Nav-Index: A High-Performance, Adaptive Index for Shortest Path Queries in RDBMS

Maximilian Reif
Technical University of Munich
reif@in.tum.de

Thomas Neumann
Technical University of Munich
neumann@in.tum.de

ABSTRACT

Shortest path queries are a fundamental operation on graphs with numerous applications. Efficiently executing shortest path searches in RDBMS is challenging, as graphs can not only be static relations but might also occur as ad-hoc intermediate results of complex analytical queries. Especially single-pair shortest path queries are difficult to accelerate, because they require minimizing the search space and intermediate results cannot be shared e.g. for multiple destinations. To address these challenges, we present Nav-Index, an adaptive index for relational DBMS that accelerates large-scale shortest path queries with seamless integration into SQL and relational algebra. Nav-Index switches adaptively between three modes: an ad-hoc generic mode using Dijkstra's algorithm, an ad-hoc optimized mode for sparse graphs leveraging contraction hierarchies (CH), and a table index mode for static graphs also leveraging CHs.

Graph DBMS typically provide specific implementations for such queries and are valuable for many workloads. In practice, however, existing systems show limited throughput for large-scale graphs such as road networks and offer limited optimization for analytical relational processing. Embedding Nav-Index into an RDBMS allows for a seamless composition with other relational operators, query decorrelation, cost-based optimization, morsel-driven parallelism, and pipelining. In the case of sparse graphs, Nav-Index's CH mode transforms the graph into a contraction hierarchy, allowing algorithmic speedups by orders of magnitude.

We implemented Nav-Index into the compiling RDBMS Umbra and evaluate it extensively on road networks from small cities to the whole planet and social graphs. In our experiments, Nav-Index outperforms all other database systems that we have tested by orders of magnitude due to algorithmic advantages and an efficient implementation. It can index the road network of Europe in under 12 minutes and Earth in less than 68 minutes, allowing planet-scale shortest path queries in milliseconds in a general-purpose RDBMS.

PVLDB Reference Format:
Maximilian Reif and Thomas Neumann. Nav-Index: A High-Performance, Adaptive Index for Shortest Path Queries in RDBMS. PVLDB, 19(11): 3344 - 3356, 2026.
doi:10.14778/3836663.3836693

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://gitlab.db.in.tum.de/max.reif/nav-index-reproducibility.

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836693

Figure 1: Planet-scale shortest path queries on road networks.

1 INTRODUCTION

Relational Database Management Systems (RDBMS) play a key role in analyzing and querying very large datasets due to the versatility of the relational model and the efficient optimization and execution of analytical queries. However, over the past years, dedicated graph systems have been introduced to improve usability and performance on graph workloads. We observe two main challenges with graph data in RDBMS which make graph systems seem appealing.

- Graph queries (such as shortest path queries) can be difficult to express in classic SQL.
- Specialized graph algorithms, such as shortest path computations, can be inefficient when relying on traditional relational operators due to growing intermediate results.

The Appeal of Graph DBMS. To address both challenges, systems with specialized query languages and dedicated graph algorithms were introduced, improving usability and algorithmic performance [14, 25, 35, 38]. However, in many scenarios, RDBMS can still outperform graph systems on graph workloads if carefully implemented in SQL [2]. We attribute this to many optimizations in RDBMS that benefit from decades of research in query optimization and execution. Therefore, the question arises whether it is sensible to further extend RDBMS towards graph workloads by integrating extensions to SQL and graph algorithms directly into RDBMS.

Graph Workloads in RDBMS. The addition of SQL/PGQ to the SQL:2023 [20] standard is a first step towards expressing graph-specific queries without difficult-to-write recursive SQL. The execution of such queries, however, is more involved, since graph operations and algorithms can be diverse and are not always mappable to classic relational operators. While pattern matching can be efficiently executed using multiple joins [2, 6], shortest path queries (especially single-pair shortest path queries) can be more involved due to unnecessarily large intermediate results.

Shortest Path Queries. We focus on shortest path queries on directed weighted graphs with non-negative edge weights. Especially single-pair shortest path (SPSP) queries are challenging because their performance heavily depends on minimizing the search space. A very prominent shortest path use case in daily life is route planning in road networks (running example), as shown in Figure 1.

A road map can be described using a single relation *roads(src, dest, weight)*. To determine the shortest path from *src* to *dest* in SQL/PGQ, we use the following MATCH clause:

```
match path = any shortest
(a:nodes where a.id = <src>)-[r:roads]->*(b:nodes
    WHERE b.id = <dest>)
as sp;
```

We use the following SQL-like syntax, which is closely related to the syntax used by the PostgreSQL extension pgRouting [31] (which also offers shortest path search). It can be derived from PGQ [38], offers similar functionality, and is easier to read and write:

```
select shortest_path(
  table (select * from roads), orig, dest
);
```
(Q1)

Q1 returns the shortest path from *orig* to *dest* using the *roads* relation as graph input. Considering a dataset of many *trips(orig, dest, dist)*, we can ask for all trips that did not take the shortest route (e.g., detours made by delivery drivers due to traffic):

```
select t.dist, weight(shortest_path(
  table (select * from roads), t.orig, t.dest
)) from trips t;
```
(Q2)

Q2 can be used to analyze the efficiency of actual trips compared to the optimal route and is our running example. The *shortest_path* function uses source and destination from the *trips* relation. Note that both parameters can be arbitrary (even correlated) columns.

```
select avg(
  length(shortest_path(
    table (select * from roads), t.orig, t.dest))
  -length(shortest_path(
    table (select src, dest, weight + 1 from roads),
    t.orig, t.dest))
)
from trips t;
```
(Q3)

Q3: How many turns could be saved by incorporating the turn count into the weight metric (assuming each edge represents a turn in the road network)?

These examples are just a glimpse of the variety of shortest path queries on road networks that can be expressed in SQL. While Q1 resembles a typical *map application* scenario with a single path lookup, Q2 and Q3 are executed for many tuples of the *trips* relation and could be part of a data analytics use case. While Q1 and Q2 query a persistent roads table which could be optimized using a table index, Q3 uses an intermediate result of a subquery as a graph and requires an ad hoc graph index to be executed efficiently.

Faster than Dijkstra with Contraction Hierarchies. The classical approach to shortest path search is Dijkstra's algorithm [12]. However, in road networks, which are typically sparse graphs, specialized

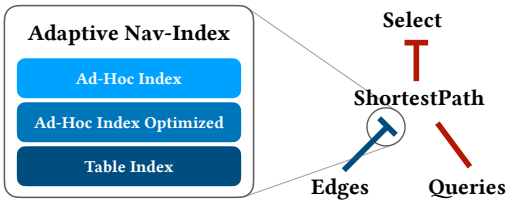

Figure 2: Nav-Index's three adaptive modes: 1) A general-purpose shortest path index for small graphs or few queries, 2) an ad-hoc contraction hierarchy (CH) index for sparse graphs, and 3) a CH table index for static graphs.

shortest path algorithms such as contraction hierarchies (CH) [17] can accelerate shortest path queries significantly. Contraction hierarchies split pathfinding into a build and a query phase, accelerating path lookups by orders of magnitude. While contraction hierarchies are well-studied in the context of shortest path algorithms [3], implementations in general-purpose RDBMS and Graph DBMS are rare. For example, the graph database systems Neo4j [25] and TigerGraph [35] both implement Dijkstra and A* search [19] but do not support contraction hierarchies. pgRouting (a routing extension for PostgreSQL) [31] implements Dijkstra and other classic shortest path algorithms. Recently, an experimental function for contraction hierarchies was added to pgRouting [32] but is still too limited for practical use, as shown in our evaluation (Section 5). However, contraction hierarchies are not suitable for all scenarios. The build phase can be time-consuming and is only justified for many queries on the same graph. For arbitrary graphs or ad-hoc graphs (e.g., intermediate results) with only a few queries, classic Dijkstra might be preferable if the build cost cannot be amortized. Therefore, DBMS have to trade off between generality and performance.

Nav-Index. Nav-Index bridges the gap between high-performance pathfinding algorithms and the generality of RDBMS by accelerating shortest path queries on large-scale graphs with three modes of adaptivity: 1) a general-purpose shortest path index for Dijkstra searches, 2) an ad hoc CH index for sparse graphs, and 3) a CH table index for static graphs.

Additionally, a deep integration into an RDBMS is appealing to offer an easy-to-use declarative SQL interface for complex analytical queries involving shortest path computations. Therefore, we integrate Nav-Index into the compiling database system Umbra [28] as an index structure paired with a relational shortest path operator, as shown in Figure 2. This allows Umbra to optimize shortest path searches alongside other relational operators to allow for query unnesting [27, 29], cost-based optimization, morsel-driven parallel execution [22], and pipelining [26].

As a running example throughout the paper, we use Query 2 introduced above. The actual workload depends on the graph (roads relation), the cardinality of trips (|trips|), and the presence of a table index on the roads relation. With Nav-Index the user does not declare which strategy to use. The best evaluation strategy (Nav-Index mode) depends on these workload properties and is automatically chosen by the system, similar to *physical operator selection*. The system identifies from the schema if a table index is present (mode 3) and uses cardinality estimates and runtime observations to decide between mode 1 and 2. In the case of a small

number of trips, it is efficient to perform Dijkstra's algorithm (mode 1) for each trip. However, if the shortest path function is evaluated for many trips, it is more efficient to build an ad-hoc contraction hierarchy (CH) index for the graph (mode 2) and execute lookups for each trip.

We evaluate our adaptive implementation using planet-scale real-world road networks (sparse, large diameter) [30] and social networks (Twitter Social Circles, denser, small diameter) [23]. We compare it to existing implementations in RDBMS, dedicated graph systems and the routing library OSRM [24]. Our results show that Nav-Index significantly outperforms existing database systems on both single shortest path queries and analytical workloads involving many shortest path queries by orders of magnitude for both datasets. Umbra with Nav-Index can index the whole planet with over 756 million edges within 68 minutes and answers shortest path queries in a few milliseconds. The road network of Europe with over 146 million edges can be indexed in under 12 minutes and could therefore be rebuilt frequently to support live traffic updates.

The remainder of this paper is structured as follows: Section 2 describes the design and algorithmic details of Nav-Index. The optimizer decision and the cost model are described in Section 3. Section 4 shows how the build phase is parallelized. Section 5 evaluates our implementation against existing systems. Section 6 discusses related work, and Section 7 summarizes the results.

2 NAV-INDEX

To enable efficient shortest path computations for ad-hoc and static, sparse and dense graphs in an RDBMS, we propose an index structure that adapts to the graph characteristics: the Nav-Index. We consider shortest path queries on directed weighted graphs with non-negative edge weights, which can be specified by a single *edges* relation $E : \{[source, destination, weight]\}$.

In the case of sparse graphs, shortest path queries can be accelerated by transforming the graph into a contraction hierarchy. A shortest path search of our running example is shown in Figure 3. It compares a shortest path search on the German road network on a contraction hierarchy to a classic bidirectional Dijkstra search. Each line in the figure is an edge touched by the search. While the unoptimized Dijkstra search has to explore a significant portion of the graph in full detail, the optimized contraction hierarchy allows finding the same shortest path by taking shortcuts and exploring only a small fraction of the graph. In this example, the basic bidirectional Dijkstra search settles over 60,000 nodes, while the search on the contraction hierarchy settles fewer than 800 nodes.[1] This results in significant speedups for shortest path queries by orders of magnitude, as shown in the evaluation (Section 5).

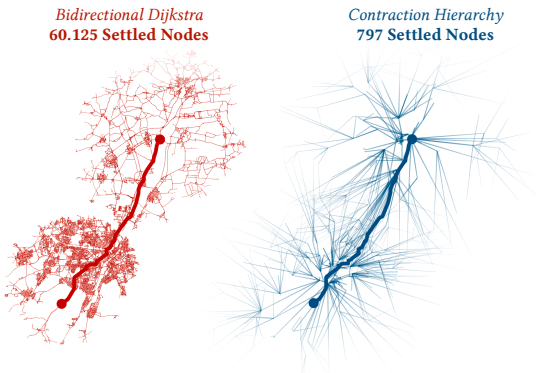

Figure 3: Dijkstra vs. contraction hierarchy: Search space comparison for a shortest path query near Munich.

2.1 Three adaptive modes

To handle generic graphs efficiently while enabling additional optimizations for sparse graphs, our implementation uses three adaptively chosen modes of operation:

- **Ad-Hoc Index (Dijkstra)** For small or dense graphs and few queries where the construction of a contraction hierarchy would not pay off.
- **Ad-Hoc Index Optimized (CH)** For sparse graphs and bulk queries on the same graph where the build cost of a contraction hierarchy can be amortized over multiple queries.
- **Table Index (CH)** For static graphs. Here, the construction cost of the contraction hierarchy is amortized over many individual queries.

All three modes share the same graph representation and Dijkstra implementation, allowing highly tuned and optimized common operations, benefiting all three modes.

2.2 From SQL to Nav-Index

For seamless integration into SQL and relational algebra, we implement shortest path search into the compiling database system Umbra as a table function *shortest_path(table edges, src, dest)*. This function takes an *edges* relation as input and returns the shortest path from *src* to *dest* as an array of nodes. It is mapped to a binary relational *ShortestPathOperator* with two input relations: the *edges* relation and a temporary table with a single tuple consisting of *source* and *destination*.

Correlated shortest path queries. If source and destination are not constant but columns of other relations, the *ShortestPathOperator* becomes a *dependent ShortestPathOperator* analogous to a *dependent join*. The query can then be unnested using general unnesting techniques [27]. This allows shortest path queries to be combined with other relational operators without any special treatment.

Nav-Index mode selection. The *ShortestPathOperator* can be executed in one of the three modes described above. If the *edges* relation is a base table with an existing table Nav-Index on the required columns, the *Table Index CH* mode is chosen. If the *edges* relation is an intermediate result or no table index exists on the base table, the ad-hoc *Optimized CH* or *Dijkstra* mode is chosen based on a cost model which is described in Section 3.

Probing the Nav-Index. For all shortest path queries, the *ShortestPathOperator* probes the Nav-Index with source and destination. The graph representation is chosen such that both Dijkstra searches and CH lookups can be executed identically using an optimized bidirectional Dijkstra implementation. The only difference is that in the case of contraction hierarchies the graph representation contains

[1] A node is called *settled* once its shortest-path distance is final and will not be updated further.

additional shortcut edges, and the search only needs to traverse the hierarchy upwards. Since the lookup is a read-only operation, multiple shortest path queries can be executed in parallel and can be pipelined with other relational operators without materializing the tuples. Consequently, as long as the index fits into memory, shortest path queries can be executed on larger-than-memory relations without spooling intermediate results to disk.

2.3 Optimized Version - Contraction Hierarchy

In the case of large graphs where additional build time can be amortized over multiple queries, Nav-Index builds a contraction hierarchy [17]. While CHs are a prominent index structure for shortest path search in road networks, implementations in general-purpose DBMS are uncommon. A rare example is pgRouting [31], which provides an experimental contraction hierarchy function, highlighting the desirability for such features in database systems.

While modern research on contraction hierarchies often focuses on extending or further optimizing them for the special case of road networks, we try to keep our approach independent of any other metrics besides the weighted edges themselves. For a full explanation of contraction hierarchies we refer to [17]. Here, we give a brief overview of the main concepts relevant for Nav-Index:

Conceptually, a contraction hierarchy is a hierarchical version of the input graph allowing faster shortest path lookups without compromising correctness. The main idea is to assign an importance to each node, which yields the hierarchy. The main goal is that shortest path queries only need to search upward in this hierarchy. To ensure that shortest paths cannot be missed during this upward-only search, shortcuts are introduced. Shortcuts represent (and thus preserve) shortest paths that would otherwise not be found by an upward-only search.

Contraction Hierarchy Lookup. Figure 4 depicts a contraction hierarchy lookup for our running example. Graphically, it depicts an excerpt of the road network where nodes are arranged according to their importance (the higher the node, the more important it is) and do not represent the actual spatial layout of the graph. Edges of the original graph are represented as solid arrows; shortcuts are depicted as dashed arrows. The contraction hierarchy consists of two subgraphs: an up-graph and a down-graph. Edges from lower-importance nodes to higher-importance nodes belong to the up-graph. Edges from higher-importance nodes to lower-importance nodes belong to the down-graph. A bidirectional Dijkstra search from source S to destination D only searches forward in the up-graph and backward in the down-graph. Both searches meet at an intermediate node M, which lies on the shortest path from S to D. To obtain the full shortest path from S to D in the original graph, the path from S to M in the up-graph and the path from M to D in the down-graph are combined, and the shortcuts are recursively unpacked.

Contraction Hierarchy Construction. Building a contraction hierarchy that is *correct* and *fast* for lookups consists of two conceptual steps which are interleaved in practice:

(1) Finding a node importance order that (in practice) leads to faster lookup times due to a *balanced* hierarchy.
(2) Adding shortcuts such that shortest paths cannot be missed during the upward-only search phase.

(1) is a *soft* challenge and is typically solved using a heuristic; (2) must be correct such that a path search is guaranteed to find a shortest path and is achieved by performing node contractions in the order determined by (1).

Node contraction. In order to preserve shortest paths in an upward-only search, we introduce shortcuts by performing *node contractions*. The contraction is performed bottom-up from the node with the lowest score. Whenever a node is contracted that is part of a shortest path between two higher-level nodes, a shortcut is introduced to preserve the shortest path information. If a shorter path (also referred to as a witness path) exists, no shortcut is introduced. This witness path search is performed using locally bounded Dijkstra searches. Figure 5 depicts the build phase of a contraction hierarchy. As before, nodes are arranged according to their importance (the higher the node, the more important it is). The figure depicts a contraction of the first two nodes (for simplification of the example, we assume identical edge weights). First, node 1 is contracted. Since node 1 is not part of shortest paths of higher-level nodes, no shortcuts have to be introduced. Next, node 2 is contracted. Since it is part of the shortest path between nodes 6 and 4, it would be missed in an upward-only shortest path search from 6 to 4. Consequently, we introduce a shortcut to preserve the shortest path information. The algorithm continues until all nodes are contracted.

Node order and heuristic. Clearly, the resulting contraction hierarchy heavily depends on the contraction order. In simple terms: the *better* the node order, the *faster* the average lookup. An easy-to-grasp heuristic (and a widely used one) is the so-called *edge*

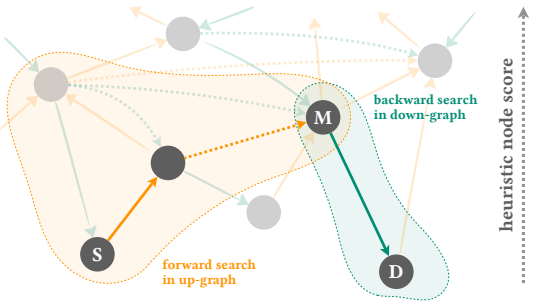

Figure 4: Running Example: Search from S to D. Forward in the up- and backwards in the down-graph, meet at node M.

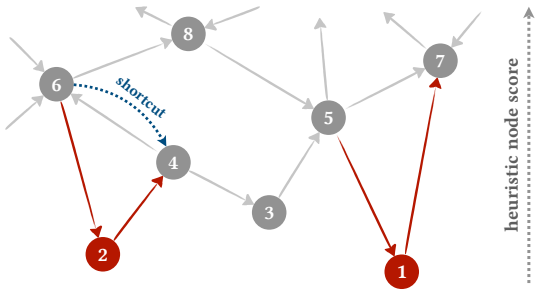

Figure 5: Running Example: Contraction Hierarchy. Contraction of the first two nodes.

difference [17], which is defined as follows: For each node, the edge difference is computed as the number of shortcuts that would be introduced during its contraction minus the number of edges that would be removed during contraction. The main idea here is that smaller graphs are faster to query. Therefore, nodes with many edges that only introduce few shortcuts during their contraction should be contracted first. This keeps the edge count in the resulting contraction hierarchy low. Many heuristics are conceivable and used in practice. We chose a combination of edge difference and a hop-count metric [17].

Interleaved heuristic update and contraction. Initially, all nodes are scored using the heuristic. The node with the lowest score is contracted first. Since each contraction changes the graph and therefore the heuristic, the heuristic scores change and need to be recomputed. To avoid recomputation of the heuristic for all nodes, only the heuristics of neighbors of contracted nodes are recomputed and updated. To efficiently determine the node with the lowest heuristic score, nodes are kept in a priority queue. To allow for priority updates, the priority queue requires adjust-key operations.

Pairing heap for priority queue. Supporting adjust-key operations in priority queues efficiently is non-trivial. For binary heaps, a main challenge is to keep track of the position of elements in the heap so they can be found if the key is updated. Tracking elements can be achieved by maintaining a hash table; however, whenever elements are moved, the hash table needs to be updated, which introduces significant overhead. Fibonacci heaps natively support decrease-key operations efficiently with asymptotic guarantees [16]. However, in practice they are complex to implement and have significant constant factor overheads. Pairing heaps offer a significantly simpler implementation [15] and are faster in practice [15]. Our experiments confirm that pairing heaps practically outperform binary heaps with hash-table element tracking for maintaining the contraction order priority queue.

3 WORKLOAD MODEL AND COST-BASED MODE SELECTION

As Nav-Index supports the (previously described) three adaptive modes (ad-hoc Dijkstra, ad-hoc CH, table-index CH), the query optimizer has to choose when each mode makes sense to execute. The Nav-Index mode selection is analogous to the physical join algorithm selection during physical optimizations (e.g. index nested loop join, hash join, nested loop join). Similarly, the query optimizer uses the schema, table cardinality estimates, and runtime observations to choose the most suitable Nav-Index mode.

3.1 Workload Definition.

The shortest_path(edges, src, dest) function has three arguments: an edges relation (specifying the graph) and two node identifiers (src, dest). Src and dest specify a path lookup request (query) on the graph and can be either constants or variables. Therefore, the function can be translated into a binary operator with the input relations E (edges) and Q (queries). We consider every tuple of Q to be a separate SPSP query, which is executed by probing the Nav-Index with the corresponding src and dest. We denote the number of edges with |E|, the number of nodes with |V|, and the number of queries |Q|. The edges relation E can be either a base table or the result of a subquery. In case of a base-table, the schema may contain a nav-table-index (analogous to a b-tree).

The workload is defined by the presence of a base table index, the number of edges |E|, the number of queries |Q|, and the average out degree d (which can be calculated from |E| and |V|). For the mode selection, the query optimizer has to determine these workload parameters. They are obtained from different sources and are summarized by the following table:

Input	Source	Known at
table index	schema	compile time (exact)
$\lvert\hat{Q}\rvert \approx \lvert Q\rvert$	cardinality estimates	compile time (estimate)
$\lvert E\rvert$	materialized input	execution time (exact)
$\lvert V\rvert$	materialized input	execution time (exact)
d	materialized input	execution time (exact)

3.2 Cost-Based Mode Selection

To choose the most efficient execution strategy (mode), the query optimizer uses a cost model and graph density gate to choose the appropriate mode based on the workload parameters. In pseudocode, the mode selection can be expressed as follows:

```
if table_index_exists(edges)
  return table-index ch
else if d > ρ
  return ad-hoc dijkstra
else if T_chBuild(|E|) + |Q̂| · T_chLookup(|E|) < |Q̂| · T_dijkstra(|E|)
  return ad-hoc ch
else
  return ad-hoc dijkstra
```

First, ρ, the graph density threshold, is compared to the average out degree d to determine if the graph is sparse or dense. If the graph is dense, contraction hierarchies are susceptible to introducing many shortcuts which increases build-time and space consumption. To avoid this, the Dijkstra mode is chosen for dense graphs (we use $\rho = 10$). If the graph is sparse, the cost model is used to compare the estimated cost of building a contraction hierarchy and executing lookups against the estimated cost of executing Dijkstra searches. If the build cost plus lookup cost for contraction hierarchies is lower than the lookup cost for Dijkstra, then the contraction hierarchy mode is chosen. Otherwise, Dijkstra is chosen. Based on literature and our own experiments we estimate the cost to be:

Component	Formula	Constant	R^2
T_{chBuild}	$\alpha_b \cdot \lvert E\rvert$	$\alpha_b \approx 3.94\,\mu\text{s}$	0.94
T_{dijkstra} [10]	$\alpha_d \cdot (\lvert V\rvert + \lvert E\rvert) \log_2 \lvert V\rvert$	$\alpha_d \approx 0.398\,\text{ns}$	0.97
T_{chLookup}	≈ 0	—	—

The constants are determined using road networks on the machine used in the evaluation section. We observed in practice that the parallel CH construction time is linear in the number of edges, which is supported by our experiments. The Dijkstra cost estimate is based on the theoretical time complexity of Dijkstra's algorithm using binary heaps. The lookup time for contraction hierarchies

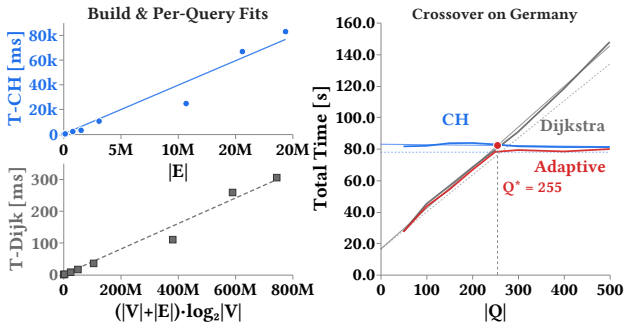

Figure 6: Cost model calibration and example: Per-graph CH build time (top left), per-query Dijkstra execution time (bottom left) and example of mode selection for the Germany road network (right, predicted vs. measured runtimes).

is assumed to be negligible compared to the CH build time and Dijkstra's execution time. Figure 6 depicts the measured CH build time for road networks of different sizes (top left) and the measured Dijkstra execution time per shortest path query (bottom left). Additionally we depict the fitted cost model for both.

Running Example. The right side of Figure 6 shows the actual runtimes for the Germany road network with $|V| = 9.81M$ nodes, $|E| = 15.6M$ edges, average out degree $d = 1.59$ for different numbers of queries $|Q| \in [50, 500]$. The schema has no table-index, so the optimizer has to choose between the ad-hoc Dijkstra and ad-hoc CH modes. As $d = 1.59 < \rho = 10$, the cost model is used to compare the two modes. For less than $Q^* = 255$ queries, the Dijkstra mode (gray) is faster than the contraction hierarchy mode. The cost model (thin dashed lines) slightly underestimates the actual runtimes, but correctly predicts the point where the contraction hierarchy mode becomes faster than Dijkstra. For more than Q^* queries, the contraction hierarchy mode (blue) is faster than Dijkstra. The adaptive mode (red) correctly chooses the faster mode.

4 DATA STRUCTURES AND PARALLELIZATION MODEL

In order to handle large graphs efficiently, both the index construction and query processing of Nav-Index are parallelized. This section describes the data structures used in our implementation and the parallelization model.

4.1 Graph Representation

The graph representation is a crucial component of Nav-Index. The memory overhead should be low while still allowing fast access to outgoing and incoming edges. At the same time edges must be concurrently updatable to allow an efficient parallel construction. During construction of a contraction hierarchy two graphs are maintained: the contraction graph which is modified during contraction, and the contraction hierarchy output graph which stores the final contraction hierarchy. During the build phase nodes are contracted iteratively until all nodes are contracted. Each contracted node is removed from the contraction graph and added to the contraction hierarchy output graph along with its shortcuts. For the Dijkstra-only mode, the graph representation is identical and just collects the edges provided by the input relation.

Graph. For fast navigation of edges (forward and reverse) we need to access outgoing and incoming edges with minimal overhead. Therefore we store edges redundantly in two hash tables: one for outgoing and one for incoming edges. Each hash table points to an adjacency list of edges.

Edge. Edges only store the destination node and the cost. The source node is implicitly given by the hash table entry. In the case of contraction hierarchies, additional information is required. Therefore, Nav-Index stores an optional mid node. When a shortcut is created during contraction, the mid node stores the contracted node. This allows recursively unpacking shortcuts during shortest path lookups. For the priority heuristics during contraction, Nav-Index also stores a hop count metric which allows more balanced contraction hierarchies [17]. For the basic Dijkstra search graph, this information is not necessary.

Concurrency. To allow concurrent updates during contraction hierarchy construction, each adjacency list is protected by a lock. The hash tables are lock-free and thread-safe and allow concurrent inserts. The locks on the adjacency lists are only required during edge insertions and removals and are fine-grained enough to allow efficient parallelization in practice.

4.2 Pathfinders

The Pathfinder component implements a bidirectional Dijkstra search on the graph representation described above. To avoid frequent allocations during shortest path searches, each thread uses its own thread-local pathfinder instance. Forward and backward searches are encapsulated in a Dijkstra pass which maintains the priority queue and the tentative distances and parent for each node.

While the asymptotic complexity of Dijkstra's algorithm depends on the asymptotic complexity of the priority queue used, in practice, other factors dominate the runtime. So while using a pairing heap improves performance for maintaining the contraction queue, for shortest path searches a binary heap performs better in practice. Therefore the overhead of maintaining more complex data structures such as pairing heaps does not pay off. Instead of decreasing the key in the priority queue, Nav-Index simply reinserts the node with the new distance. When extracting the minimum from the priority queue, nodes with outdated tentative distances are skipped.

In practice, we observed another performance bottleneck during the frequent shortest path searches during the contraction hierarchy construction. After each shortest path search, the tentative distances need to be cleared. To reduce allocation or resetting overhead, Nav-Index uses an amortized clearing approach using a *generation counter*. All elements inserted into the *generation map* are tagged with the current generation. If an element in the hash table is retrieved the generation of the element is compared to the generation counter of the hash table. Clearing can then be achieved by incrementing the hash table generation. Whenever the generation counter overflows or the hash table size exceeds a certain threshold, a full clear of the hash table is performed to avoid unnecessary memory over-allocation.

4.3 Concurrent Contraction Hierarchy Construction

The parallel construction of contraction hierarchies is always a trade-off since the contraction order is sequential by design. Contracting a node impacts the heuristic scores of neighboring nodes which in turn impacts the contraction order. Therefore, a naive parallelization by contracting multiple nodes at once can lead to suboptimal contraction orders and thus slower query performance.

A variety of approaches exist to parallelize the contraction hierarchy construction which are mainly based on the idea of finding independent nodes which can be contracted in parallel without impacting the contraction order significantly. This was first proposed by Vetter [36].

The parallel contraction algorithm is depicted in Figure 7. The algorithm starts by first initializing the global contraction queue with all nodes and their heuristic scores. Then the following steps are repeated until all nodes are contracted: First the set of independent nodes (red set) is retrieved from the global contraction queue. Then each red node is contracted in parallel. After contraction, the heuristic scores of neighboring nodes are recomputed in parallel. When all nodes are contracted, the contraction hierarchy output graph is returned.

Figure 7: Parallelization Model of Nav-Index

Independent Node Set. The general idea is that nodes which are not direct neighbors can be contracted in parallel since their contractions do not impact each other directly. Such nodes are called independent [36]. Strictly speaking, nodes are still not really independent because an update of the heuristic scores of neighbors of freshly contracted nodes might move them within the priority queue. Therefore, a parallel contraction of nodes will most likely influence the contraction order and is not equivalent to a sequential contraction. However, if *reasonably* independent nodes are selected, the impact on the contraction order becomes negligible. The challenge is to find a large set of independent nodes which can be contracted in parallel while still honoring the contraction order as much as possible. A very recent approach, proposed by Wan et al. [37], can scale extremely well by selecting nodes with minimal heuristic scores in their neighborhood. This however has conceptually multiple disadvantages: (1) A locally minimal node set does not necessarily honor the global heuristic score order. (2) If the graph is (or becomes) denser, it is harder to find independent nodes because fewer nodes are locally minimal in their neighborhood. This effect is pronounced if the heuristic scores have coarse granularity with many ties (typically the node id is used as a tiebreaker). Vetter [36] suggests to only consider nodes independent if they are minimal in their k-neighborhood with k ≥ 2. While larger values of k are more robust considering the contraction order, they lead to smaller independent sets and therefore limit parallelism.

Sequential node selection for robustness. To parallelize the contraction hierarchy construction, the global node order has to be relaxed to some degree. However, we want to keep the impact on the contraction order as small as possible and ideally bounded by design.

We propose and implemented an alternative approach which, to the best of our knowledge, has not been described before. The intuition is twofold: We iteratively expand an independent set of nodes respecting the global score and ignore direct neighbors of nodes already present in the set. Second, we only add nodes to the set if we don't need to ignore *too many* nodes.

Red-Blue Set. We call the set of independent nodes the *red set* and the set of ignored dependent nodes the *blue set*. The algorithm is described in the following pseudocode:

getRedSet(PQ) :
$Red \leftarrow \emptyset, Blue \leftarrow \emptyset$
$limit \leftarrow |PQ| \cdot c_l + 1$
while $PQ \neq \emptyset \ \wedge \ limit > 0 \ \wedge \ |Red| \cdot c_r \geq |Blue|$ do
  $v \leftarrow$ deleteMin(PQ)
  if $(neighbors(v) \cap Red = \emptyset)$ then
    $Red \leftarrow Red \cup \{v\}$
  else
    $Blue \leftarrow Blue \cup \{v\}$
  $limit \leftarrow limit - 1$
return Red

The algorithm iteratively retrieves nodes from the global priority queue PQ which is ordered by the global heuristic score as shown in Figure 8. If the retrieved node is not a neighbor of any red node it is added to the red set. Otherwise it is added to the blue set. The algorithm stops if the size of the blue set exceeds the size of the red set by a constant factor. Additionally, we also limit the maximum number of processed nodes to a constant factor of the number of remaining nodes in the contraction graph to avoid too large parallel contraction steps which would also violate the contraction order. This algorithm guarantees by design that both the ratio of ignored nodes and the ratio of processed nodes is bounded. The constants c_l and c_r can be used to tune the aggressiveness of the parallelization. Higher values lead to larger red sets and therefore more parallelism but also have a higher impact on the contraction order. In practice, we found that values of $c_l = 0.25$ and $c_r = 1.8$ work well.

As an intuition we can use the metaphor of a gold mine. In this metaphor the red nodes are gold and blue nodes are rock. We are looking for gold as long as we are finding more of it than rock. The advantages of this approach are that (1) it retrieves a globally minimal independent node set and (2) it stops doing so if finding more independent nodes would change the order significantly.

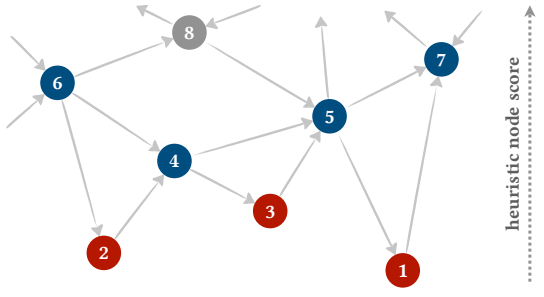

Figure 8: Running Example: Red-Blue selection of low-score independent nodes (red set) and dependent nodes (blue set). Red nodes can be contracted in parallel

A drawback of this approach is that this algorithm is inherently sequential and requires a global priority queue. As shown in the evaluation, the impact of this single-threaded step is a downside at high core counts. However, we found that the approach is more robust and limits the deviation from a pure sequential order by design and therefore appealing for a general-purpose DBMS.

Parallel Contraction. After retrieving the red set of independent nodes, each node in the set can be contracted in parallel. Each thread picks a morsel consisting of several red set nodes and contracts it using its own pathfinder instance (for the witness path search). The contraction itself requires multiple updates to the global graph representation. It should be noted that, for correctness, updates to the graph must be deferred until all independent nodes are contracted [36]. Since the graph representation is thread-safe, multiple threads can safely update the graph concurrently. Due to the fine-grained locking on the adjacency lists, contention is low in practice and we observe excellent scalability for the parallel contraction step.

Parallel Heuristic Computation. The most time-consuming step in the construction of the contraction hierarchy is the repeated heuristic computation. The heuristic score needs to be computed for all nodes initially and is repeated for neighbors of contracted nodes. Since the heuristic computation is a read-only operation on the graph it can easily be parallelized as long as the priority queue inserts/updates are synchronized. Nav-Index parallelizes the heuristic computation by assigning each thread morsels of nodes for which it computes the heuristic scores. After computing the heuristic scores for its morsel, each thread updates the global contraction queue. At high core counts the contention on the contraction queue could become a bottleneck. We mitigate this effect by maintaining thread-local pairing heaps. The global priority queue consists of a binary heap containing the minimum element of each local queue. Thus the global contraction queue can be viewed as a binary heap of pairing heaps. During heuristic updates, threads can update heuristic scores in local queues of other threads without locking the global queue. The local locks are fine-grained enough to avoid contention.

5 EVALUATION

In this section, we evaluate the performance of Nav-Index in Umbra on shortest path queries using real-world datasets. While we mainly focus on road networks due to their scale and complexity, we also evaluate Nav-Index on the Twitter Social Circles Dataset [23]. We analyze the impact of graph size, scalability regarding parallelism and compare our implementation with other established systems and the routing library OSRM [24].

5.1 Road Networks

As motivated in the Introduction, shortest path queries in road networks are a use case of daily life and at the same time difficult to solve efficiently in RDBMS. Compared to social network graphs, road networks produce long paths due to their high diameter[2] and sparse connectivity. Classic BFS or bidirectional Dijkstra searches tend to explore large parts of the graph. This makes road networks an interesting use case for contraction hierarchies and Nav-Index.

[2]The diameter is the longest shortest path between any two nodes in the graph.

For our benchmark, we use real-world road network datasets from OpenStreetMap (OSM)[3] for different regions of increasing size, from a small city map to the planet-wide road network. We convert the datasets into an expanded graph, where each road segment is represented as a node and each intersection turn is represented as a directed edge. This way of modeling allows for incorporating turn restrictions and turn costs into the graph, which is required for real-world navigation. For some related work, turn costs are ignored and edges represent road segments; in many cases, it is unfortunately not specified. However, the expansion of the graph changes the graph topology and increases the number of edges.

5.2 Hardware and Software Setup

All experiments (if not specified separately) are conducted on an AMD Ryzen 9 5950X 16-Core Processor with 32 hardware threads and 64 GB RAM running Ubuntu 25.04. To report end-to-end numbers (on unseen data), the measurements of Umbra report the end-to-end time including database startup, data loading from CSV, index creation, query execution, result reporting, and teardown. For the runtimes in table-index mode, the database is started and a graph table with a table-index is created before the measurements.

5.3 Nav-Index CH - Graph Size

Real-world road networks can vary significantly in size, ranging from small city maps to continent-wide networks. To compare the performance of Nav-Index on different graph sizes, we measured the end-to-end build time for *Bremen*, *Saarland*, *Oberbayern*, *Switzerland*, *Spain*, *Germany*, *DACH*[4], *Europe* and *Earth*[5]. The following table shows graph sizes for several regions:

Region	Edges	Nodes	Build Time
Bremen	53 211	33 894	1.0 s
Saarland	163 766	105 143	1.2 s
Oberbayern	827 682	516 569	3.6 s
Switzerland	1 550 507	1 008 060	5.4 s
Spain	10 719 292	6 212 963	44.8 s
Germany	15 631 438	9 811 877	79.5 s
DACH	19 399 503	12 247 256	99.0 s
Europe[5]	146 598 088	87 007 242	691.4 s
Earth[5]	756 666 368	412 031 092	4 021.1 s

Memory-Overhead of Contraction Hierarchies. During the contraction hierarchy construction, shortcuts are added to the graph. For the road networks, the number of shortcuts is between 55% and 65% of the final number of edges. The impact on memory usage, however, is far smaller, because edges in the contracted graph only have to be stored either forward or backward, not both. Thus they do not have to be duplicated compared to the graph representation for bidirectional Dijkstra. For Germany, the number of edges increases from 15.6M to 40.4M (increase by 2.58×, 61% shortcuts). However, the memory usage only increases from 1.0 GB to 1.29 GB.

[3]Data is obtained from Geofabrik [18] and converted into an expanded graph. Included road types are: motorway, primary, primary_link, road, secondary, secondary_link, residential, tertiary, tertiary_link, unclassified, trunk, trunk_link, motorway_link.

[4]DACH is an abbreviation for Germany, Austria and Switzerland

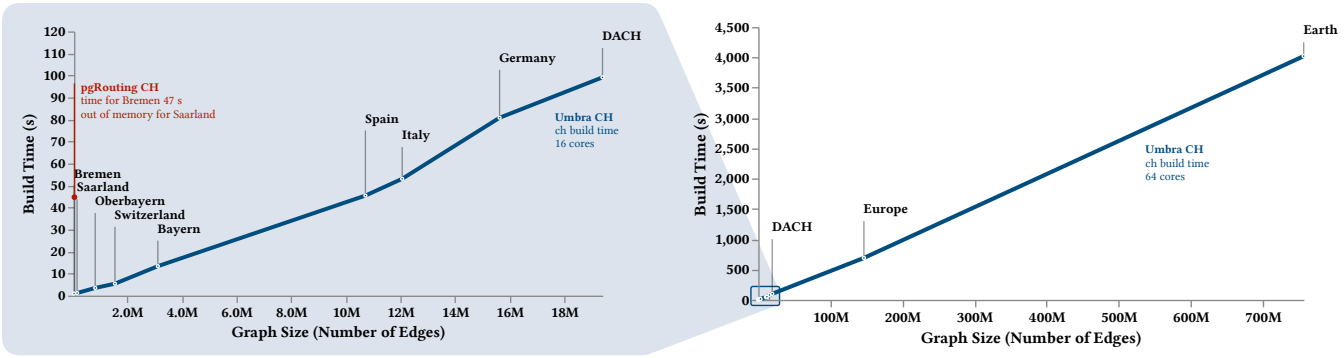

Figure 9: Contraction-Hierarchy Construction Time vs. Graph Size for Different Regions from Bremen to Earth

Figure 9 shows the construction time of the optimized Nav-Index (CH) depending on the graph size.[5] It can be observed that the construction time increases almost linearly with the graph size. This can be explained by the fact that significant parts of the graphs are relatively independent. Only in the final iterations of the contraction hierarchy construction global dependencies arise with a large number of shortcut additions. However, since most nodes of the graph can be contracted independently, these global dependencies do not dominate the total runtime. This independence property is an emergent result of the heuristics and not explicitly modeled.

Comparison to routing engines. To compare our implementation of Nav-Index to a dedicated routing engine, we evaluate the Open Source Routing Machine (OSRM) [24], which provides a parallel contraction hierarchy implementation for road networks and implements the independent node set approach by Vetter [36]. Additionally, we report the runtimes for RoutingKit [11] (a well-known single threaded baseline). OSRM operates on an edge-expanded graph internally, derived from OpenStreetMap data. As OSRM preprocesses and generates the edge-expanded graph with different filters, we export the OSRM expanded graph and import it into Umbra and RoutingKit for a fair comparison. We then measure the build time with 32 threads (Umbra and OSRM) for the contraction hierarchy and the number of shortcuts (%SC, Umbra and RoutingKit). OSRM does not expose this through its CLI.

		OSRM	RoutingKit		Umbra	
Dataset	Edges	Build	Build	%SC	Build	%SC
Bremen	195 k	4 s	2 s	58 %	**1 s**	55 %
Saarland	398 k	6 s	4 s	59 %	**2 s**	55 %
Oberbayern	2 552 k	40 s	37 s	59 %	**6 s**	56 %
Switzerland	4 208 k	43 s	57 s	58 %	**9 s**	54 %
Spain	13 974 k	163 s	375 s	62 %	**55 s**	60 %
Germany	41 395 k	531 s	837 s	60 %	**119 s**	56 %

RoutingKit is faster than OSRM for small graphs but slower for larger graphs due to its single-threaded implementation. Umbra however, always outperforms RoutingKit and OSRM due to its low overhead and efficient parallelization. The number of shortcuts is comparable between the implementations.

[5] Experiments for Europe and Earth were carried out on a single NUMA node of a larger machine (AMD EPYC 7713 64-Core, 500 GB memory) due to memory constraints.

Comparison to pgRouting CH. The Postgres extension pgRouting offers an experimental function for contraction hierarchies [32]. As shown in Figure 9, pgRouting was able to build a contraction hierarchy for the smallest dataset (Bremen) within 47 seconds. For larger datasets pgRouting ran out of memory (given 64 GB). Clearly, the current implementation of contraction hierarchies in pgRouting is not yet suitable for practical use on large road networks.

5.4 Ablation Study

To gain insights on the different optimizations, we performed an ablation study of the different design and implementation decisions. Figure 10 shows the cumulative ablation of the optimizations for the contraction hierarchy construction. The baseline is a naive single threaded contraction hierarchy implementation within Umbra that contracts node by node, updating the priority of all neighbors after each contraction.

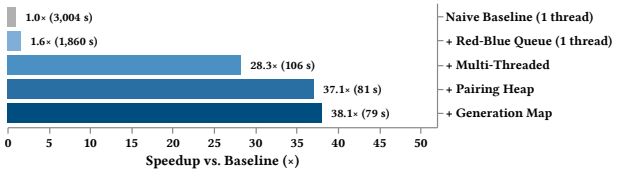

Figure 10: Cumulative ablation of CH build optimizations.

The first optimization is the use of the proposed red-blue queue to find independent nodes that can be contracted batchwise. This already helps due to the reduced number of priority updates (e.g. common neighbors of independent nodes) and improves the runtime by a factor of 1.6. The largest speedup of 13.75× is achieved by parallelizing the contraction of independent nodes and the priority updates. Next, the use of pairing heaps instead of binary heaps for the priority queue helps and yields another 1.33× speedup. Finally, the generation map optimization described in Section 4 leads to a further speedup of 1.04×. Overall, the optimized parallel contraction hierarchy construction is 30.5× faster than the single-threaded baseline implementation.

Parallelization. To further evaluate the scalability of our implementation, we measure the CH build time on the OSM Germany dataset while varying the number of threads. Figure 12 plots the speedup depending on the number of cores for the road graph of Germany.

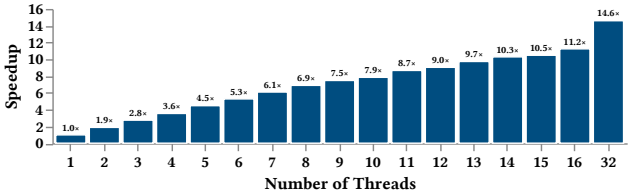

Figure 12: Parallelization of Nav-Index CH build phase. Speedup over single-threaded execution.

The build time improves significantly with an increasing number of threads, demonstrating the effectiveness of our parallelization approach. For 32 threads (including hyperthreading), we achieve a speedup of 14.6× compared to single-threaded execution. The numeric difference to the speedup in the ablation study is due to all optimizations being applied. The increase in speedup from 16 to 32 threads can mainly be attributed to cache misses caused by the graph-traversal access pattern being hidden by hyperthreading.

5.5 Planet Scale CH Construction Analysis

To gain more insights into the performance of the parallel contraction hierarchy construction, we perform a detailed analysis of the build phase for the Earth dataset on an AMD EPYC 7713 64-core machine with 128 threads, as shown in Figure 11. The figure shows the time spent in each iteration of the CH build phase.

During the first iterations, most of the time is spent in the single-threaded selection phase (red) of the red set due to the large number of independent nodes found at the beginning. Due to the heuristics, nodes with few neighbors are likely to be contracted first, which also accelerates the parallel phases. In later iterations, the average degree of nodes increases due to shortcut additions and the order determined by the heuristics. Additionally, shortcuts which span longer distances in the original graph are introduced in later iterations, leading to more time-consuming witness path searches. This leads to an increase in time spent in the parallel contraction phase (blue) in later iterations. However, the number of remaining nodes decreases rapidly, leading to a decrease in iteration time.

While in the first iterations the single-threaded red set selection is a bottleneck in the parallel contraction hierarchy construction, we chose this approach due to its robustness by design (regarding the contraction order) and sacrificed the scalability that could be achieved using the approach by Wan et al. [37].

5.6 Comparison with Other Systems

For a comprehensive evaluation, we compare our implementation to several established graph and relational database systems (supporting shortest path queries). The systems are Neo4j [25], DBMS-X, pgRouting [31], Kuzu [14], and DuckDB PGQ [33, 38]. For each system, we measure the total time taken to compute the running example query Q2 on the OSM Oberbayern dataset. The SQL query executes 1 000 random (but fixed) shortest path queries (for each tuple in the trips relation). All edge weights are set to 1 for compatibility with DuckDB. Since the interfaces of the systems differ significantly, we describe the setup for each system in the following paragraphs.

Neo4j. For the performance evaluation in Neo4j (version 2025.10.1), we use the built-in shortest path functionality exposed via their Cypher query language. To accelerate shortest path searches for a set of queries, we use the Dijkstra implementation from their Graph Data Science library to compute the shortest paths. Since Neo4j exposes a Dijkstra streaming implementation, we can execute multiple shortest path queries in bulk. (Neo4j is running and data is preloaded).

DBMS-X. For the performance evaluation in DBMS-X, we use the built-in shortest path functionality. The interface allows specifying a set of source and destination nodes for which all shortest paths are computed. We use one source and n destination nodes for our evaluation. In theory, it would be possible to optimize this query using a single-source shortest path search. However, DBMS-X seems to execute individual shortest path queries for each destination node, which would be required for a SPSP scenario.

pgRouting. PgRouting, an extension for PostgreSQL, provides routing functionality for geospatial data. For our evaluation, we use the bidirectional Dijkstra implementation. The syntax is very close to our SQL syntax for shortest path queries. The edges are provided via a subquery, while the source and destination nodes are provided via another subquery. Since April 2025, pgRouting also supports the creation of contraction hierarchies. The build phase is evaluated in Subsection 5.3. However, the shortest path search does not support contraction hierarchies yet [32]. Therefore, we only evaluate the bidirectional Dijkstra implementation (pgRouting is running on version 15-3.5-3.8 and data is preloaded).

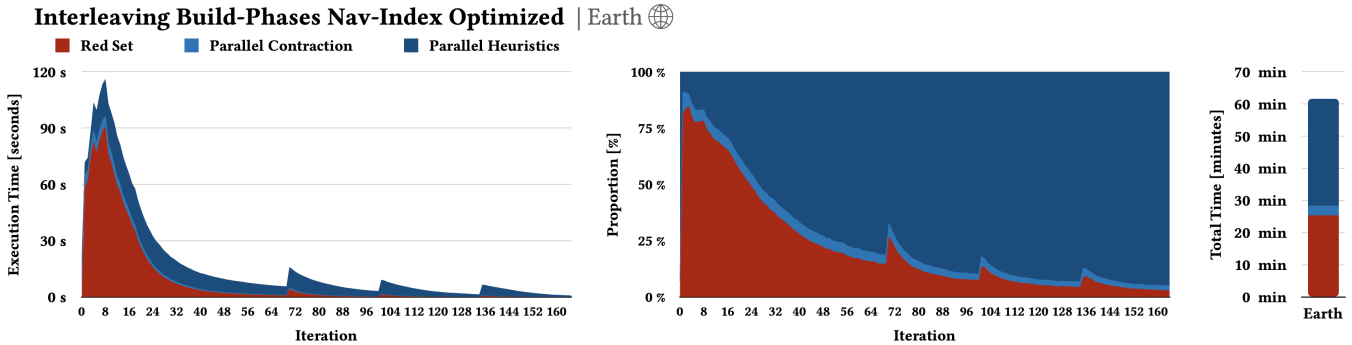

Figure 11: Detailed analysis of CH-index construction time for the Earth dataset. First 165 iterations shown. Red areas indicate the time spent in the red set selection phase. Blue areas indicate the time spent in parallel contraction and priority heuristics.

DuckDB PGQ. For the performance evaluation in DuckDB PGQ, we use the built-in shortest path functionality exposed via their PGQ (version 1.4.1) extension [34]. Since DuckDB PGQ does not seem to support decorrelation of shortest path source and destination nodes yet, we use the same destination node for all shortest path queries and provide multiple source nodes. Our experiments show that DuckDB PGQ executes a multi-source BFS search for all nodes and does not leverage bidirectional- or reverse single-source search optimizations. This allows an imperfect but comparable evaluation to other systems using multiple (src, dest) pairs. Ten Wolde's thesis [34] provides more details on the implementation of shortest path queries in DuckDB PGQ and also evaluates a Bellman-Ford implementation which would allow weighted path searches. However, Ten Wolde states a significant performance disadvantage compared to BFS for unweighted searches. Therefore, we only evaluate the BFS implementation here.

Kuzu. For the performance evaluation in Kuzu (version 0.11.3), we use the built-in shortest path functionality exposed via their Cypher-like query language. Kuzu does not support bulk execution of single-pair shortest path queries, so we execute individual shortest path queries for each src, dest pair with multiple threads and measure the total time. While bulk execution of shortest path queries can be expressed in Kuzu using filters on the query, the execution engine cannot optimize that case: it executes an all-pairs shortest-path algorithm and filters the results, and thus does not terminate. We do not include the VLDB 2025 Kuzu version [8] here, as it overflows for paths longer than 255 hops and these paths cannot be reconstructed.

Umbra. For the performance evaluation in Umbra, we use our Nav-Index implementation as described in this paper. We are using Q2 from our running example. For the table-index mode, we first create and load data into a table and execute the following SQL statement:

```
create index on edges using nav(src, dest, cost);
```

Afterwards, the same Q2 query is executed. Umbra's query optimizer automatically selects the table index for execution.

Results. Figure 13 shows the throughput in queries per second for 1 000 randomly chosen shortest path queries on the OSM Oberbayern dataset (827 682 edges). The average hop count of paths found is 273. It can be observed that the table index mode outperforms the other modes of Nav-Index and other systems by more than two orders of magnitude. This is expected as the precomputed table index (build time 3.6 seconds) allows for fast shortest path queries (sub millisecond, depending on the actual path). The adaptive mode builds a contraction hierarchy on-the-fly and performs the queries leveraging the CH graph. Increasing the number of queries could amortize the build time of the contraction hierarchy even better such that the adaptive throughput would approach the table index throughput. The forced Dijkstra-only mode performs worse than the adaptive mode due to its algorithmic disadvantages.

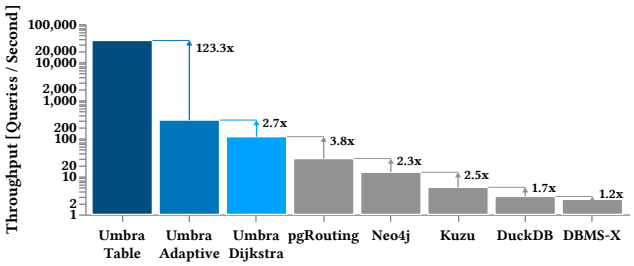

Figure 13: Throughput in queries per second for 1000 shortest path queries on OSM Oberbayern dataset

If we increase the number of queries |Q|, the adaptive mode outperforms the Dijkstra-only mode, benefiting from the contraction hierarchy acceleration. However, the table-index lookups play in a different complexity class: 2 000 queries on the Umbra Nav-Table-Index take 39 ms. Consequently, for analytical workloads with many shortest path queries on road networks, Nav-Index *adaptive* and *table-index* are the only approaches capable of delivering interactive query performance. This demonstrates the significance of data structures and algorithms for single-pair shortest path queries in RDBMS. For single pairs, the only efficient option is to drastically reduce the search space as intermediate results cannot be reused for multiple destination nodes (compared to single-source Dijkstra or BFS). These SPSP optimizations are still missing from many modern GraphDBMS, only pgRouting targets them specifically, which is clearly visible in the performance results.

5.7 Twitter Social Circles Dataset

To evaluate the performance of Nav-Index on social network graphs, we use the (combined) Twitter Social Circles Dataset [23], which contains 81 306 nodes and 2 420 766 unweighted (weight = 1) directed edges. We measure the total time taken to compute randomly chosen shortest path queries on this dataset using the same systems as in Section 5.6. Compared to road networks, social network graphs typically have a lower diameter and higher connectivity, resulting in shorter paths with fewer hops. For this Twitter dataset, the average shortest path length is 4.5 hops and the diameter is 7.

System	Time
Nav-Index Table Index	34 ms
Nav-Index Adaptive (optimizer chooses Dijkstra)	3 557 ms
Kuzu	6 220 ms
pgRouting	8 986 ms
DBMS-X	13 422 ms
DuckDB PGQ	18 330 ms
Neo4j	20 043 ms

Clearly, Nav-Index in Umbra outperforms all other systems significantly. Again, the table-index mode outperforms both other Nav-Index modes by a factor of 105. In adaptive mode, Nav-Index considers building an ad-hoc table index but notices the increased density (higher average degree) of the graph during execution and avoids building a contraction hierarchy. Instead, it employs the Dijkstra-only search, such that both modes achieve identical performance. In our experiments, the CH table index takes 204 seconds to build, which cannot be amortized for an ad hoc index but can still speed up analytical workloads (especially for interactive data exploration) significantly.

6 RELATED WORK

We categorize related work into two groups: existing algorithms and index structures for shortest path computations, and existing database systems supporting shortest path queries.

Algorithms and Index Structures for Shortest Path Computations. Due to its importance in computer science, shortest path computation has been extensively researched, yielding numerous algorithms and indexing approaches. A comprehensive survey is given by Bast et al. [3]. Classical algorithms for shortest path search include Dijkstra's algorithm [12] and A* [19]. While Dijkstra's algorithm is the classical generic approach for shortest path search, A* is a heuristic-guided approach that requires domain-specific knowledge to define a suitable heuristic function, going beyond the graph's weighted edges. Dijkstra's algorithm can be further improved using bidirectional search [3] or combining it with Bellman-Ford [5, 13]. However, both algorithms still require exploring a large portion of the graph for long-distance queries. If the graph is static for multiple queries, index structures can speed up shortest path computations. Contraction hierarchies, introduced by Geisberger et al. [17], are one of the central index structures for shortest path computations in road networks. Other techniques include hub labeling [1, 9] and Transit Node Routing [4], which typically use or extend contraction hierarchies to improve performance. Customizable contraction hierarchies [11] support dynamic edge weights through an additional customization phase but are more susceptible to graphs that incorporate turn restrictions and turn costs [7, 21]. Although contraction hierarchies were originally designed with road networks in mind, our evaluation shows that they can also be effectively applied to other graphs such as social networks. Nav-Index combines the generality of Dijkstra's algorithm with the performance of contraction hierarchies in an adaptive index structure suitable for shortest path queries in RDBMS. Compared to standalone implementations of shortest path algorithms, our Nav-Index implementation in Umbra is tightly integrated into relational algebra and can be optimized and executed alongside other relational operations. This allows Nav-Index to achieve performance comparable to dedicated routing engines while providing the versatility of a declarative SQL interface with interleaved relational and graph operations.

Database Systems Supporting Shortest Path Queries. Two of the most prominent graph database systems supporting shortest path queries are Neo4j [25] and TigerGraph [35]. Both systems offer query languages to express graph queries, including shortest path queries. Both systems implement specialized operators for shortest path computations using Dijkstra's algorithm [12] or A* [19]. Among relational DBMS, pgRouting [31] is an extension for the relational database system PostgreSQL to support geospatial and routing queries. It implements various shortest path algorithms, such as Dijkstra, A*, and Bellman-Ford. However, pgRouting does not integrate shortest paths tightly into relational algebra. Instead, relations for the graph and queries are passed as query strings to pgRouting functions. Recently, a first experimental implementation of contraction hierarchies was added to pgRouting [32], highlighting that such an implementation is desirable. However, their current CH implementation is still very limited in its functionality. For example, it cannot leverage contraction hierarchies in its shortest path functions (yet), and our tests show that it is not capable of handling large graphs (e.g., Saarland, a small German state, is already too large). DuckDB [33] added support for SQL/PGQ [38], which includes shortest path queries. DuckDB PGQ currently supports weighted and unweighted shortest path queries with significant performance advantages for unweighted Breadth-first search (BFS) [34]. Still, even with BFS, the performance is still limited on large graphs as shown in our evaluation. Kuzu [8, 14] similarly accelerates shortest paths through parallel BFS-style traversal. Like DuckDB, it is optimized for single/multi-source exploration rather than the single-pair queries Nav-Index targets, which is reflected in our evaluation. Compared to existing systems, Umbra with Nav-Index combines the versatility of RDBMS with the performance of contraction hierarchies. The adaptivity of Nav-Index allows Umbra to bridge the gap between the generality of Dijkstra searches and the performance of contraction hierarchies on sparse graphs. The tight integration into relational algebra allows even correlated shortest path queries and mixed relational and graph operations.

7 CONCLUSION

Nav-Index is an adaptive index structure to efficiently support shortest path queries in relational database management systems, with three modes of operation: a generic mode using a bidirectional version of Dijkstra's algorithm for general directed graphs with non-negative edge weights, an optimized ad-hoc mode leveraging contraction hierarchies for sparse graphs such as road networks, and a table index mode also leveraging contraction hierarchies for static graphs. We implemented Nav-Index in Umbra and evaluated it using real-world road networks and social networks. Our results show that Nav-Index significantly outperforms existing implementations in both relational and graph database systems on both single shortest path queries as well as analytical workloads involving many shortest path queries. The implementation of Nav-Index scales to large graphs with hundreds of millions of edges and parallelizes well on high core counts. A Nav-Table-Index on the whole planet with over 756 million edges can be constructed in 1h 8m and answers shortest path queries in a few milliseconds. The road network of Europe with 146 million edges can be indexed in under 12 minutes and could therefore be rebuilt frequently to support live traffic updates. To the best of our knowledge, Nav-Index is the first implementation of contraction hierarchies in a general-purpose RDBMS that is able to scale to such large graphs and efficiently support shortest path queries, bridging the gap between specialized shortest path algorithms and general-purpose relational database systems. The integration on the relational algebra level allows Nav-Index to leverage existing optimizations such as cost-based optimization, parallel execution, pipelining, code generation, and allows shortest path queries to be seamlessly integrated into larger analytical workloads. Future work may leverage Nav-Index for new applications and analytics in logistics, transportation, and social networks. Future research may also further optimize single-source or multi-source scenarios. In-Umbra experiments showed that optimizing for MSSP and SSSP can accelerate the search further by 1.2× to 1.5× if source or destination nodes recur or are constant. These optimizations are however orthogonal to Nav-Index's design and can benefit from related work and be explored in future work.

REFERENCES

[1] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. 2011. A hub-based labeling algorithm for shortest paths in road networks. In *International Symposium on Experimental Algorithms*. Springer, 230–241.

[2] Renzo Angles, János Benjamin Antal, Alex Averbuch, Altan Birler, Peter Boncz, Márton Búr, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, et al. 2020. The LDBC social network benchmark. *arXiv preprint arXiv:2001.02299* (2020).

[3] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F Werneck. 2016. Route planning in transportation networks. In *Algorithm engineering: Selected results and surveys*. Springer, 19–80.

[4] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. 2007. Fast routing in road networks with transit nodes. *Science* 316, 5824 (2007), 566–566.

[5] Richard Bellman. 1958. On a routing problem. *Quarterly of applied mathematics* 16, 1 (1958), 87–90.

[6] Altan Birler, Alfons Kemper, and Thomas Neumann. 2024. Robust join processing with diamond hardened joins. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3215–3228.

[7] Valentin Buchhold, Dorothea Wagner, Tim Zeitz, and Michael Zündorf. 2020. Customizable contraction hierarchies with turn costs. In *20th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2020)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 9–1.

[8] Anurag Chakraborty and Semih Salihoglu. 2025. Robust Recursive Query Parallelism in Graph Database Management Systems. *Proc. VLDB Endow.* 18, 11 (2025), 4465–4477. https://doi.org/10.14778/3749646.3749706

[9] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355.

[10] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.

[11] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. 2016. Customizable contraction hierarchies. *Journal of Experimental Algorithmics (JEA)* 21 (2016), 1–49.

[12] Edsger W Dijkstra. 2022. A note on two problems in connexion with graphs. In *Edsger Wybe Dijkstra: his life, work, and legacy*. 287–290.

[13] Ran Duan, Jiayi Mao, Xiao Mao, Xinkai Shu, and Longhui Yin. 2025. Breaking the Sorting Barrier for Directed Single-Source Shortest Paths. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing*. 36–44.

[14] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. 2023. Kùzu graph database management system. In *The Conference on Innovative Data Systems Research*, Vol. 7. 25–35.

[15] Michael L Fredman, Robert Sedgewick, Daniel D Sleator, and Robert E Tarjan. 1986. The pairing heap: A new form of self-adjusting heap. *Algorithmica* 1, 1 (1986), 111–129.

[16] Michael L Fredman and Robert Endre Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)* 34, 3 (1987), 596–615.

[17] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International workshop on experimental and efficient algorithms*. Springer, 319–333.

[18] Geofabrik GmbH. 2024. Geofabrik OpenStreetMap Data Extracts. https://download.geofabrik.de Accessed: 2026-07-20.

[19] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

[20] ISO/IEC. 2023. ISO/IEC 9075-16:2023: Information technology — Database languages SQL — Part 16: Property Graph Queries (SQL/PGQ). https://www.iso.org/standard/79473.html Accessed: 2025-12-11.

[21] Olivier Klukas. 2024. *Efficient Customizable Contraction Hierarchies for Road Networks*. Master's thesis. Technical University of Munich, Munich, Germany.

[22] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 743–754.

[23] Jure Leskovec and Julian Mcauley. 2012. Learning to discover social circles in ego networks. *Advances in neural information processing systems* 25 (2012).

[24] Dennis Luxen and Christian Vetter. 2011. Real-time routing with OpenStreetMap data. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (Chicago, Illinois) *(GIS '11)*. ACM, New York, NY, USA, 513–516. https://doi.org/10.1145/2093973.2094062

[25] Neo4j. 2025. Neo4j. https://neo4j.com/ Accessed: 2026-07-20.

[26] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.

[27] Thomas Neumann. 2025. Improving Unnesting of Complex Queries. In *Datenbanksysteme für Business, Technologie und Web (BTW 2025)*. Gesellschaft für Informatik, Bonn, 25–47.

[28] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*.

[29] Thomas Neumann and Alfons Kemper. 2015. Unnesting arbitrary queries. *Datenbanksysteme für Business, Technologie und Web (BTW 2015)* (2015).

[30] OpenStreetMap contributors. 2017. Planet dump retrieved from https://planet.osm.org . https://www.openstreetmap.org. Accessed: 2026-07-20.

[31] pgRouting Community. 2025. pgRouting. https://pgrouting.org/ Accessed: 2026-07-20.

[32] pgRouting Community. 2025. pgRouting Contraction Hierarchies. https://docs.pgrouting.org/latest/en/pgr_contractionHierarchies.html Accessed: 2026-07-20.

[33] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.

[34] Daniël ten Wolde. 2022. *Integrating SQL/PGQ into DuckDB*. Ph.D. Dissertation. Centrum Wiskunde & Informatica.

[35] TigerGraph. 2025. TigerGraph. https://www.tigergraph.com/ Accessed: 2026-07-20.

[36] Christian Vetter. 2009. *Parallel time-dependent contraction hierarchies*. Student Research Project. Universität Karlsruhe (TH). 134 pages. https://ae.iti.kit.edu/download/vetter_sa.pdf Accessed: 2026-07-20.

[37] Zijin Wan, Xiaojun Dong, Letong Wang, Enzuo Zhu, Yan Gu, and Yihan Sun. 2025. Parallel contraction hierarchies can be efficient and scalable. In *Proceedings of the 39th ACM International Conference on Supercomputing*. 670–688.

[38] Daniel ten Wolde, Gábor Szárnyas, and Peter Boncz. 2023. DuckPGQ: Bringing SQL/PGQ to DuckDB. *Proceedings of the VLDB Endowment* 16, 12 (2023), 4034–4037.