*) AS members
  FROM TABLE(FUNCTION wcc('org')) t(r)
 GROUP BY 1;
```

community	members
1	8

the whole company is one connected component.

```
-- Louvain: the two teams (each labelled by its smallest member id, 1 and 5)
SELECT t.r.community, COUNT(*) AS members
  FROM TABLE(FUNCTION louvain('org', 1.0)) t(r)
 GROUP BY 1 ORDER BY 1;
```

community	members
1	4
5	4

two teams, each labelled by its smallest member id (1 and 5).

```
-- Label Propagation merges both teams across the single bridge
SELECT COUNT(DISTINCT t.r.community) AS communities
  FROM TABLE(FUNCTION label_propagation('org', 20)) t(r);
```

communities
1

one giant community: plain LPA cannot resolve weakly-bridged groups.

5.3.5. Distance and similarity — SSSP, Jaccard and weighted cosine

Weighted shortest path turns "hours/week" into collaboration distance; similarity finds the people who work with the same colleagues. Distances from Ana (1): her teammates are 3 away, Eve is one 1-hour bridge further (4), and the rest of Team Beta is 7.

```
SELECT t.r.node, t.r.score AS distance
  FROM TABLE(FUNCTION sssp('org', 1)) t(r)
 ORDER BY 2, 1;
```

node	name	distance
1	Ana	0.0
2	Ben	3.0
3	Cara	3.0
4	Dan	3.0
5	Eve	4.0
6	Finn	7.0
7	Gus	7.0
8	Hugo	7.0

```
-- Who collaborates with the same people as Ana? (Jaccard of neighbour SETS)
SELECT t.r.node, t.r.score AS jaccard
  FROM TABLE(FUNCTION node_similarity('org', 1, 10)) t(r)
 ORDER BY 2 DESC, 1;
```

```
  node  name   jaccard
     2  Ben      0.500
     3  Cara     0.500
     4  Dan      0.400
     5  Eve      0.167

 Ben and Cara share two of Ana's three collaborators.
```

```
-- The weighted companion: cosine over hours-weighted collaboration vectors
SELECT t.r.node, t.r.score AS cosine
  FROM TABLE(FUNCTION node_similarity_weighted('org', 1, 10)) t(r)
 ORDER BY 2 DESC, 1;
```

```
  node  name   cosine
     2  Ben     0.667
     3  Cara    0.667
     4  Dan     0.655
     5  Eve     0.109
```

Every number above is asserted on a live Informix engine against this exact dataset by the project's automated test suite.

6. Benchmark 1 — Blade vs Recursive SQL (real x86 hardware)

MACHINE tier — Informix 14.10 and PostgreSQL 16 on separate LAN servers, over a BOM-shaped DAG of **46,750 nodes / 67,500 edges** (6 levels, out-degree 10). The dataset is our own deterministic generator; every result row is verified against a BFS/Dijkstra reference before timing.

Dataset. the synthetic DAG as a plain SQL file (table + `INSERT` s), ready to load and reproduce — synthetic-dag.sql.

6.1. Recursive CTE vs the blade — same answer, very different cost

Both compute the explosion; the recursive CTE **enumerates paths** (row count grows as fan-out^{depth}), while the blade does a **visit-once BFS** over the CSR (bounded by nodes + edges). They return the same 27,736 components — but the CTE materialises 111,110 path-rows to get there.

```
-- Recursive CTE  (Informix uses plain WITH; it ENUMERATES every path):
WITH reach (node, lvl) AS (
        SELECT dst, 1 FROM graph_bench_edges WHERE src = 100000
        UNION ALL
        SELECT e.dst, r.lvl + 1
          FROM graph_bench_edges e JOIN reach r ON e.src = r.node
         WHERE r.lvl < 5)
SELECT COUNT(DISTINCT node) FROM reach;
```

```
  count
  27736       -- correct, but 111,110 path-rows were materialised
```

```
-- Blade  (visit-once BFS over the CSR — each node processed exactly once):
SELECT COUNT(*)
  FROM TABLE(FUNCTION reachable('bench', 100000, 5)) t(r)
 WHERE t.r.depth > 0;
```

```
  components
       27736
```

Weighted shortest path is the sharper case: a CTE can only do it by enumerating `every` path and taking the minimum — combinatorial, not viable at depth — whereas the blade runs Dijkstra with a priority queue, which SQL recursion cannot express:

```
-- Recursive CTE: enumerate all paths, then MIN  (does not scale):
WITH paths (node, cost, lvl) AS (
        SELECT dst, w, 1 FROM graph_bench_edges WHERE src = 100000
        UNION ALL
        SELECT e.dst, p.cost + e.w, p.lvl + 1
          FROM graph_bench_edges e JOIN paths p ON e.src = p.node
         WHERE p.lvl < 6)
SELECT MIN(cost) FROM paths WHERE node = 631480;

-- Blade: one Dijkstra call:
EXECUTE FUNCTION shortest_path_cost('bench', 100000, 631480);   -- 462
```

Table 3 — Latency — recursive CTE vs the blade (synthetic 46,750-node DAG, MACHINE tier: IDS 14.10 + PostgreSQL 16 on LAN)

Query	PostgreSQL CTE	Informix CTE	Informix blade	Blade vs best CTE
Explosion depth 3 (720 nodes)	1.18 ms	7.85 ms	3.77 ms	0.3×
Explosion depth 4 (4,265 nodes)	5.03 ms	54.50 ms	**6.54 ms**	0.8×
Explosion depth 5 (27,736 nodes)	47.85 ms	558.52 ms	**26.19 ms**	**1.8× faster**
Where-used (117 products)	1.30 ms	4.64 ms	3.04 ms	0.4×
Weighted shortest-path cost	39.43 ms	440.46 ms	**5.73 ms**	**6.9× faster**

At shallow depth the set-based CTE wins on raw latency; the story is the **slope**. By depth 5 the blade overtakes even PostgreSQL's CTE (1.8×), and on the weighted shortest path it is ~7× faster than the best CTE and ~77× faster than Informix's own. Real BOMs run 20–40 levels deep, where path enumeration is not viable at all.

6.2. Execution and expected result

```
-- (a) Explosion: components of the product (100000) within 5 levels
SELECT COUNT(*) AS components
  FROM TABLE(FUNCTION reachable('bench', 100000, 5)) t(r)
 WHERE t.r.depth > 0;
```

```
 components
------------
      27736
```

```
-- (b) Where-used: assemblies containing the most-shared component (400149)
SELECT COUNT(*) AS assemblies
  FROM TABLE(FUNCTION reachable('bench_rev', 400149, 0)) t(r)
 WHERE t.r.depth > 0;
```

```
  assemblies
------------
         117
```

```
-- (c) Weighted shortest-path cost: product (100000) -> costliest leaf (631480)
EXECUTE FUNCTION shortest_path_cost('bench', 100000, 631480);
```

```
  (expression)
---------------
           462
```

7. Benchmark 2 — Informix vs Apache AGE (openCypher) parity

Apache AGE (openCypher on PostgreSQL) is the reference property-graph engine. The same DAG is loaded on the Informix side (the CSR snapshot) and into an AGE graph; each capability is asserted equal to the Java reference (**Informix = AGE = Java**) before timing. CONTAINER tier (local Docker, emulated — read relatively).

Dataset. the synthetic DAG as a plain SQL file (table + `INSERT` s), ready to load and reproduce — synthetic-dag.sql.

Table 4 — Capability parity — Informix vs Apache AGE (openCypher), on the synthetic DAG (CONTAINER tier, emulated — read relatively)

Capability	Informix UDR	AGE openCypher	Informix ms	AGE ms	Result (identical)
Out-neighbours	`neighbors()`	`MATCH (s)-[]->(m)`	4.55	1.40	10 nodes
Degree	`degree()`	`MATCH (s)-[r]->() RETURN count(r)`	2.28	0.80	10
Reachable, ≤ 3 hops	`reachable(...,3)`	`-[*1..3]->`	3.10	7.14	720 nodes
Connectivity	`is_reachable()`	`-[*]->`	4.34	1.79	reachable (t)
Weighted shortest cost	`shortest_path_cost()`	**no native operator**	2.32	—	Informix-only: 462

> **Verdict. Informix matches AGE exactly** on neighbours, degree, depth-bounded reachability and connectivity. AGE's strengths are openCypher and a native mutable property-graph model; Informix's is the **weighted shortest path**, which this AGE release cannot express natively — delivered on the Informix data in place, with no separate graph store.

7.1. Execution and expected result

Each capability run on the Informix side and as the equivalent Apache AGE openCypher (AGE runs Cypher through its `cypher()` table function), returning the identical result on the synthetic graph — node 100000 is the product. The rows below match the parity table one-to-one.

```
-- Out-neighbours
-- Informix:
SELECT COUNT(*) FROM TABLE(FUNCTION neighbors('bench', 100000)) t(r);
-- Apache AGE:
SELECT * FROM cypher('bench', $$ MATCH (s)-[]->(m) WHERE s.id = 100000 RETURN count(m)
 $$) AS (n agtype);
```

```
  count
-------
     10
```

```
-- Degree
-- Informix:
EXECUTE FUNCTION degree('bench', 100000);
-- Apache AGE:
SELECT * FROM cypher('bench', $$ MATCH (s)-[r]->() WHERE s.id = 100000 RETURN count(r)
 $$) AS (n agtype);
```

```
┌──────────────┐
│ (expression) │
├──────────────┤
│           10 │
└──────────────┘
```

```
-- Reachable within 3 hops (distinct nodes)
-- Informix:
SELECT COUNT(*) FROM TABLE(FUNCTION reachable('bench', 100000, 3)) t(r) WHERE t.r.depth
 > 0;
-- Apache AGE:
SELECT * FROM cypher('bench', $$ MATCH (s)-[*1..3]->(m) WHERE s.id = 100000 RETURN
 count(DISTINCT m) $$) AS (n agtype);
```

```
┌───────┐
│ count │
├───────┤
│   720 │
└───────┘
```

```
-- Connectivity: is there any path s -> t ?
-- Informix:
EXECUTE FUNCTION is_reachable('bench', 100000, 631480);
-- Apache AGE:
SELECT * FROM cypher('bench', $$ MATCH (s)-[*]->(t) WHERE s.id = 100000 AND t.id =
 631480 RETURN count(*) > 0 $$) AS (ok agtype);
```

```
┌──────────────┐
│ (expression) │
├──────────────┤
│            t │
└──────────────┘
```

```
-- Weighted shortest-path cost
-- Informix:
EXECUTE FUNCTION shortest_path_cost('bench', 100000, 631480);
-- Apache AGE: no native weighted shortest-path operator in this release — it must
 enumerate paths.
```

```
┌──────────────┐
│ (expression) │
├──────────────┤
│          462 │
└──────────────┘
```

8. Benchmark 3 — Real Road Network (DIMACS) Weighted Shortest Path

Benchmarks 1 and 2 use a controlled synthetic DAG to isolate the depth-vs-slope effect. This third benchmark is the **real-world credibility case**: weighted shortest path on a genuine, large public graph — a road network from the **9th DIMACS Implementation Challenge** (distance graph) — which is exactly the operation a recursive SQL CTE cannot do economically. Every queried pair's cost is asserted **equal to an in-JVM Dijkstra reference**, exact to the unit (road distances are integers and the totals stay below 2^{24}, so the blade's arithmetic is exact).

Dataset. the public DIMACS New York distance graph — original DIMACS source (regions BAY, COL, …, USA at the same location), or the ready-to-load roadnetwork-ny.sql.

Table 5 — New York road network — 264,346 nodes / 733,846 arcs (CONTAINER tier, emulated — indicative)

Step	Measurement
Load 733,846 arcs (JDBC batch INSERT)	~10.4 s (one-time)
`graph_create` — build the CSR snapshot	~7.9 s (one-time)
shortest_path_cost(1 → 90644), path cost **1,440,081 m**	**~53 ms** (mean of 5)
Correctness	blade cost = Java Dijkstra cost, **exact** (3 pairs)

> **Real-world verdict.** The blade answers an arbitrary point-to-point cheapest route across the entire state — a 1,440 km path over 264 k intersections — in **tens of milliseconds**, on the Informix data in place, and exactly matching the Dijkstra ground truth. A recursive-CTE weighted shortest path over 733 k arcs is not viable; this is the capability gap the blade closes.

8.1. Execution and expected result

```
-- cheapest route across New York: intersection 1 -> intersection 90644
EXECUTE FUNCTION shortest_path_cost('road', 1, 90644);
```

```
(expression)
1440081
```

9. Equivalent or Capability? When the Blade Is Actually Required

It is worth being blunt about this, because it decides whether the blade belongs in a given system at all. For a large class of queries the recursive CTE and the blade return the **identical answer** — the blade is merely faster, and on a small or shallow graph "faster" means a few milliseconds nobody will notice. **Do not adopt the blade for a few milliseconds.** The blade earns its place only where one of two things happens: the CTE **degrades past the point of usefulness** as the graph scales, or the CTE **cannot produce a correct answer at all**. Those are different thresholds, and it matters which one you are facing.

9.1. Three regimes

Table 6 — When are blade and recursive CTE equivalent, and when is the blade required?

Regime	Graph shape	CTE result	Recommendation
A — Equivalent	Small or shallow (a few thousand nodes, ≤ 3–4 levels), acyclic	Same answer, both in single-digit ms	**Use the CTE.** Simpler, always-current, no snapshot to maintain. The blade gain is noise.
B — Feasibility	Deep or highly-shared acyclic (real BOMs, 10–40 levels)	Same answer, but work grows as fan-out^{depth} until the query no longer completes	**Blade for scale.** Equivalent results; the blade keeps the query runnable where the CTE times out / exhausts memory.
C — Capability	**Cyclic** graphs, and weighted shortest-path / cost on any non-trivial graph	Cannot terminate unbounded; with a hop bound it is **wrong** below the optimal hop count and explodes at it	**Blade required.** Only a settled-set + priority-queue traversal (Dijkstra) is correct and bounded. SQL recursion structurally cannot do this.

9.2. The capability boundary, measured — weighted shortest path on a cyclic graph

A random **cyclic** weighted digraph (400 nodes, out-degree 4, 1,600 edges; cycles are unavoidable with random back-edges). The true shortest path from node 0 to the farthest reachable node is **cost 59, 9 hops** (Java Dijkstra ground truth). The blade runs Dijkstra directly; the recursive CTE has no priority queue and no settled set, so it must enumerate every **walk** up to a hop bound H and take the minimum cost reaching the target. Measured on Informix 14:

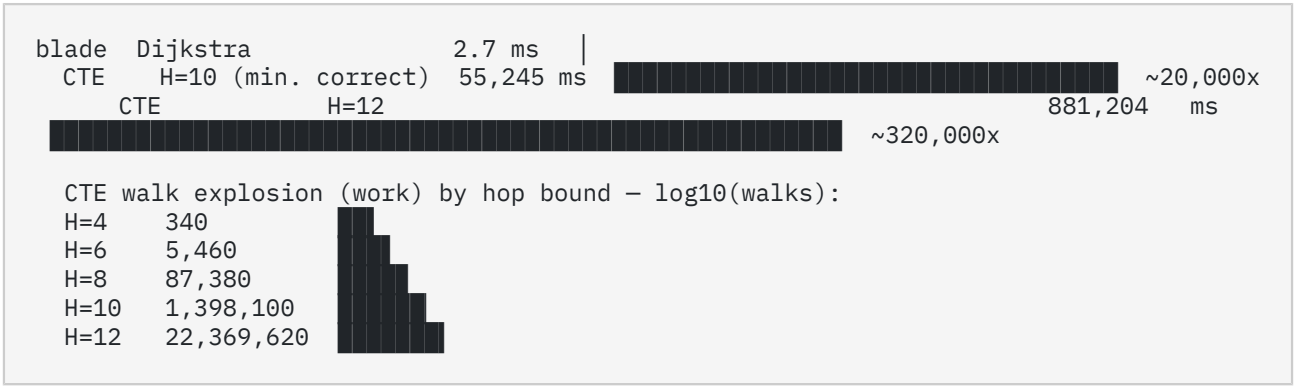
Table 7 — Cyclic-graph shortest path — blade Dijkstra vs hop-bounded recursive CTE

Method	Walks enumerated	Time	Result vs true optimum (59)
Blade — Dijkstra	— (visits each node once)	**2.7 ms**	**59 — correct, cycle-immune**
CTE — hop bound H=4	340	30 ms	target unreached (H too small)
CTE — hop bound H=6	5,460	216 ms	target unreached
CTE — hop bound H=8	87,380	3.4 s	**WRONG — 63** (a longer detour; optimum needs 9 hops)
CTE — hop bound H=10	1,398,100	**55 s**	59 — correct, but ≈ 20,000× slower

Method	Walks enumerated	Time	Result vs true optimum (59)
CTE — hop bound H=12	22,369,620	**14.7 min**	59 — correct, ≈ 320,000× slower
CTE — unbounded	∞	never terminates	— (infinite walks around cycles)

Two failures, not one. Below the optimal hop count (H=8) the CTE is not merely slow — it returns the **wrong cost** (63), because a hop-limited walk search has not yet found the 9-hop optimum. To become correct it must reach H≥9, and the walk count grows ~ degree^H : 340 → 5,460 → 87,380 → 1.4M → 22.4M. By the time it is correct it is 4–6 orders of magnitude slower than the blade, and on an unbounded cyclic graph it cannot terminate at all. This is the line between "slower" and "cannot".

Time to the correct answer (cost 59) — note the axis would need a log scale to even show the blade's bar:



9.3. Choosing the approach

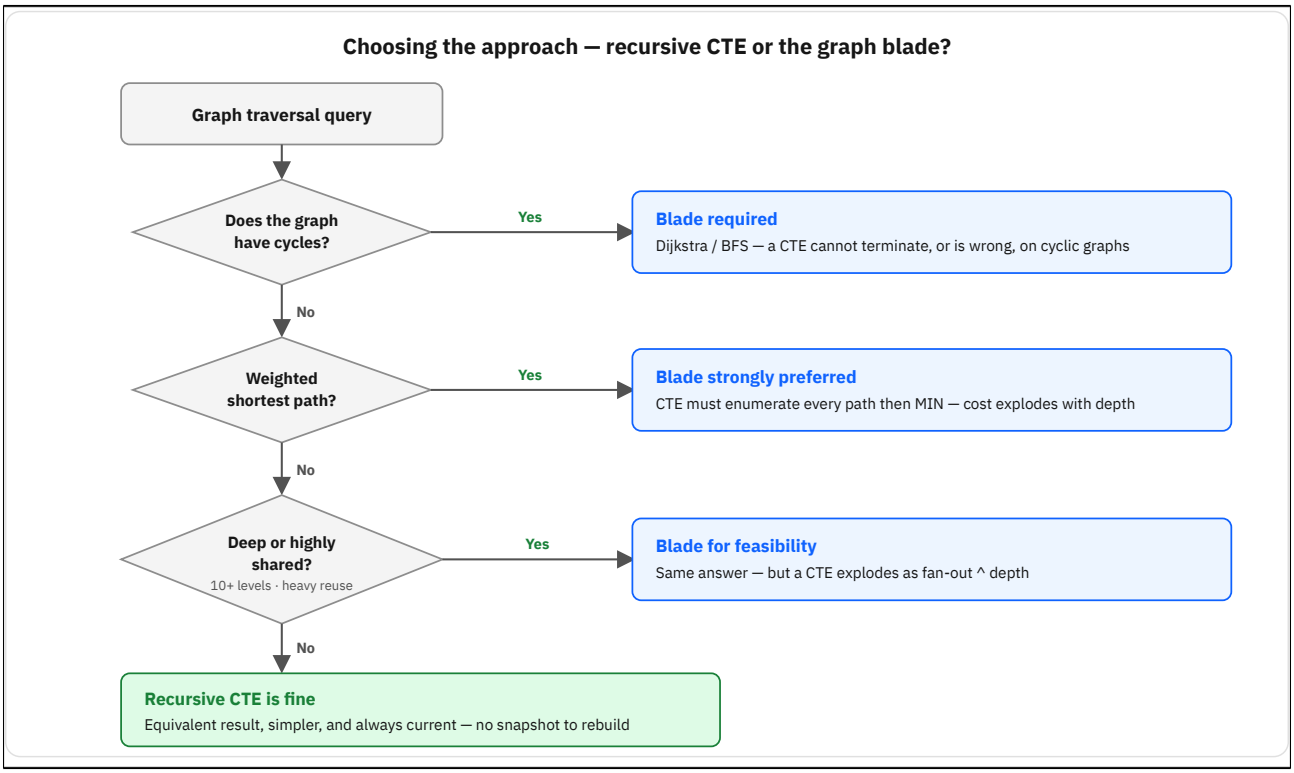


Figure 2 — Choosing the approach — recursive CTE or the graph blade?

9.4. Real-world cases that fall on the capability side

- **Logistics / routing** — road, rail or network paths are cyclic and weighted; cheapest/fastest route is Dijkstra. A CTE cannot express it.
- **Payment netting & settlement** — inter-account transfer graphs contain cycles; minimum-cost settlement paths require weighted shortest path.
- **Fraud / AML ring detection** — the question `is` "is there a cycle back to the origin?" — a cycle search, which SQL recursion can only do with explicit, expensive path-tracking guards.
- **Dependency & impact graphs** — microservice call graphs, build graphs and permission graphs routinely contain cycles; reachability and shortest-impact paths must be cycle-safe.
- **Deep Bill of Materials** — acyclic but 20–40 levels with heavy part reuse: the answer is the same as a CTE, but only the blade finishes (Regime B).

9.5. Why a graph engine (Apache AGE) exists at all

The same logic explains why Apache AGE — a whole graph database bolted onto PostgreSQL — exists despite PostgreSQL already having recursive CTEs. If the CTE could do everything, AGE would be redundant. It is not, because a recursive CTE is a **set-based fixpoint iteration over paths**; it lacks the primitives a graph traversal needs:

- a **settled / visited set** — so it cannot visit each node once, and cannot terminate on cycles without explicit, costly bookkeeping;
- a **priority queue** — so it cannot expand the cheapest frontier first, i.e. cannot run Dijkstra / A* (weighted shortest path);

- **graph pattern matching** — variable-length and shaped patterns (`(a)-[:OWNS*]->(b)<-[:OWNS]-(c)`) that don't map to relational joins.

AGE solves this with a native, mutable property-graph store and the openCypher language. The Informix graph blade solves the **traversal** half of the same problem — it brings the missing primitives (visited set, priority queue, BFS/Dijkstra) to the data **in place in Informix**, with no separate graph store to provision, synchronise or secure. And on the one capability AGE's openCypher lacks — **weighted** shortest path — the blade is ahead of AGE as well (see Benchmark 2). What AGE additionally offers, and the blade does not, is the declarative Cypher language, a live mutable graph, and whole-graph analytics; those are the reasons to choose a dedicated graph engine over an in-database traversal blade.

10. Architecture Comparison

Table 8 — Three approaches to graph traversal in a relational platform

Dimension	Informix graph blade	Recursive CTE (SQL)	Apache AGE
Model	In-memory CSR snapshot of an edge table	None — re-derives on each query	Native stored property graph
Traversal cost	O(nodes + edges) — visit once	O(fan-out^{depth}) — path enumeration	Path-pattern; var-length can enumerate
Weighted shortest path	**Yes** (Dijkstra)	Enumerate + MIN (explodes)	No native operator
Mutability	Read-only snapshot; `graph_refresh()` to rebuild	Always live (reads base table)	Live mutable graph (Cypher CREATE/SET)
Query language	SQL function calls	Standard SQL	openCypher
Data location	In the Informix database	In the database	Separate PostgreSQL/AGE store
Best for	Repeated deep traversals over a stable graph	Shallow, ad-hoc, always-current	Cypher-native graph applications

11. Limitations

11.1. Snapshot freshness

The CSR snapshot is a point-in-time copy in named memory. Edge-table changes are not seen until `graph_refresh()` rebuilds it. This is the right trade-off for graphs that are queried far more often than they change (BOM, org chart, network topology), but for a write-heavy graph the rebuild cadence must be managed. The recursive-CTE approach, by contrast, is always live (it reads the base table each time) at the cost of per-query work.

11.2. Memory residency

The snapshot lives in `PER_SYSTEM` shared memory and must fit alongside the server's other structures. `graph_stats()` reports its footprint; very large graphs should be sized against available shared memory before building.

11.3. Directed traversal from a single source

The UDRs traverse forward from a start node (use a reverse-edge snapshot for where-used). Whole-graph analytics (PageRank, community detection, all-pairs) are out of scope — this is a single-source traversal engine, not a general graph-analytics platform.

12. Compilation and Registration

The blade is C compiled to a shared object (`graph.so`) and registered as UDRs. In this project the compile happens **on the Informix server**: the install tooling (`InformixBladeModule`) streams `graph.c` over JDBC, writes it via `LOTOFILE` , and invokes gcc through an SPL `SYSTEM` call, then runs the registration SQL. The server therefore needs a C compiler on its `PATH` . The steps below are the manual equivalent.

12.1. Compiling the shared object

Standard DataBlade C-UDR compile: a position-independent shared object with the Informix public headers on the include path. Use `-march=native` only on the exact CPU that will run the engine.

```
gcc -O3 -fPIC -shared graph.c -I$INFORMIXDIR/incl/public -lm -o graph.so
```

Table 9 — Libraries and flags

Item	Why
`-fPIC -shared`	Dynamically loadable shared object (required for EXTERNAL NAME).
`-I$INFORMIXDIR/incl/public`	DataBlade API headers (`mi.h` , `milib.h`).
`-lm`	libm (math in the traversal/cost routines).
`-O3` (opt. `-march=native`)	Optimisation; native only on the engine's CPU.

12.2. Registering the routines

Each entry point is bound with `CREATE FUNCTION ... EXTERNAL NAME` , where the path placeholder is the directory of the compiled `graph.so` . Row-returning traversal functions are registered `WITH (ITERATOR)` (see `graph_install.sql`).

```
CREATE   FUNCTION   graph_create(LVARCHAR,LVARCHAR,LVARCHAR,LVARCHAR,LVARCHAR,BOOLEAN)
 RETURNING INTEGER
    EXTERNAL NAME '<sopath>/graph.so(graph_create)' LANGUAGE C;
CREATE FUNCTION reachable(LVARCHAR,BIGINT,INTEGER) RETURNING graph_reach_t
    WITH (ITERATOR) EXTERNAL NAME '<sopath>/graph.so(graph_reachable)' LANGUAGE C;
```

12.3. Deployment prerequisites

The target is an ordinary logged database; graph data is small (BIGINT ids, SMALLFLOAT weights), so no special page size is needed — only the default sbspace the routines reference.

```
-- 1. C compiler on the Informix server (e.g. microdnf install -y gcc).
-- 2. Default sbspace present (onstat -d shows an 'S' space).
-- 3. Create the logged database; graph.c is compiled on the server and the UDRs
 registered.
CREATE DATABASE graphdb WITH LOG;
```

13. Glossary — Graph and Traversal Terms

Terms specific to graph theory and traversal, for readers fluent in Informix but new to the graph domain. (Informix/DataBlade terminology is assumed known.)

Table 10 — Graph glossary

Term	Definition
Graph	A set of **nodes** (vertices) connected by **edges**. Here, edges are rows (src, dst, weight).
Directed graph	Edges have a direction (parent → child). Traversal follows edge direction.
DAG	Directed Acyclic Graph — a directed graph with no cycles. A BOM is a DAG (a part never contains itself).
Out-degree / in-degree	Number of edges leaving / entering a node. High in-degree marks a heavily shared component.
Neighbours	The nodes directly reachable from a node by one edge.
Path	A sequence of edges from one node to another. Two nodes may be joined by many distinct paths.
Reachability	Whether (and within how many hops) one node can be reached from another by following edges.
Transitive closure	The full set of nodes reachable from a start node at any depth — what "explode the BOM" computes.
Explosion	Forward transitive closure: all components contained in an assembly, recursively.
Where-used (implosion)	Reverse traversal: all assemblies that contain a given component — impact/recall analysis.
BFS	Breadth-First Search — explores level by level, marking nodes visited so each is processed once. The basis of `reachable()`.
DFS	Depth-First Search — explores one branch fully before backtracking.
Visited set	The bookkeeping (a bitmap here) that stops a traversal from re-processing a node — the very thing a recursive CTE lacks.
Weighted graph	Each edge carries a numeric weight (cost, distance, quantity). Shortest-path uses these.
Shortest path	The path between two nodes minimising total edge weight (weighted) or hop count (unweighted).
Dijkstra's algorithm	The classic single-source weighted shortest-path algorithm, using a priority queue to expand the cheapest frontier first.

Term	Definition
Priority queue / heap	The data structure that lets Dijkstra always pick the lowest-cost frontier node — absent from SQL recursion.
Adjacency list	Per-node list of outgoing edges — the natural in-memory graph representation.
CSR (compressed sparse row)	A compact adjacency representation: all neighbours in one contiguous array plus a per-node offset index. Cache-friendly and allocation-free at query time.
Fan-out	Average out-degree. Recursive-CTE path enumeration grows as fan-out^{depth}.
Recursive CTE	A SQL Common Table Expression that references itself (`WITH` / `WITH RECURSIVE`) — expresses transitive closure but enumerates paths.
Property graph	A graph model where nodes and edges carry key/value properties — the model of Apache AGE and openCypher.
openCypher / Cypher	The declarative graph query language used by Apache AGE (e.g. `MATCH (a)-[*1..3]->(b)`).
Variable-length path	A Cypher pattern matching paths of a range of lengths (`-[*1..3]->`) — the openCypher analogue of depth-bounded reachability.
Graph analytics	Whole-graph iterative algorithms (influence, communities, centrality, similarity) — the differentiated layer Neo4j GDS and Oracle PGX monetise, here run in-engine over the CSR.
PageRank	Node influence as the stationary distribution of a damped random walk; scores sum to 1, damping default 0.85.
Weakly connected component (WCC)	A maximal set of nodes connected when edge direction is ignored; computed by union-find, labelled by the smallest member id.
Community detection	Partitioning a graph into densely-connected groups; available as Label Propagation and Louvain.
Label Propagation	A near-linear community algorithm: each node repeatedly adopts the most frequent label among its neighbours.
Louvain	Modularity-maximising multi-level community detection: local moving then aggregation, repeated. Splits a connected graph where Label Propagation merges it.
Modularity	How much denser within-community edges are than chance (−1…1); the quantity Louvain maximises. The `resolution` parameter tunes granularity.

Term	Definition
Centrality	A measure of a node's structural importance; available as closeness, betweenness and eigenvector centrality.
Closeness centrality	(reach−1)/Σ distances from a node — high for nodes near everything else.
Betweenness centrality	Fraction of all shortest paths passing through a node (Brandes algorithm) — high for brokers, bottlenecks and single points of failure.
Eigenvector centrality	Influence defined recursively — a node is central if its neighbours are central (principal eigenvector of the adjacency matrix).
SSSP	Single-source shortest path — weighted distances from one node to every reachable node.
Jaccard similarity	|N(a)#N(b)| / |N(a)#N(b)| over neighbour sets — the basis of `node_similarity`.
Cosine similarity	Cosine of two nodes' weighted neighbour vectors (scale-invariant) — the basis of `node_similarity_weighted`.

14. License, Trademarks and Disclaimer

This DataBlade is a **proprietary, commercial software product** owned and published by **Airtool Technologies** (airtool.io). **All rights reserved.** It is licensed, not sold, and its use is governed by a commercial license agreement. No right to use, copy, modify, distribute, sublicense, or create derivative works is granted except as expressly permitted in a written license from Airtool Technologies. Unauthorized reproduction or distribution is prohibited. Contact info@airtool.io for licensing terms.

Purpose of this document. This document is provided for information only. It describes the product as at the date of publication, is subject to change without notice, forms no part of any agreement, and creates no representation, warranty or commitment. The terms governing any use of the software are those of the written license agreement between you and Airtool Technologies.

Warranty. Warranties, if any, are those expressly set out in that agreement. Except as expressly so provided, and to the maximum extent permitted by applicable law, the software is provided **"as is"**, and Airtool Technologies disclaims all other warranties and conditions, whether express, implied or statutory, including the implied warranties of merchantability, fitness for a particular purpose and non-infringement.

Limitation of liability. Except as expressly provided in that agreement, and to the maximum extent permitted by applicable law: neither party is liable for indirect, incidental, special, punitive or consequential damages, or for loss of profits, revenue, data or business, however caused; and each party's total aggregate liability is limited as set out in that agreement. Nothing in this document excludes or limits liability for death or personal injury caused by negligence, for damage to tangible property caused by negligence, for fraud or fraudulent misrepresentation, for wilful misconduct, or for any other liability that cannot be excluded by law.

Trademarks — no affiliation. IBM® and **Informix®** are trademarks or registered trademarks of International Business Machines Corporation; Informix is currently developed and distributed by HCL under license. This project is **independent** and is **not** affiliated with, sponsored by, or endorsed by IBM or HCL. Product names are used solely for identification and interoperability (nominative fair use); no third-party logos are distributed.

Commercial support. Licensing, production support (with SLAs), custom development and consulting are available from **Airtool Technologies** — airtool.io · info@airtool.io. Support entitlements are defined by your commercial license or support agreement. Bundled third-party components retain their own licenses.