American Journal of Information Science and Technology
2026, Vol. 10, No. 3, pp. 82–100
https://doi.org/10.11648/j.ajist.20261003.11

SciencePG
Science Publishing Group

Research Article

Optimization of Query Processing Through Nested Persistent Storage Implementation of Relational Tables

Hammad Alenezi*, Janusz Getta, Tianbing Xia

School of Computing and Information Technology, University of Wollongong, Wollongong, Australia

Abstract

Relational databases use physical design techniques such as indexing, clustering, and partitioning to improve performance without altering the logical schema seen by applications. This paper proposes nesting of persistent storage structures as a new physical optimisation technique. Related tables can be implemented as a single nested persistent storage structure, eliminating join operations when processing complex queries and removing redundant foreign key repetitions on the many side of the relationship. The paper describes two independent nesting strategies. The schema driven approach selects nesting candidates from the logical schema before any workload is known. This selection subject to storage optimisation and database organisation constraints, formulating the selection as a constrained maximum-weight branching problem. The workload driven approach determines nesting decisions subject to aggregate query processing frequencies, applying nesting transformations in descending order of join frequency. To preserve logical transparency, mapping metadata records each attribute's physical access path. The paper shows how SQL queries expressed over the logical schema can be automatically transformed into equivalent queries over the nested physical structures, keeping the nested representation hidden from applications. A prototype implementation in PostgreSQL on the TPC-H benchmark demonstrates the feasibility of the proposed approach and confirms that standard indexing techniques can recover the query performance overhead introduced by nesting. By reducing storage redundancy and eliminating join operations while remaining compatible with existing relational interfaces, the technique preserves the relational programming model.

Keywords

Physical Database Design, Schema Nesting, Join Elimination, Workload-driven Optimisation, Query Rewriting, Logical Transparency, Mapping Metadata

1. Introduction

The structural design and implementation of relational tables have the most important impact on performance of relational databases. A correct mapping of a *logical view* of a database into its *physical view* may significantly improve system performance and storage efficiency. In a traditional approach, a *logical view* of a relational database is a set of well-normalized relational tables. A *physical view* of a relational database includes information about the persistent storage structures used to keep the contents of relational tables and about data access methods to efficiently process data kept in relational tables. Well-known specialized data access techniques, such as indexing, clustering, and partitioning, are commonly used to improve the performance of data processing. They change the ways in which tuples are stored and accessed. In this work

*Correspondence: Hammad Alenezi (hgha144@uowmail.edu.au)

Received: 8 June 2026; **Accepted:** 22 June 2026; **Published:** 28 July 2026

we consider implementation of relational tables as nested persistent storage structures to improve the performance of query processing.

At the physical level, a relational table can be implemented in several ways. The most common implementation stores tuples as sequences of flat records in data blocks. Columnar implementation of relational tables stores data column by column rather than row by row, consolidating values of each attribute into several consecutive data blocks. This layout improves access efficiency for analytical queries that scan specific attributes across many rows. Rows from a group of related tables may alternatively be physically co-located in the same sequence of data blocks through clustering. Indexing extends these base storage layouts with access structures that support efficient selective retrieval without altering the underlying row organization. Partitioning divides a table's rows across separate storage segments according to a partition key, reducing the volume of data scanned by queries that target a restricted key range. Although these implementations differ in how data is physically organized and accessed, they share a common property that the logical schema of a database remains unchanged in all cases. Applications continue to see a collection of flat relational tables and express queries over that view.

This paper proposes nesting of the rows from logically related relational tables as a physical implementation technique, treating it in the same spirit as indexing, clustering, and partitioning. Instead of storing each relational table as an independent collection of flat rows, logically related tables are implemented as a single nested persistent storage structure in which rows of a dependent table are embedded within the rows of their parent table. This approach is conceptually related to clustering, as both techniques aim to improve data locality between related tables. However, the two differ in how this is achieved. Clustering preserves logically separate tables while co-locating their rows within shared data blocks. Nesting, by contrast, embeds dependent rows directly within their parent rows, forming a hierarchical physical structure.

Nesting of persistent storage structures is beneficial under two circumstances. First, it adjusts the physical view of a database to the characteristics of a logical view. The fragments of hierarchically oriented logical views can be implemented in a natural way as the nested persistent storage structures. When a dependent table contains many rows per parent row, the foreign key attributes of the dependent table are repeated across all its tuples, forming a significant portion of its storage cost. In this case nesting eliminates these repetitions, reducing the total volume of stored data. Minimizing storage consumption is critical for in-memory database systems. In-memory database systems, in which the entire working dataset is maintained in main memory rather than on persistent storage, represent a rapidly growing class of database deployments [1]. The storage reduction achieved by nesting is particularly relevant in this context. In disk-based systems, a reduction in stored data volume reduces I/O cost and improves cache utilization; however, the large capacity of persistent storage limits the practical impact of moderate savings on overall system behaviour. In main-memory systems, by contrast, every reduction in data volume directly shrinks the working set held in RAM, where capacity is substantially more constrained and the cost per unit of storage is considerably higher [1]. The elimination of redundant foreign key repetitions through nesting can therefore meaningfully reduce memory pressure and improve system throughput in in-memory database environments, making the storage objective of the schema-driven nesting phase directly applicable to this increasingly prevalent class of systems.

The second advantage of nesting is the elimination of frequent join operations. When a workload repeatedly joins two or more tables, nesting them eliminates the need for a runtime join, significantly reducing query execution costs. These two advantages of nesting correspond to different stages of the database lifecycle: the first arises before any workload is known, while the second requires a workload profile to guide nesting decisions. Accordingly, this paper addresses the nesting selection problem using two different strategies.

Schema-driven nesting operates on the logical database schema graph before any workload is known. It identifies one-to-many relationships in which repeated foreign key values account for a significant portion of storage cost and selects nesting decisions that maximise storage savings. *Workload-driven nesting* operates on a frequency-weighted schema graph derived from a representative workload. It selects nesting transformations guided by the execution frequency of join operations, prioritizing the joins that contribute most to overall workload cost. Both strategies produce a nested schema graph that encodes all nesting decisions and serves as input to the mapping and query rewriting procedure. Database administrators can further guide both strategies by supplying constraints that identify relationships which must not be nested or must always be nested. Having established the criteria for selecting nesting decisions, it remains to show how such decisions can be implemented in practice using existing database system capabilities.

Nesting in modern relational database systems can be quite easily implemented through relational tables with columns that contain nested JSON documents. In such approach, at a logical level a database schema is still a collection of flat and well normalized relational tables like in a traditional view of relational databases. At a physical level, implementation of selected relational tables is different. Some of the chains of relational tables linked through referential integrity constraints are implemented as nested JSON documents stored in the relational tables in the columns of type JSON. Such approach requires transformation of SQL statements operating on flat relational tables into SQL statement operation on relational tables and JSON documents stored within the relational tables.

Beyond enabling physical nesting, this transformation layer

offers a significant benefit to application developers by simplifying the programming and view of a database. Without such a way, applications that access data stored in JSON columns must operate at two incompatible levels of abstraction within a single query. Standard SQL at the outer relational level, and a path-based navigational syntax for accessing fields within the embedded JSON structures. This approach makes queries significantly harder to write, read, and maintain. By exposing applications to a flat relational view while internally generating the appropriate mixed SQL and JSON access expressions, the transformation layer allows all queries to be written in pure relational SQL, regardless of how the underlying data is physically stored. This benefit holds independently of the specific motivation for adopting nesting. Whether nesting is introduced for storage savings, join elimination, or both, the transformation layer uniformly preserves the relational programming model for all applications.

Since nesting only alters the physical storage structures rather than the logical schema, queries remain compatible with the original relational view. To process such queries, the system must internally rewrite them. This rewriting is driven by *metadata*, which contains information on how each logical attribute is represented in the nested physical schema. Using this metadata, the system automatically rewrites queries before execution, keeping the underlying physical structure entirely transparent to applications and users. Achieving logical transparency of nesting is a primary design objective of the proposed approach, as it preserves the strict separation between logical design and physical tuning. Application developers do not need to understand or track nesting decisions. Database administrators can introduce, adjust, or remove nesting over time without modifying any application code. Nesting is also compatible with other physical techniques. The system can combine nesting with indexing, clustering, and partitioning while the logical interface exposed to applications stays unchanged. Against this background, the paper makes the following three contributions.

This paper pursues three research objectives: (1) to develop a set of rules to decide how to nest persistent storage structures based on both the logical view of data and how data is used, (2) to define mapping metadata and a system-independent query rewriting procedure that preserves logical transparency when the physical schema is nested, and (3) to demonstrate that the framework can be realized in a concrete database system using standard features without modification to the logical schema.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 introduces the basic concepts and definitions used throughout the paper. Section 4 presents the schema-driven nesting phase. Section 5 discusses workload-driven nesting. Section 6 defines the mapping metadata M and the query transformation process. Section 7 describes the implementation in PostgreSQL.

2. Related Work

The foundational separation between logical and physical database design was established by Codd [2], whose relational model specified that applications should interact with data through a logical schema of flat normalized tables while the physical organization of that data remains independently adjustable by the system. This principle of physical data independence has informed several decades of research on techniques that modify physical storage while preserving the relational interface visible to applications. Indexing, clustering, partitioning, and their automated selection across diverse workloads have been subjects of extensive investigation, as surveyed by Zhou et al. [3]. The present work extends this tradition by introducing schema nesting as a further physical optimization technique applied at the storage level.

The observation that hierarchically related data can be represented more compactly through nesting has a long history in the database literature. Schek and Scholl [4] introduced the relational model with relation-valued attributes, commonly referred to as the NF^2 model. This model permits table cells to contain relations rather than scalar values and provides a formal basis for hierarchical data representation, alongside corresponding algebraic and logical foundations. These works demonstrate that nesting can eliminate redundant key values, reduce storage overhead, and simplify query expressions over hierarchically related data. However, they treat nesting as a logical modeling decision. The nested structure forms part of the schema presented to applications, and query languages must be redesigned to operate natively over nested collections. By contrast, the present work treats nesting strictly as a physical implementation choice, leaving the logical schema and application queries unchanged.

The problem of selecting physical structures to improve query performance over a fixed logical schema has been studied extensively within the tradition of automated physical database design. A succession of workload-driven physical design advisors demonstrated the viability of automated tuning systems capable of jointly selecting indexes, materialized views, and partitioning structures guided by optimizer cost estimates. Kossmann et al. provides a rigorous comparative evaluation of these systems, analyzing trade-offs between solution quality and computational cost across a representative range of approaches and workloads [5]. More recently, reinforcement learning has been applied to physical database design. Perera et al. [6] proposed HMAB, a hierarchical bandit framework that integrates index and view selection in a single adaptive search space and adjusts its recommendations in response to workload changes at runtime. Machine learning has also been applied more broadly to automate physical layout decisions, including partitioning for cloud-deployed databases [7]. Leis et al. provides an empirical characterization of join optimization quality in industrial systems using the Join Order Benchmark [8]. Their findings show that cardinality estimation errors propagate significantly into multi-join plans even in mature production optimizers. This observation reinforces the value of eliminating joins entirely through nesting when workload statistics support this decision. The present work is

positioned within this same tradition but differs in that it restructures the physical storage layout of the tables themselves, rather than adding auxiliary structures on top of an unchanged layout.

Vertical partitioning offers a further point of comparison, as it also modifies the physical layout of data to reduce the volume accessed by a query without changing the logical schema. Navathe et al. [9] formulated vertical partitioning as a graph problem over attribute access affinity and solved it with combinatorial algorithms that maximize I/O efficiency for a given workload. The comprehensive survey by Abadi et al. [10] demonstrates that separating attribute storage across independent columns yields substantial performance gains for read-intensive analytical workloads. These gains arise primarily through projection of only the required attributes and tighter compression. Alkowaileet and Carey [11] extend these ideas to nested columnar layouts within an operational document store, showing that hierarchical physical encodings can be combined with columnar projection to support both relational and semi-structured access patterns. Li et al. [12] further demonstrate that universal columnar file formats originally designed for analytical workloads can be adapted to serve fast transactional operations. The storage efficiency benefits shared across these physical layout techniques are further amplified in main-memory database environments [1]. In such systems, the entire working dataset resides in RAM, capacity is substantially more constrained than on disk, and every reduction in data volume directly improves system throughput. Schema nesting is independent of these layout approaches and can be viewed as a compatible technique within the broader physical design space.

The growth of JSON as a data interchange format has introduced a class of systems in which semi-structured data is managed within or alongside relational engines. Bourhis et al. [13] provide the first formal data model and query language specification for JSON documents, establishing the theoretical foundation for reasoning about nested key-value structures and their query semantics. The PostgreSQL JSONB binary type [14] provides the concrete storage substrate for the implementation described in Section 7 of this paper, supporting field-access operators, LATERAL unnesting via jsonb_array_elements, and GIN and B-tree expression indexing. Durner et al. [15] addressed the related problem of accelerating analytics over semi-structured JSON in their JSON Tiles system. The system automatically identifies frequently accessed keys and extracts them into a relational-like physical layout while preserving the original document interface. The present work takes the opposite direction: it starts from a relational schema and applies nesting to improve performance, keeping the relational interface unchanged.

The problem of preserving query correctness across a transformation of the physical schema is closely related to the general problems of query processing under views and data exchange. Halevy [16] surveyed the conditions under which a query expressed over a source schema can be reformulated to use only available view definitions, establishing the maximally contained rewriting framework as the theoretical standard for correctness preserving query translation. Fagin et al. [17] formalized data exchange as the problem of mapping data from a source schema to a target schema while preserving query semantics. They characterized the classes of queries for which translated answers are guaranteed to be complete. Kossmann et al. survey how data dependencies enable a broad range of query rewriting and join elimination optimizations in relational systems [18]. These dependencies include functional dependencies and inclusion dependencies that support primary-key–foreign-key relationships. Their work provides a formal inventory of dependency-driven techniques closely related to the join elimination step in the rewriting procedure. The present work draws on these general frameworks but instantiates them in a specific and constructive form suited to the context of physical nesting within a relational system.

3. Basic Concepts

The logical schema of relational database is represented as a schema graph $G = (N, E, L)$ where each node $n \in N$ corresponds to a base table of the schema, $E \subseteq N \times N$ is a set of directed edges such that each edge $(n_i, n_j) \in E$ represents a referential integrity constraint where a foreign key in a table n_j references a primary key in table n_i. The labelling L attaches to each node $n \in N$ the schema of the corresponding table.

On the schema graph G, the cost model assigns to each edge $e \in E$ a numeric value $w(e)$, called the weight of e representing the storage benefit of nesting along that edge. The full computation of $w(e)$ is defined in Section 4.3.

The nested logical schema is represented as a nested graph $G_N = (N, E_s, E_d)$ where each node $n \in N$ corresponds to a table (either an original base table that remains unnested or a host table that contains one or more tables nested within it). Solid edges $E_s \subseteq N_i \times N_j$ represents referential integrity constraints that remain as candidates for nesting.

Dashed edges $E_d \subseteq N_i \times N_j$ represents committed nesting decisions: when a nesting transformation is applied along an edge, the corresponding solid edge is removed from E_s and a dashed edge is added to E_d.

A workload W is a set $\{q_1, \ldots, q_k\}$ of SQL queries. For each query $q \in W$, the query graph G_q is the subgraph of the schema graph G that contains tables referenced in a query q and the referential constraints used as join conditions in a query q.

The mapping metadata M describes how each logical table and attribute is represented in the current physical schema, including any nesting decisions. Formally, M maps each logical attribute reference to a complete dot-separated physical access path, defined as $M : A \rightarrow P$ where: A is the set of attribute references in the original schema, each of the form $R.a$, with R a table and a an attribute; and P is the set of dot-separated paths of the form $r.n_1.n_2.\ldots.n_k.a$, where r is the physical root table, $n_1, n_2, \ldots, n_k$ are the nested components traversed in order from r, and a is the target attribute name. For a root-level

attribute, the path reduces to $R.a$. The root table is always recoverable as the first element of the path, and the expansion requirement is derived from the sequence of intermediate components between the root and the attribute name. M is used to rewrite queries automatically so that nesting remains transparent to applications.

4. Schema-driven Nesting

After a stage of logical database design, a logical schema is known, but there is no detailed information about the workload yet. For example, a database may be designed in advance for an application that is still under development. The schema-driven nesting operates purely at the schema level. It takes the schema graph and basic statistics as input, together with DBA constraints, and proposes a nested physical schema before any workload is known. This nesting looks for the relational tables implementing one-to-many relationships where repeated values of foreign key dominate storage cost. The repetitions are replaced with nested structures, subject to limits on table size, row length, and storage overhead. The core task is to determine which one-to-many relationships should be implemented as nesting. There are two stages before applying the main algorithm to the logical schema graph G. First, DBA anti-nesting constraints are applied as edge deletions on the schema graph and second, weights are assigned to the edges of the schema graph. Schema-driven nesting is formulated on a weighted directed schema graph. Each candidate one-to-many edge carries a weight that reflects the storage benefit of nesting along that edge. We treat the problem as a constrained maximum-weight branching problem and solve it with Chu–Liu/Edmonds algorithm, extended with DBA and physical constraints.

4.1. Optimization Objectives and Constraints

Consider a relationship between tables R and S where the foreign key in S references the primary key of R. If each row of R is referenced on average by n rows of S, then the key attributes from R appear roughly n times. When n is too large, the repeated key values form a significant part of the storage cost of the many-side table. Implementing this relationship as nesting removes these repetitions: the key from R is stored once in the host row and the dependent rows of S are stored as a nested collection. The effect is a reduction in the number of bytes dedicated to key attributes and a reduction in overhead from separate tables and blocks.

Schema-driven nesting is therefore governed by two competing objectives. The first objective is to maximize storage savings by eliminating the repeated values of foreign keys through nesting. The second objective is to limit the width of the host row after nesting, since embedding dependent rows increases the row length of the host table and, if left unconstrained, can produce rows that are too wide to store or process efficiently.

Formally, these two objectives impose the following constraints on any candidate nesting. A nesting is only admissible if (i) the storage saving obtained by eliminating repeated key values outweighs the overhead introduced by the nested collection, and (ii) the resulting row length of the host table after nesting does not exceed a configurable maximum row-length threshold ρ. Both constraints must be satisfied simultaneously; a candidate nesting that achieves high storage savings but violates the row-length bound is rejected.

The schema-driven algorithm must therefore select edges where the storage saved by eliminating repeated keys outweighs the extra space needed for nested collections and must discard any candidate whose post-nesting row length exceeds ρ, avoiding configurations that would produce rows that are too wide or for which the total size of the nested table is too large.

4.2. DBA Constraints

DBA constraints capture predefined information about which edges of the schema graph may or may not be used for nesting. For the logical schema graph G, where each potential nesting is represented by a directed edge $e = (u,v) \in E$ from node u to node v. We formalize DBA constraints as a pair $f=(F_{forbid},F_{force})$ where $F_{forbid} \subseteq E$ is a set of *forbidden nesting* (anti-nesting constraints) and $F_{force} \subseteq E$ is a set of *forced nesting* (pro-nesting constraints). Forbidden nesting is applied by removing all edges in F_{forbid} from E, which prevents the corresponding pairs (u,v) from being used in any nesting hierarchy. Forced nesting are applied by identifying the subgraphs produced by edges in F_{force} and collapsing each such subgraph into a single node, redirecting incoming and outgoing edges to this new node while preserving connectivity. This reduction is performed only on a working copy of the schema graph; the original logical schema graph G is preserved so that all referential integrity constraints remain available for implementation and for mapping back to the relational view. After applying constraints in F, we obtain a transformed schema graph that already reflects both anti-nesting and pro-nesting decisions. All subsequent steps of the schema-driven nesting algorithm, including weight assignment and the use of Chu–Liu/Edmonds' algorithm, operate on this graph.

4.3. Cost Model

The cost model measures the storage impact of candidate nesting and supplies the edge weights $w(e)$ used by the schema-driven algorithm. Its role is to determine how valuable it is to apply a certain nesting and to decide the best nesting when several alternatives exist on the same fragment of the schema. Consider an edge $e= (u,v)$ that remains after applying DBA constraints, where rows of v are candidates to be nested into rows of u. Let $rows(u)$ and $rows(v)$ denote the number of rows in tables u and v, respectively. Each row of u is on average referenced by

$$n = rows(u) / rows(v) \quad (1)$$

Let *attrs(v)* be the set of attributes of *v* that are stored inside *u* in nested form and let *size(attrs(v))* denote the total size of these attributes for a single row of *v*. Under the nesting, the expected amount of data from table ***v*** added to each row of table *u* is then

$$d = n * size(attrs(v)) \quad (2)$$

Let ***r*** denote the average row length of *u* before nesting and ***r′*** the average row length after nesting. The model sets

$$r' = r + d \quad (3)$$

The row-length constraint introduced in Section 4.1 requires that $r' \leq \rho$, where ρ is the maximum admissible row length. Any candidate edge $e = (u, v)$ for which $r + d > \rho$ is immediately disqualified and removed from consideration, regardless of its storage saving.

Let *t(u)* and *t(v)* denote the total storage occupied by tables *u* and *v* before nesting, then the corresponding total storage before nesting S_0 is approximated by

$$S_0(e) = t(u) + t(v) \quad (4)$$

Once nesting applies, the total storage after nesting S_1 is calculated using the formula:

$$S_1(e) = t(u) + (d* rows(u)) \quad (5)$$

Using the formulas given above, we obtain for each edge $e = (u, v)$ a weight *p*. In prepossessing step, we need to find the benefit value that will be assigned to each edge. For each edge *e,* we estimate the storage cost before and after nesting as calculated in (4) and (5), using values provided. The benefit used as the edge weight is:

$$p(e) = S_0(e) - S_1(e) \quad (6)$$

This result will be attached to the edge as required by Chu–Liu/Edmonds' algorithm. The full vector *w(e)* is retained for later stages, where complete nesting candidates are checked against row-length and storage constraints.

4.4. Identifying Nesting Candidates

After preprocessing, which applies the DBA constraints $f = (F_{forbid}, F_{force})$ to the logical schema graph *G*, we obtain an updated graph G_f where forbidden edges are removed and forced nesting are collapsed into single nodes. Then, we run Chu–Liu/Edmonds' algorithm on the weighted graph *G′*. Given a directed weighted graph and a designated root node, Chu–Liu/Edmonds' algorithm returns a maximum-weight arborescence, a directed tree that spans all nodes reachable from the root and maximizes the sum of edge weights. In our solution, this corresponds to selecting, for each node, at most one incoming edge, chosen as the one that yields the greatest storage saving.

Let $R \subseteq N$ be the set of nodes with no incoming edges in *E*; these serve as root candidates. By restricting Chu–Liu/Edmonds' algorithm to roots in *R,* we reduce the number of times the algorithm must be run. This is because many hierarchies rooted at internal nodes would be strictly contained in the upper hierarchies. For each root $r \in R$ with at least one outgoing edge, we run Chu–Liu/Edmonds' algorithm on graph *G* with *r* as the root, obtain a maximum-weight arborescence A_r, and interpret each edge $(u, v) \in A_r$ as a proposed nesting of table *v* into table *u*. The generated nesting is denoted H_r, and the cost model is used to compute its total storage saving S_r. This produces a pair of candidate nesting and total storage saving $<H_r, S_r>$. In a second step, we refine these candidates by filtering them against physical and DBA limits, resolving conflicts where different hierarchies propose an overlapping nesting. The remaining hierarchies are then merged into a single nesting graph G_N, which encodes all schema-driven nesting decisions and serves either as the physical schema or as the starting point for workload-driven nesting. If no candidate nesting satisfies all constraints, G_N is set equal to *G*, indicating that no nesting is applied.

Algorithm 1: Schema-Driven Nesting

Input:

schema graph *G*

statistics: for each table $n \in N$, the row count *rows(n)*, the average row length *r(n)*, and the total storage size *t(n)*.

DBA constraints: $F=(F_{forbid}, F_{force})$

Maximum row-length threshold ρ

Output:

nested graph G_N.

Procedure:

1. Preprocess DBA constraints

1.1 Let G_f be a working copy of *G*.

1.2 For each edge e = *(u, v)* in E:

1.3 If $e \in F_{forbid}$

1.3.1 remove *e* from G_f

1.4 If $e \in F_{force}$

1.4.1 collapse the subgraph produced by edge in F_{force} into a single node,

1.4.2 redirecting incoming and outgoing edges in the working graph.

1.5 End for

1.6 *G* is preserved to retain referential integrity information.

2. Assign weights to edges

2.1 For each edge $e = (u, v)$ in G_f :

2.1.1 compute *n* using equation (1)

2.1.2 compute *d* using equation (2)

2.1.3 compute *r′* using equation (3)

2.2 Check row-length constraint:

2.2.1 if $r' > \rho$

2.2.1.1 remove e from G_f and proceed to next edge.

2.3 End for

2.4 For all remaining edges

2.4.1 compute S_0 using equation (4)

2.4.2 compute S_1 using equation (5)

2.4.3 compute $p(e)$ using equation (6)

2.4.4 *set* $w(e) = p(e)$

3. Generate candidate hierarchies

3.1 Let $R \subseteq N$ be the set of nodes with no incoming edges.

3.2 Initialise $H := \emptyset$ as the set of candidate hierarchy.

3.3 For each root $r \in R$

3.3.1 Run Chu–Liu/Edmonds' algorithm on G_f with r as the designated root to obtain a maximum-weight H_r.

3.3.2 Use the cost model to compute the total storage saving S_r of H_r as the sum of $p(e)$ over all edges $e \in H_r$.

3.3.3 Add the pair (H_r, S_r) to H.

4. Resolving Conflicts Between Candidate Hierarchies

4.1 If two hierarchies H_i and $H_j \in H$ propose nesting the same table under different host tables.

4.2 While a conflict exists between H_i and $H_j \in H$:

4.2.1 Compare the total storage savings S_i and S_j.

4.2.2 Retain the hierarchy with the greater storage saving

4.2.3 If $S_i = S_j$ go to manual tuning (Step 5).

5. Manual tuning

5.1 For any candidate nesting that violates the row-length constraint ρ or the maximum table size constraint, and for any unresolved conflict from Step 4, the final decision is to the database designer.

5.2 The designer may override the algorithm's selection by explicitly choosing which nesting to retain or discard.

6. Return final nested graph G_N

6.1 The process terminates with a nesting graph that encodes all schema-driven nesting decisions, ready to be used as the physical schema.

5. Workload-driven Nesting

The schema-driven approach introduced in Section 4 proposes nesting decisions on the basis of structural properties of the schema alone, before any workload information is available. This is appropriate when query patterns are unknown or when a physical design must be committed to in advance. When a representative workload is available, however, structural properties of the schema are insufficient to guide nesting decisions optimally. Different queries impose different access patterns on the same tables, and a nesting that eliminates costly joins for one query may introduce additional overhead for another. In such settings, nesting decisions should be informed directly by the expected execution cost of the workload rather than by storage considerations alone.

This section presents the workload-driven nesting approach. Its input is the logical schema graph, together with a set of SQL queries and workload statistics. The optimisation objective is to reduce the total execution cost of the workload, subject to constraints on acceptable storage overhead. Nesting is used as a physical implementation technique to eliminate join operations in the query plans of high-priority queries. All joins are assumed to be on primary-key–foreign-key pairs, so every join condition corresponds directly to an edge in the schema graph. This section focuses on how workload information is used to assign aggregate execution frequencies directly to the edges of the schema graph and apply a greedy sequence of nesting decisions guided by these edge frequencies.

A key property of this phase is that its outcome depends on the order in which nesting transformations on join edges are applied. Because each transformation modifies the schema and potentially alters the access paths available to subsequent nestings, different processing sequences can yield structurally different nesting decisions and different overall performance. The best nesting strategy is the central challenge addressed in this section. We analyse how to assign aggregate execution frequencies to the edges of the schema graph and derive a frequency-weighted graph that induces an effective sequence of nesting decisions without exhaustive enumeration of permutations.

5.1. Input, Representation, and Optimization Goal

The input to the workload-driven phase consists of:

1) Schema graph: The original logical schema graph $G = (N, E, L)$ as defined in section 3.
2) Workload W: A finite set of SQL queries, each expressed over the logical relational schema.
3) Workload statistics: For each query $q_i \in W$, an execution frequency $f(q_i)$ is provided.

For each query $q_i \in W$ we also consider the set of join conditions $J(q_i)$, where each element of $J(q_i)$ is a primary-key–foreign-key equality join condition corresponding to an edge in E. This correspondence is captured by the mapping $\varphi : C \rightarrow E$, where C denotes the set of all join conditions expressible over the schema, and φ maps each join condition to its unique corresponding edge in E. The goal of the workload-driven phase is to determine a set of nesting decisions that minimises the total execution cost of the workload through elimination of join operations. Since all queries are expressed over the logical relational schema, every nesting decision is evaluated with respect to its effect on the current physical schema and on the remaining edges affected by each nesting decision.

5.2. Frequency-weighted Schema Graph and Nesting Algorithm

The algorithm assigns aggregate execution frequencies directly to the edges of the schema graph, derives a frequency-weighted graph by removing edges with zero weight, and selects nesting transformations in a greedy order determined by

these frequencies. For each edge $e \in E$, we assign the total frequency as

$$w(e) = \sum_{\{q:\, e \in \{\varphi(j) \mid j \in J(q)\}\}} f(q) \quad (7)$$

so that $w(e)$ records how frequently the join condition mapped to edge e by φ is used by the queries in W. These edge frequencies will guide the subsequent selection of nesting transformations. Because each nesting transformation modifies the schema and updates the available access paths, applying nestings in different orders can produce structurally different nesting decisions and lead to different total execution costs. A nested table early defines part of the nested structure; subsequent nestings must be consistent with this structure and may invalidate alternative nestings that would have benefited other queries. Once a nesting transformation is applied, every node that becomes nested in the resulting structure is treated as a nested node. For each nested node, all incoming edges to that node are removed from the current schema graph. Therefore, an earlier nesting decision not only changes the physical schema but also eliminates conflicting nesting alternatives for later joins. The ordering of edges by their aggregate frequencies determines which compatible nesting decisions are committed first.

Exhaustive enumeration of all permutations of possible nesting sequences is intractable for realistic workloads. Instead of exploring permutations, the edge with the highest frequency is the next candidate for nesting. Edges in the frequency-weighted schema graph are therefore examined in descending order of their aggregate weight $w(e)$. Intuitively, a join that is exercised by many high-frequency queries should be nested earlier than a join that appears rarely or only in infrequent queries, because it offers greater potential to reduce the overall workload cost. This edge-centred view avoids random choices between queries with similar importance and directly ranks the join conditions that drive the nesting process.

With the frequency-weighted schema graph established, nesting transformations are applied iteratively. The algorithm processes on a nested graph G_N which is a copy of original schema graph G, selecting nesting transformations in descending order of edge weight. The nested graph is represented as $G_N = (N, E_s, E_d)$, which captures any possible nesting structure: solid edges E_s represent referential integrity constraints and are candidates for nesting, and dashed edges E_d represent all committed nestings so far. When a nesting is applied, the original solid edge in E_s is replaced by a dashed edge and it is added to E_d representing the nesting. The node pointed to by this dashed edge is referred to as the nested node. All incoming edges to the nested node are removed from the schema graph G_N. It is because an edge is no longer valid once its target node has been nested into another node, since a node cannot be nested into more than one node. This greedy, frequency-driven procedure terminates when no valid solid edges remain in E_s. Not all queries are equally frequent; some are executed rarely by the workload, and nesting such join would create structural overhead without yielding meaningful query performance benefit. A minimum frequency threshold τ is therefore introduced, and only edges whose weight meets or exceeds τ are considered candidates for nesting.

Algorithm 2: Workload-Driven Nesting

Input:

Logical schema graph $G = (N, E, L)$

A set of workload queries $W = \{q_1, \ldots, q_k\}$

Workload frequency function $f: W \rightarrow N$ where N denotes the set of positive integer numbers and $f(q_i)$ is the execution frequency of q_i

A minimum frequency threshold τ, specifying the minimum total execution frequency

For each query $q_i \in$ W, let $J(q_i)$ denote the set of join conditions used by q_i. Let φ : C → E be the mapping from join conditions to their corresponding edges in E

Output:

Nested logical schema graph G_N.

Procedure:

1. Initialize edge weights

1.1 For each edge e ∈ E, set $w(e) = 0$.

2. Assign frequencies from query statistics

2.1 For each edge $e \in E$:

2.1.1 assign its frequency $w(e)$ using equation (7), so that $w(e)$ is the total execution frequency of all queries in W that use the join condition represented by e.

3. Apply Nesting Transformations

3.1 Initialize the nested schema as $G_N = G$ where $E_s = \{e \in E : w(e) \geq \tau\}$ and $E_d = \emptyset$.

3.2 While E_s is not empty:

3.2.1 Select edge $e \in E_s$ with the largest weight $w(e)$

3.2.2 Apply the nesting to G_N by replacing the solid edge with a dashed edge.

3.2.3 Update E_s : remove edges that are no longer valid.

4. Return final nested graph G_N

4.1 The result is the nested schema graph G_N, which encodes all workload-driven nesting decisions.

5.3. Workload-driven Index Recommendation for Nested Tables

Algorithm 2 (Workload-Driven Nesting) returns a nested schema graph G_N encoding all workload-driven nesting decisions. A consequence of nesting is that the physical size of a root table increases, since each root row embeds the records of all its nested components. A query that accesses only one logical table T that has been nested within a root table R must, after the nesting transformation, scan the entire root structure rather than the smaller flat table T alone. This means more data is read than was necessary before nesting. To mitigate this cost, a post-processing step examines W for queries that access individual nested tables and identifies the predicate attributes of those queries as candidates for index creation. The decision of

which index type to create and how to express the index over the physical representation is discussed in Section 7.6.

Algorithm 3: Index Recommendation for Nested Tables

Input:

Nested graph $G_N = (N, E_s, E_d)$

A set of workload queries $W = \{q_1, \ldots, q_k\}$

Output:

A set I of pairs (T, a), where T is a nested logical table and a is an attribute of T.

Procedure:

1. Identify candidate tables for indexing.

1.1 Let $N_{indexed}$ denote the set of tables to be indexed.

1.2 Initially $N_{indexed} = \emptyset$

1.3 For each edge $e = (u, v) \in E_d$

1.3.1 Add v to $N_{indexed}$

2. Identify qualifying single-table queries.

2.1 Let I = Ø.

2.2 For each query q ∈ W:

2.2.1 If q references one logical table T and $T \in N_{indexed}$

2.2.1.1 For each attribute a of T that appears in q, add the pair (T, a) to I.

3. Return I.

3.1 Each pair (T, a) in I identifies a logical attribute on a nested table for which index is suggested.

5.4. Computational Complexity and Solution Quality

This section analyses the computational complexity of the algorithm and the quality of the solutions it produces. The two are closely related: the algorithm achieves linear complexity by making greedy local decisions, and this efficiency comes at the cost of optimality on certain schema graphs. On acyclic schema graphs, every nesting decision is independent that starting with the highest-weight edge cannot invalidate a higher-benefit alternative elsewhere in the graph. The greedy sequence is therefore optimal on acyclic schema graphs. On schema graphs that contain cycles, however, this independence no longer holds. Resolving a cycle in one direction eliminates the opposing edge and may propagate further invalidations to edges outside the cycle. The greedy rule selects the highest-weight edge at each step has no mechanism to anticipate these downstream consequences. It may therefore commit to an edge that yields a high immediate benefit but excludes a larger aggregate benefit available through the alternative. Finding the globally optimal nesting on a cyclic schema requires considering all combinations of conflicting edges, and the search space grows exponentially with the number of cycles. The algorithm is a heuristic; it is optimal on acyclic schemas and produces good-quality solutions in practice but offers no optimality guarantee when cycles are present. Suboptimality on cyclic schema graphs is the price paid for the near-linear efficiency of the greedy approach. Optimal handling of cyclic subgraphs is left as a direction for future work. The following example illustrates this limitation. Consider the schema graph in Figure 1, where tables W, R, S, and V have aggregate edge frequencies as shown. Two choices of nesting are available for the cycle between R and S, and each invalidates the other. The first nesting, Nest R within S (edge $R \to S$, $w = 8$): invalidates $S \to R$ and $V \to S$. Remaining valid nesting: $W \to R$. Total benefit: 8 + 5 = 13. The second nesting, nest S within R (edge $S \to R$, $w = 10$): invalidates $R \to S$ and $W \to R$. Remaining valid nesting: $V \to S$. Total benefit: 10 + 2 = 12. The greedy rule selects $S \to R$ because it carries the higher individual weight, yet this choice yields the lower total benefit. The difficulty arises because the invalidations triggered by resolving the R–S cycle propagate to edges outside the cycle, and the greedy rule evaluates each edge in isolation without accounting for these consequences. This example is representative of a broader class of cyclic schemas on which the greedy heuristic is suboptimal, and it motivates the complexity result above: linear greedy selection is efficient precisely because it commits to local decisions without the exponential search that global optimality on cyclic inputs would require.

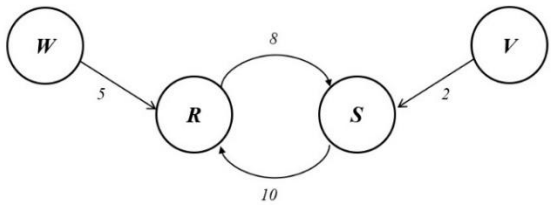

Figure 1. *A sample cyclic schema graph.*

6. Query Rewriting

Nesting is only useful if applications can continue to query a familiar relational schema without being aware of how data is stored. A metadata mapping is the component that makes this possible. It provides an abstraction layer between the logical schema seen by applications and the transformed physical schema produced by the nesting process. Queries are written over the logical schema, and the metadata mapping is used to rewrite them into equivalent queries that operate on the nested tables. In this sense, the metadata mapping is the mechanism that preserves logical transparency while allowing the physical schema to change. At a high level, the metadata mapping plays two roles. First, it acts as a catalogue of metadata definitions that describe how logical tables and attributes are represented in the nested physical schema. Second, it supports an automatic rewriting procedure that translates a query over the logical schema into a plan that follows the recorded physical access paths and introduces any required expansion of nested structures.

6.1. Metadata Mapping

The metadata mapping M is defined as follows. Let A denote the set of all logical attribute references of the form $R.a$,

where R is a logical table and a is an attribute of R. To describe the physical location of each attribute after nesting, a path uses a dot-separated sequence of names of the form

$$\text{path(a)} = \text{r.n}_1\text{.n}_2\text{. … .n}_k\text{.a}$$

where r is a root node name, $n_1, n_2, \ldots, n_k$ are the names of nested components traversed in order from r, and a is the name of the target attribute. For an attribute held directly at the root level, the path is $R.a$. Since the root node is always the first element of the path, it can be recovered from the path directly whenever needed. Then M is defined as a function

$$\text{M} : A \rightarrow P$$

where, for each logical attribute reference $R.a \in A$, the value $\text{M}(R.a) = r.n_1.n_2. \ldots .n_k.a$ is a complete dot-separated path that begins at the physical root table and ends at the attribute a. The root table is always recoverable as the first element of the path. The nested schema and the mapping M are constructed in a single top-down traversal of G'_N. To record multi-level access paths, a path variable $path(n)$ is maintained for each node n during traversal. For each root node r, $path(r)$ is initialised to r. When a dashed edge $e = <u, v>$ is encountered, $path(v)$ is set to $p(u).v'$, extending the accumulated path by the primed name of the child node. This ensures that at any depth in the hierarchy, the full path from the root to the current node is available before its attributes are recorded in M. For each dashed edge $e = <u, v>$, the attributes of v are incorporated into the schema of u as a nested component accessible under the name v'. Foreign key attributes in v that reference the primary key of u are removed, since the PK-FK association is now encoded structurally by the nesting. The mapping entry for each attribute a of v is then set to $\text{M}(v.a) = path(v).a$, recording its complete access path from the root.

6.2. Construction of Nested Physical Schema and Metadata Mapping

To construct the mapping M, the nested physical schema must be derived from the nested schema graph G_N. This procedure takes G_N as input and produces a nested schema definition that reflects the hierarchical structure encoded by the dashed edges. Formally, let $schema(n)$ denote the set of attributes of a node $n \in N$. As a preprocessing step, all solid edges in E_s are removed from G_N, and any nodes that become disconnected upon their removal are also eliminated; the resulting graph is denoted G'_N. All subsequent steps in this procedure operate on G'_N. A node is called a root node r if it has no incoming dashed edges in G'_N. Root nodes represent physical top-level tables that are not nested inside any other table and therefore serve as entry points for all access paths. Consequently, all remaining nodes are reachable from exactly one root node by following dashed edges outward, since the nesting constraint ensures that each node has at most one incoming dashed edge.

For query rewriting, the expansion requirement of an attribute reference is derived directly from $\text{M}(R.a)$. If $\text{M}(R.a) = R.a$, the attribute is available at the top level of root r, and no structural expansion is required. Otherwise, the intermediate components of the path determine the ordered sequence of nested components that must be traversed to reach the attribute, and this sequence is translated into a corresponding series of expansion operations in the FROM clause. Each intermediate component in the path introduces one expansion step: starting from the physical root table r, the first intermediate component is expanded and assigned an alias; each subsequent component is then expanded from within the alias introduced by the previous step, until the component directly containing the target attribute is reached. The final alias is then used to project the required attribute.

To illustrate, consider three tables $R(a, b)$, $S(b, c)$, and $T(c, d)$, where b is the primary key of R and a foreign key in S, and c is the primary key of S and a foreign key in T. Suppose there are dashed edges $e_1 = <R, S>$ and $e_2 = <S, T>$ in G_N. Since R has no incoming dashed edge, R is the root node. Traversing e_1, the attributes of S are incorporated into the schema of R as a nested component S', and the foreign key attribute b is removed from $schema(S)$ since this association is now captured by the nesting. Traversing e_2, the attributes of T are incorporated into the schema of S' as a further nested component T', and the foreign key attribute c is removed from $schema(T)$. The resulting nested schema is therefore $R(a, b, S'(c, T'(d)))$, where R retains its own attributes, S contributes its non-foreign-key attribute c as a nested component within each R row, and T contributes its non-foreign-key attribute d as a nested component within each S' row. At the same step, M is updated. When R is first visited, each attribute of R receives a root-level path entry: $\text{M}(R.a) = R.a$ and $\text{M}(R.b) = R.b$ When edge e_1 is processed, attribute c of S receives the entry $\text{M}(S.c) = R.S'.c$, recording the complete access path from root R through nested component S' to attribute c. When edge e_2 is processed, attribute d of T receives the entry $\text{M}(T.d) = R.S'.T'.d$, recording the complete access path from root R through nested components S' and T' to attribute d.

Algorithm 4: Nested Schema Construction and Mapping

Input:

Nested schema graph $G_N = (N, E_s, E_d)$

Output:

A set R of nested schemas.

Mapping $\text{M} : A \rightarrow P$

Preprocessing: Construct G'_N

P.1 Let $G'_N = G_N$

P.2 Remove all edges in E_s from G'_N.

P.3 Remove all nodes that have become disconnected.

1. Identify root nodes

1.1 Let *Roots* denote the set of all root nodes.

1.2 Initially *Roots* = Ø

1.3 For each node $n \in G'_N$:

1.3.1 if n has no incoming dashed edge, add n to

Roots.

2. Traverse dashed edges and update nested schemas

2.1 For each root node $r \in Roots$: perform traversal of G'_N following dashed edges outgoing from r:

2.1.1 Initialize $path(r) = r$

2.1.2 For each attribute a of r, set $M(r.a) = path(r).a$

2.1.3 For each dashed edge $e = (u, v)$ encountered during traversal do:

2.1.3.1 Set $path(v) = path(u).v'$

2.1.3.2 Remove from *schema(v)* any foreign key attribute that references the primary key in *schema(u)*.

2.1.3.3 Append v' (*schema(v)*) to *schema(u)*

2.1.3.4 For each attribute a of v, set $M(v.a) = path(v).a$

3. Return nested schema definitions

3.1 Return the set R of nested schema for each root node $r \in Roots$.

6.3. Application of Metadata Mapping to Query Rewriting

When a query q is expressed over the logical schema, the system uses M to rewrite it into an equivalent query q' that operates on the nested physical schema. The rewriting eliminates all join conditions whose structural associations are already captured by nesting. It then reconstructs the FROM clause by deriving the expansion operations implied by the path values recorded in the mapping M. Finally, it replaces every logical attribute reference in the SELECT and WHERE clauses with its corresponding physical access expression, using the paths recorded in M. The rewriting procedure proceeds as follows.

In the first step, join conditions in q are examined against G_N. For each join condition $c \in J(q)$, the mapping φ, defined in Section 5.1, is used to identify the corresponding edge φ(c) ∈ E. If φ(c) appears as a dashed edge in E_d of G_N, then the association represented by c has already been captured structurally by nesting, and the join condition is removed. Otherwise, the join condition is retained unchanged. In this way, only join conditions eliminated through nesting are removed, while all other join conditions remain in the rewritten query.

In the second step, all paths required by the query are collected. For each attribute reference $R.a$ appearing in the SELECT and WHERE clauses of q, the entry $M(R.a) = path(a)$ is retrieved from M. All retrieved paths are gathered into a set of paths P, which serves as the sole input to the subsequent FROM clause reconstruction and attribute rewriting steps.

In the third step, the FROM clause of q is reconstructed using the set of paths P. Each distinct physical root table identified as the first element of each path p required by the query is included as a top-level table reference. For a $path = r.n_1.n_2.\ldots.n_k.a$, this step produces a sequence of k expansion operations applied in order over the nested components $n_1, n_2, \ldots, n_k$. Each component is expanded in turn from the result of the previous expansion, beginning with n_1 expanded directly from root r, until the final component n_k is reached. Attribute a is then accessed through ex_k, the result of the final expansion in the sequence. Each distinct root reference and each distinct expansion sequence is inserted only once.

Before describing the fourth step, the expansion operator is defined. For a nested component x, $expand(x)$ denotes the operation that unfolds x so that its rows become individually accessible in the query. For a $path = r.n_1.n_2 \ldots .n_k.a$, $expand(path)$ denotes the corresponding sequence of expansion operations applied in order over the nested components $n_1, n_2, \ldots, n_k$. This notation is system-independent; the concrete database-specific function of $expand(path)$ is discussed in Section 7.

In the fourth step, every attribute reference $R.a$ in the SELECT, WHERE and retained join clauses is rewritten using the paths established in Step 2. If $M(R.a) = R.a$, then the attribute remains at the top level of the physical root table and $R.a$ is rewritten as a direct reference to attribute a under root R. If $path = r.n_1.\ldots.n_k.a$, the attribute is stored inside a nested component, and $R.a$ is rewritten as a reference to attribute a through the result of $expand(path)$ introduced in Step 3. The result is that every logical attribute reference of the form $R.a$ is replaced by a reference to a through ex_k, the result of the final expansion in the sequence.

Thus, the mapping φ identifies which join conditions correspond to logical-schema edges and is therefore used to determine whether a join condition can be removed. The mapping M determines, for each logical attribute reference, a complete dot-separated path p beginning at the physical root node and ending at the target attribute. The root is always recoverable as the first element of p. The expansion requirement is derived from the nested component names between the root and the attribute name in p, and the final attribute access is determined from the last element once the appropriate alias has been introduced in the FROM clause.

Algorithm 5: Query Rewriting Using Metadata Mapping

Input:

Query q expressed over the logical schema

Metadata mapping $M : A \rightarrow P$

Join condition mapping $\varphi : C \rightarrow E$

Nested schema graph $G_N = (N, E_s, E_d)$

Output:

Query q' expressed over the nested physical schema

Procedures:

1. Eliminate join conditions

1.1 For each join condition $c \in J(q)$:

1.1.1 Compute $e = \varphi(c)$.

1.2 If $e \in E_d$

1.2.1 remove c from q.

1.3 Else

1.3.1 retain c unchanged.

2. Collect required paths

2.1 Let $P = \emptyset$.

2.2 For each attribute reference $R.a$ in the SELECT and WHERE clauses of q:

2.2.1 Look up M($R.a$) to obtain *path* to a
2.2.2 Add *path* to P.

3. Reconstruct the FROM clause

3.1 For each distinct root r appearing as the first element of any $p \in P$, include r as a top-level table reference.
3.2 Let ex_i denote the result of expanding the nested component i along the path, used to access attributes stored at that level or to drive the next expansion.
3.3 For each *path* = $r.n_1, \ldots, n_k.a$:
3.3.1 Expand($r.n_i$) and denote the result ex_i.
3.3.2 For each subsequent component n_{i+1}, expand($ex_i.n_{i+1}$) and denote the result ex_{i+1}.
3.3.3 Access attribute a through ex_k.
3.3.4 Insert each root reference and each expansion sequence.

4. Replace attribute references

4.1 For each attribute reference $R.a$ in the SELECT, WHERE and retained join clauses of q:
4.1.1 Look up $M(R.a)$ to obtain *path* to a
4.1.2 If $M(R.a) = R.a$
4.1.2.1 $R.a$ remains unchanged
4.1.3 Else
4.1.3.1 replace $R.a$ with $ex_k.a$
4.4.4 End if

5. Return rewritten query

5.1 Return q' as the executable query over the nested physical schema.

6.4. Example

We use four tables from the TPC-H benchmark database: CUSTOMER, ORDERS, LINEITEM, and SUPPLIER. The schemas of the tables have been simplified, retaining only the attributes needed for joining and querying. The tables are linked by primary key to foreign key relationships: CUSTOMER to ORDERS via C_CUSTKEY, ORDERS to LINEITEM via O_ORDERKEY, and SUPPLIER to LINEITEM via S_SUPPKEY.

```
CUSTOMER(
C_CUSTKEY      INT PRIMARY KEY,
C_NAME         VARCHAR)
ORDERS(
O_ORDERKEY  INT   PRIMARY KEY,
O_CUSTKEY      INT,
O_ORDERDATE DATE)
SUPPLIER(
S_SUPPKEY      INT   PRIMARY KEY,
S_NAME         VARCHAR,
S_NATION       VARCHAR)
LINEITEM(
L_ORDERKEY  INT,
L_SUPPKEY      INT,
L_QUANTITY  DECIMAL,
L_PRICE        DECIMAL,
L_SHIPDATE     DATE)
```

We consider a single query that touches all four tables, retrieving customer names alongside their order dates, lineitem details, and the supplying supplier, filtered to lineitems with a price below 100:

```
SELECT C_NAME, O_ORDERKEY, O_ORDERDATE,
L_QUANTITY, L_PRICE, L_SHIPDATE,
S_NAME
FROM CUSTOMER
JOIN ORDERS    ON C_CUSTKEY = O_CUSTKE
JOIN LINEITEM ON O_ORDERKEY = L_ORDERKEY
JOIN SUPPLIER ON L_SUPPKEY = S_SUPPKEY
WHERE L_PRICE < 100;
```

First, we construct G'_N and identifying root nodes. The analysis produces two dashed edges corresponding to the join conditions that being realised as nesting: e_1 = <CUSTOMER, ORDERS> and e_2 = <ORDERS, LINEITEM>. The join condition between SUPPLIER and LINEITEM yields a solid edge e_3 = <SUPPLIER, LINEITEM>, which remains solid because LINEITEM is already nested within ORDERS and cannot be nested a second time. The solid edges, including e_3, are removed during preprocessing to yield G'_N, which retains only the dashed edges e_1 and e_2. SUPPLIER becomes a disconnected node in G'_N and is therefore also removed, remaining as a flat physical table. CUSTOMER has no incoming dashed edge in G'_N and is identified as the sole root node.

Then we construct the nested schema and mapping M. The traversal of G'_N proceeds top-down from CUSTOMER as follows:

Path (CUSTOMER) = CUSTOMER each attribute of CUSTOMER receives a root-level mapping entry:

M(CUSTOMER.C_CUSTKEY) = CUSTOMER.C_CUSTKEY
M(CUSTOMER.C_NAME) = CUSTOMER.C_NAME
Traversing e_1 = <CUSTOMER, ORDERS>:
Path (ORDERS) = CUSTOMER.ORDERS.

The foreign key O_CUSTKEY is removed from the schema ORDERS and the remaining attributes of ORDERS are incorporated into the schema of CUSTOMER as nested component ORDER, the mapping entries are recorded:

M(ORDERS.O_ORDERKEY) =
CUSTOMER.ORDERS.O_ORDERKEY
M(ORDERS.O_ORDERDATE) =
CUSTOMER.ORDERS.O_ORDERDATE
Traversing e_2 = <ORDERS, LINEITEM>:
Path(LINEITEM) = CUSTOMER.ORDERS.LINEITEM

The foreign key L_ORDERKEY is removed from the schema LINEITEM, and the remaining attributes of the schema are incorporated into the schema ORDERS as nested component LINEITEM, the mapping entries are recorded:

M(LINEITEM.L_QUANTITY) =
CUSTOMER.ORDERS.LINEITEM.L_QUANTITY
M(LINEITEM.L_PRICE) =
CUSTOMER.ORDERS.LINEITEM.L_PRICE
M(LINEITEM.L_SHIPDATE) =
CUSTOMER.ORDERS.LINEITEM.L_SHIPDATE.

After constructing the nested schema and creating the metadata, the resulting schema will be as follows:

CUSTOMER_N(
C_CUSTKEY INT PRIMARY KEY,
C_NAME VARCHAR(50) NOT NULL,
ORDERS JSONB)
SUPPLIER(
S_SUPPKEY INT PRIMARY KEY,
S_NAME VARCHAR,
S_NATION VARCHAR)

The next step is rewriting the original query to be compatible with nested schema. Following the procedure described in Section 6.2, the result is:

Step 1: Eliminate join conditions:

Only the join conditions corresponding to dashed edges realised as nesting edges are removed, so the following joins are removed from SQL:

JOIN ORDERS *ON* C_CUSTKEY = O_CUSTKEY
JOIN LINEITEM *ON* O_ORDERKEY = L_ORDERKEY

The join condition corresponding to the retained solid edge is:

JOIN SUPPLIER *ON* L_SUPPKEY = S_SUPPKEY

Step 2: Collect required paths:

The attribute references in the query are resolved against M, yielding the following Paths set:

C_NAME → CUSTOMER.CNAME
O_ORDERKEY → CUSTOMER.ORDERS. O_ORDERKEY
O_ORDERDATE → CUSTOMER.ORDERS.O_ORDERDATE
L_SUPPKEY → CUSTOMER.ORDERS.
LINEITEM.L_SUPPKEY
L_QUANTITY → CUSTOMER.ORDERS.
LINEITEM.L_QUANTITY
LPRICE → CUSTOMER.ORDERS.LINEITEM. LPRICE
LSHIPDATE → CUSTOMER.ORDERS.
LINEITEM. L_SHIPDATE
S_NAME → SUPPLIER.S_NAME

Step 3: Reconstruct the FROM clause:

CUSTOMER_N is included as the top-level table reference. Two expansion operations are required: ORDERS is expanded from CUSTOMER_N via expand(CUSTOMER.ORDERS′), aliased as order_item. LINEITEM is then expanded from that result via expand(order_item.LINEITEM), aliased as line_item. Table SUPPLIER is included as a regular table reference, joined on the retained condition line_item.L_SUPPKEY = S_SUPPKEY.

Step 4: Replace attribute references:

Each attribute reference is rewritten through the alias introduced for its final nested component. C_NAME is accessed directly from CUSTOMER_N. The attributes O_ORDERKEY and O_ORDERDATE are accessed through order_item alias. The attributes L_QUANTITY, L_PRICE, and L_SHIPDATE are stored inside LINEITEM and are accessed through line_item alias. S_NAME is accessed directly from SUPPLIER. The WHERE clause L_PRICE < 100 is rewritten as line_item.L_PRICE < 100.

The rewritten query expressed using the system independent EXPAND notation is:

SELECT C_NAME,
order_item.O_ORDERKEY,
order_item.O_ORDERDATE,
line_item.L_QUANTITY,
line_item.L_PRICE,
line_item.L_SHIPDATE,
S_NAME
FROM CUSTOMER_N,
EXPAND(CUSTOMER.ORDERS) AS order_item,
EXPAND(order_item.LINEITEM) AS line_item
JOIN SUPPLIER *ON*
line_item.L_SUPPKEY = SUPPLIER.S_SUPPKEY
WHERE line_item.L_PRICE < 100;

7. Implementation in a Database Management System (DBMS)

The nested schema graph G_N produced by the algorithms in Section 5, and the mapping metadata M defined in Section 6, must be implemented as concrete physical structures before any query can be executed. This section describes how that implementation is performed, using PostgreSQL as a sample DBMS. Each nesting in G_N is encoded using PostgreSQL's native JSONB type, which supports structured data stored within a single column, together with efficient traversal via its built-in unnesting functions. Tables that are not involved in any nesting are left as ordinary flat relations. The logical relational view seen by applications is preserved throughout: queries are written over the original schema and are rewritten automatically using M before execution, so that no structural change is visible to the application. In the current implementation, this rewriting is implemented through a query rewriting layer that translates incoming logical queries into their physical equivalents using M before passing them to the PostgreSQL execution engine.

7.1. Representing Nested Structures Using JSONB

The nested schema produced by the algorithm 4 (Nested Physical Schema Construction and Mapping M) is encoded in PostgreSQL by replacing each nested component with a JSONB column in the root table. PostgreSQL provides two types for storing JSON data: JSON and JSONB. The JSON type stores the raw input text literal, preserving insertion order and whitespace, but requires the value to be re-parsed on every access. The JSONB type instead stores data in a decomposed binary format, which eliminates the need for repeated parsing and enables operator-based traversal and indexing. Because the query rewriting procedure in Sections 7.3 and 7.4 accesses

nested attributes through field-access operators applied at query execution time, and because Section 7.6 introduces GIN-based indexing over nested paths, JSONB is used throughout this implementation in preference to JSON. For each table v nested within a table u, the attributes of v that remain after foreign key removal are no longer stored in a separate flat table. Instead, they are stored as an array of JSONB objects embedded within the corresponding column of u. The first level of nesting is represented as a JSONB column in the root table. Each further level of nesting is represented by embedding JSON objects within the JSONB structure, so that all nesting levels are contained within the single root-level JSONB column. Tables that are not involved in any nesting are left as ordinary flat relations and require no structural modification. For the TPC-H example considered in the previous section, the nesting <CUSTOMER, ORDERS> and <ORDERS, LINEITEM> produce a two-level nesting. The physical table CUSTOMER_N is defined as follows:

```
CUSTOMER_N(
C_CUSTKEY INT      PRIMARY KEY,
C_NAME    VARCHAR(50) NOT NULL,
ORDERS    JSONB);
```

Each row in CUSTOMER_N table stores a customer's root-level attributes alongside an ORDERS JSONB column containing an array of objects representing customer orders. Each order object in turn embeds a LINEITEMS array containing objects representing the corresponding lineitems. The foreign keys O_CUSTKEY and L_ORDERKEY, which were used to express the joins in the logical schema, are removed from the nested structure since the association they encoded is now captured by the containment. This encoding preserves the information content of the original relational schema while eliminating the need for processing join operations across the nested associations at query time.

7.2. Storing and Maintaining Mapping Metadata M in PostgreSQL

The mapping M defined in Section 6.1 is a function from logical attribute references to dot-separated physical access paths. For M to support automatic query rewriting at runtime, it needs to be stored as a relational table within PostgreSQL rather than maintained only as an abstract definition. This is achieved by materialising M as a dedicated metadata catalogue table, referred to here as M_CATALOGUE which is defined as follows:

```
M_CATALOGUE(
logical_table VARCHAR,
logical_attr  VARCHAR,
physical_root      VARCHAR,
full_path          VARCHAR)
```

Each row in M_CATALOGUE records one entry of M. The logical_table and logical_attr columns together identify the logical attribute reference $R.a$. The physical_root column records the first element of the path, identifying the physical root table under which $R.a$ is stored. The full_path column stores the complete dot-separated path $r.n_1.n_2.\ldots.n_k.a$ produced during the top-down traversal of G'_N described in Section 6.1. For root-level attributes, full_path takes the form $R.a$, indicating that no expansion is required. The query rewriting procedure in Sections 7.3 and 7.4 reads from M_CATALOGUE to determine, for each attribute reference in an incoming query, which expansion chain to construct and which alias to use for the final attribute access.

M_CATALOGUE is populated once when the nested physical schema is constructed and must be updated whenever the nesting configuration changes, for example, when a new dashed edge is introduced or an existing one is removed. For the TPC-H example used in the worked example in section 6.3, the entries recorded during traversal of G'_N are as follows:

```
(CUSTOMER,C_CUSTKEY,CUSTOMER_N,
CUSTOMER.C_CUSTKEY)
(CUSTOMER,C_NAME,CUSTOMER_N,CUSTOMER.
C_NAME)
(ORDERS,O_ORDERKEY,CUSTOMER_N,
CUSTOMER.ORDERS.O_ORDERKEY)
(ORDERS,O_ORDERDATE,CUSTOMER_N,
CUSTOMER.ORDERS.O_ORDERDATE)
(LINEITEM,L_SUPPKEY,CUSTOMER_N,
CUSTOMER.ORDERS.LINEITEM.L_SUPPKEY)
(LINEITEM,L_QUANTITY,CUSTOMER_N,
CUSTOMER.ORDERS.LINEITEM.L_QUANTITY)
(LINEITEM,L_PRICE,CUSTOMER_N,CUSTOMER.
ORDERS.LINEITEM.L_PRICE)
(LINEITEM,L_SHIPDATE,CUSTOMER_N,
CUSTOMER.ORDERS.LINEITEM.L_SHIPDATE)
(SUPPLIER,S_SUPPKEY,SUPPLIER,SUPPLIER.
S_SUPPKEY)
(SUPPLIER,S_NAME,SUPPLIER,SUPPLIER.S_NAME)
```

These entries correspond exactly to the M entries constructed in the worked example of Section 6.3, confirming that M_CATALOGUE is a direct physical implementation of the abstract mapping M.

7.3. Implementing the Expansion Operator in PostgreSQL

In PostgreSQL, each level of nesting is unfolded using the built-in JSONB_ARRAY_ELEMENTS function, which takes a JSONB array and returns its elements as individual rows. For a nested structure with k levels, this function is applied once per level, producing a chain of expansions in the FROM clause. The first expansion is applied directly to the JSONB column of the root table r, unfolding the array stored under the first nested component n_i: JSONB_ARRAY_ELEMENTS (r -> n_i) AS ex_i, where -> is the PostgreSQL JSONB operator that retrieves the value of a named field as a JSONB object, and ex_i is the alias assigned to each element produced at this level. Each subsequent level is then unfolded from the result of the previous expansion. Given the alias ex_i at level i, the

next level is expanded as: JSONB_ARRAY_ELEMENTS (ex_i -> n_{i+1}) AS ex_{i+1}, where n_{i+1} is the name of the nested component at the next level and ex_{i+1} is the alias assigned to each element produced at that level. This process continues until all nesting levels have been unfolded, with each alias making the objects at its level individually accessible for attribute reference in the SELECT and WHERE clauses.

Applying this to the FROM clause of the query, the three original JOIN clauses are handled as follows. The condition JOIN ORDERS ON C_CUSTKEY = O_CUSTKEY is eliminated, since the association between CUSTOMER and ORDERS is now captured by the first JSONB_ARRAY_ELEMENTS expansion over the ORDERS column of CUSTOMER_N, which produces the alias order_item. The condition JOIN LINEITEM ON O_ORDERKEY = L_ORDERKEY is likewise eliminated, since the association between ORDERS and LINEITEM is captured by the second expansion over the LINEITEMS array embedded within each order object, which produces the alias line_item. The condition JOIN SUPPLIER ON L_SUPPKEY = S_SUPPKEY is retained, since SUPPLIER is not part of any nesting and remains a flat relational table. The rewriting of its attribute reference and the transformation of the SELECT and WHERE clauses are described in Section 7.4. For the TPC-H example, the ORDERS nesting is unfolded first, producing order_item:

JSONB_ARRAY_ELEMENTS(CUSTOMER_N -> ORDERS) AS order_item

The LINEITEM nesting is then unfolded from order_item, producing line_item:

JSONB_ARRAY_ELEMENTS(order_item -> LINEITEMS) AS line_item

7.4. Rewriting Attribute References Using JSONB Access Operators

Once the FROM clause has been reconstructed with the appropriate expansion chain, each logical attribute reference $R.a$ in the SELECT and WHERE clauses is rewritten using the JSONB access operators supported by PostgreSQL. This step is the concrete realisation of Step 4 of the rewriting algorithm.

PostgreSQL provides two operators for accessing fields within a JSONB object. The -> operator, introduced in Section 7.3, returns the value of a field as a JSONB object, and the ->> operator returns it as text. For attribute references that appear in the SELECT clause and require a typed value, ->> is typically used, with an explicit cast applied where the original attribute type must be preserved. For example, an integer attribute O_ORDERKEY stored in a JSONB object is retrieved and cast as (order_item->>'O_ORDERKEY')::INT. For attributes used in WHERE clause comparisons, the appropriate operator is selected based on the comparison context.

For an attribute reference $R.a$ whose $path = r.n_1.\ldots.n_k.a$, the rewritten reference takes the form: ex_k ->> $'a'$ where ex_k is the alias introduced for the final expansion. For root-level attributes whose path is simply $path = R$, the attribute is accessed directly from the root table without any JSONB operator.

For the TPC-H example, the attribute references are rewritten as follows:

CNAME → STOMERN.CNAME
O_ORDERKEY → order_item->>'O_ORDERKEY'
O_ORDERDATE→ order_item->>'O_ORDERDATE'
L_SUPPKEY → line_item->>'L_SUPPKEY'
L_QUANTITY → line_item->>'L_QUANTITY'
L_PRICE → line_item->>'L_PRICE'
L_SHIPDATE → line_item->>'L_SHIPDATE'
S_NAME → SUPPLIER.S_NAME

Root-level attributes such as C_NAME are accessed directly on CUSTOMER_N without any operator, since they are stored as ordinary relational columns. Attributes at deeper levels are accessed through the alias of their corresponding expansion result using the ->> operator.

More generally, any join condition in the original query that references a nested attribute is rewritten in the same way: the logical attribute reference is replaced by the alias of its corresponding expansion result together with the appropriate JSONB access operator, exactly as for SELECT clause references. Join conditions that reference only root-level attributes of the physical root table require no rewriting.

Applying this to the query introduced in Section 7.2, the SELECT clause rewrites for each logical attribute reference follow directly from the table above, with each reference replaced by the alias and operator corresponding to its full_path entry. Three structural transformations not captured by the table are as follows. First, the retained join condition JOIN SUPPLIER ON L_SUPPKEY = S_SUPPKEY is rewritten as JOIN SUPPLIER S ON (line_item->>'L_SUPPKEY') = S.S_SUPPKEY, replacing the logical reference L_SUPPKEY with the line_item alias in accordance with its path entry in the table. Second, the two nested join conditions eliminated in Section 7.3 do not appear in the rewritten query, since the associations they expressed are captured by the JSONB_ARRAY_ELEMENTS expansions in the FROM clause. Third, the WHERE clause L_PRICE < 100 is rewritten as (line_item->>'L_PRICE')::NUMERIC < 100, applying the same alias-based mapping as the SELECT clause with an explicit cast to numeric type. The full query resulting from combining the FROM clause reconstruction of Section 7.3 with these transformations is given in Section 7.5.

The same rewriting applies to attribute references appearing in WHERE clause conditions and join conditions. Here, the filter on the logical attribute L_PRICE is rewritten as: (line_item->>'L_PRICE')::NUMERIC < 100

7.5. The Complete Rewritten Query

Combining the FROM clause reconstruction from Section 7.3 and the attribute reference rewriting from Section 7.4, the full PostgreSQL query for the TPC-H example is:

SELECT C_NAME,
order_item->>'O_ORDERKEY' AS O_ORDERKEY,

```
order_item->>'O_ORDERDATE'AS O_ORDERDATE,
line_item->>'L_QUANTITY' AS L_QUANTITY,
line_item->>'L_PRICE'   AS L_PRICE,
line_item->>'L_SHIPDATE' AS L_SHIPDATE,
S.S_NAME
FROM CUSTOMER_N,
JSONB_ARRAY_ELEMENTS(
CUSTOMER_N.ORDERS) AS order_item,
JSONB_ARRAY_ELEMENTS(
order_item->'LINEITEMS') AS line_item
JOIN SUPPLIER S
ON (line_item->>'L_SUPPKEY')= S.S_SUPPKEY
WHERE (line_item->>'L_PRICE')::NUMERIC < 100;
```

This query is semantically equivalent to the original three-table join over the logical schema. The two explicit join conditions have been eliminated, since the associations they expressed are now captured by the JSONB containment structure. The join condition between LINEITEM to SUPPLIER is retained and rewritten using the line_item alias, since SUPPLIER remains a flat relational table. The FROM clause replaces the join with a chain of unnesting operations, and each attribute reference in the SELECT clause reaches its value through the appropriate expansion result. The WHERE clause condition on L_PRICE is rewritten using the alias line_item and the ->> operator, applying the same mapping-based rewriting as the SELECT clause references. The query operates on two tables CUSTOMER_N and SUPPLIER produces the same result set as the original query over the flat relational schema.

7.6. Indexing JSONB Columns

The algorithm 2 (Workload-Driven Nesting) in Section 5.2 embeds nested tables within a root table, increasing the storage consumption of the root table. As identified in Section 5.3, queries that access a single nested table T must after rewriting scan the full root table rather than T alone, incurring a higher data access cost than before nesting. For each pair (T, a) ∈ I produced by Algorithm 3 (Index Recommendation for Nested Tables), an index is created over the JSONB field access expression corresponding to attribute a, derived from the full path recorded in M. The appropriate index type depends on whether the target attribute *a* is stored as a scalar field within a JSONB object column, or as a field inside a JSONB array. Each case requires a different indexing strategy, as described below. A B-tree expression index is applicable when the target attribute *a* is stored as a scalar field directly within a JSONB object column, rather than inside a JSONB array. In this case, the expression resolves exactly one scalar value per row, satisfying the fundamental requirement of B-tree indexing. In PostgreSQL, such an index takes the form:

```
CREATE INDEX index_name
ON physical_root (((jsonb_column >>'attribute_name')::attribute_type));
```

where physical_root is the root table, jsonb_column is the relevant JSONB column, attribute_name is the attribute being indexed, and attribute_type is its declared SQL type. When the JSONB column stores an array rather than a single object, the expression jsonb_column->>'attribute_name' returns NULL for every row, since key access cannot resolve to a scalar value across multiple array elements. B-tree indexing requires that the indexed expression evaluate to exactly one scalar value per row; because a JSONB array column does not satisfy this requirement, the standard field-access expression yields NULL rather than a usable scalar, making B-tree indexing inapplicable in this configuration. In this case a B-tree expression index cannot be applied, and a Generalized Inverted Index (GIN) index must be used instead. In the schema used throughout this paper, the ORDERS column of CUSTOMER_N stores a JSONB array, so B-tree expression indexing is not applicable to attributes nested within ORDERS. B-tree expression indexing is therefore applicable only in configurations where a JSONB column stores a single scalar object per row rather than an array. GIN is applied when the target attribute *a* resides inside a JSONB array. Unlike B-tree, a GIN index does not require a single scalar value per row; instead, it builds an inverted index over all keys and values within the JSONB structure, including those inside arrays. In PostgreSQL, a GIN index using the jsonb_path_ops operator class is created as follows:

```
CREATE INDEX index_name
ON physical_root
USING GIN (jsonb_column jsonb_path_ops);
```

The jsonb_path_ops operator class is preferred over the default jsonb_ops class because it produces a smaller and faster index for path-based queries. A GIN index of this form supports queries using the containment operator @> and the jsonpath match operator @?, enabling the query planner to substitute a Bitmap Index Scan for a sequential scan when evaluating equality predicates on attributes of array-nested components. To demonstrate GIN indexing on array-nested attributes, an experiment is conducted on the nested schema produced earlier. In the experiment CUSTOMER_N is the root table, ORDERS is nested as a JSONB array under each customer row, and LINEITEM is nested within each order object, using a synthetic TPC-H dataset. The experiment considers a query that retrieves order attributes from the ORDERS array nested within CUSTOMER_N, filtered by a specific order key:

```
SELECT
order_item->>'O_ORDERKEY' AS O_ORDERKEY,
order_item->>'O_ORDERDATE' AS O_ORDERDATE
FROM customer_n,
JSONB_ARRAY_ELEMENTS(customer_n.orders)     AS order_item
WHERE orders @> '[{"O_ORDERKEY": 100}]'
AND (order_item->>'O_ORDERKEY')::INT = 100;
```

The WHERE clause contains two predicates serving distinct roles. The first predicate, expressed using the containment operator @>, is evaluated against the GIN index at the root row level and identifies which rows of CUSTOMER_N

contain at least one order with O_ORDERKEY equal to 100. The second predicate, applied to the unnested alias order_item, filters the individual matching element from within the ORDERS array after unnesting. Both predicates are necessary: the first drives index-based row selection, and the second isolates the specific matching element that the GIN index alone cannot identify. This query accesses O_ORDERKEY and O_ORDERDATE, which are stored inside the ORDERS JSONB array of CUSTOMER_N. Without an index, the query requires a full sequential scan over all rows of CUSTOMER_N, as confirmed by the baseline query plan reported in Appendix. A GIN index is then created over the ORDERS column:

CREATE INDEX idx_gin_orders
ON CUSTOMER_N
USING GIN (ORDERS jsonb_path_ops);

After index creation, the planner switches to a Bitmap Index Scan, as shown in the indexed query plan in Appendix. The full query scripts are provided in Appendix. The results confirm that a GIN index using jsonb_path_ops applies directly to the ORDERS JSONB array column of CUSTOMER_N produced by nesting, reducing execution time to 0.095 ms compared to 1.723 ms which uses sequential scan. This indicates that the physical representation produced by the nesting framework can be further enhanced through GIN indexing, recovering the query performance penalty identified in Section 5.3 without any modification to the logical schema or the query rewriting procedure.

8. Conclusion and Summary

The structural design of relational tables at the physical level has an impact on the performance of relational database systems. Established techniques such as indexing, clustering, and partitioning modify how tuples are stored and accessed without altering the logical schema. This paper has introduced the nesting of persistent storage structures as a further physical optimisation technique in the same tradition. Related tables are implemented as a nested structure, in which the rows of a dependent table are embedded within the rows of their parent table. This approach addresses two sources of inefficiency. The first is the redundant repetition of foreign key values on the many-side of one-to-many relationships. The second is the recurring cost of join operations across frequently queried tables.

To determine which relationships should be nested, the paper developed two selection strategies. The schema driven approach operates on the logical schema graph before any workload is known. It assigns weights to candidate edges and selects nesting decisions as a constrained maximum-weight branching problem. This problem is solved with the Chu–Liu/Edmonds algorithm, subject to row-length and storage constraints and to DBA demands. The workload driven approach operates on a frequency weighted schema graph derived from a representative workload. It greedily applies nesting transformations in descending order of aggregate join frequency, eliminating the joins that contribute most to overall workload cost. This procedure is optimal on acyclic schema graphs and serves as an efficient heuristic on cyclic schemas, where global optimality would require an exponential search.

A central contribution is the preservation of logical transparency through the mapping metadata M. This metadata records the complete access path of each logical attribute within the nested physical schema. A system independent query rewriting procedure uses metadata to eliminate nested join conditions, reconstruct the FROM clause through expansion operations, and rewrite attribute references. Queries thus continue to be expressed over the original flat relational schema. The nested structure remains hidden from applications, and nesting stays compatible with other physical techniques.

Feasibility was demonstrated through a prototype implementation in PostgreSQL. Nested structures are encoded using the native JSONB type, and metadata is materialised as a metadata catalogue table that drives automatic query rewriting. A worked example over four TPC-H tables illustrated the complete path from a logical query to its rewritten physical equivalent. The paper further showed that the query performance overhead introduced by nesting can be recovered through standard indexing. A GIN index using the *jsonb_path_ops* operator class reduced the execution time of a representative containment query from 1.723 ms to 0.095 ms. This improvement required no modification to the logical schema or the rewriting procedure.

These results confirm that schema nesting can be treated as a transparent physical optimisation technique. It reduces storage redundancy and eliminates join operations while remaining compatible with existing relational interfaces and physical structures. The storage reductions are particularly relevant for in-memory database systems, where capacity is constrained and every reduction in the working set directly improves throughput. The optimal handling of cyclic schema graphs remains an open problem and is left as a direction for future work.

Abbreviations

GIN	Generalized Inverted Index
JSON	JavaScript Object Notation
JSON	JSON Binary (PostgreSQL storage type)
NF^2	Non-First Normal Form (Nested Relational Model
SQL	Structured Query Language
TPC-H	Transaction Processing Performance Council Benchmark H

Author Contributions

Hammad Alenezi: Conceptualization, Data curation, Formal Analysis, Investigation, Methodology, Resources, Software, Validation, Writing - original draft, Writing - review &

editing

Janusz Getta: Conceptualization, Methodology, Supervision

Tianbing Xia: Conceptualization, Methodology, Supervision

Data Availability Statement

The data that supports the findings of this study can be found at:
https://uowmailedu-my.sharepoint.com/:f:/g/personal/hgha144_uowmail_edu_au/IgBL8JMlDxL-T5JFcvlD-GttzAbJi-RMUFRNZOCGpk1__3eU)

Conflicts of Interest

The authors declare no conflicts of interest.

Appendix: JSONB Indexing Experiment: SQL Scripts & Planner Output

This appendix provides the full SQL scripts and PostgreSQL planner output for the indexing proof of concept described in Section 7.6. All results were obtained on a 1,000-row synthetic TPC-H dataset

Step 1. Baseline query on O ORDERKEY without an index. The following query is used throughout the experiment:

SELECT order_item->>'O_ORDERKEY' AS O_ORDERKEY, order_item->>'O_ORDERDATE' AS O_ORDERDATE

FROM customer_n, JSONB_ARRAY_ELEMENTS(customer_n.orders) AS order_item *WHERE* orders @> ' [{"O_ORDERKEY": 100}] ';

The planner produces a sequential scan over all rows of CUSTOMER_N:

QUERY PLAN

Nested Loop (cost=10000000000.00..10000000287.50 rows=1000 width=64)(actual time=0.203..1.689 rows=5 loops=1)

-> Seq Scan on customer_n (cost=10000000000.00..10000000262.50 rows=10 width=1766)(actual time=0.185..1.667 rows=1 loops=1) Filter: (orders @> ' [{"O_ORDERKEY": 100}] '::jsonb) Rows Removed by Filter: 999

-> Function Scan on jsonb_array_elements order_item (cost=0.00..1.00 rows=100 width=32) (actual time=0.013..0.014 rows=5 loops=1)

Planning Time: 0.117 ms

Execution Time: 1.723 ms

Step 2. GIN index created over the ORDERS column using the jsonb ops operator class:

CREATE INDEX idx_gin_orders ON CUSTOMER_N USING GIN (ORDERS jsonb_path_ops);

Step 3. Same containment query after GIN index creation. The planner switches to a Bitmap Index Scan:

QUERY PLAN

Nested Loop (cost=12.88..67.15 rows=10 width=64) (actual time=0.041..0.043 rows=1 loops=1)

-> Bitmap Heap Scan on customer_n (cost=12.88..47.00 rows=10 width=1766) (actual time=0.025..0.026 rows=1 loops=1) Recheck Cond: (orders @> ' [{"O_ORDERKEY": 100}] '::jsonb) Heap Blocks: exact=1

-> Bitmap Index Scan on idx_gin_orders (cost=0.00..12.87 rows=10 width=0) (actual time=0.014..0.014 rows=1 loops=1)

Index Cond: (orders @> ' [{"O_ORDERKEY": 100}] '::jsonb)

-> Function Scan on jsonb_array_elements order_item (cost=0.00..2.00 rows=1 width=32) (actual time=0.013..0.013 rows=1 loops=1)

Filter: (((value ->> 'O_ORDERKEY'::text))::integer = 100)

Rows Removed by Filter: 4

Planning Time: 0.172 ms

Execution Time: 0.095 ms

(11 rows)

References

[1] Faerber, F., Kemper, A., Larson, P.-Å., Levandoski, J., Neumann, T., & Pavlo, A. (2017). Main memory database systems. *Foundations and Trends® in Databases*, *8*(1–2), 1–130. https://doi.org/10.1561/1900000058

[2] Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, *13*(6), 377–387. https://doi.org/10.1145/362384.362685

[3] Zhou, X., Chai, C., Li, G., & Sun, J. (2022). Database meets artificial intelligence: A survey. *IEEE Transactions on Knowledge and Data Engineering*, *34*(3), 1096–1116. https://doi.org/10.1109/TKDE.2020.2994641

[4] Schek, H.-J., & Scholl, M. H. (1986). The relational model with relation-valued attributes. *Information Systems*, *11*(2), 137–147. https://doi.org/10.1016/0306-4379(86)90003-7

[5] Kossmann, J., Halfpap, S., Jankrift, M., & Schlosser, R. (2020). Magic mirror in my hand, which is the best in the land? An experimental evaluation of index selection algorithms. *Proceedings of the VLDB Endowment*, *13*(11), 2382–2395. https://doi.org/10.14778/3407790.3407832

[6] Perera, R. M., Oetomo, B., Rubinstein, B. I. P., & Borovica-Gajic, R. (2022). HMAB: Self-driving hierarchy of bandits for integrated physical database design tuning. *Proceedings of the VLDB Endowment*, *16*(2), 216–229. https://doi.org/10.14778/3565816.3565824

[7] Hilprecht, B., Binnig, C., & Röhm, U. (2020). Learning a partitioning advisor for cloud databases. In *Proceedings of the 2020 ACM International Conference on Management of Data* (pp. 143–157). Association for Computing Machinery. https://doi.org/10.1145/3318464.3389704

[8] Leis, V., Radke, B., Gubichev, A., Mirchev, A., Boncz, P., Kemper, A., & Neumann, T. (2018). Query optimization through the looking glass, and what we found running the Join Order Benchmark. *The VLDB Journal*, *27*(5), 643–668. https://doi.org/10.1007/s00778-017-0480-7

[9] Navathe, S. B., Ceri, S., Wiederhold, G., & Dou, J. (1984). Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems*, *9*(4), 680–710. https://doi.org/10.1145/1994.2209

[10] Abadi, D. J., Boncz, P. A., Harizopoulos, S., Idreos, S., & Madden, S. R. (2013). The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases*, *5*(3), 197–280. https://doi.org/10.1561/1900000024

[11] Alkowaileet, W. Y., & Carey, M. J. (2022). Columnar formats for schemaless LSM-based document stores. *Proceedings of the VLDB Endowment*, *15*(10), 2085–2097. https://doi.org/10.14778/3547305.3547314

[12] Li, T., Butrovich, M., Ngom, A., Lim, W. S., McKinney, W., & Pavlo, A. (2021). Mainlining databases: Supporting fast transactional workloads on universal columnar data file formats. *Proceedings of the VLDB Endowment*, *14*(4), 534–546. https://doi.org/10.48550/arXiv.2004.14471

[13] Bourhis, P., Reutter, J. L., & Vrgoč, D. (2020). JSON: Data model and query languages. *Information Systems*, *89*, 101478. https://doi.org/10.1016/j.is.2019.101478

[14] PostgreSQL Global Development Group. (2024). *PostgreSQL documentation: JSON types*. Retrieved May 12, 2026, from https://www.postgresql.org/docs/current/datatype-json.html

[15] Durner, D., Leis, V., & Neumann, T. (2021). JSON tiles: Fast analytics on semi-structured data. In *Proceedings of the 2021 ACM International Conference on Management of Data* (pp. 445–458). Association for Computing Machinery. https://doi.org/10.1145/3448016.3452809

[16] Halevy, A. Y. (2001). Answering queries using views: A survey. *The VLDB Journal*, *10*(4), 270–294. https://doi.org/10.1007/s007780100054

[17] Fagin, R., Kolaitis, P. G., Miller, R. J., & Popa, L. (2005). Data exchange: Semantics and query answering. *Theoretical Computer Science*, *336*(1), 89–124. https://doi.org/10.1016/j.tcs.2004.10.033

[18] Kossmann, J., Papenbrock, T., & Naumann, F. (2022). Data dependencies for query optimization: A survey. *The VLDB Journal*, *31*(1), 1–22. https://doi.org/10.1007/s00778-021-00676-3