Exploring Topographic Data with Transitive Closure on Graphs

Adam Nelson-Archer, Christoph F. Eick, and Carlos Ordonez

University of Houston, USA

Abstract. We present a lightweight recursive query framework in Python for evaluating slope-constrained reachability on terrain graphs. Recursive query evaluation is traditionally confined to database systems due to its complexity, but we show that database-style fixpoint semantics can be implemented effectively using Pandas. We define a reproducible pipeline that transforms Digital Elevation Models (DEMs) into directed graphs parameterized by a slope threshold, and evaluate transitive closure using a seminaive algorithm. We support correctness with an inductive proof sketch and empirical validation against a SQLite baseline, achieving zero discrepancy across all configurations. We demonstrate that varying the slope threshold induces a sharp structural phase transition in graph connectivity, shifting from fragmented components to a dominant giant component. This behavior enables the terrain graph to act as a controllable benchmark for recursive query evaluation, producing workloads of varying density from a single dataset. Beyond performance, we show that the resulting closure supports expressive queries that reveal meaningful topographical structure.

1 Introduction

Spatial measurements, such as elevation grids, sensor arrays, and GPS traces, are commonly stored as CSV files and analyzed in Python. CSV files load in any environment, and Pandas handles datasets up to roughly a million rows without stability issues. The challenge is methodological. Without a principled framework, Python analysis of graph-structured data often reduces to ad hoc traversal loops that produce results without a clear correctness guarantee or complexity analysis. This paper shows that applying database theory directly in Python on CSV files is a feasible, useful, and correct approach that remains practical for moderate-scale exploratory analysis without the overhead of deploying a full database system.

Transitive closure is a foundational problem that requires recursion. Recursive queries allow solving list problems, gradient descent, clique enumeration, and connectivity [1]. Datalog was designed with recursion in mind and remains the theoretical gold standard. We do not aspire to match Datalog, but we motivate extending Python with recursive-query capabilities by borrowing its fixpoint semantics to provide reproducible, database-style query evaluation.

The core engine leverages a seminaive evaluation strategy. By storing the directed edge relation as a Pandas DataFrame [8] and executing merge-based joins, the recursive iteration computes the transitive closure efficiently by eliminating redundant recomputation. This implementation provides a flexible environment for Python users without replacing a full relational database. We validate our implementation's correctness empirically by using a SQLite recursive view as a baseline oracle, confirming that our engine produces exact matches across all tested configurations.

A Digital Elevation Model (DEM) provides an ideal domain for evaluating this approach and generating benchmark workloads. A DEM represents terrain as a regular raster grid of elevation measurements. By constructing a directed graph where cells act as vertices and adjacent cells form edges, we create a network whose connectivity is governed by physical geometry. A slope passability threshold S acts as a query parameter filtering the allowable edges. Altering this threshold modifies which paths are traversable, allowing us to treat the terrain as a parameterized dataset without rewriting the underlying graph structure. The construction applies to any raster DEM; our experiments instantiate it on a USGS 3DEP crop of the Mount Rainier region [14].

Sweeping S over this terrain reveals a sharp structural phase transition near $S^* \approx 20°$. Below this threshold the graph fragments into hundreds of small isolated components; above it a giant mutually reachable component covers the majority of the terrain. This transition makes the terrain graph an exceptionally useful recursive-query benchmark: the workload moves from trivially small closures to near-quadratic output within a narrow 5° window, allowing researchers to generate workloads of arbitrary density in a controlled way.

Contributions

We define a reproducible pipeline from DEM to slope-filtered graph to transitive closure in Python, validate both graph construction and closure correctness, and show that varying the slope threshold changes workload structure in a controlled way. Specifically:

1. We define a formal DEM-to-directed-graph data model that treats the slope passability threshold as a parameter, giving a relation family applicable to any raster DEM. Changing this threshold predictably alters graph density, making the framework a tunable benchmark generator for evaluating recursive queries.
2. We show that the Pandas seminaive fixpoint loop provides a correct approach that is practical for moderate-scale datasets for evaluating recursive queries, establishing that reachability naturally increases monotonically as the slope constraints are relaxed.
3. We experimentally characterize both graph construction and closure evaluation. We demonstrate empirically that the Python engine matches SQLite on correctness at every tested threshold, and induces a sharp structural phase transition near 20° across two dataset scales (n = 1,000 and n = 10,000), confirming its utility as a controllable workload generator.

2 Related Work

Recursive query evaluation traces to Datalog and seminaive evaluation [12, 1]. Modern systems expose recursion via recursive views and CTEs [9, 15, 2], with optimization techniques spanning both row and column-oriented [5] systems. Relational evaluation relies heavily on efficient join strategies [7]. Ordonez [11] benchmarks direct vs. seminaive transitive closure in SQL across a full set of relational algebra operators, providing the closest comparison for our SQL vs. Pandas experiments. We apply the seminaive framework via Pandas [8]. Zhou et al. [16] compute transitive closure in Python using bitwise and block-based matrix algorithms, establishing precedent for non-DB evaluation strategies. Reachability queries have also been optimized using graph labeling schemes such as 2-hop labels [3].

Terrain analysis over Digital Elevation Models (DEMs) has been studied extensively in Geographic Information Systems (GIS) and hydrology. Flow-direction models like D8 [10] and D-infinity [13] define directed graphs over raster cells for drainage-basin delineation. Standard GIS software, such as GRASS GIS, utilizes algorithms to find optimal routes by incorporating anisotropic costs based on slope and terrain friction [6]. However, these traditional pathfinding methods are often optimized for single-source queries (e.g., A*) and require specialized algorithmic implementations to handle massive slope constraints [4].

3 Data and Problem Definition

3.1 Graph Definition

Let $G = (V, E)$ be a directed graph, where V is a set of $n = |V|$ vertices and $E \subseteq V \times V$ is a set of $m = |E|$ directed edges. Each vertex $v \in V$ corresponds to a distinct spatial location, and each directed edge $(u, v) \in E$ represents a traversable step from u to v. We consider the problem of evaluating recursive reachability queries over G. In our primary application, G is instantiated as a topographic terrain graph.

3.2 DEM Source

A Digital Elevation Model (DEM) is a regular raster grid where each cell records an elevation measurement in meters. By interpreting this grid as a graph, continuous "long" paths across the terrain are approximated by a sequence of short vertical, horizontal, and diagonal micro-segments between adjacent cell centers. To ensure our framework evaluates over realistic topography, all experiments instantiate the graph on crops of the USGS 3DEP 1/3 arc-second DEM [14] centered on Mount Rainier, WA (latitude 46.852° N, longitude −121.726° W). The spatial resolution is approximately 7.1 m/cell. We use a 32×32 crop ($n = 1{,}024$ valid cells) covering the summit region (elevation 2,989–3,178 m) for baseline behavior, and a 100×100 crop ($n = 10{,}000$ cells) for scale analysis. Fig. 1 shows the primary crop.

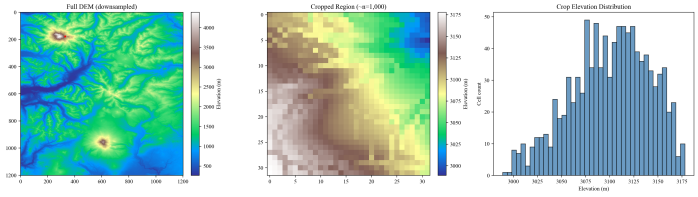


Fig. 1: USGS 3DEP DEM crop used for the n=1,000 experiments. The 32×32 tile covers the Mount Rainier summit region (elevation 2,989–3,178 m) at $r \approx 7.1$ m/cell.

3.3 Slope Threshold Concept

Terrain passability for travelers or vehicles is heavily constrained by the steepness of the terrain. We define the slope angle θ between two adjacent grid cells as the arctangent of the absolute elevation difference over the horizontal distance between the cell centers. A slope passability threshold, S (in degrees), determines which cell-to-cell transitions are feasible. Rather than creating a static dataset per slope threshold, we treat S as a query-time parameter. This parameter filters the base terrain edges, providing a predictable, tunable knob for generating evaluation workloads of varying densities from a single physical dataset.

3.4 Target Task: Slope-Constrained Reachability

The core computational task is slope-constrained reachability: identifying all pairs of terrain cells (u, v) such that v is reachable from u via a directed, continuous path where no single step exceeds the slope threshold S. This is equivalent to computing the transitive closure over the filtered edge set. By keeping the underlying DEM constant and varying S, we study how structural changes in the reachable terrain impact the performance and output size of recursive query evaluation.

4 Graph Construction

4.1 Vertex and Edge Construction

We define a formal DEM-to-directed-graph data model. The set of valid DEM cells instantiates the vertex set V. For each vertex, we inspect its eight grid neighbors (horizontal, vertical, and diagonal). For every valid adjacent cell, we emit a directed edge (u, v). The signed elevation difference dictates the edge's directionality (uphill or downhill), while the slope θ serves as the filtering criterion. The resulting graph is directed and asymmetric. The 8-connectivity model yields up to 8 outgoing edges per interior cell, guaranteeing that the edge count scales linearly with the vertex count ($m \leq 8n$). Because Δh is signed, every edge (u, v) has a reverse edge (v, u) with the negated elevation difference but the same unsigned slope θ.

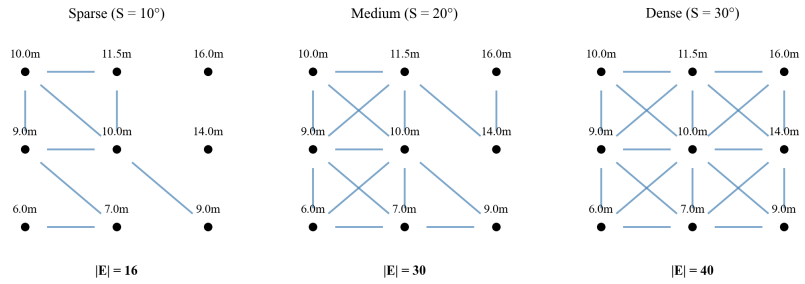


Fig. 2: Graph construction on a 3×3 grid. Raising the slope threshold S increases graph density while the underlying edge distance is fixed ($r = 10\,\text{m}$). Symmetric edge pairs are drawn as single lines for visual clarity.

Slope threshold example. To illustrate how the threshold S affects graph density, consider a 3×3 elevation grid with $r = 10\,\text{m}$. Fig. 2 shows the edges retained at $S \in \{10°, 20°, 30°\}$. At a strict $10°$ threshold, only paths along relatively flat contours are valid ($m_{10} = 16$). Relaxing the threshold to $20°$ incorporates moderately steep edges ($m_{20} = 30$), and at $30°$ the graph becomes fully connected along all 8-neighbor adjacency paths ($m_{30} = 40$). By tuning S, the framework serves as a dataset generator that produces graphs of arbitrary density over the exact same spatial grid.

4.2 Construction Invariants

To ensure the mapping from raster to graph is geometrically sound, the construction process must maintain strict structural invariants. We automatically validate four properties on the generated graph. First, for antisymmetry, every edge (u, v) must have a corresponding reverse edge (v, u) with exactly $\Delta h_{(v,u)} = -\Delta h_{(u,v)}$. Second, to prevent self-loops, no edge connects a vertex to itself. Third, for degree expectations, interior cells must have exactly 8 outgoing edges, and boundary cells must have 3 or 5 depending on their position (assuming a dense crop). Finally, for slope exactness, the slope angle θ must precisely match the trigonometric formula over the sampled DEM elevations. These invariants ensure the integrity of the graph before any recursive querying begins. Both the $n = 1{,}000$ and $n = 10{,}000$ graphs pass all checks with zero discrepancies.

4.3 Graph Statistics

The baseline n=1,000 instantiation has $m = 7{,}812$ edges (3,844 diagonal, 3,968 cardinal), mean out-degree 7.63, and weight distribution symmetric about zero (range $[-33.8, +33.8]\,\text{m}$). Fig. 3 shows the spatial edge structure and per-vertex slope map, highlighting the geometric complexity that the recursive evaluation framework must navigate.

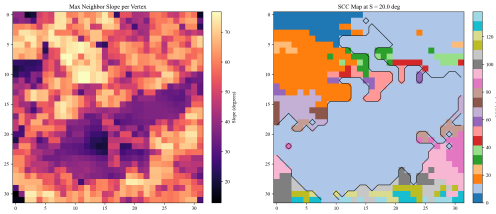


Fig. 3: Terrain graph for n=1,000: (Left) maximum slope θ per vertex, showing the rugged topography; (Right) strongly connected components (SCCs) at S=20°, illustrating the structural phase transition where disparate regions become mutually reachable.

4.4 Graph Generation Complexity

Because each cell inspects a constant number of neighbors (at most 8), the time to build the vertex set V and the base edge set E is strictly $O(n)$. The memory footprint is also $O(n)$, scaling linearly with the geographic extent of the cropped DEM. This linear construction step ensures that creating the benchmark graph from a CSV or raster source introduces minimal overhead relative to the subsequent recursive evaluation.

5 Recursive Evaluation Framework

While our experiments utilize a terrain dataset, this framework is general-purpose and can evaluate recursive queries over any directed graph. The system defines a strict order of operations to ensure correct and efficient evaluation. The framework takes as **input** a base graph (vertices V and directed edges E) alongside filtering constraints, such as a slope threshold S. The slope filtering is applied *first* to the base edges to produce the feasible edge set E_S. We explicitly define $G_S = (V, E_S)$ as the slope-filtered graph induced by E_S. The recursive computation then operates strictly over this filtered set E_S, without applying query-specific filters during the recursion. The **output** is the materialized directed transitive closure G_S^+, which explicitly lists every reachable source-destination pair. By defining the graph via filtering *before* recursion, we maintain the structural monotonicity required for the computation to converge.

5.1 Evaluation Strategies

The core operational mechanism for computing the slope-constrained reachability is *seminaive evaluation* (Algorithm 1). We define transitive closure G_S^+ as the least fixpoint of a monotone, inflationary operator over the base filtered edge set E_S. This approach avoids redundantly recomputing known paths. Here, TC represents the accumulated closure, Δ is the delta relation containing newly discovered paths, and each iteration derives only new tuples to avoid redundant

Algorithm 1 Seminaive Transitive Closure

Require: Edge relation $E(src, dst)$
1: $TC \leftarrow E$
2: $\Delta \leftarrow E$
3: **while** $\Delta \neq \emptyset$ **do**
4: $\quad New \leftarrow \Delta \bowtie E$
5: $\quad New \leftarrow New \setminus TC$
6: $\quad TC \leftarrow TC \cup New$
7: $\quad \Delta \leftarrow New$
8: **end while**
9: **return** TC

work. A fundamental property of most common recursive queries is that they exhibit linear tail recursion. This allows the recursive formulation to be "unfolded" and executed efficiently as a plain iterative loop. In our Pandas framework, the relation is stored as a DataFrame, and the merge-based join logic provides a lightweight implementation of this unfolded recursive evaluation.

We contrast this with a *naive evaluation* strategy, which joins the entire accumulated closure with E_S at every step. Naive evaluation performs massive amounts of redundant work. We include naive evaluation in our benchmarks solely as a baseline for comparison to highlight the necessity of the seminaive algorithm, which serves as the actual proposed evaluation method.

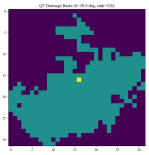
5.2 Correctness and Query Expressiveness

We establish the correctness of our seminaive evaluation through the following proposition.

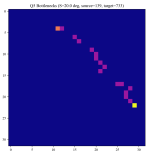
Proposition 1. *The seminaive evaluation converges to the least fixpoint because:*

1. *New tuples are added monotonically.*
2. *Duplicate tuples are removed.*
3. *The relation domain is finite.*
4. *Therefore the process terminates after finitely many iterations.*

Let F_k denote the frontier of newly discovered pairs at step k, and C_k denote the accumulated closure up to step k. As a base case, the initial frontier and closure are exactly the base edge set: $F_0 = E_S$ and $C_0 = E_S$. In the inductive step k, the algorithm joins the newly discovered frontier F_{k-1} with the base edge set E_S to extend paths by one hop, and retains only the novel pairs by removing those already in C_{k-1}, defining $F_k = (F_{k-1} \bowtie E_S) \setminus C_{k-1}$. The closure is then updated as $C_k = C_{k-1} \cup F_k$. By induction, C_k contains all reachable pairs of path length at most $k + 1$. Because the graph is finite and the operator is monotonic, the sequence $C_0 \subseteq C_1 \subseteq \cdots \subseteq C_k$ strictly increases until it must eventually converge to a least fixpoint G_S^+, where an iteration produces an empty frontier ($F_k = \emptyset$). We validate this theoretical correctness empirically by using a SQLite recursive view as a baseline oracle on identical edge sets, confirming exact matches on row outputs across all test thresholds.



(a) Q2: in-reach (drainage) basin of outlet t.



(b) Q5: bottleneck vertices (s–t choke points).

Fig. 4: Example query outputs on n=1,000 at S=20°.

Once the transitive closure G_S^+ is materialized, users can express complex queries using standard Pandas operations (filtering, joining, grouping) directly on the results. This makes the framework a flexible exploratory tool where queries are parameterized operations over DataFrames. The materialized relation effectively turns complex multi-hop path traversals into simple relational lookups and aggregations over the materialized closure. This approach works efficiently in Python because DataFrame operations inherently map to relational algebra: projections correspond to column selections, filtering maps to row selections, and recursive evaluation is fundamentally a series of iterative merge-based joins. By leveraging these optimized, vectorized operations in Pandas, the framework remains practical for moderate-scale datasets. Furthermore, because the underlying data structure is a standard tabular `DataFrame`, users can easily leverage Large Language Models (LLMs) to translate natural language questions into these executable operations.

From a theoretical standpoint, queries over the closure differ by their difficulty and relationship to monotonicity. Monotonic queries (where adding edges never invalidates previous results) are straightforward to answer using the materialized fixpoint, ensuring that evaluation converges efficiently. Non-monotonic queries, or those requiring intermediate structural information beyond reachability, break this property and are fundamentally harder. We present six representative queries in order of increasing complexity:

- **Q1: Point-to-point reachability** (simple filter for $u \to^+ v$ over G_S^+).
- **Q2: Drainage basin** (reverse reachability projection $\{u \in V \mid u \to^+ t\}$ from a sink t).
- **Q3: Hub detection** (join of $G_S^+(u, v)$ with base graph out-degrees for source u, plus aggregation).
- **Q4: SCC** (self-join $G_S^+(u, v) \bowtie G_S^+(v, u)$ to find equivalence classes $C \subseteq V$ such that $\forall u, v \in C$, $u \to^+ v$ and $v \to^+ u$).
- **Q5: Bottleneck detection** (non-monotonic path evaluation from s to t).
- **Q6: Shortest path** (requires distance-aware augmented recursion for $u \to^+ v$).

5.3 Example Topographical Outputs

Fig. 4 illustrates example outputs for Q2 and Q5 on the n=1,000 graph at $S = 20°$, showing how simple lookups translate into rich topographical maps.

6 Experiments

6.1 Experimental Setup

Experiments were conducted on a desktop machine (AMD Ryzen 7 2700X, 32 GB RAM, Windows 11). The implementation uses Python 3.12 with `pandas` 2.2 [8] for in-memory DataFrames, `numpy` for array operations, and `sqlite3` for baseline validation. SQLite was selected because it provides standards-compliant recursive query evaluation in a lightweight environment that can be directly compared against the Python implementation. All timing and correctness comparisons use SQLite as a baseline oracle.

6.2 Graph Construction Correctness and Performance

We first validate the graph generation process on the n=1,024 terrain crop (N = 1,024, m = 7,812). Our invariant checks confirm that the built graph contains exactly 0 missing reverse edges, 0 self-loops, and 0 degree mismatches (900 interior cells have exactly deg =8). The exact trigonometric slope formula on a sample of 500 edges produced a max absolute error of 9.4×10^{-7}, confirming numerical stability.

To demonstrate that the initial $O(N)$ construction overhead is manageable even at massive scale, we extrapolated the graph generation step on synthetic grids ranging from n = 10,000,000 to 1,600,000,000 cells (Table 1). Note that these billions of cells apply strictly to the graph construction step to prove its linear scalability. Because evaluating the full transitive closure generates near-quadratic output, the subsequent recursive evaluation and query experiments are conducted on the exact n=1,024 and n=10,000 terrain crops to ensure a feasible laptop-scale memory footprint. Building the vertex geometry remains in the single-digit seconds until $n \geq$ 40,000,000, while edge generation peaks at 7,535.7 seconds (about 2.1 hours) for 1.6 billion cells with roughly 1.3 billion edges. Peak memory increment stays proportional (28.8 GB for the largest tier), validating that the pipeline provides a manageable workload generator even for out-of-core scale graphs.

Table 1: Graph construction runtime and memory footprint across grid sizes. The purpose of this table is to understand the efficiency of the graph generation step and to verify that building vertex (V) and edge (E) sets scales linearly with grid size.

n	m	V time (s)	E time (s)	Mem Inc (GB)
10,000,000	79,880,040	1.2	218.4	1.8
20,000,000	160,008,240	2.5	495.5	3.6
40,000,000	319,760,040	8.1	1110.0	7.2
80,000,000	639,919,840	12.9	2827.9	14.4
160,000,000	1,279,520,040	21.5	7535.7	28.8

6.3 Closure Correctness Validation

The Pandas seminaive closure result is validated against a SQLite recursive view on every tested configuration. The two result sets are sorted and compared row-wise. Table 2 reports the validation results on the n=1,024 terrain graph across all tested slope thresholds. Zero discrepancies were found at every threshold.

Table 2: Pandas seminaive vs. SQLite recursive view on the n=1,024 terrain graph. All thresholds agree to zero row discrepancy.

S (°)	m_S	$\lvert G_S^+ \rvert$ (Pandas)	$\lvert G_S^+ \rvert$ (SQLite)
15	1,878	22,786	22,786
20	2,688	223,524	223,524
25	3,436	592,865	592,865
35	4,904	960,618	960,618

SNAP baseline comparison. To contextualize the engine's behavior before extending it to the terrain setting, we also validated on the Facebook ego-network and directed cycle graphs of increasing size. As shown in Fig. 5, Pandas runtime grows steeply as join-based deduplication over the accumulating closure dominates. SQL (SQLite) scales predictably via spill-to-disk at the cost of higher per-iteration overhead. These results establish the laptop-scale limits ($n \approx 10^4$) and directly motivate why our terrain experiments target the n=1,000 and n=10,000 tiers.

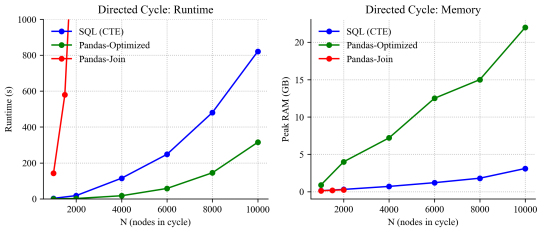


Fig. 5: Directed cycle stress test (linear scale): runtime and peak memory vs. n. Pandas-Optimized avoids relational overhead to stay fast in-memory, whereas the naive relational approach (Pandas-Join) suffers from severe runtime explosion as redundant pairs accumulate. The right-hand plot demonstrates that the substantial speedup associated with Pandas comes at the cost of higher memory consumption compared to SQL, which scales predictably by spilling to disk but with higher per-iteration overhead.

Without slope filtering, the terrain graph becomes strongly connected, saturating the closure to all $1{,}024^2 = 1{,}048{,}576$ pairs in 30 iterations and 13.5 s. This unfiltered state is computationally trivial relative to the structural intricacies revealed when a threshold is enforced.

6.4 Naive vs. Seminaive Performance Benchmark

While dynamic-programming approaches (e.g., Warshall's algorithm or Floyd-Warshall) skip many intermediate results, they typically require in-place updates

that perform poorly at the boundary of main memory and storage. Conversely, logarithmic evaluation algorithms (squaring the adjacency matrix to find paths of length $2, 4, 8, \dots$) are optimized for graphs with long diameters and disjoint paths, which does not suit the dense, localized structure of our topographic graphs. Therefore, our baseline characterization directly compares naive against seminaive evaluation for relational fixpoints. Table 3 reports runtime for both algorithms, alongside seminaive peak RSS and iteration count, on the terrain graph at four slope thresholds and both dataset scales.

The iteration count peaks near the phase transition ($S = 20°$, 70 seminaive iterations) rather than at the densest threshold ($S = 35°$, 48 iterations). This occurs because the graph diameter is longest when the giant SCC is first forming and decreases as higher S introduces denser connectivity.

Seminaive reduces runtime by 1.5–3.3× on the n=1,000 tier and consistently by ∼3.1× on n=10,000. The speedup on the n=1,000 tier grows with S because larger $|G_S^+|$ means more redundant pairs re-joined per round by the naive algorithm. Memory savings are modest (5–14%) since both methods must materialize the same final G_S^+.

Table 3: Naive (direct) vs. seminaive TC on the terrain graph. Speedup is $t_{\text{naive}}/t_{\text{semi}}$.

n	S (°)	Naive Time (s)	Semi Time (s)	Speedup	Semi RSS (MB)	Semi Iter.	$\lvert G_S^+ \rvert$
1,024	15	0.55	0.37	1.49×	572.9	25	22,786
	20	13.40	5.14	2.61×	579.1	70	223,524
	25	35.21	11.01	3.20×	592.0	65	592,865
	35	50.67	15.18	3.34×	602.7	48	960,618
10,000	15	1852.3	599.6	3.09×	879.4	133	9,632,120
	20	7972.8	2580.7	3.09×	989.8	111	67,761,650
	25	9261.0	2997.6	3.09×	1033.0	95	82,831,495
	35	10336.0	3345.6	3.09×	1083.0	73	95,932,298

6.5 Phase Transition: Tunable Workloads

The real utility of the DEM terrain graph lies in sweeping the slope parameter S to vary the workload complexity. Table 4 illustrates that as S increases, G_S undergoes a sharp structural phase transition near $S^* \approx 20°$.

At $S = 15°$, the graph fragments into 270 components, computing trivially small closures. Between 20° and 22°, the giant component (Q4) nearly doubles from 44.0% to 66.5% of vertices, and the reachable-pair ratio jumps from 21.3% to 44.7%. By $S = 35°$, a single dominant component covers 95.7% of the terrain. This abrupt, predictable transition allows a single $n = 1{,}024$ grid to serve as an entire suite of queries, moving from disconnected pathlets to near-quadratic TC output simply by tuning S.

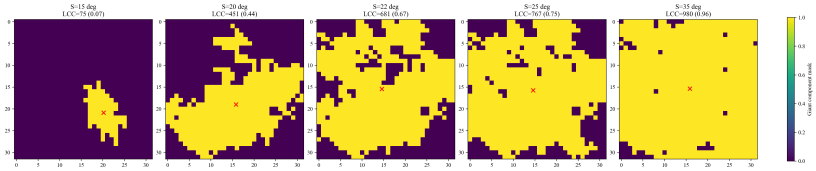


Fig. 6: Giant weakly connected component of G_S vs. S (n=1,024): yellow in-component, purple out; expansion past $S^*\approx 20°$.

Table 4: Phase transition on the n=1,024 graph. The sharp jump near $S^* \approx 20°$ demonstrates that S provides a tunable workload control.

S (°)	m_S	Edge %	$\lvert G_S^+ \rvert$	Reach %	Components	Giant (%)
15	1,878	24.0	22,786	2.2	270	75 (7.3%)
20	2,688	34.4	223,524	21.3	139	451 (44.0%)
22	2,992	38.3	468,975	44.7	103	681 (66.5%)
25	3,436	44.0	592,865	56.5	68	767 (74.9%)
35	4,904	62.8	960,618	91.6	19	980 (95.7%)

6.6 Spatial Footprint and Scale Validation

The drainage basin and giant component footprint spatially reflect this phase transition. Table 5 quantifies the giant component's centroid and bounding box at each threshold. At $S \geq 22°$ the bounding box expands to fill the entire 32×32 grid, confirming that the giant component is geographically contiguous across the full terrain crop. Fig. 6 shows the spatial footprint at five thresholds. At low S the largest reachable cluster is small and confined to the relatively flat plateau region. As S crosses 20°, the giant component expands rapidly outward, driven by the progressive inclusion of steeper edges along the summit flanks.

Table 5: Giant component spatial properties on the n=1,024 terrain graph at selected slope thresholds. Centroid (row, col) is 0-indexed; bounding box is in grid cells.

S (°)	Components	Giant (n)	Giant (%)	Centroid (r, c)	Bbox (h × w)
15	270	75	7.3	(20.9, 20.2)	15 × 10
20	139	451	44.0	(19.0, 15.8)	31 × 32
22	103	681	66.5	(15.4, 14.6)	32 × 32
25	68	767	74.9	(15.7, 14.6)	32 × 32
35	19	980	95.7	(15.4, 15.8)	32 × 32

6.7 Topographical Interpretation of Queries

While benchmarking validates the engine's performance, the primary value of the framework lies in reading the geographical map directly from the relational

query results. By examining these outputs on the n=1,024 dataset, users can deduce critical topographical features without needing specialized GIS software.

Running Q1 (Point-to-Point Reachability) between a trailhead and a summit returns a simple Boolean. Checking reachability from a lower plateau to the summit yields `False` at a low slope tolerance ($S = 20°$), correctly identifying that technical climbing gear (or a gentler zigzag path) is required. At $S = 35°$, it returns `True`.

Evaluating Q2 (Drainage Basin) for a specific valley outlet returns the set of uphill cells that drain into it. In our dataset, Q2 at $S = 20°$ identifies a contiguous basin of 215 cells (10,838 m^2) feeding into the eastern ravine (Fig. 4a). This query is highly valuable for hydrological modeling, indicating exactly where rainfall will accumulate.

Q3 (Reachable Hub Detection) identifies the most central, flat gathering point accessible from a starting cell. In a search-and-rescue scenario, running Q3 for a stranded hiker at (12, 10) ($S = 20°$) reveals the optimal helicopter landing zone at (15, 15) with a flat out-degree of 8, located 41.3 m away.

The size of the Q4 (Mutually Reachable Component) giant component explicitly measures the traversable area of the mountain. The phase transition has a direct topographical meaning: below $S = 20°$, the terrain is dominated by impassable ridges. As S crosses the critical 20° threshold, isolated meadows abruptly connect via steeper corridors, instantly opening up the map (growing from 7.3% at 15° to 44.0% at 20°).

Q5 (Bottleneck Detection) identifies critical geographical choke points. Running Q5 at $S = 20°$ reveals a narrow land bridge connecting two plateaus (Fig. 4b), with 6 choke points identified out of over 1,000 paths evaluated. Identifying these bottleneck vertices is crucial for infrastructure planning, as washing out any of these specific cells would completely sever the walking route.

While Q1 proves a path exists, Q6 (Shortest Distance Path) evaluates the actual travel cost. Because paths must zigzag to avoid steep cliffs, the shortest slope-constrained path is rarely a straight line. Computing Q6 for a target 500 m away linearly yields a path distance of 1,240.5 m at $S = 25°$, emphasizing why distance-aware extensions are critical for accurate travel-time estimations.

Query Execution Complexity. Table 6 summarizes the computational cost of evaluating representative queries on the materialized n=1,024 graph ($S = 20°$). The results experimentally confirm the theoretical difficulty progression. Simple filters (Q1) operate instantaneously, aggregations and self-joins (Q3, Q4) introduce relational overhead, and non-monotonic or distance-aware queries (Q5, Q6) require either expensive path validation or modified multi-iteration algorithms, significantly increasing the execution time.

Table 6: Execution complexity for representative queries over the n=1,024 terrain graph ($S = 20°$).

Query	Difficulty Class	Runtime (s)
Q1 (Reachability)	Easy (Direct Lookup)	<0.01
Q3 (Reachable Hub)	Medium (Join + Agg)	0.05
Q4 (Mutually Reachable)	Medium (Self-Join)	0.12
Q5 (Bottleneck)	Hard (Non-Monotonic)	0.84
Q6 (Shortest Path)	Hard (Beyond-Closure)	1.85

6.8 Scale Validation

This topological shift holds at larger scales: evaluating on n=10,000, we observed the same phase transition across the 18°–24° window, yielding 79.2% reachability at 24° on a workload producing $\approx 80 \times 10^6$ resulting rows. This scaling behavior confirms that the DEM pipeline acts as a robust, controllable dataset generator for reachability analysis.

7 Conclusion

We addressed how to evaluate recursive reachability queries over terrain data in Python while preserving strict database-style correctness. By modeling slope-constrained reachability as transitive closure over a thresholded edge relation, we built and validated a DEM-to-graph pipeline that evaluates queries using a Pandas seminaive algorithm. Our experiments show that this Python implementation correctly reproduces the semantics of SQL recursive views with zero discrepancy against SQLite. We empirically characterized the runtime, memory footprint, and scaling of both graph generation and transitive closure evaluation on Mount Rainier DEM crops up to 10,000 vertices, confirming practical performance.

Varying the slope passability threshold induces a sharp phase transition in the graph's structural connectivity, moving the output from highly fragmented, small closures to near-quadratic coverage within a narrow parameter window. This characteristic changes graph density and recursion depth in a highly controlled, predictable way. Consequently, this framework is exceptionally useful as a reproducible, scalable benchmark family for recursive-query evaluation. Future work will compare against PostgreSQL recursive CTEs and larger-scale graph processing systems. Future work will also extend this pipeline to support more complex spatial graph structures, including non-local edges for long-range visibilities and Delaunay triangulation methods for higher-fidelity terrain modeling. The next major step will involve applying neural networks, specifically Graph Neural Networks (GNNs) with a temporal aspect, to predict path viability and terrain evolution into the future (e.g., predicting paths next week, next month). We plan to release the implementation and benchmark datasets to support future research.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Reading, MA (1995), classic reference for Datalog/RA/FOL and recursion
2. Banerjee, J., Quass, D., Motwani, R., Widom, J.: Implementation of recursion in SQL: The power and limitations of Recursive query processing. In: Proceedings of the ACM SIGMOD Conference. ACM (1987)
3. Cohen, E., Halperin, E., Kaplan, H., Zwick, U.: Reachability and distance queries via 2-hop labels. In: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 937–946. SODA '02, Society for Industrial and Applied Mathematics, USA (2002)
4. Contributors, S.O.: How to limit elevation over distance using the a* search algorithm? `https://stackoverflow.com/questions/76624324` (2023), discussion on modifying traditional single-source pathfinders for global terrain gradient constraints.
5. Di Tria, F., Lefons, E., Tangorra, F.: Revisiting recursive query optimization in column-oriented database systems. In: Proceedings of the ACM DOLAP Workshop. ACM (2014)
6. GRASS Development Team: Grass gis manual: r.walk. `https://grass.osgeo.org/grass-stable/manuals/r.walk.html` (2024), anisotropic cumulative cost and optimal path routing over raster elevations.
7. Kitsuregawa, M., Tanaka, H., Moto-oka, T.: Application of hash to data base machine and its architecture. In: Proceedings of the International Conference on Very Large Data Bases (VLDB). pp. 88–101 (1983), early work on hash-based join processing
8. McKinney, W.: Data structures for statistical computing in python. In: Proceedings of the 9th Python in Science Conference (SciPy). pp. 51–56 (2010)
9. Melton, J.: Understanding SQL and SQL3: A guide for database users. In: Proceedings of ACM SIGMOD. ACM (1993), early discussion of recursive SQL
10. O'Callaghan, J.F., Mark, D.M.: The extraction of drainage networks from digital elevation data. Computer vision, graphics, and image processing **28**(3), 323–344 (1984)
11. Ordonez, C.: Optimization of linear recursive queries in SQL. IEEE Transactions on Knowledge and Data Engineering **22**(2), 264–277 (2010)
12. Ramakrishnan, R., Srivastava, D.: Magic templates: A spellbinding approach to logic programs. In: Proceedings of SIGMOD. ACM (1992), classic paper on efficient fixpoint and Datalog evaluation
13. Tarboton, D.G.: A new method for the determination of flow directions and upslope areas in grid digital elevation models. Water resources research **33**(2), 309–319 (1997)
14. U.S. Geological Survey: 3D Elevation Program (3DEP). `https://www.usgs.gov/3d-elevation-program` (2026), accessed 2026-03-09
15. Zhao, G., Rundensteiner, E.: Optimizing recursive queries in relational databases. IEEE Transactions on Knowledge and Data Engineering (TKDE) **22**(6), 812–826 (2010)
16. Zhou, X., Farouzi, A., Bellatreche, L., Ordonez, C.: Bitwise algorithms to compute the transitive closure of graphs in Python. In: Database and Expert Systems Applications (DEXA). pp. 364–378. Springer (2023)