Storing and Indexing Multiple Tables by Interesting Orderings: For Efficient Joins, Groupings, and Updates in Relational Databases

Wenhui Lyu
University of Wisconsin–Madison
wenhui@cs.wisc.edu

Goetz Graefe
Google Inc.
goetzg@google.com

ABSTRACT

Relational database systems often face a trade-off between supporting multi-table queries and frequent updates. Materialized join views can drastically speed up queries, but they slow down updates and may consume significant storage. Conversely, query-time joins over tables and their indexes optimize update performance at the cost of query latency. Our recent study of two-table joins introduced "merged indexes" (a form of multi-table index) to break this trade-off, approaching the query performance of materialized views without sacrificing the update efficiency of traditional single-table indexes. This study generalizes this technique to "order-sharing pipelines"—multi-table joins and grouping operations on shared keys. By incorporating interesting orderings into the physical database design, merged indexes partially pre-compute these order-sharing pipelines, closely approaching the query performance of materialized views. Meanwhile, they match traditional indexes on update performance.

PVLDB Reference Format:
Wenhui Lyu and Goetz Graefe. Storing and Indexing Multiple Tables by Interesting Orderings: For Efficient Joins, Groupings, and Updates in Relational Databases. PVLDB, 19(11): 3621 - 3634, 2026.
doi:10.14778/3836663.3836713

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/alicia-lyu/vldb19-artifact.

1 INTRODUCTION

Transactional updates and joins are the two critical operations behind the popularity of relational DBMS, and these features continue to differentiate them from Key-Value stores and NoSQL systems. Relational database systems face a trade-off between efficient frequent updates and fast complex queries requiring joins and groupings. Currently, databases typically choose between two extreme approaches to this trade-off. On the one hand, materialized join views can drastically speed up query processing, but they may impose penalties on update performance and space efficiency. On the other hand, query-time computation with tables and their indexes (e.g., via hash joins) minimizes update overhead but provides little preprocessing to accelerate queries. Database systems typically lack a solution that offers great query performance, update performance, and space efficiency simultaneously. The middle ground between these extremes—partial pre-computation—has seen recent interest [46, 58], and this study contributes to this line of work.

Scope and Definition. This paper focuses on pipelines of multi-table joins and grouping operations on shared keys. A pipeline can be an entire query or part of a more complex query. More formally, this study targets pipelines of the form:

$$Q = \gamma_{A,F}(\sigma_P(R_1 \bowtie_A R_2 \bowtie_A \ldots \bowtie_A R_n))$$

Notation: $\gamma_{A,F}$ represents the grouping on attributes A with aggregation set functions F; the join keys and the grouping key are equal column sets, A. More precisely, all join/grouping keys must form a prefix chain compatible with one shared sort order (details in Section 3.1). TPC-H Q3 is an example—keys differ, yet form a common sort order.

This structural pattern often arises because relational databases use normal forms to represent complex application objects across multiple tables. For example, TPC-H represents customers with their orders across three tables: Customer, Orders, and Lineitem. This complex application object may be reconstructed at query time by primary key–foreign key joins.[1] Consequently, these query components appear frequently in relational database workloads. They appear in 14 of 22 queries in the TPC-H benchmark [60] and in all 33 queries in the Join Order Benchmark (JOB) [32].

Rather than proposing an entirely new system, this technique improves classic relational database systems, which remain the backbone of most production workloads. Therefore, existing and robustly-tested techniques, such as hash joins, single-table indexes, and traditional materialized views, are available and should be chosen for queries that do not fit the scope of this work.

This scope represents a generalization of our previous work on two-table joins [37]. It builds on the key insight from Selinger et al. [55] that identifies the cost-saving opportunities during query planning where a sort order produced by an upstream operator can be reused by a downstream operator. The new observation here is that this shared sort order can be embedded in *one* merged index. A single index scan then replaces what would otherwise be a series of joins and grouping operators.[2]

Our recent work [37] shows that two-table merged indexes approach the query performance of materialized views and match the update and space efficiency of traditional single-table indexes,

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836713

[1] Although the two joins do not share the join keys, functional dependence can make their required join orders compatible, as detailed in Section 3.1.
[2] In comparison, according to our previous study, each join would require its own two-table merged index, and the grouping operations were not addressed at all. Moreover, a single multi-table merged index has a combinatorial advantage: it supports order-sharing pipelines over any subset of the indexed tables, thus subsuming multiple two-table merged indexes.

as summarized in Table 1. The findings of this study show that multi-table merged indexes also offer great query performance, update performance, and space efficiency, breaking the traditional trade-off between pre-computation and query-time computation.

Table 1: Performance trade-offs of storage structures.

	Query Performance	Update Performance	Space Efficiency
Query-time computation with traditional indexes	–	+	+
Pre-computation, i.e., materialized views	+	–	–
Two-table Merged Index	+	+	+
Multi-table Merged Index	**+**	**+**	**+**

Traditional approaches—query-time computation (tables + traditional indexes) and pre-computation (materialized views)—make trade-offs across query performance, update performance, and space efficiency. Two-table merged indexes break this trade-off [37]; this paper extends the result to multi-table merged indexes for order-sharing pipelines.

Contributions. **(a) Generalization of scope**: Merged indexes accelerate order-sharing pipelines that comprise multi-way joins (inner, outer, and semi joins) and grouping operations, a generalization beyond two-table joins from our previous work [37]. **(b) Storage layout and implementation**: We describe the physical layout that embeds interesting orderings in multi-table merged indexes, with practical implementations over two ordered storage structures, B-trees (LeanStore [33]) and LSM-trees (RocksDB [16]), which optimize for different workloads. **(c) Query execution**: We provide algorithms that use merged indexes as a form of partial pre-computation, requiring moderate computation at query time and moderate pre-processing at update time. **(d) Experiments and performance**: Merged indexes closely approach the query speed of materialized views, the practical upper bound, while providing the update efficiency of traditional single-table indexing.

Section Outline. Following the related work in Section 2, three core sections bridge physical database design and query performance, corresponding to Contributions (b)–(d). Section 6 concludes the paper.

2 RELATED WORK

The proposed techniques in this study lie at the intersection of physical database design, indexing, and query execution; it also builds on the insights from query optimization and database schema design. Section 2.1 covers prior techniques that provide the foundation for this work. Sections 2.2 and 2.3 cover prior approaches to complex multi-table queries. They may be computed at query time or pre-computed and stored as materialized views; they may also be partially pre-computed and later require moderate query-time computation. Section 2.4 covers our previous two studies about merged indexes and prior problems addressed with this technique.

2.1 Foundational Techniques

Three groups of prior techniques provide the foundation for this work: ordered storage, interesting orderings in query optimization, and relational normal forms.

2.1.1 Ordered Storage. Ordered storage is an essential component of relational database systems. This paper builds on established systems on *external storage* for sorting and indexing, including the ubiquitous B-trees [4, 11, 18, 20] (e.g., LeanStore [33], WiredTiger [43]). In addition, LSM-trees [28, 49] (e.g., RocksDB [16], Monkey [15], WiscKey [36], KVell [34]) offer excellent write performance. No contributions are claimed in this area. Due to the robustness, efficiency, and popularity of ordered storage, data are often stored in an ordered format, even if this sort order may be under-utilized in query execution.

2.1.2 Interesting Orderings. Interesting orderings are a query optimization technique introduced by Selinger et al. [55] that recommends successive sort-based operators when they can share a common sort order. It identifies sort orders produced by one operator (e.g., an index scan, a sort, or an earlier merge join) that are required by a subsequent operator, thereby avoiding redundant sorting. This technique has since become standard in major relational databases [41, 52, 59].

For instance, TPC-H Q3 includes two primary key–foreign key joins among Customer, Orders, and Lineitem. A naive plan using two consecutive merge joins (Lineitem ⋈ Orders on `orderkey`, then ⋈ Customer on `custkey`) would require four sort operations. Based on such a cost estimate, the query optimizer may choose hash joins. Yet computing the first merge join on composite key `(custkey, orderkey)`—semantically equivalent to using `orderkey` due to functional dependency (see Figure 4 in Section 3 for details)—produces a stream already sorted by `custkey`, as required by the second merge join, eliminating one of the four sorts. After updating the cost estimate, the query optimizer may favor merge joins over hash joins.

Interesting orderings are the primary inspiration for this work; bringing renewed attention to this underexplored technique is itself part of this paper's contribution. While the original technique avoids redundant sorting during query planning and execution, merged indexes embed these interesting orderings into the physical database design, as Section 3 will discuss.

2.1.3 Relational Normal Forms and Complex Application Objects. Complex application objects are application-level entities that include information too complex for a flat-table format, such as customers with their associated orders and order items in TPC-H. Modern RDBMSs manage this complexity through normalization (e.g., 3NF or BCNF) [10, 29]. Normalization decomposes complex application objects across multiple tables tied by relationships, for example, Customer, Orders, and Lineitem tables are tied by primary key–foreign key relationships on `custkey` and `orderkey`.

This decomposition brings about benefits such as controlled redundancy but necessitates the reconstruction of objects at query time via primary key–foreign key joins. Notably, normalization in logical design does not dictate physical storage—by storing multiple tables in one B-tree, merged indexes cluster records representing

these complex objects without violating relational principles and speed up reconstruction queries.

2.2 Query-time Computation and Full Pre-computation

Relational databases divide the workload of computing a query generally between two phases, query time and update time. This section reviews prior work under this framework on the more general problem of multi-table queries, since no previous work except Selinger et al. [55] appears to focus specifically on order-sharing pipelines—a gap this study fills.

Relational database systems typically compute a query on the fly (i.e., at query time). They may run moderate pre-processing on database updates to later accelerate query execution, typically in the form of index creation and index maintenance. Columnar stores often perform even less pre-processing at update time since they may not maintain ordered indexes. This represents one extreme that devotes almost all effort to query time.

On the other end, materialized views fully pre-compute and store a query, shifting almost all effort to update time. At query time, databases only perform a simple retrieval (often an index retrieval from an indexed view [42]). They pay the cost of view storage and view maintenance. Some database systems refresh materialized views only occasionally, and each refresh may even require a full re-evaluation of the query [26, 53]. Incremental view maintenance (IVM) greatly mitigates this cost [7, 22, 23, 30, 44, 65]. However, many systems impose strict constraints on which views can be incrementally maintained, such as the lack of support for outer joins [14, 42, 45]. Oracle's support for IVM is impressively comprehensive, despite the complexity and remaining constraints [50].

Recent academic studies have proposed several high-performance IVM engines, but they also present some trade-offs. DBToaster [1, 46, 57] and Naiad [8, 38] achieve excellent IVM performance by maintaining intermediate views and differential traces in memory, respectively. However, their memory consumption, especially that of the former, can be prohibitive for large datasets. Umbra DB's "split maintenance" [64] achieves both a high insert rate and low query latency, but it lacks support for joins without a subsequent aggregation.

This trade-off between query-time work and update-time work applies to entire queries as well as to order-sharing pipelines embedded in larger queries. In the latter case, a rather similar analysis pertains only to the pipeline, as deferred to Sections 4.5.

2.3 Partial Pre-computation

The middle ground between query-time computation and materialized views is partial pre-computation. Two classic studies, join indices [48, 62] and combined images [24], fall into this category. They are the conceptual predecessors of merged indexes. As Table 2 lays out, join indices store base row IDs (e.g., $(r_{\text{id}}, s_{\text{id}}, t_{\text{id}})$) of each joined row. Combined images co-group the rows without joining them, mapping each join key to sets of matching row IDs in each relation. In comparison, merged indexes generalize combined images by storing actual records (or selected columns) per key.

Recent techniques for partial pre-computation—often introduced under the broad umbrella of IVM—appear in the following studies.

Table 2: Join index [48, 62], combined image [24], and merged index.

	Content	Cardinality
Join index	$(r_{\text{id}}, s_{\text{id}}, t_{\text{id}})$ Joined row IDs	$\lvert R \bowtie S \bowtie T \rvert$
Combined image	key $\rightarrow (\{r_{\text{id}}\}, \{s_{\text{id}}\}, \{t_{\text{id}}\})$ Co-grouped row IDs	Count of unique keys, which is no more than $\lvert R \rvert + \lvert S \rvert + \lvert T \rvert$
Merged index	key $\rightarrow (\{\pi(r)\}, \{\pi(s)\}, \{\pi(t)\})$ Co-grouped records or their projection	$\lvert R \rvert + \lvert S \rvert + \lvert T \rvert$

Three approaches to $R \bowtie S \bowtie T$. r_{id} = row ID of a record in R; $\pi(r)$ = projection of record r.

Factorized IVM [46, 47] stores factorized representations of query results, but it only offers an in-memory implementation. Berkholz et al. [6], Idris et al. [27] offer strong theoretical guarantees but target main-memory environments. In addition, two hash-based approaches are highly complementary to this paper's focus on order-based query operators. IQP [58] persists and maintains intermediate states of query processing, most notably hash tables. Symmetric hash tables [25, 31, 61, 63] may also be considered a form of partial pre-computation.

2.4 Merged Indexes and Their Advantage in Two-table Joins

Merged indexes belong to a broader lineage of multi-table clustering strategies. The network data model, with its explicit owner–member sets traversed record-by-record [3], can be viewed as an early form of multi-table clustering. Object-oriented database systems [13] continued this lineage.[3] More recently, the idea of clustering has surfaced in diverse contexts, including distributed systems (e.g., Spanner's interleaved tables [12]), NoSQL systems (e.g., Cassandra's wide partition [9]), columnar storage (e.g., nested data [39, 54]), and query execution (e.g., Google Flume's "co-group-by" [2]).

Our two previous studies proposed merged indexes [17] and showed their advantage for two-table joins [37], respectively.

2.4.1 Design of merged indexes [17]. A merged index [17] maps a search key to a set of records from different tables, as shown previously in Table 2. A sort order accomplishes this mapping by physically interleaving records from multiple tables and clustering "relevant" records. The interleaving and clustering within a merged index rely on a record structure [17] that accommodates variable record types. This format consists of three components:

(a) Key fields with domain tags enable byte-wise comparison between different types of keys in the same merged index. A domain tag that precedes each key field names its column type (i.e., its "domain"), bounds its extent, and indicates the position of the next domain tag. **(b) Index identifier:** Immediately following the key, a special domain tag (`I` in Table 3) indicates that the next few bytes are not a key field but an index identifier. An index identifier indicates the source index of the record. It is part of the sort key, enabling the interleaving of records with the same join key but different source

[3] In contrast to these early examples, merged indexes, thanks to physical data independence, implement clustering while upholding the relational model.

indexes. **(c) Payload:** The remaining non-key columns of a record are stored as the payload following the index identifier, usually in a binary format without any special tags and requiring parsing based on schema information. It does not participate in the sort order.

Table 3: An Orders record in a merged index with variable-size keys.

Field value	Field type	Denoting...
C	Domain tag	custkey is next
C_1	Data value	The custkey of this record
O	Domain tag	orderkey is next
O_3	Data value	The orderkey of this record
I	Domain tag	Index identifier is next
o	Index identifier	This record is from an index on Orders
'2006/12/20'	Data value	The orderdate of this record
'Urgent'	Data value	The shippriority of this record
...	Data values	Other field values

Field order = byte order in the stored record. Domain tags (pink) bound and name each key field; the index identifier (black) marks the source index. Adapted from Figure 3 of Graefe [17]; see Listing 1 for the implementation.

2.4.2 Merged indexes for two-table joins. Lyu and Graefe [37] recommended merged indexes for two-table joins, because they excel simultaneously in query performance, update performance, and space efficiency. On the one hand, a two-table merged index is simply a reorganization (interleaving by sort order) of the two single-table indexes. On the other hand, by co-locating matching entries from two indexes, the merged index partially pre-computes the join and is only one step away (i.e., the step of record assembly) from the materialized view.

3 STRUCTURE OF MULTI-TABLE MERGED INDEXES—"DATA AT REST"

This section describes the physical layout of multi-table merged indexes in four parts: running examples, the interesting orderings they embed, design trade-offs, and practical cost.

3.1 Running Examples

Physical data layout in a multi-table merged index depends on how normalized schemas decompose complex application objects into linked tables with overlapping keys. Shared keys that naturally form compatible sort orders fall into one of three structural shapes: identical keys, hierarchical prefix chains, or branching trees. Sections 3.1.1, 3.1.2, and 3.1.3 characterize each shape with a running example over a unified schema. A merged index can physically interleave records according to any of these three sort orders.

We ground our discussion in a familiar benchmark schema. The schema shown in Figure 1 includes the Customer, Orders, and Lineitem tables as defined in TPC-H [60]; it is also extended with an Invoice table to support reasoning about more complex multi-table relationships and join patterns. Over this schema, a database system can construct three distinct merged-index designs, summarized in Table 4, which we use as running examples throughout our study.

3.1.1 Identical sort keys. In $\mathbf{MI_A}$, the indexes for Customer, Orders, and Invoice all share custkey as their leading sort key column. Although Orders and Invoice retain their own additional key fields

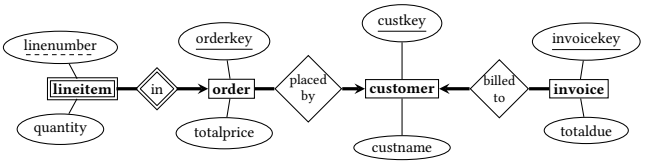


Figure 1: ER diagram extended upon TPC-H schema: a customer with orders, line items, and invoices.

Sections 3.1.1–3.1.3 use this schema to discuss three merged-index options, summarized in Table 4. Thick lines denote total participation; thick arrows point to the key constraint (the "one" side of the relationship). Double borders indicate weak entity sets and their identifying relationships, while dashed underlines denote partial keys.

Table 4: Three options to build a merged index over the schema in Figure 1.

Merged index label	Included indexes[1]	Key shape	Described in
$\mathbf{MI_A}$[2]	I_{cust}, I_{ord}, I_{inv}	Identical	Section 3.1.1
$\mathbf{MI_B}$	I_{cust}, I_{ord}, I_{line}	Hierarchical	Section 3.1.2
$\mathbf{MI_C}$	I_{cust}, I_{ord}, I_{line}, I_{inv}	Tree-shaped	Section 3.1.3

[1] The included indexes are: I_{cust}, primary index of Customer, key custkey; I_{ord}, secondary index of Orders, key (custkey, orderkey); I_{line}, secondary index of Lineitem, key (custkey, orderkey, linenumber); I_{inv}, secondary index of Invoice, key (custkey, invoicekey).
[2] $\mathbf{MI_A}$ forms a shared sort order and clusters records from multiple sources based on custkey only, even though I_{ord} and I_{inv} retain their full keys.

(orderkey and invoicekey), the shared custkey establishes a compatible sort order across all three tables. This shared key enables master-detail clustering [17]: all detail records (orders and invoices) belonging to a given customer are physically co-located alongside their master Customer record.

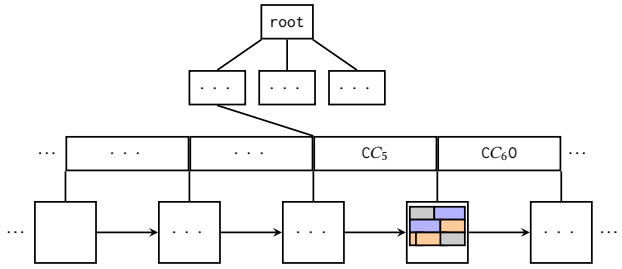
Figure 2a illustrates such a merged index stored in the B-tree format. Compared with a traditional B-tree index, the branch nodes in Figure 2a store keys of variable sizes up to the index identifiers in order to sort and interleave all types of records. Figure 2b magnifies and aligns the example records from Figure 2a, showing orders and invoices nested under the sort range of a customer.

3.1.2 Hierarchical sort keys. Hierarchical sort keys arise when the keys of participating indexes form a prefix chain, where each detail table extends the key of its parent. Because lexicographical sorting by a composite key (e.g., (A, B)) preserves the order of its prefix (A), these keys form nested, multi-level sort ranges.

In $\mathbf{MI_B}$, Customer, Orders, and Lineitem form such a prefix chain on (custkey), (custkey, orderkey), and (custkey, orderkey, linenumber). By contrast, in $\mathbf{MI_A}$ (Figure 2), orders and invoices act as siblings whose keys belong to distinct "subdomains" separated by domain tags. In $\mathbf{MI_B}$, the sort ranges form two nested levels: line items are clustered directly inside their parent order (a "complex sub-object"), which is itself clustered under the customer.

3.1.3 Tree-shaped keys. Tree-shaped keys combine sibling subdomains and hierarchical prefix chains into a branching key structure, as illustrated by $\mathbf{MI_C}$ in Figure 3. Under this layout, all four tables interleave under custkey: invoices and orders form distinct sibling "subdomains", while orders further nest their associated line items.

This unified key tree gives $\mathbf{MI_C}$ the greatest query flexibility among the three designs, enabling a single index to serve queries



(a) MI_A as stored in a B-tree.

Branch keys sort Customer (gray), Orders (blue), and Invoice (orange) records; leaf nodes interleave records from all three tables.

c	C_5	I c	customer info	
c	C_5	o O_3	I o	order info
c	C_5	o O_7	I o	order info
c	C_5	v V_2	I v	invoice info
c	C_5	v V_8	I v	invoice info
c	C_6	I c	customer info	

(b) Example records from MI_A.

The shared prefix is shown in gray to indicate nested sort ranges. Legends: Table 3; implementation: Listing 1.

The three record types share a sort order by `custkey`. Key fields after `custkey` separate invoices and orders within the same customer cluster. They also enable lookups of greater granularity inside customer clusters.

Note: `custkey` is not in Lineitem's schema, but because of `orderkey` → `custkey` (functional dependence), each Lineitem record can be extended with `custkey`, as they are in MI_B and MI_C.

Figure 2: Layout of MI_A—a merged index storing the primary index of the Customer table and secondary indexes of the Orders and Invoice tables.

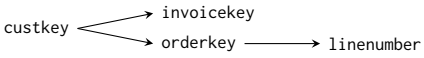


Figure 3: Tree-shaped key structure of MI_C.

The tree describes both the sort-key shape and the record-clustering shape. At each node, the composite key uniquely identifies one record from the table whose key terminates there (1 record per node). Tables with longer keys appear as children, so a single parent-key value can have many child records.

joining any subset of the participating tables (subsuming both MI_A and MI_B). At the physical storage level, however, clustering all detail records per customer object increases the record size, potentially causing large customer instances to span multiple B-tree pages.

3.2 Embedding Interesting Orderings for Multi-table Queries into Merged Indexes

While Section 2.1.2 introduced interesting orderings as a query optimization technique to avoid redundant sorts, they also naturally divide a query plan tree into distinct parts. When consecutive operators (such as merge joins and grouping operators) share and preserve the same tuple order, they form a single *order-sharing pipeline*. This partitions the query plan tree at sort boundaries into pipeline segments where intermediate results require no re-sorting [55]. Merged indexes embed such orderings into physical storage and, as Section 4.1 will discuss, provide an execution algorithm for order-sharing pipelines. This order-sharing pipeline model incorporates grouping operators in addition to multi-table joins, distinguishing our approach from join indices [48, 62], combined images [24], and prior work on two-table joins [37].

In comparison to traditional query plans that recognize shared sort orders at the time of *query planning*, a merged index embeds the shared sort order into *the physical design*, while the latter is also not an entirely new practice. In conventional execution, interesting orderings can leverage single-table indexes as pre-sorted inputs at the scan layer, allowing the optimizer to eliminate explicit sorting before joins. Figure 4 illustrates this for TPC-H Q3 [60][4], where the plan selects three single-table indexes sharing compatible keys to form an order-sharing pipeline (gray box).

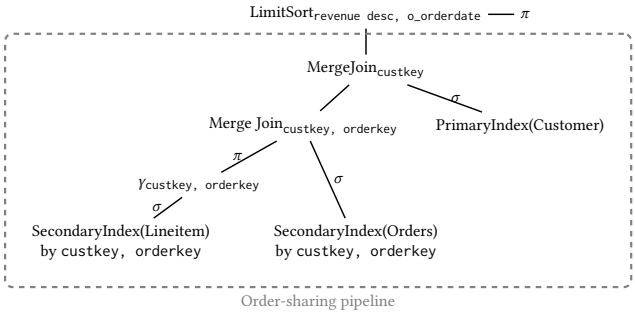


Figure 4: TPC-H Q3 [60]: A possible execution plan, selecting the primary index of the Customer table and secondary indexes of the Orders and Lineitem tables.

Lineitem ⋈ Orders on `(custkey, orderkey)` ≡ join on `orderkey` alone, since `orderkey` → `custkey` (functional determination). `custkey` is readily available if a Lineitem secondary index sorted by `custkey` exists (e.g., MI_B).

Gray box: the order-sharing pipeline (Section 4.3); MI_B can replace all three indexes simultaneously.

Whereas traditional single-table indexes separately embed the sort order of a single table, a merged index embeds these interesting orderings across multiple tables directly into physical storage. Instead of coordinating the three index scans in a query plan, MI_B interleaves all three indexes in its physical layout, embedding the shared sort order required by the query. As Section 4.1 will discuss, TPC-H Q3 becomes a scan over MI_B.

Summary of Section 3.2. Query patterns over complex application objects (Section 2.1.3) naturally produce order-sharing pipelines: multi-table joins reconstruct objects from normalized tables, while aggregations compute complex-object summaries. Both schema structure and these query patterns drive physical design; while Section 3.1 addresses schema key shapes, this subsection focuses on physically embedding their order-sharing pipelines.

[4] Although TPC-H Q3 joins Lineitem and Orders only on `orderkey`, the plan joins them on `(custkey, orderkey)`; this is syntactically additional but logically equivalent, because the foreign-key chain Lineitem → Orders → Customer makes `l_custkey` functionally determined by `l_orderkey`, so propagating it into the join predicate adds a constraint that is already implied. c.f. Section 2.1.2.

3.3 Cost and Implementation

Implementation. The primary implementation requirement for merged indexes is support for a specialized record structure with variable-size keys. Listing 1 sketches the record type for an Orders entry inside $\mathbf{MI_B}$, i.e., a Customer-Orders-Lineitem merged index, illustrating how Table 3 maps onto a concrete implementation.

Listing 1: C++ implementation for the Orders record type that can fold into a merged index.

```
struct orders_sec_by_custkey_t { // secondary index keyed on custkey
   struct Key {
      static constexpr int IDX_ID = 2;
      int custkey;
      int orderkey;
      static void fold2COL(KeyBuilder& out, const Key& k) {
         out << COLDomainTag::CUSTOMER << k.custkey
             << COLDomainTag::ORDERS   << k.orderkey
             << COLDomainTag::INDEX << IDX_ID;
      } // Folds the key into the merged-index layout:
   };
   Timestamp  o_orderdate;
   int        o_shippriority; // Payload: Q3-projected fields only
};
```

Target merged index: $\mathbf{MI_B}$ (COL, short for Customer-Orders-Lineitem). `fold2COL` converts each key to the layout required by the merged index. Full code (incl. endianness handling) at this repository.

Beyond this, the implementation leverages existing components of the storage backend, most importantly, sorting and indexing. Other details, such as latching mechanisms, are also inherited directly. We implement merged indexes on two common storage backends: a B-tree (LeanStore [33]) and an LSM-tree (RocksDB [16]). The feasibility of building merged indexes on these widely used structures suggests a practical path for their integration into existing relational database systems.

Space requirements. Compared with an entry in a traditional index, only the domain tags and index identifier constitute the storage overhead of an equivalent entry in a merged index. Since a domain tag or an index identifier usually takes up no more than one byte, the storage overhead is minimal, as supported by the experiments.

Factorized tables [47] can also be seen as partially executed queries that store join views in a space-efficient way. The focus of merged indexes on sorting and interesting orderings represents a difference in approach and scope. This approach allows a practical implementation of merged indexes on top of ordered storage, which may be a viable, scalable, and persistent representation of factorized databases.

3.4 Design Space and Trade-offs

Indexing and storing views. While this paper focuses on merged indexes storing (indexes of) tables, they can also store indexed views, i.e., query results. For example, consider $\mathbf{MI_B}$ for Figure 4. Instead of storing a secondary index of the full Lineitem table, it can store an aggregate view—the revenue of line items per order—indexed on `orderkey`.[5] Supporting views in merged indexes provides a wider range of design choices, and this decision depends on the workload of the system. We implemented this variant for TPC-H Q10 and evaluated the performance in Section 5.3. The merged indexes in this paper create participating indexes only for base tables or simple aggregate views, instead of complex views. The full scope of storing views in merged indexes is left to future work.

Primary and secondary indexes. Both primary and secondary indexes can participate in a merged index. One merged index can even comprise the primary index and a secondary index of the same table, such as the primary index of the Employee table (sorted by `employeeid`) and its secondary index sorted by `managerid`. A merged index might not store additional copies of the participating indexes but rather replace their original storage. [6]

Included columns. A participating index includes all columns if it is the primary index. Otherwise, a merged index should ideally cover all columns required by all queries it serves, enabling index-only retrieval and eliminating the need to access base tables during query execution. This design choice follows the increasing use of covering indexes in the industry, which reflects a broader trend where the boundary between relational databases and key-value stores continues to blur.

Backend choice; row and columnar storage. Merged indexes, similar to other ordered indexes, are primarily explored in the context of row storage. We choose two row stores, LeanStore [33] and RocksDB [16], as the backend. Yet technically, merged indexes are not limited to row-oriented backends. The core requirements are (A) variable-size keys and payloads, (B) an ordered iterator supporting forward scans and seek-to-key, and (C) standard write operations. Although row stores more commonly optimize for these operations, there are columnar stores that also fit these requirements. Requirement A can be met in several ways. The composite, tagged key fits into a single column of variable-length byte strings. The payload is the harder part and has several feasible designs.[7] For Requirement B, for example, Apache Kudu [35] stores data column-wise but keeps each table sorted by its primary key and exposes a row-oriented scan iterator.

Fundamentally, the design of merged indexes is independent of any specific storage configuration—a major difference from previous clustering strategies, such as Spanner's interleaved tables [12]. The clustering by merged indexes is built upon sort orders, and any ordered storage structure may theoretically underlie a merged index. However, different backends may deliver vastly divergent performance, due to disparate cost profiles for the operations that merged indexes rely on, such as incremental sorting, range scans, and point reads. Our experiments show such divergences between

[5] Standard TPC-H puts `GROUP BY` after the join; eager aggregation lets this single-table aggregate answer the query.

[6] This design is space-efficient and keeps maintenance minimal, but it incurs read overhead when a query needs only a subset of the participating indexes, and write overhead when an update touches only a subset. If this overhead matters for the workload, a separate copy of a participating index can be stored. Merged indexes are designed for join-heavy workloads in which multiple record types are accessed together; they are not universally superior to traditional indexes.

[7] Three example strategies: (i) a wide union schema covering every source table's columns with NULL-padding for inapplicable fields—simple but wasteful; (ii) per-source-table column groups co-located within the shared key order, so each record writes only its own columns; (iii) a single variant column (e.g., a typed BLOB or semi-structured value) carrying the type-specific payload.

B-trees and LSM-trees; we leave a columnar-backend implementation of merged indexes and an empirical comparison against column-store baselines to future work.

Summary of Section 3. Merged indexes organize data into clusters by interleaving records; these "record clusters" often represent instances of complex application objects. This layout materializes the interesting orderings that enable efficient joins and groupings. These two aspects, database schema design and query patterns, often coincide and drive the design of merged indexes. As opposed to join views that represent denormalization at the logical level, merged indexes move denormalization to the level of physical database design. Merged indexes store each record in normal form, offering benefits such as controlled redundancy. This structure lays the foundation for the high-performance query execution techniques discussed in the next section.

4 QUERY EXECUTION AND INDEX MAINTENANCE—"DATA IN FLIGHT"

This section describes query execution over and maintenance of merged indexes in three parts: execution of order-sharing pipelines via a scan, execution of complex queries that embed such a pipeline, and maintenance under data updates. This section eventually shows that merged indexes are a form of partial pre-computation.

4.1 Multi-table Query Execution via a Scan

A multi-table order-sharing pipeline can be executed efficiently via a single-pass scan over an appropriate merged index. The algorithms presume a cost-based query optimizer that can select appropriate merged indexes and instantiate these plans when applicable.

4.1.1 Multi-table join operator over a merged index. The execution operator of multi-table joins over a merged index functions as a non-blocking iterator that processes the interleaved record stream sequentially and produces joined results on the fly. Algorithm 1 shows a generic N-way join.

Algorithm 1 Multi-Table Join Operator over Merged Index (reused for aggregation pipelines by replacing CartesianProduct; see Section 4.1.2).

```
   State: iterator over a merged index I; current key k ← null; buffers of input
   records B[1 . . . n]; output queue Q
1: function Next
2:   if Q not empty then return Q.Pop()
3:   while I.HasNext() do
4:     r ← I.Next(); k′ ← key(r)
5:     if k′ ≠ k and k ≠ null then                          // key change
6:       Q.PushAll(CartesianProduct(B))
7:       for i ← 1 to n do
8:         if Match(B[i].First(), r) < 0 then      // Falling behind r's sort range
9:           B[i].Clear()    // Clear buffer that doesn't match r or future records
10:      k ← k′
11:      if Q not empty then return Q.Pop()
12:    B[r.source].Add(r)
13:  if any B not empty then
14:    Q.PushAll(CartesianProduct(B)); ClearAll(B)
15:    if Q not empty then return Q.Pop()
   return null
```

As the operator scans the interleaved records of the merged index, it accumulates records matching the current join key into the respective index buffers. A change in the join key signifies the end of a group or a subgroup (e.g., a customer group or an order group). It triggers two actions: computing the Cartesian product of all non-empty buffers and generating all joined results for the old join key. A sort-order-aware Match function invalidates stale buffers that "fall behind" the current sort range.[8] The cost is bounded by coarsest groups, e.g., customer groups. Database practitioners should choose the clustering strategies and bound the group size.

This process is analogous to a merge join over coordinated sequential scans of pre-sorted inputs. For joins over a small key range, merged index saves over traditional merge joins on seek count and read co-locality. For larger reads, the per-seek saving fades into the scan cost, and the two approaches would converge if every byte of the inputs had to be read. Realistic queries, however, often carry a selective predicate on one or more inputs (e.g., the `mktsegment` filter on Customer in TPC-H Q3). Both approaches skip on every side upon a rejected join key—traditionally through a "zigzag merge join" [19] and for a merged index through a "skip scan" that we implemented. This way, a larger scan is effectively many smaller scans "stitched" together, yielding similar savings on seek count and read co-locality. The savings scale with the number of gaps in the key range—typically, the more selective a predicate is, the faster the merged index executes a join.

In addition, this algorithm flattens the join chain and requires no join-order optimization within such a pipeline, potentially yielding faster and easier query optimization. This is an advantage that a merge-join chain does not have. Our ongoing work integrates merged indexes into Apache Calcite [5], producing query plans comprising merged index scans.

4.1.2 Execution of Multi-table Aggregation. The execution logic for a pipeline with aggregation is very similar to Algorithm 1, with the main difference in the output generation step. Specifically, the CartesianProduct function at Lines 6 and 14 in Algorithm 1 is replaced with a query-specific function that computes the aggregate from the buffered records. For TPC-H Q3, this function should calculate the revenue from the line items per order and extend the order summary row with customer information. The rest of the logic, such as key change detection, remains the same. Alternatively, if the merged index stores the aggregate view of order revenue as suggested back in Section 3.4, this pipeline in TPC-H Q3 reduces to a three-way join between Customer, Orders, and the aggregate view.

Aside from grouping operators, `ORDER BY` and `SELECT DISTINCT`, among many other SQL constructs, also have order-based execution algorithms and are thus executable by merged indexes.

4.1.3 Limitations. Merged indexes are not optimal in every setting; we note several limitations below. The co-locality benefits of merged indexes depend on moderate sizes of "complex object instances", e.g., a customer object that can fit all their orders and invoices within one B-tree leaf page. Small dimension tables (e.g., Region

[8] Match($B[i]$.First(), r) returns a negative value when the cursor at r has advanced past the longest sort-key prefix that $B[i]$.First() and r share, meaning no future record can match $B[i]$.First() on its join columns and $B[i]$ is safe to clear. Concretely, in $\mathbf{MI}_{\mathrm{C}}$ (Customer-Orders-Lineitem-Invoice), an invoice and a lineitem share only the `custkey` prefix; the invoice buffer is therefore cleared when `custkey` changes, not when an intervening `orderkey` or `invoicekey` advances within the same customer.

or Nation) may not be the best root object, since their children easily span GiBs of leaf pages, and most of the child records have to resort to a random seek to retrieve the dimension record, essentially voiding the co-locality benefits.

Our experiments show $\mathbf{MI_C}$'s edge over the baselines is smaller than $\mathbf{MI_B}$'s, presumably because the extra invoice records increase the chances of spreading a customer object across pages. Yet, including additional record types increases the combinatorial advantage of merged indexes. For example, $\mathbf{MI_C}$ covers all queries covered by $\mathbf{MI_A}$ and $\mathbf{MI_B}$. It is important that database practitioners strike a balance between combinatorial advantage and co-locality. Since practical workloads usually have a schema more complex than TPC-H, it may not be wise to merge each and every table using a shared key in one huge merged index. We recommend starting with a modest number of tables that frequently join together, while the exact number depends on schema, key value distribution, workload, and backend configurations (e.g., the page size). See Section 5.6 for additional workload-side trade-offs.

Summary of Section 4.1. Despite being logically multiple operators, an order-sharing pipeline is executed through a (range) scan over an appropriate merged index. This scan and record assembly constitute the entire query-time effort.

4.2 Query Coverage

Merged indexes can support queries with various join multiplicities, join types, and various aggregation functions. Functional dependencies further broaden their query coverage. Essentially, their relatively broad coverage comes from the minimal pre-processing, as Section 4.5 will discuss.

Support for various join multiplicities and join types. Algorithm 1 supports both many-to-many and many-to-one joins, since each buffer may contribute one or multiple records to the Cartesian product. Customer-Orders-Lineitem in $\mathbf{MI_B}$ is a hierarchical many-to-one example, while Customer-Orders-Lineitem-Invoice in $\mathbf{MI_C}$ matches multiple orders and invoices within a customer.[9] CartesianProduct in Algorithm 1 may be replaced with a function specific to outer joins, semi joins, or anti joins, and all join types can be executed within this single-pass scan.

Aggregation functions. Materialized views often only support commutative aggregation functions (or "algebraic" [21] aggregation functions) including `COUNT`, `SUM`, or `AVG`, because these aggregated views can be easily maintained [40, 51]. In contrast, merged indexes support relatively complex aggregations, such as `MEDIAN`. For example, consider computing the median invoice amount per customer. A merged index stores all invoices for each customer and, within each customer cluster, sorts the invoices by their amounts at query time. This is similar to how a traditional index of invoices sorted by `custkey` also supports this query.

[9]Such a many-to-many relationship is limited within the sort range of a customer. A classic many-to-many relationship is, for example, Student and Course linked by an Enrollment bridge table. A merged index keyed on `studentid` clusters each student with their enrollments, while a second one keyed on `coursenumber` clusters each course with its enrollments. A join between students and courses involves two disparate sort orders and is technically no longer an order-sharing pipeline. We defer these queries to future work.

Utilizing functional dependency. Functional dependency recognizes shared sort orders that may not be immediately obvious from the key composition. It is a fundamental technique in query optimization [56] that is also applicable to the design of merged indexes. As shown by our running example, it allows the Lineitem table to be indexed on `custkey`, thus sharing a sort order with the other participating indexes in $\mathbf{MI_A}$, $\mathbf{MI_B}$, or $\mathbf{MI_C}$. Additional order sharing brings about greater combinatorial advantage and greater query coverage—for example, $\mathbf{MI_C}$ can handle all pipelines requiring a subset of the participating indexes, including those covered by $\mathbf{MI_A}$ and $\mathbf{MI_B}$.

4.3 Execution of Complex Queries

An order-sharing pipeline may be a subtree of a larger, more complex plan. For example, the order-sharing pipeline of TPC-H Q3 is the subtree marked by the gray box in Figure 4. The trailing LimitSort and projection sit outside the pipeline and run as the system usually processes queries without merged indexes.

More generally, a complex query splits into operators that a merged-index scan can execute and operators that it cannot. The latter range from trivial post-pipeline steps (as in TPC-H Q3) to anything that the existing system can process. For example, while a rather similar Customer-Orders-Lineitem pipeline appears in TPC-H Q5, its output is joined with three more tables, Supplier, Nation, and Region.

We presume a query optimizer to correctly identify each order-sharing pipeline, since interesting orderings have become standard in modern relational databases. The missing gap is to convert such an order-sharing plan to one based on a scan of the appropriate merged index; our ongoing work will provide a prototype implementation to fill this gap. This pipeline identification is the core strategy for generalizing merged indexes from two-table joins. It is also the bridge to more complex queries, i.e., queries that comprise multiple order-sharing pipelines, each with its own sort order—analogous to multiple sets of interesting orderings in the plan. Ongoing work investigates such complex queries and how to execute them with multiple merged indexes.

4.4 Index Maintenance

For standard insertions, updates, or deletions of base table records, the system simply updates the corresponding entry in the merged index. Similar to the case of traditional indexes, this is a 1-to-1 operation: one table update triggers one index update. This 1-to-1 maintenance characteristic contrasts with materialized join views, where a single base table update requires accessing other tables and may generate multiple view updates (1-to-N).

Furthermore, physical clustering in merged indexes speeds up updates to "relevant" records across several tables common in relational databases, such as TPC-H RF1 that inserts orders with their line items. Clustering may benefit complex updates even more, such as referential integrity checks and cascading deletes. The latter becomes a range-delete operation since child records are stored within the sort range of the parent.

There are factors that may slow down maintenance of merged indexes. Compared with traditional indexes, an equivalent merged index stores more data in one storage structure and may incur a

higher access cost (e.g., longer tree traversal), but this slowdown is not shown in experiments. Moreover, when only a subset of the participating indexes is frequently updated, the other static indexes may incur extra overhead, such as disrupting a range insert. If such operations are frequent and the performance gain in queries cannot justify the additional cost, systems may instead maintain traditional indexes. Merged indexes are only an addition to relational databases and do not seek to replace existing indexing techniques.

In summary, merged indexes offer comparable update performance to traditional indexes and avoid the high maintenance overhead of materialized views.

4.5 Spectrum of Pre-computation

A pipeline supported by a merged index may be an entire query or a subtree. While the pre-computation spectrum described in Section 2.2 applies to entire queries, a rather similar analysis applies to query subtrees. The three strategies—traditional indexes, merged indexes, and materialized views—apply to order-sharing pipelines within a larger plan.

On one end of the spectrum, a materialized view stores the fully computed output of the pipeline, e.g., the initial join result in Figure 4. This intermediate view eliminates query-time work for that pipeline but requires the final processing. A single base update must undergo evaluation of this pipeline and generate the view delta. This may result in numerous view updates (1-to-N).

On the other end of the spectrum, traditional indexes store the inputs of this pipeline, e.g., Lineitem and Orders tables. All operators in the pipeline, as well as the final processing of the pipeline output, are evaluated at query time.

In the middle of the spectrum, a merged index stores the *sources* of the pipeline, similar to traditional indexes, but it co-locates and pre-sorts multiple types of records, so that the pipeline reduces to a sequential scan and on-the-fly record assembly. At update time, both traditional indexes and merged indexes exhibit a lightweight 1-to-1 maintenance cost, where one base table modification triggers a single, corresponding index update.

Table 5: Three different approaches to distribute pipeline processing between query time and update time.

	Pipeline processing	**Query coverage**
Materialized views	Primarily at update time	This pipeline only
Traditional indexes	Primarily at query time	Any pipeline over indexed tables
Merged indexes	Update time: sort the delta; query time: scan & assemble joined records	Pipelines sharing the join predicate(s) over any subset of the indexed tables

Workload distribution also affects query coverage, as shown in the last column of Table 5. The more of a pipeline's computation is shifted to update time, the more specialized the resulting structure becomes, thus supporting a narrower range of pipelines. Traditional indexes remain general-purpose, serving any pipeline over the indexed tables, including index nested loop joins, merge joins, and predicate evaluation. A materialized view, by storing the output of one specific pipeline, is optimized for that pipeline but may not be able to serve many others.[10] Again, merged indexes occupy the middle ground: they serve any pipeline over a subset of their tables that can utilize their sort orders. For example, $\mathbf{MI_C}$ is useful for both TPC-H Q3 and a binary join between Orders and Customer.

In summary, while all three approaches require the same amount of query processing *beyond* the pipeline, they represent varying distributions of work between update time and query time *within* the pipeline. Merged indexes partially pre-compute a pipeline by shifting sorting and clustering to update time, leaving only scan-and-assemble to query time. They avoid the excessive pre-processing of materialized views, which produces structures useful for only a narrow range of pipelines. Focused on order-sharing pipelines, merged indexes are a principled form of partial pre-computation.

Summary of Section 4. The physical layout of merged indexes enables multi-table query execution via a scan. By handling the sorting and clustering of data at update time, this technique eliminates what is often the most expensive component of query processing—sorting or hashing—while avoiding the maintenance overhead associated with materialized views. This confirms the merged index's position as partial pre-computation—the middle ground between traditional indexes and materialized views.

5 EXPERIMENTS

This section evaluates multi-table merged indexes on TPC-H order-sharing pipelines. We first state the three hypotheses the experiments test and describe the workload and environment. Next, we discuss query performance, maintenance overhead, and space requirements. This section ends with divergences observed between B-tree and LSM-tree backends and a summary of trade-offs.

5.1 Hypotheses

As a reminder, the research goal of this study is to investigate the performance of multi-table merged indexes for order-sharing pipelines. The context is the classic dilemma between query-time computation (with traditional indexes) and pre-computation (i.e., materialized views). Table 6 breaks down this broad claim into specific hypotheses that the experiments test.

Table 6: Experiment Hypotheses H1, H2, H3

	Query Performance	Update Performance	Space efficiency
Query-time computation with traditional indexes	Moderate	Fast	High
Materialized views with traditional IVM	Fast	Slow	Low
…with DBToaster [1]		Very Fast	Extremely low
Multi-table merged indexes	**Fast (H1)**	**Fast (H2)**	**High (H3)**

Hypotheses tested: merged indexes closely approach the best baseline on query performance (**H1**); match the best baselines on update performance (**H2**) and space efficiency (**H3**).

[10] As a practical example, widening one view to cover additional pipelines may be prohibitive because join fan-out replicates every added payload column across all joined rows; see Table 7 for details.

Notably, merge joins using pre-sorted inputs are already a strong baseline, so even moderate gains over them are meaningful. Furthermore, a materialized view serves order-sharing pipelines directly from pre-computed output and thus sets the practical upper bound on query latency. Closely trailing this bound (**H1**) while preserving update performance (**H2**) and space efficiency (**H3**) of traditional indexes is the central claim for merged indexes.

5.2 Experiment Setup

Workload. We evaluate selected TPC-H queries [60] that consist of an order-sharing pipeline; this set covers the query components that this paper targets. Two pipeline shapes are evaluated, each matched to a merged index from Table 4. For update performance, we test the TPC-H refresh functions. Specifically, we evaluate—

(a) Q3, Q5, Q10: three standard TPC-H queries consisting of a Customer–Orders–Lineitem pipeline (matched to $\mathbf{MI_B}$). **(b) Q3i, Q5i, Q10i**: TPC-H Q3, Q5, Q10 extended with an Invoice table based on the schema in Figure 1. These Invoice-extended queries include a Customer–Orders–Invoice–Lineitem pipeline (matched to $\mathbf{MI_C}$). See https://github.com/alicia-lyu/vldb19-artifact for the specification. **(c) RF1, RF2**: the standard TPC-H refresh functions, which insert and delete order–lineitem groups, respectively. The update size is set to 1 (no batching).

Baselines. Table 7 shows the three baselines, each defined by a pair of a query-execution strategy and a corresponding storage strategy. The experiments vary the query execution only of the targeted order-sharing pipelines and keep everything beyond the pipeline consistent across all approaches, e.g., all joins outside the pipeline are either index nested loop joins or hash joins. The first two baselines, Base-Hash and Base-Merge, represent query-time computation in Table 6, while the third baseline, Mat-View, precomputes and materializes the pipeline output. Each secondary structure stores the covering columns for the queries it serves.

Environment. Experiments run on a CloudLab c220g2 node. It has two Intel Xeon E5-2660 v3 CPUs at 2.60 GHz and 160 GB of DDR4 memory. For storage, it has an Intel DC S3500 480 GB SATA SSD and two 1.2 TB 10K RPM SAS HDDs. Results use the SSD by default; HDD results are reported where explicitly noted. The primary evaluation targets a beyond-memory workload[11]: the engine DRAM budget is fixed at 1.0 GiB with the OS page cache bypassed. Results for other memory budgets are reported where informative. The experiments run on two storage engines, LeanStore (B-tree) [33] and RocksDB (LSM-tree) [16], with specific configurations reported in https://github.com/alicia-lyu/vldb19-artifact.

Each measured query (e.g., Q3) runs against four background threads. Three execute TPCH queries (Q3, Q5, Q10 or Q3i, Q5i, Q10i) using the same storage option as the foreground query[12]; another issues a uniform-random point-lookup stream over all base tables. Queries are measured over five repetitions with rotating substitution parameters. We report the median and IQR of per-query latency. Update throughput is measured at a steady RF1+RF2 rate after a warm-up phase.

Table 7: Four Approaches[1] to Q3, Q5, and Q10[2], in terms of storage strategies and physical execution. The four approaches to Q3i, Q5i, and Q10i mirror these designs, e.g., Merged-Idx requires $\mathbf{MI_C}$ only.

Label	Physical Execution of the Pipelines	Extra Storage structures	DB Size[2] (GiB)
Base-Hash	Hash-based	None	8.57
Base-Merge	Order-based (e.g., merge joins)	Secondary indexes equivalent to $\mathbf{MI_B}$, as pre-sorted inputs	10.81
Mat-View	Pre-computed and materialized	Materialized views, one for each query[3]	31.48[4]
Merged-Idx	Order-based	$\mathbf{MI_B}$	11.06

[1] Mat-View: a parameter-reusable indexed view, not a per-query result cache. The latter would dominate any index on query performance, but is infeasible across the substitution parameters that TPC-H (and most analytic workloads) require.
[2] Total DB Size is the LeanStore (B-tree) on-disk footprint in GiB. Each value is the size of one image jointly supporting all three queries, Q3, Q5, and Q10; the two families (TPCH and TPCHi) live in separate images.
[3] In theory, one materialized view suffices, i.e., the three-way join of MI_B, but the performance turned out to be prohibitive. Since a merged index stores each record once, widening its payload to cover additional queries costs proportionally; a materialized view replicates each row by its join fan-out, so adding payload columns for better query coverage turned out to inflate the view disproportionately.
[4] This number counts the naive 16.88 GiB three-way join for Q10; using the partially aggregated view instead (see Section 5.3) lowers it to 15.84 GiB.

5.3 Query Performance

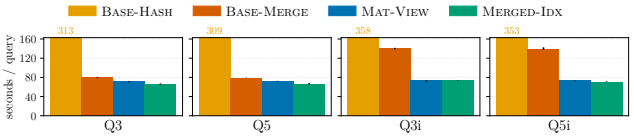
Across both backends, query latency of Q3, Q3i, Q5, and Q5i (Figure 5) shows a general trend: Base-Hash > Base-Merge > Mat-View. In this order, each approach invests more pre-processing and, in return, delivers faster queries. Q10 and Q10i depart from this pattern, as discussed separately below.

Across Q3, Q3i, Q5, and Q5i, Merged-Idx closely trails Mat-View—and on several queries matches or surpasses it—while it beats the other two baselines. We believe the reason is that Merged-Idx has a smaller working set ($\mathbf{MI_B}$) than Mat-View (a view per query). The one regime where Merged-Idx trails Mat-View by a non-negligible margin is the Invoice-extended queries on LSM-trees, where Merged-Idx runs at roughly 1.6× the latency of Mat-View, as Section 5.5 further analyzes.

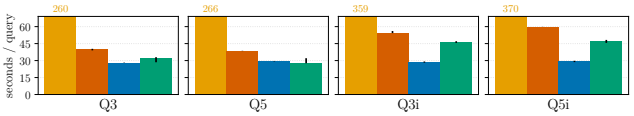
Q10 and Q10i deviate from the general pattern, as shown in Figure 6, primarily in that Mat-View performs poorly. The naive Mat-View stores the full three-way join between Customer, Orders, and Lineitem, especially replicating `c_comment` (max. 117 bytes), incurring space and scan overhead.[13] A common optimization, partial pre-aggregation, mitigates this, as shown in Figure 6. However, this optimization also applies to Merged-Idx and achieves even better query performance.[14]

Comparing each TPC-H query with its Invoice-extended version, Merged-Idx loses its edge in the latter. As more records are clustered under the sort range of one customer, we believe the locality

[11] Per-backend scale factor chosen so both base images reach a similar on-disk footprint (~ 8.6 GiB for B-tree, ~ 7.9 GiB for LSM-tree). RocksDB packs ~ 2.7× denser than LeanStore (LSM compaction vs. half-empty B-tree pages). All structures within a backend share one SF/DRAM pair; absolute numbers are not compared across backends.
[12] Q10/Q10i with Mat-View, when running only as a background query, use the partially aggregated variant (see Section 5.3). The naive view for Q10 would thrash the buffer pool, and the foreground query would heavily regress.
[13] TPC-H Q10 aggregates "lost revenue" from returned items per customer over a given quarter. `orderdate` is a substitution parameter, so per-customer pre-aggregation is



(a) B-tree backend (LeanStore).



(b) LSM-tree backend (RocksDB).

Figure 5: Query latency for TPC-H Q3, Q5, and their Invoice-extended variants Q3i, Q5i, across the four approaches.

Lower is better. The color bars show the median, while the whiskers span the interquartile range (Q1, Q3) over 5 repetitions. Clipped Base-Hash bars exceed the y-axis range; the orange number above each marks their median latency (seconds). Q10/Q10i: see Figure 6.

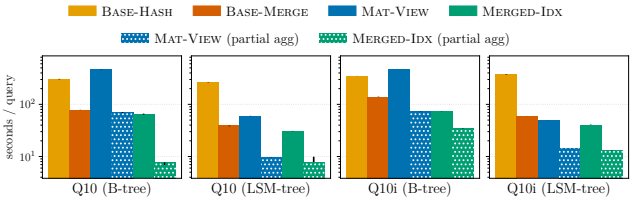


Figure 6: Query latency for Q10 and Q10i, with the partial-aggregation variants of Mat-View and Merged-Idx.

Lower is better; y-axis is log-scaled. Partial pre-aggregation shrinks the merged index enough that its working set fits in memory; LeanStore is optimized for this workload, explaining the strong performance of partial-agg Merged-Idx on B-trees.

benefit of Merged-Idx wanes, e.g., the records of one customer may scatter across multiple B-tree leaf pages, as detailed in Section 4.1.3.

Summary. Merged-Idx closely trails Mat-View—and on several queries matches or surpasses it—while it beats the other two baselines. This confirms **H1**.

5.4 Maintenance Overhead and Space Requirements

Index maintenance overhead under TPC-H refresh functions. On B-trees, Merged-Idx processes the TPC-H refresh functions at roughly 1.4× the throughput of the traditional indexes used by Base-Merge, and much faster than Mat-View, as shown in the left part of Figure 7. Beyond the default 1 GiB memory budget, we also evaluate a 20 GiB budget, the minimum required for DBToaster to run. DBToaster provides a performance gain around a factor of 2 at the expense of a memory footprint an order of magnitude larger.

not possible; Mat-View thus stores a naive three-way join, amounting to a 16.88 GiB view.

[14]The "partial-agg" variants pre-aggregate per order rather than per customer: the merged index stores customer records, order records, and per-order lost revenue; the view stores their join. At query time, partial aggregates are filtered by `orderdate`, then the revenue of surviving orders is summed per customer.

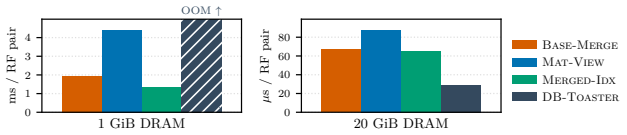


Figure 7: Maintenance overhead measured by latency of TPC-H refresh functions (B-tree backend).

Lower is better. RF pair = TPC-H RF1+RF2. DBToaster: Out-Of-Memory at 1 GiB. Backend: LeanStore (except DBToaster, which uses its own indexing format). The 20 GiB DRAM runs do not use background threads, to stay consistent with DBToaster. Base-Hash maintains no secondary structure, so it is not a meaningful baseline for **H2**. Its updates run at ∼ 2.7× and ∼ 0.93× the latency of Base-Merge, on B-trees and LSM-trees, respectively. the LSM compaction path absorbs much of the secondary-maintenance cost but penalizes random access with which background queries retrieve the build-side rows on hash hit (see SST-read and background-compaction breakdown available at https://github.com/alicia-lyu/vldb19-artifact).

Figure 8 further shows that Merged-Idx can offer an update latency even lower than Base-Merge. Because Merged-Idx clusters an order with its line items, it runs RF1/RF2 even more efficiently than Base-Merge, especially given a tight memory budget using the B-tree backend. On LSM-trees, it no longer leads but still matches Base-Merge, as discussed in Section 5.5.

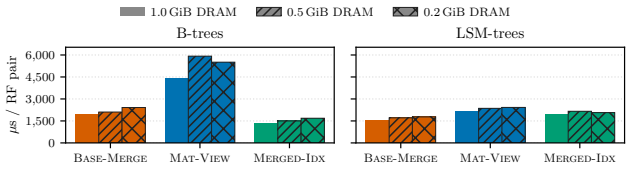


Figure 8: Maintenance overhead measured by latency of TPC-H refresh functions across the B-tree and LSM-tree backends.

Lower is better. RF pair = TPC-H RF1+RF2. Merged-Idx leads on B-trees and matches Base-Merge on LSM-trees.

Space requirements. Merged indexes occupy slightly more disk space than traditional secondary indexes, as shown in Table 7. $\mathbf{MI_B}$ occupies 2.49 GiB, whereas the equivalent secondary indexes require 2.23 GiB.

Compared with Mat-View, Merged-Idx offers much greater space efficiency. First, with the B-tree format, $\mathbf{MI_B}$ (2.49 GiB) is smaller than the materialized view for Q3 (2.83 GiB) and for Q5 (3.20 GiB). For Q10, the partial-aggregation variant adds 0.29 GiB on top of $\mathbf{MI_B}$, while the equivalent view consumes 1.24 GiB. Second and more importantly, $\mathbf{MI_B}$ is shared across Q3 and Q5 and may also be used by Q10, whereas Mat-View must create one view for each query. The views amount to 7.26 GiB in total across the three family queries, even after replacing the Q10 join view with the partially aggregated variant.

Summary. The experiments confirm both **H2** and **H3**: Merged-Idx matches the best baselines on update performance and space efficiency.

5.5 B-Trees vs. LSM-Trees

While all three hypotheses largely hold on both backends, the following divergences are observed:

LSM-trees shrink Mat-View disproportionately. Compared with Merged-Idx, Mat-View materializes fewer but wider rows, and neighboring rows share longer key prefixes. LSM-tree's prefix truncation likely exploits this difference and shrinks each view on disk. On RocksDB, $\mathbf{MI_B}$ occupies 1.81 GiB while the materialized views for Q3 and Q5 require only 1.74 GiB and 1.68 GiB, inverting the LeanStore order. Merged-Idx still requires less space in total than Mat-View because $\mathbf{MI_B}$ is shared across queries.

LSM-trees improve the query performance of Mat-View disproportionately. The bespoke storage-layout properties also accelerate scans for Mat-View: fewer but wider rows reduce per-record processing overhead along the LSM-tree scan path. On the contrary, Merged-Idx stores many narrower rows and incurs higher per-record processing overhead.

The Invoice-extended queries fold one additional table (Invoice) into the merged index, add to the narrow records for Merged-Idx, and widen the joined records for Mat-View; this is likely what widens the gap in query performance on LSM-trees, as noted in Section 5.3. Interestingly, this LSM-specific gain of Mat-View vanishes on HDD (results available at https://github.com/alicia-lyu/vldb19-artifact), perhaps because the IO cost of HDDs dominates the comparison and dwarfs this processing overhead.

Maintenance overhead of LSM-tree indexes and B-tree indexes. Figure 8 shows that, on LSM-trees, Merged-Idx no longer leads but still matches Base-Merge. $\mathbf{MI_B}$ is the largest storage structure across four approaches; this is likely why Merged-Idx takes the longest to compact (SST-read and background-compaction breakdown available at https://github.com/alicia-lyu/vldb19-artifact). While **H2** holds for both backends, Merged-Idx takes relatively longer to maintain.

Summary. While all three hypotheses still hold on LSM-trees, B-trees favor Merged-Idx in multiple aspects and yield its largest advantage.

5.6 Trade-offs of Merged Indexes

The experiments focus on the target workload of order-sharing pipelines. For other workloads, merged indexes may not be the optimal choice; see also Section 4.1.3 for query-shape limitations. In addition, merged indexes are optimized for workloads where queries utilize the full set of interleaved tables or a large subset. Other query patterns, such as those accessing a subset of the tables (e.g., accessing only customer entries in a Customer-Order-Invoice index), entail read overhead due to the traversal of interleaved data, even though merged indexes support such queries. If such queries are common, a system may choose to maintain additional indexes on individual tables, but this comes at the cost of increased space and update overhead. Ultimately, merged indexes are designed for join-heavy workloads where multiple tables are accessed together.

Additionally, because a merged index captures the outer join result, executing inner joins requires filtering non-matching records. This overhead is most pronounced in low-selectivity inner joins where few input rows yield matches. Our previous work provided experiment results setting different levels of join selectivity for two-table joins, showing that merged indexes underperform materialized views at low selectivity [37]. Future research may address this by partitioning merged indexes to concentrate inner join candidates. Conversely, merged indexes can efficiently support outer joins while materialized inner joins cannot.

Crucially, merged indexes are an addition to classic relational database systems. All existing and robustly tested techniques, such as hash joins, traditional indexes, and traditional materialized views, remain available for workloads that do not fit the target patterns of merged indexes.

6 CONCLUSION

Relational database systems often face a trade-off between efficient frequent updates and complex, multi-table queries. Existing approaches often force a compromise: materialized views optimize query performance at the expense of update performance and storage, and vice versa for traditional query-time joins. This paper focuses on order-sharing pipelines, i.e., multi-way joins and grouping operations that can share a sort order, which can be complete queries or parts of more complex queries. This paper revives the concept of interesting orderings [55] and recognizes the pattern of shared sort orders in such pipelines, so that a single-pass scan of a merged index can support multiple joins and grouping operations. This technique breaks the trade-off between materialized views and traditional query-time computation, offering fast queries, fast updates, and high space efficiency. This paper provides two practical implementations of multi-table merged indexes, utilizing two ordered storage structures, B-trees and LSM-trees, respectively.

While merged indexes might seem to violate relational principles, merged indexes exist only at the physical level and, due to physical data independence, cannot violate any logical database design. In fact, materialized join views represent logical denormalization, violate the relational principle of normal forms, and forgo benefits such as controlled redundancy. In contrast, merged indexes move denormalization from logical database design to physical database design, keeping to relational principles at the logical level.

Merged indexes are a form of partial pre-computation that shifts sorting and clustering to update time. At query time, the system performs only a sequential range scan over a merged index and record assembly, eliminating the need for sorting or hashing. Focusing on order-sharing pipelines, merged indexes are a principled approach to partial pre-computation.

Future work on complex queries will use order-sharing pipelines as building blocks. While a significant portion of standard workloads [32, 60] includes order-sharing pipelines, complex queries can require multiple pipelines, each with a different sort order. Ongoing work studies the application of merged indexes in such complex queries.

ACKNOWLEDGMENTS

We thank our colleagues for their feedback and support throughout this research, including Prof. AnHai Doan, Prof. Aaron Elmore, Yifei Yang, Chenhao Ye, Wenjie Hu, and Riki Otaki.

REFERENCES

[1] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. 2012. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. In *Proceedings of the VLDB Endowment*, Vol. 5. VLDB Endowment, Istanbul,

Turkey, 968–979. https://doi.org/10.14778/2336664.2336670 This work was supported by ERC Grant 279804. Articles from this volume were invited to present their results at The 38th International Conference on Very Large Data Bases, August 27th–31st 2012, Istanbul, Turkey.

[2] Apache Beam. 2024. CoGroupByKey. https://beam.apache.org/documentation/transforms/python/aggregation/cogroupbykey/ Accessed: 2024-09-30.

[3] Charles W. Bachman. 1973. The Programmer as Navigator. *Commun. ACM* 16, 11 (1973), 653–658. ACM Turing Award Lecture, 1973.

[4] Rudolf Bayer and Edward McCreight. 1972. Organization and maintenance of large ordered indexes. *Acta Informatica* 1, 3 (1972), 173–189. https://doi.org/10.1007/BF00289509

[5] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. ACM, 221–230. https://doi.org/10.1145/3183713.3190662

[6] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. 2017. Answering Conjunctive Queries under Updates. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '17)*. Association for Computing Machinery, New York, NY, USA, 303–318.

[7] José A. Blakeley and Nancy L. Martin. 1990. Join Index, Materialized View, and Hybrid-Hash Join: A Performance Analysis. In *ICDE*. 256–263.

[8] Mihai Budiu, Leonid Ryzhyk, Gerd Zellweger, Ben Pfaff, Lalith Suresh, Simon Kassing, Abhinav Gyawali, Matei Budiu, Tej Chajed, Frank McSherry, et al. 2025. DBSP: automatic incremental view maintenance for rich query languages: M. Budiu et al. *The VLDB Journal* 34, 4 (2025), 39.

[9] J. Carpenter and E. Hewitt. 2022. *CASSANDRA: The Definitive Guide, (revised) Third Edition*. O'Reilly Media, Incorporated. https://books.google.com/books?id=MKVaEAAAQBAJ

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

[11] Douglas Comer. 1979. The ubiquitous B-tree. *Comput. Surveys* 11, 2 (1979), 121–137. https://doi.org/10.1145/356770.356776

[12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (Aug. 2013), 22 pages. https://doi.org/10.1145/2491245

[13] Jérôme Darmont and Le Gruenwald. 1996. A comparison study of object-oriented database clustering techniques. *Information Sciences* 94, 1 (1996), 55–86. https://doi.org/10.1016/0020-0255(96)00119-3

[14] Databricks. 2018. Introducing Stream-Stream Joins in Apache Spark 2.3. https://www.databricks.com/blog/2018/03/13/introducing-stream-stream-joins-in-apache-spark-2-3.html Accessed: 2026-07-20.

[15] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 79–94.

[16] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. *ACM Trans. Storage* 17, 4, Article 26 (10 2021), 32 pages. https://doi.org/10.1145/3483840

[17] Goetz Graefe. 2007. Algorithms for merged indexes. In *Datenbanksysteme in Business, Technologie und Web (BTW 2007)–12. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme"(DBIS)*. Gesellschaft für Informatik e. V., 112–131.

[18] Goetz Graefe. 2011. Modern B-Tree Techniques. *Found. Trends Databases* 3, 4 (2011), 203–402.

[19] Goetz Graefe. 2011. Modern B-tree techniques. *Foundations and Trends in Databases* 3, 4 (2011), 203–402.

[20] Goetz Graefe. 2024. More Modern B-Tree Techniques. *Found. Trends Databases* 13, 3 (2024), 169–249.

[21] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. 1996. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *Proceedings of the Twelfth IEEE International Conference on Data Engineering*. IEEE, New Orleans, LA, 152–159.

[22] Ashish Gupta, Inderpal Singh Mumick, et al. 1995. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin* 18, 2 (1995), 3–18.

[23] Himanshu Gupta and Inderpal Singh Mumick. 2006. Incremental maintenance of aggregate and outerjoin expressions. *Information Systems* 31, 6 (2006), 435–464. https://doi.org/10.1016/j.is.2004.11.011

[24] Theo Haerder. 1978. Implementing a generalized access path structure for a relational database system. *ACM Trans. Database Syst.* 3, 3 (Sept. 1978), 285–298. https://doi.org/10.1145/320263.320284

[25] W. Hong and M. Stonebraker. 1993. Optimization of Parallel Query Execution Plans in XPRS. *Distributed and Parallel Databases* 1, 1 (1993), 9–32.

[26] IBM Db2. 2024. Refreshing system-maintained materialized query tables. https://www.ibm.com/docs/en/db2-for-zos/13?topic=tables-refreshing-system-maintained-materialized-query-table Accessed: 2024-09-01.

[27] Muhammad Idris, Martin Ugarte, and Stijn Vansummeren. 2017. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1259–1274.

[28] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and Rama Kanneganti. 1997. Incremental Organization for Data Recording and Warehousing. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 16–25.

[29] William Kent. 1983. A simple guide to five normal forms in relational database theory. *Commun. ACM* 26, 2 (Feb. 1983), 120–125. https://doi.org/10.1145/358024.358054

[30] Per-Åke Larson and Jingren Zhou. 2007. Efficient Maintenance of Materialized Outer-Join Views. In *2007 IEEE 23rd International Conference on Data Engineering*. 56–65. https://doi.org/10.1109/ICDE.2007.367851

[31] Ramon Lawrence. 2005. Early hash join: A configurable algorithm for the efficient and early production of join results. In *Proceedings of the 31st international conference on Very large data bases*. 841–852.

[32] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[33] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. Leanstore: In-memory data management beyond main memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 185–196.

[34] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 447–461.

[35] Todd Lipcon, David Alves, Dan Burkert, Jean-Daniel Cryans, Adar Dembo, Mike Percy, Silvius Rus, Dave Wang, Matteo Bertozzi, Colin Patrick McCabe, Andrew Wang, and Michael Yoder. 2015. *Kudu: Storage for Fast Analytics on Fast Data*. Technical Report. Cloudera, Inc.

[36] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *ACM Trans. Storage* 13, 1, Article 5 (March 2017), 28 pages. https://doi.org/10.1145/3033273

[37] Wenhui Lyu and Goetz Graefe. 2025. Indexing Join Inputs for Fast Queries and Maintenance. In *Datenbanksysteme für Business, Technologie und Web (BTW 2025)*. Gesellschaft für Informatik, Bonn, Berlin, Germany, 71–93.

[38] Frank McSherry, Michael Isard, and Derek G Murray. 2013. Differential dataflow. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR)*. CIDR, Asilomar, CA, USA, 1–11.

[39] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theodore Vassilakis. 2011. Dremel: Interactive Analysis of Web-Scale Datasets. *Commun. ACM* 54 (2011), 114–123. http://cacm.acm.org/magazines/2011/6/108648-dremel-interactive-analysis-of-web-scale-datasets/fulltext

[40] Microsoft Corporation. 2025. Create Indexed Views (SQL Server). https://learn.microsoft.com/en-us/sql/relational-databases/views/create-indexed-views. Accessed: 2026-05-05.

[41] Microsoft Corporation. 2025. Merge Join Showplan Operator (SQL Server). https://learn.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms189961(v=sql.105). Accessed: 2026-05-05. Documents that the SQL Server merge-join operator consumes ordered input streams—typically supplied by an index scan or an upstream sort-aware operator—thereby reusing one operator's sort order downstream.

[42] Microsoft SQL Server. 2025. Create indexed views. https://learn.microsoft.com/en-us/sql/relational-databases/views/create-indexed-views?view=sql-server-ver17 Accessed: 2025-11-30.

[43] MongoDB. 2024. WiredTiger Storage Engine. https://www.mongodb.com/docs/manual/core/wiredtiger/ Accessed: 2026-02-26.

[44] Inderpal Singh Mumick, Dallan Quass, and Barinderpal Singh Mumick. 1997. Maintenance of data cubes and summary tables in a warehouse. *SIGMOD Rec.* 26, 2 (jun 1997), 100–111. https://doi.org/10.1145/253262.253277

[45] MySQL. 2024. Updatable and Insertable Views. https://dev.mysql.com/doc/refman/8.4/en/view-updatability.html Accessed: 2024-09-01.

[46] Milos Nikolic and Dan Olteanu. 2018. Incremental View Maintenance with Triple Lock Factorization Benefits. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 365–380. https://doi.org/10.1145/3183713.3183758

[47] Dan Olteanu and Maximilian Schleich. 2016. Factorized Databases. *SIGMOD Rec.* 45, 2 (Sept. 2016), 5–16. https://doi.org/10.1145/3003665.3003667

[48] Patrick O'Neil and Goetz Graefe. 1995. Multi-table joins through bitmapped join indices. *SIGMOD Rec.* 24, 3 (Sept. 1995), 8–11. https://doi.org/10.1145/211990.212001

[49] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-tree (LSM-Tree). *Acta Inf.* 33, 4 (1996), 351–385.

[50] Oracle. 2024. Refreshing Materialized Views. https://docs.oracle.com/en/database/oracle/oracle-database/23/dwhsg/refreshing-materialized-views.html Accessed: 2024-09-01.

[51] Oracle Corporation. 2025. Oracle Database 26c Data Warehousing Guide: Basic Materialized Views. https://docs.oracle.com/en/database/oracle/oracle-database/26/dwhsg/basic-materialized-views.html. Accessed: 2026-05-05.

[52] Oracle Corporation. 2025. Oracle Database SQL Tuning Guide: Joins (Sort Merge Join). https://docs.oracle.com/en/database/oracle/oracle-database/21/tgsql/joins.html. Accessed: 2026-05-05. Describes how Oracle's optimizer exploits already-sorted index access paths to skip the sort phase of a sort-merge join.

[53] PostgreSQL. 2023. Incremental View Maintenance. https://wiki.postgresql.org/wiki/Incremental_View_Maintenance#Outer_Join_Support Section: Outer Join Support, Accessed: 2024-05-17.

[54] Alice Rey, Michael Freitag, and Thomas Neumann. 2023. Seamless Integration of Parquet Files into Data Processing. In *Datenbanksysteme für Business, Technologie und Web (BTW 2023) (Lecture Notes in Informatics (LNI))*, Birgitta König-Ries et al. (Eds.), Vol. P-331. Gesellschaft für Informatik, Bonn, Dresden, Germany, 235–258.

[55] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 23–34. https://doi.org/10.1145/582094.582109

[56] David Simmen, Eugene Shekita, and Timothy Malkemus. 1996. Fundamental Techniques for Order Optimization. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. ACM, Montreal, Quebec, Canada, 57–67.

[57] Christoforos Svingos, Andre Hernich, Hinnerk Gildhoff, Yannis Papakonstantinou, and Yannis Ioannidis. 2023. Foreign Keys Open the Door for Faster Incremental View Maintenance. *Proc. ACM Manag. Data* 1, 1, Article 40 (May 2023), 25 pages. https://doi.org/10.1145/3588720

[58] Dixin Tang, Zechao Shang, Aaron J. Elmore, Sanjay Krishnan, and Michael J. Franklin. 2019. Intermittent query processing. *Proc. VLDB Endow.* 12, 11 (July 2019), 1427–1441. https://doi.org/10.14778/3342263.3342278

[59] The PostgreSQL Global Development Group. 2025. PostgreSQL Source Code: `src/backend/optimizer/path/pathkeys.c`. https://github.com/postgres/postgres/blob/master/src/backend/optimizer/path/pathkeys.c. Accessed: 2026-05-05. Implements pathkeys, PostgreSQL's representation of interesting orderings used to plan merge joins, sort-based grouping, and order-by/distinct elimination.

[60] Transaction Processing Performance Council. 2017. *TPC BENCHMARK H (Decision Support) Standard Specification, Revision 2.17.3.* Technical Report. Transaction Processing Performance Council (TPC), San Francisco, CA, USA. http://www.tpc.org/tpch/ Revision 2.17.3.

[61] Tolga Urhan and Michael J. Franklin. 2000. XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Engineering Bulletin* 23, 2 (2000), 27–36.

[62] Patrick Valduriez. 1987. Join indices. *ACM Trans. Database Syst.* 12, 2 (June 1987), 218–246. https://doi.org/10.1145/22952.22955

[63] A. N. Wilschut and P. M. G. Apers. 1991. Dataflow Query Execution in a Parallel Main-Memory Environment. In *Proceedings of the 1st International Conference on Parallel and Distributed Information Systems*. IEEE, Miami Beach, FL, USA, 68–77.

[64] Christian Winter, Tobias Schmidt, Thomas Neumann, and Alfons Kemper. 2020. Meet me halfway: split maintenance of continuous views. *Proc. VLDB Endow.* 13, 12 (July 2020), 2620–2633. https://doi.org/10.14778/3407790.3407849

[65] Yue Zhuge, Héctor García-Molina, Joachim Hammer, and Jennifer Widom. 1995. View maintenance in a warehousing environment. *SIGMOD Rec.* 24, 2 (may 1995), 316–327. https://doi.org/10.1145/568271.223848