PAT: Towards Transaction Routing with Page Affinity in Shared-Cache Databases

Zhongqin Tan[1], Haoyuan Zhang[1], Yanfeng Zhang[1] , Zeshun Peng[1],
Weixing Zhou[1], Jinyu Zhang[2], Yang Ren[2], Guoliang Li[3], Ge Yu[1]
[1]*Northeastern University, China;* [2]*Huawei Company, China;* [3]*Tsinghua University, China*
{tanzq1, zhanghy46}@mails.neu.edu.cn, {pengzs, zhouwx}@stumail.neu.edu.cn, {zhangyf, yuge}@mail.neu.edu.cn,
{zhangjinyu.zhang, renyang1}@huawei.com, liguoliang@tsinghua.edu.cn

***Abstract*—Shared-cache architectures decouple compute from storage and employ local caches in compute nodes to reduce the latency of accessing shared storage, achieving high availability and elasticity. However, this design suffers from local cache misses and cache coherence overhead. Transaction routing has been widely used to mitigate these issues by routing transactions that access the same data to the same nodes, improving cache locality. Most existing routing approaches rely on row affinity, i.e., routing transactions that access the same set of rows to the same nodes. Since shared-cache databases typically maintain distributed cache coherence at the page level, this mismatch can cause redundant coherence traffic and degrade performance.**

In this paper, we present `PAT`, a shared-cache database system with page affinity-based routing, which routes transactions that access frequently co-accessed pages to the same compute node, reducing local cache misses and cache coherence overhead. Since SQL does not reveal which pages will be accessed before execution, `PAT` abstracts pages using key ranges to enable page affinity-based routing. This is based on the ordering property of widely used clustered indexes. Moreover, page updates may cause key ranges to become misaligned with pages, leading to significant cache coherence overhead. To address this issue, we introduce the route-aware page reorganization mechanism. Experiments show that `PAT` achieves 1.03×-14.36× higher throughput than state-of-the-art approaches under TPC-C and YCSB.

***Index Terms*—Transaction Routing, Shared-Cache Database, Data Partitioning, Page Reorganization.**

I. Introduction

The shared-cache architecture has emerged as a promising design for cloud-native databases due to its availability and elasticity [1]–[9]. It decouples compute from storage, enabling independent and elastic scaling of resources. Each compute node caches frequently accessed pages to accelerate subsequent transactions. With high-speed interconnects such as RDMA [10], [11] and CXL [12], these local caches are combined to form a coherent, distributed shared cache. This design reduces data access latency by allowing compute nodes to fetch cached pages directly from other compute nodes rather than from the slower shared storage.

Shared-cache databases face expensive local cache misses and cache coherence costs. Compute nodes must load all relevant pages into their local cache before executing a transaction [2], [3], [6]. When required pages are not cached locally, expensive remote accesses are incurred. Due to the orders-of-magnitude latency gap between accessing remote nodes (including remote caches and storage nodes) and local caches, such local cache misses can lead to substantial transaction latency [1], [13]. Second, when multiple compute nodes cache the same pages, updates to any replica require synchronization among page replicas on all nodes. This update blocks concurrent transactions accessing that page on another node to ensure consistency, resulting in increased transaction latency and degraded performance [3], [5].

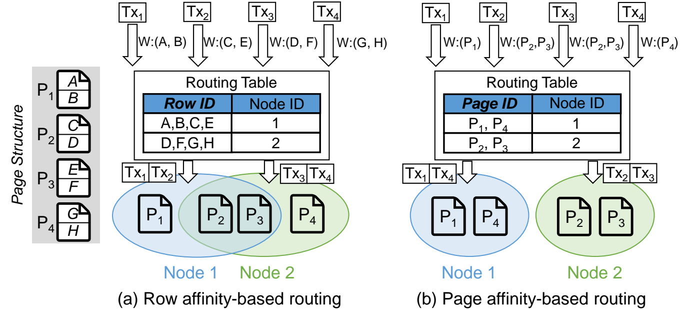


Fig. 1: Comparison of different routing approaches. Four transactions are routed based on different routing approaches. The routing table represents data affinity. Frequently co-accessed data has the same Node ID, which means that the transactions accessing them are routed to the same node.

Transaction routing, which routes transactions that access the same data to a single node, is widely used to improve cache locality and minimize the overhead of cache coherence. Many existing routing methods [14]–[35] rely on *row affinity*, i.e., the co-access frequency between rows. They route transactions that frequently access the same set of *rows* to the same nodes. This causes a page to be accessed by multiple nodes when the rows it contains are updated by transactions routed to different nodes. As shown in Figure 1a, based on the accessed row ID and the routing table, transactions Tx_1 and Tx_2 are routed to node 1, while Tx_3 and Tx_4 are routed to node 2. However, the rows updated by Tx_2 and Tx_3 are stored in the same pages P_2 and P_3. As a result, P_2 and P_3 will be updated by both nodes, resulting in cache coherence overhead and degrading performance. The reason can be attributed to the mismatch of *row affinity-based routing* and *page-based data organization*, which brings a fundamental challenge to row affinity-based routing. Therefore, it is essential to route transactions based on page affinity instead of row affinity.

Zhongqin and Haoyuan contributed equally.

To this end, we propose PAT, a shared-cache database system that routes transactions based on ***page affinity***. As shown in Figure 1b, with page affinity-based routing, Tx_1 and Tx_4 are routed to node 1, while Tx_2 and Tx_3 are routed to node 2. Compared to the row affinity-based routing, this reduces cache coherence overhead on P_2 and P_3. While PAT improves cache locality and reduces coherence overhead, it introduces several challenges.

First, it is difficult to determine the set of pages a transaction will access before executing. SQL statements typically specify only table names and predicates, without revealing the pages that will be accessed until execution. To address this challenge, PAT abstracts pages through key ranges based on the ordering property of clustered indexes [36], [37]. Since clustered indexes store rows in primary key order, rows represented by a given key range are typically located in a contiguous sequence of pages. As a result, a key range typically maps to a set of pages. This abstraction eliminates the need to identify specific pages during routing, while enabling page affinity to be modeled through key range affinity.

Second, page affinity evolves due to page updates and workload shifts. Operations such as page splits and merges can redistribute rows across pages, changing previously observed page affinity. Similarly, changes in transaction access patterns can also change page affinity, as transactions may begin to access different sets of pages over time [38]. To address this challenge, PAT models transactions as graph updates and constructs a key range co-access graph over a period of time to capture the most recent key range affinity. By periodically partitioning this graph, PAT identifies frequently co-accessed ranges and routes the transactions that access them to the same node, thereby adapting to changing page affinity.

In addition, static key ranges cannot perfectly align with pages under page updates. Consequently, a page may still be accessed by multiple nodes, incurring cache-coherence overhead, especially for hot pages. To mitigate this, we propose *route-aware page reorganization*, which periodically reorganizes hot pages to align them with key ranges, thereby reducing cache coherence overhead.

In summary, we make the following contributions.

- We propose page affinity-based transaction routing for shared-cache databases. By abstracting pages with key ranges and capturing their affinity in a key range co-access graph, our approach groups frequently co-accessed ranges using graph partitioning. This enables page affinity-based routing that reduces both local cache misses and cache coherence overhead.
- We propose route-aware page reorganization to address the misalignment between key ranges and pages. This misalignment often causes a page to be accessed by multiple nodes, resulting in cache coherence overhead. To mitigate this issue, PAT reorganizes page layouts so that they are aligned with key ranges.
- We design and implement a shared-cache database prototype, PAT, which routes transactions based on page affinity to reduce both local cache misses and cache coherence overhead. Experimental results show that PAT achieves $1.03\times$-$15.53\times$ performance improvement over state-of-the-art routing schemes.

II. Related Work

Table I summarizes existing partitioning/routing methods.

A. Architectures and Partitioning/Routing

In shared-nothing databases, data partitioning and transaction routing are tightly coupled. Data is partitioned across multiple nodes, and transactions are routed to the nodes that host the required data. When a transaction spans multiple partitions (i.e., distributed transactions), the system must coordinate across nodes (e.g., 2PC) to ensure atomicity and consistency. As a result, most shared-nothing systems aim to minimize distributed transactions through careful partitioning [22], [24]–[28], [34], [39]. Moreover, T-Part [29] and Hermes [41] improve performance through routing in deterministic databases.

On the other hand, in shared-cache databases, transaction routing is decoupled from data partitioning due to the separation of the compute and storage [42]. Compute nodes are responsible for processing transactions. The storage layer manages data partitioning, which is transparent to the compute nodes. A transaction can be routed to any compute node without relying on the storage layer's data partitioning. When pages that are accessed by a transaction are not cached by the compute node, these pages are fetched from other compute nodes or storage nodes. This decoupling enables the system to exploit transaction routing to improve cache locality and reduce cache coherence overhead across nodes.

B. Affinity Granularity and Technique

Affinity Granularity. Most existing work leverages row affinity, since they explicitly co-locate frequently co-accessed rows to minimize distributed transactions [22], [24]–[29], [34], [41]. Particularly, T-Part models dependencies between transactions through accessed rows. As a result, it makes routing decisions based on row affinity. However, shared-cache databases organize data at the page level. Even if two transactions access disjoint rows, they may still contend on the same page across nodes, triggering page-level cache coherence overhead. In contrast, E-Store [40] ignores data affinity and partitions hot and cold data independently across nodes, resulting in numerous distributed transactions. Accordion [39] co-locates data partitions that are frequently accessed together to reduce distributed transactions and preserve partition affinity under elastic scaling. However, partition is coarse-grained and cannot capture the rapidly changing page affinity.

Technique. There have been many graph-based methods [22], [24], [25], [28], [29] that model the workload as a graph to capture its characteristics. For example, Schism [22], SWORD [24], and Clay [28] model co-access relationships among rows through a co-access graph, where vertices represent rows and edges denote that two rows are co-accessed by

TABLE I: Summary of Existing Partitioning and Routing Methods.

System	Architecture	Partitioning / Routing	Affinity Granularity	Technique		
				Type	Vertex	Edge
Horticulture [26]	Shared-nothing	Partitioning	Row-level	Large-neighborhood search	-	-
JECB [27]	Shared-nothing	Partitioning	Row-level	Join-extension	-	-
Accordion [39]	Shared-nothing	Partitioning	Partition-level	MILP	-	-
E-Store [40]	Shared-nothing	Partitioning	-	Two-tier data placement	-	-
G-Store [34]	Shared-nothing	Partitioning	Row-level	Key group	-	-
Hermes [41]	Shared-nothing	Routing + Partitioning	Row-level	Prescient transaction routing	-	-
Schism [22]	Shared-nothing	Partitioning	Row-level	Graph-based	Row	Co-access
SWORD [24]	Shared-nothing	Partitioning	Row-level	Graph-based	Row	Co-access
Chiller [25]	Shared-nothing	Partitioning	Row-level	Graph-based	Row/Txn.	Contention
Clay [28]	Shared-nothing	Partitioning	Row-level	Graph-based	Row	Co-access
T-Part [29]	Shared-nothing	Routing	Row-level	Graph-based	Txn.	Row Dep.
PAT (ours)	**Shared-cache**	Routing	**Page-level**	Graph-based	**Key Range**	Co-access

the same transaction. Chiller [25] represents each transaction as a star, where the center vertex corresponds to the transaction, and the outer vertices represent the rows it accesses. Edges indicate contention on rows between transactions. T-Part relies on deterministic concurrency control, where a transaction's read/write set must be known before execution. This enables T-Part to build a graph that models inter-transaction dependencies based on a batch of transactions. T-Part then routes transactions based on the graph partitioning results. As a result, the graph must be recomputed for every batch and is highly dynamic, so routing decisions are transient and cannot directly leverage longer-term locality patterns. These methods transform data partitioning or transaction routing into a graph partitioning problem.

In contrast to graph-based methods, several methods [26], [27], [34], [39]–[41] explore alternative techniques. Horticulture [26] employs a large-neighborhood search algorithm to find efficient partitioning schemes. JECB [27] leverages key-foreign key relationships to enlarge the search space. It then uses a divide-and-conquer strategy, the database schema, and the transaction SQL code to search for a good partitioning solution. Accordion [39] uses server capacity estimation and mixed integer linear programming (MILP) to co-locate predefined partitions. E-Store [40] adopts a two-tier placement strategy: cold data is partitioned into large chunks, while hot data is partitioned at the row level. G-Store [34] dynamically groups rows and assigns each group to a node. Hermes [41] proposes a prescient transaction routing algorithm that uses heuristics to optimize routing and data partitioning.

III. Preliminaries

In this section, we first introduce the overall architecture and workflow of shared-cache databases, which will be used as the baseline. We then review clustered indexes and analyze page affinity to motivate our work.

A. Page-Level Shared-Cache Database

Shared-cache architecture has been widely used in many DBMSs, such as Oracle RAC [6] and PolarDB-MP [3]. In this architecture, data pages are persisted in shared storage and cached in the local memory of compute nodes.

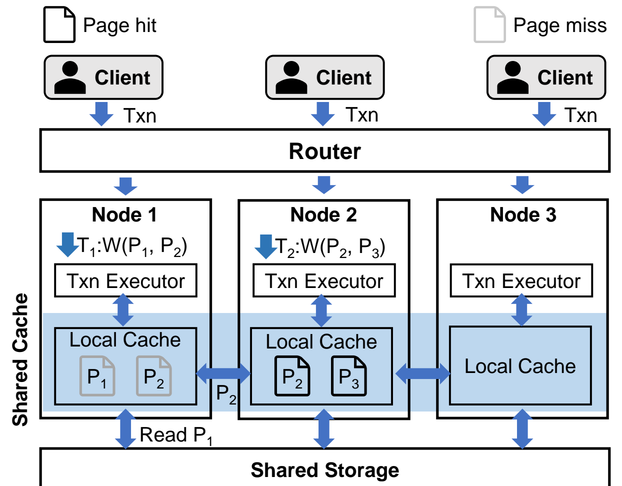


Fig. 2: The workflow of shared-cache databases.

Workflow. As Figure 2 shows, given a transaction, the router extracts the keys from the SQL predicates and looks them up in a routing table, which consists of multiple <key, node ID> pairs. The transaction is then dispatched to a compute node, which fetches all required pages to execute it. Note that if the keys of a transaction are mapped to multiple nodes, the router selects the node that appears most frequently in the lookup results and dispatches the transaction to that node. Node 1 executes transaction T_1, which writes pages P_1 and P_2. Since both pages are missing in the local cache of node 1, node 1 obtains P_2 from node 2, where it is cached, and fetches P_1 from shared storage, as it is not cached on any compute node.

Page-Level Cache Coherence. Since a page can be modified on multiple nodes, updates can lead to stale or inconsistent pages. To maintain cache coherence, shared-cache databases rely on the page invalidation mechanism and distributed locking [1], [3], [6], [43]. When a page is modified on a node, its copies on other compute nodes are invalidated to prevent accessing stale data. Additionally, distributed locks are utilized to coordinate access to cached pages. They enforce exclusive access for writes, ensuring that only one node can modify a page at any given time. This prevents conflicting updates and maintains data consistency.

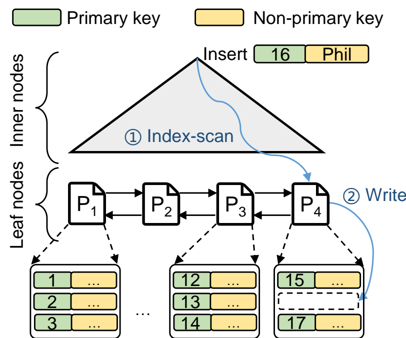


Fig. 3: The structure of the clustered index.

B. Clustered Index

To present the insight that key ranges can be used to model pages, we introduce the clustered index, also known as index-organized tables (IOTs) [36], [37], [44], [45], which have been widely adopted by InnoDB, Oracle [2], and Microsoft SQL Server [46]. A clustered index organizes rows in primary key order and stores them in the index structure, which is typically a B⁺-tree. The inner nodes are built based on the primary key, while the leaf nodes store entire rows, including both primary and non-primary key columns.

Figure 3 illustrates the structure of the clustered index. The rows are clustered in primary key order. For example, page P_3 contains rows with primary keys from 12 to 14, and P_4 contains rows from 15 to 17. To maintain this clustering, insertions begin with a primary key index scan to locate the appropriate page. The new row with primary key 16 is inserted at P_4, and subsequent rows are shifted to preserve the sorted order. This ordering ensures that rows within the same key range are stored contiguously across one or a few pages, providing an opportunity to model pages using key ranges.

C. Page Affinity

We define page affinity as the frequency with which pages are co-accessed by a transaction. Pages that are frequently co-accessed exhibit strong affinity. To analyze page affinity and its potential performance benefits for page affinity-based routing, we conduct experiments using the YCSB-A* [47] and TPC-C [48]. In YCSB-A*, the database is divided into five partitions. Each transaction only accesses rows in one partition. In TPC-C, we initialize the database with five warehouses. Most transactions access data from a single warehouse. The other experimental settings are detailed in Section VII.

To demonstrate the behavior of page affinity under different workloads, we analyze the page co-access frequency among the 50 most frequently accessed pages (i.e., hot pages) under YCSB-A* and TPC-C. The results are shown in Figure 4a. In YCSB-A*, we observe five distinct page groups with strong affinity, as pages within the same partition are frequently co-accessed. In contrast, in TPC-C, pages from different groups are also co-accessed. This can be attributed to two reasons. First, several TPC-C transactions access data from multiple warehouses, so pages from different groups are co-accessed.

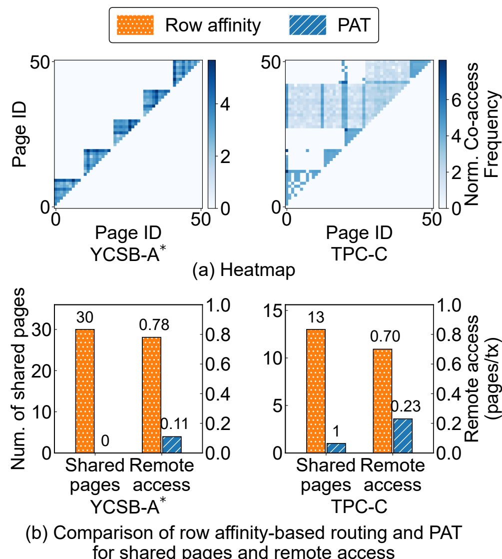


Fig. 4: (a) Page co-access frequency under different workloads. Color intensity shows the normalized co-access frequency between pages. Darker colors indicate higher co-access. (b) Row-affinity routing vs. page-affinity routing (PAT). We compare the number of shared pages and remote page accesses.

Second, since the page layout is workload-agnostic, a single page may store rows from different warehouses, leading to co-accesses across groups.

To evaluate the effectiveness of page affinity-based routing, we compare row affinity-based and page affinity-based routing (PAT) in terms of the number of shared pages and the remote page accesses. The results are shown in Figure 4b. Here, a shared page refers to a page accessed by multiple nodes. First, row affinity-based routing results in more shared pages compared to PAT. This is because it cannot capture page-level co-access patterns. Therefore, frequently co-accessed pages are not cached together. Moreover, PAT can reduce remote page accesses by 3.04×-7.09×. This highlights the potential benefits of page-affinity routing in improving cache locality and reducing cross-node communication.

IV. Page Affinity-based Routing

In this section, we give a detailed discussion on the page affinity-based transaction routing approach. First, we use key ranges as an intermediate abstraction for pages, which allows us to model page affinity through range affinity (Section IV-A). Second, PAT builds a key range co-access graph to capture the affinity between key ranges (Section IV-B). Finally, we present an incremental partitioning method (Section IV-C).

A. Modeling Page Affinity via Key Ranges

Key Ranges. Key ranges are defined by dividing the key space into equal-sized intervals. When rows are stored in order, the rows represented by a key range are stored on a set of pages. As shown in Figure 5a, the key space is divided into two ranges. Each key range can represent a set of rows

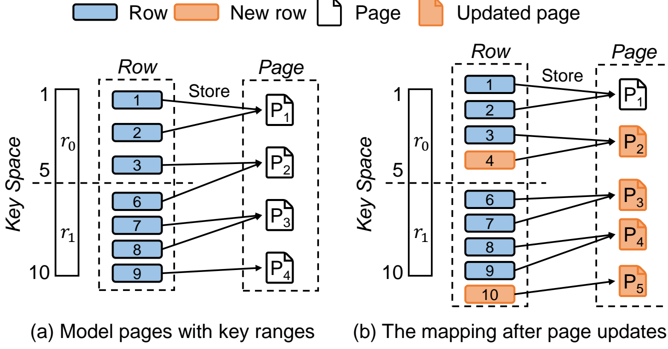


Fig. 5: An illustration of modeling pages with key ranges. Each key range maps to a group of pages, since rows are stored in primary-key order in the clustered index.

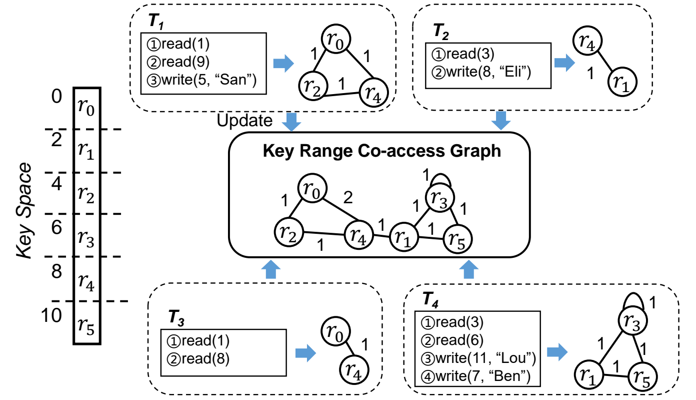


Fig. 6: A key range co-access graph derived from sampled transactions.

with continuous keys. The range r_0 is associated with rows $\{1, 2, 3\}$, as the primary keys of these rows belong to r_0. Rows $\{1, 2, 3\}$ are stored in pages P_1 and P_2 respectively. Thus, the range r_0 can be used to model P_1 and P_2. Similarly, range r_1 maps to P_2, P_3 and P_4. Notably, key ranges do not always align perfectly with pages. For example, P_2 stores rows $\{3, 6\}$, which are from both ranges. This associates P_2 with multiple key ranges, which we will detail in Section V-A. The affinity between r_0 and r_1 can be used to model the affinity between page sets $\{P_1, P_2\}$ and $\{P_2, P_3, P_4\}$.

Address Page Updates. Page updates are common in OLTP workloads. We now show that key ranges can consistently model the page affinity after such updates. As shown in Figure 5b, after inserting row 4, which is associated with r_0, pages P_2, P_3, and P_4 are updated to maintain primary key order (see Section III-B). Meanwhile, a new row with primary key 10 is inserted into range r_1. Since all existing pages are full, a new page P_5 is allocated to store it. After these updates, rows associated with r_0 are distributed across pages P_1 and P_2, so r_0 still maps to the page set $\{P_1, P_2\}$. For r_1, the corresponding pages become $\{P_3, P_4, P_5\}$. As a result, the affinity between r_0 and r_1 models the affinity between these two new page sets. It is important to note that the affinity between key ranges evolves with row insertions, deletions, and workload shifts. For example, inserting row 4 introduces a new data access pattern for range r_0, since r_0 may now be co-accessed with other key ranges. To capture these changes efficiently, we maintain range affinity through incremental updates (see Section IV-C).

Key Range Sizing and Construction. To make the range abstraction effective, PAT must (i) choose an appropriate range size (i.e., the number of rows) close to one or a few pages, and (ii) construct and maintain ranges under page updates. PAT automatically sets the key range size (i.e., the number of rows) based on database statistics. To model pages with key ranges, PAT sets the key range size so that each range covers roughly the same amount of data and is close to one or a few pages. Specifically, it estimates the range size using the average row size and the page size:

$$range_size = \alpha \left\lceil \frac{page_size}{avg_row_size} \right\rceil . \quad (1)$$

Here, α controls the expected number of pages covered by a key range, and can be tuned to balance routing granularity and maintenance cost.

Given the target range size S, PAT partitions the table into consecutive key ranges offline, each containing about S rows. As runtime updates may introduce data skew across ranges, making them less efficient for modeling pages, storage nodes maintain a lightweight row counter for each range. When a range grows beyond a split threshold (e.g., $2S$), the node notifies the router to split it into two ranges. Conversely, if the combined size of two adjacent ranges is smaller than a merge threshold (e.g., S), they are merged. Notably, PAT uses different split and merge thresholds to avoid oscillation. This method keeps each range size below a bound (e.g., at most $2S$ in the example).

B. Key Range Co-Access Graph

As discussed above, key ranges can be used as an abstraction of pages in clustered indexes. Based on this observation, we build a key range co-access graph to capture the affinity between key ranges. This graph must meet two key requirements. First, it should effectively model the co-access patterns among key ranges. This enables the system to identify ranges that are frequently accessed together. Transactions that access these ranges can then be routed to the same nodes. Second, the graph must support online incremental updates, enabling it to capture changes in access patterns and accurately maintain the corresponding affinity between key ranges.

As shown in Figure 6, we model each transaction as a graph update. Each vertex represents a key range, and an edge connects two vertices if the corresponding ranges are co-accessed by the same transaction. For example, transaction T_1 accesses rows with primary keys $\{1, 5, 9\}$, which are mapped to key ranges $\{r_0, r_2, r_4\}$. Following the above method, T_1 is modeled as a graph update that consists of three vertices

$\{r_0, r_2, r_4\}$ and three edges $\{e_{0,2}, e_{0,4}, e_{2,4}\}$. Here, $e_{i,j}$ denotes the edge between ranges r_i and r_j. Similarly, T_2, T_3, and T_4 are transformed into graph updates. Notably, r_3 has a self-loop because r_3 is accessed twice in T_4. Finally, these graph updates incrementally update the key range co-access graph. The edge weight indicates the number of transactions that co-access the two key ranges.

Because transactions are modeled as graph updates, we can use a sliding window to capture the most recent access behavior. We maintain the key range co-access graph over the current window of the transaction stream. For each new transaction, we transform it into a graph update and apply it to the key range co-access graph. At the same time, transactions that fall out of the sliding window expire. When a transaction expires, we decrement the weights of the edges to which it contributed. If the weight of an edge drops to zero, the edge is removed from the graph.

In this example, we assign equal weight to reads and writes when computing edge weights. However, reads and writes incur different costs and thus should have different impacts on edge weights. Such differences can affect the effectiveness of the transaction routing algorithm. To quantify this effect, we experimentally tune the relative contributions of reads and writes in Section VII-D.

Vertex Weights for Load Balancing. To achieve load balancing across compute nodes, the graph partitioning, which will be described later, aims to minimize the total weight of cut edges while keeping the weight of each partition approximately equal. The weight of a partition is defined as the sum of the weights of its vertices. Therefore, an appropriate definition of vertex weight plays a crucial role in balancing both data size and load. We define the weight of each vertex as follows:

$$w_v = \alpha r_{\text{size}} + (1 - \alpha) r_{\text{load}}. \tag{2}$$

Here, w_v denotes the weight of vertex v, r_{size} represents the data size associated with the vertex, and r_{load} captures the runtime load derived from factors like access frequency, CPU usage, and I/O overhead. The parameter $\alpha \in [0, 1]$ controls the trade-off between balancing data size and load. The load can be obtained from existing work [49], [50].

C. Affinity Analysis by Partitioning

In the key range co-access graph, a higher edge weight indicates that the key ranges are co-accessed more frequently. To improve cache locality, transactions that access such key ranges should be routed to the same compute node. Because the page affinity is modeled as a weighted graph, we can apply graph partitioning algorithms [51]–[61] to group frequently co-accessed key ranges into the same partition. Each partition is then mapped to a compute node. The results are stored in the *routing table* maintained by the router. The routing table consists of multiple $<r_i, n_j>$pairs. Here, r_i represents the i-th key range and n_j denotes the j-th node.

Incremental Partitioning. As workloads shift and pages are updated, key-range affinity evolves. A naive solution is to repartition the co-access graph from scratch. However, this approach leads to substantial changes in the routing table, which in turn forces cached pages to be shuffled across compute nodes. To address this, we adopt an incremental graph partitioning strategy that continuously adapts to changes.

Unlike conventional incremental or streaming graph partitioning algorithms [53]–[55], our problem requires handling dynamic updates to both edges (i.e., co-access frequencies between key ranges) and vertices (i.e., insertions and deletions of key ranges). These updates may invalidate previous assignments and thus necessitate re-evaluating the placement of affected vertices. To effectively capture such changes, we develop an incremental partitioning algorithm based on Fennel [55], which assigns vertices to the partition with the most neighbors under the load constraints.

For each graph update, we identify the affected vertices, namely those whose incident edges have changed due to weight updates, insertions, or deletions. These vertices are first removed from their current partitions. Then, we re-evaluate the partitions of these vertices according to Equation 3. Equation 3 defines the gain score $\delta g(r, \mathcal{P}_i)$, which quantifies the benefit of assigning vertex r to partition $\mathcal{P}_i$. The vertex is reassigned to the partition with the highest score.

$$\delta g(r, \mathcal{P}_i) = \beta \sum_{u \in \mathcal{P}_i \cap N(r)} w(r, u) - (1 - \beta)\gamma |\mathcal{P}_i|^{\gamma - 1}. \tag{3}$$

Here, r denotes the key range being re-evaluated, and $\mathcal{P}_i$ denotes the i-th partition. $N(r)$ represents the neighbors of r, i.e., key ranges that are co-accessed with r. $w(r, u)$ denotes the edge weight between r and u. $\sum_{u \in \mathcal{P}_i \cap N(r)} w(r, u)$ quantifies the affinity between vertex r and partition $\mathcal{P}_i$, reflecting the benefits on cache locality when r is colocated with its frequently co-accessed neighbors. $|\mathcal{P}_i|$ denotes the load of partition $\mathcal{P}_i$, which is defined as the sum of vertex weights in that partition. $(1 - \beta)\gamma|\mathcal{P}_i|^{\gamma-1}$ represents the penalty of assigning r to partition $\mathcal{P}_i$. The parameters β and γ are tunable. Specifically, $\beta \in [0, 1]$ balances the importance between minimizing edge cuts (improving cache locality) and load balancing. γ controls the penalty of load imbalancing. Since each partitioning needs to traverse all updated vertices and their incident edges, the partitioning complexity is $O(|\Delta V| + |\Delta E|)$, where $|\Delta V|$ and $|\Delta E|$ denote the number of affected vertices and their incident edges, respectively.

D. Discussion

1) Scalability: To improve scalability, we have to address two issues: (i) the size of the graph and routing table, and (ii) the cost of graph partitioning.

`PAT` can reduce the size of the graph and routing table by adjusting the range size. Coarse-grained ranges reduce the number of ranges, which in turn reduces the routing table and graph size while amortizing maintenance overhead. However, they can also combine pages with different access behaviors into one range, thus reducing the accuracy of routing decisions.

Fine-grained ranges preserve page-level affinity and support more accurate routing, but they increase the overhead of maintaining routing metadata. Moreover, PAT can also adjust the number of transactions used for graph construction to reduce the size of the graph and routing table. PAT can further limit the size of the routing table based on a replacement strategy, such as LRU. When the number of key-value pairs exceeds the limit, it has to evict a key-value pair. Since many OLTP workloads only contain small portions of hot data [62], limiting the size of the routing table has a minimal impact on performance [25], [41]. Graph partitioning can be time-consuming and computation-intensive. To reduce the partitioning overhead, we can partition the graph in parallel [63], [64].

2) Elasticity: To handle dynamic cloud environments where the number of compute nodes changes [65], we decouple graph partitioning from placement. Specifically, we partition the key range co-access graph into more partitions than the available compute nodes. We then assign partitions evenly to nodes so that the number of partitions hosted on each node is balanced. When nodes are added or removed, each partition independently decides whether to migrate using a probabilistic rule. For example, when n nodes are added to a cluster with k existing nodes, a partition remaps to one of the new nodes with probability $p = \frac{n}{k+n}$. This yields an approximately uniform distribution of partitions over the $k+n$ nodes and preserves load balance.

3) Routing on Non-Primary Keys: In general, the routing key should not be limited to the clustered index key (typically the primary key), but should be extended to any key that appears in SQL predicates. However, under a clustered index, rows are physically ordered only by the primary key [3], [36], [45], which makes page affinity-based routing on non-primary keys ineffective. PAT handles predicates on non-clustered keys via a lightweight *secondary index* maintained at the router. A secondary index stores only a mapping from non-primary keys to primary keys. The router uses this mapping to rewrite non-primary key predicates into a set of primary keys, which are then used for routing. This design incurs secondary-index maintenance overhead at the router.

4) Handling Range Updates: Any range split/merge requires updating both the co-access graph and the routing table. To reduce update overhead, PAT updates the graph lazily. It replaces the original vertex with two new vertices that have no edges. PAT updates their edges/edge weights from the split time. The old vertex stops receiving new edges/edge weights and is gradually removed as the graph evolves. Meanwhile, the new ranges inherit the original range's routing node and are appended to the routing table.

E. Routing Transactions

PAT routes transactions based on the primary key. To extract the primary keys required for routing, the router analyzes the SQL statements of each transaction. Since the route of a transaction can be determined by analyzing its individual SQL statements, we illustrate the analysis process using a single SQL example. We take the TPC-C table schema as an example to demonstrate the approach. The ITEM table has a single-column primary key, i_id, while the STOCK table has composite primary keys, {s_w_id, s_i_id}. Let $\mathcal{PK}$ denote the primary key of the table and $\mathcal{K}$ represent the set of keys accessed in the SQL statement. Based on the columns that are being accessed, there are three cases.

- $\mathcal{K} = \mathcal{PK}$, i.e., the SQL statement accesses data via primary key columns. For example, the statement SELECT * FROM ITEM WHERE i_id = 100 accesses the ITEM table by its primary key i_id. In this case, the router directly uses the primary key for querying the routing table.
- $\mathcal{K} \subset \mathcal{PK}$: the table uses a composite primary key, but the SQL specifies only part of it. For example, SELECT * FROM STOCK WHERE s_w_id=100 provides only s_w_id for the primary key {s_w_id, s_i_id}. The router fills the missing keys (e.g., s_i_id) with the minimal value to form a complete key for routing.
- $\mathcal{K} \cap \mathcal{PK} = \emptyset$, i.e., the SQL does not reference any primary key columns. For example, the statement SELECT * FROM ITEM WHERE i_price = 2 filters on i_price, which is not part of the primary key. As the routing table only supports routing based on primary keys, the router disregards such statements.

For *range queries*, traversing all keys in the routing table to make routing decisions is impractical. To reduce routing overhead, PAT samples the accessed keys for routing.

Timely Load Balancing. A node may experience transient overload during runtime, which is caused by temporary hot spots or skewed access patterns. To address this overload, the system continuously monitors the load of each compute node. Before routing a query, the system estimates its load. If the designated node for a query is overloaded, PAT reroutes it to the next-best node according to the routing table.

V. Route-Aware Page Reorganization

In this section, we present a route-aware page reorganization mechanism to mitigate the cache coherence overhead caused by the misalignment between key ranges and pages.

A. Problem Statement

Although we model page affinity via key ranges, key ranges and pages are not perfectly aligned, as discussed in Section IV-A. In practice, a single page may span multiple key ranges, and conversely, a key range may cover parts of multiple pages. This misalignment can introduce cache coherence overhead. Specifically, when key ranges that map to the same page are mapped to different nodes, transactions accessing these key ranges may concurrently access the same page from multiple nodes. As a result, the shared cache must maintain coherence for that page across nodes, introducing additional overhead. This problem becomes more severe when the page is frequently accessed (i.e., a hot page), as increased coherence traffic can lead to performance degradation.

As shown in Figure 7, both key ranges r_1 and r_2 map to the same page P_2. Transaction T_1, which accesses r_1, is routed

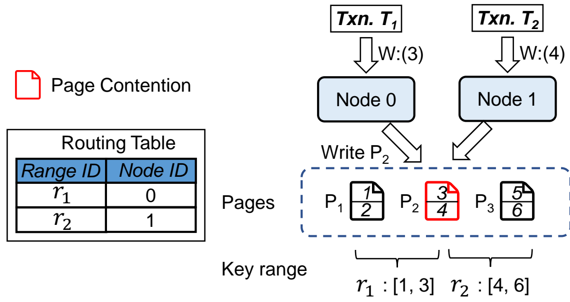


Fig. 7: An example of cache coherence overhead caused by routing. Since the key range and page are not aligned, page P_2 is mapped to both key ranges r_1 and r_2. P_2 can be accessed by both nodes concurrently with conflicting ownership (e.g., both nodes write P_2), resulting in cache coherence overhead.

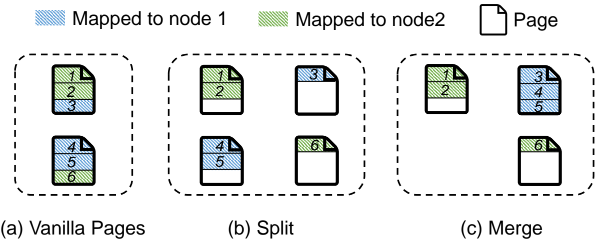


Fig. 8: Workflow of route-aware page reorganization. To preserve the primary-key order of the clustered index, only adjacent pages are eligible for merging.

to node 0, while transaction T_2, which accesses r_2, is routed to node 1. Since both transactions write rows on page P_2, the two nodes attempt to write the same page P_2 concurrently. To maintain page consistency, only one node is allowed to write the page, while the other transaction must be blocked. This coherence overhead not only increases transaction latency but also reduces concurrency. Such contention underscores the importance of aligning key ranges with pages.

B. Page Reorganization

Next, we analyze potential methods to align pages with key ranges and, based on their limitations, motivate our page reorganization approach.

1) Two Naive approaches: To align each key range with a page, two straightforward approaches may be considered. First, the size of key ranges can be dynamically adjusted so that key ranges align with pages. However, this method is prohibitively expensive because every page update requires adjusting the corresponding key ranges. Second, pages can be forced to contain rows from only one key range. The page that holds rows from multiple ranges should be split. However, data distributions are typically skewed, with some key ranges covering more rows than a single page can accommodate and others too few to utilize a page effectively. Moreover, frequent updates such as row insertions and deletions continually alter the contents of each page, breaking the one-to-one correspondence between pages and key ranges. In summary, both approaches fail because page layout is affected by skewed data distribution and ongoing updates. As a result, a single page may contain rows mapped to different partitions, which incurs additional cache coherence overhead.

2) Reorganize Pages: To mitigate this problem, our key insight is to introduce finer-grained key ranges. Specifically, by refining key ranges into *unit ranges* (i.e., key range size of one), each page can be abstracted as a consecutive set of key ranges. Although this does not guarantee strict one-to-one alignment between a page and a single key range, it ensures that any page can always be abstracted as a set of unit ranges. Based on this, we further propose route-aware page reorganization. Due to the high overhead of reorganization, this scheme is only applied to hot pages, as these pages are few in number and contribute the most to cache coherence overhead. The objective is to co-locate rows whose key ranges are mapped to the same node, as indicated by the routing table, within the same set of pages. To enable this, each compute node maintains a replica of the routing table. After reorganization, the resulting pages are primarily accessed by their designated routing node, thereby significantly reducing cache coherence overhead across nodes.

We employ a split-and-merge strategy. A page is split when it contains rows whose key ranges are routed to different nodes. This ensures that rows having distinct routes reside in separate pages. As shown in Figure 8b, rows 1 and 2 are separated from row 3, since their key ranges have different routes. Page splitting introduces page fragmentation, leading to additional storage overhead. To mitigate fragmentation, adjacent pages are merged when all of their rows share the same route. Only neighboring pages are considered for merging to preserve the primary-key order of the clustered index, and a page is merged only with its left neighbor to avoid deadlocks. This merging operation mitigates fragmentation while maintaining routing locality. As shown in Figure 8c, rows 3, 4, and 5 are co-located in a single page, since they share the same route. Finally, to improve efficiency, page split and merge operations are executed in parallel across compute nodes. Each node is responsible for reorganizing the pages that contain rows whose key ranges are mapped to it.

C. Discussion

1) Impact on Existing Page Splitting and Merging: Traditional page splitting occurs when a page reaches its capacity limit. This process does not interfere with the route-aware page organization, as the newly allocated pages continue to adhere to the same constraint. Specifically, the key ranges within the new allocated pages are mapped to the same node ID. Therefore, we focus only on modifying traditional page merging strategies in the context of route-aware page reorganization. Page merging typically occurs when the amount of data falls below a predefined threshold (e.g., 1KB). To ensure that merging does not compromise route-aware page reorganization, an additional verification step is introduced: the compute node checks whether the key ranges of both pages are mapped to the same node ID. Only pages with equal node IDs are allowed for merging.

2) ACID Guarantee: Compute nodes ensure the ACID properties of page reorganization using locking and write-ahead logging (WAL). Before splitting a page, a compute node acquires an exclusive lock from the lock manager. It then logs the original page content and the IDs of the new pages to the undo log before performing the split. In the event of a crash, the undo log enables recovery by rolling back the changes. The exclusive lock is released only after the reorganization of a page completes, preventing concurrent access or modification during the process. To minimize performance impact, each reorganization operates on one page at a time.

3) Hot Spots Detection: `PAT` identifies hot spots by monitoring the access frequency of each key range. It ranks ranges by frequency and marks the top-k ranges and their physical pages as hot. The parameter k determines the trade-off between page reorganization overhead and cache coherence overhead. A larger k marks more pages as hot, which can trigger excessive page reorganization and increase storage overhead. A smaller k reduces reorganization overhead, but leaves more hot pages misaligned with their ranges, increasing cache coherence overhead. By default, we set k to the hottest 5% of ranges, following a prior study where 99.94% of Wikipedia requests access only 5% of the database [62]. In addition, `PAT` derives the unit ranges from the primary-key predicates of queries that target the hot spot, i.e., each primary key is treated as a unit range. `PAT` replaces the original hot range with a set of unit ranges. To accommodate changing hot spots, such unit ranges are later merged back into their parent range once they are identified as cold.

4) Deferred Reorganization: To mitigate oscillating page reorganization under dynamic workloads and window-based graph processing, `PAT` employs a deferred reorganization method. First, `PAT` triggers reorganization only for pages that remain hot across several consecutive windows, avoiding unnecessary reorganization due to short-lived workload fluctuations. Second, `PAT` uses different split/merge rules to prevent immediate reversal of recent reorganizations under dynamic workloads. `PAT` splits a hot page when the routes of its unit ranges remain different over a period of time. In contrast, `PAT` merges under-filled *adjacent* hot pages only if the routes of their ranges have been the same for several windows.

5) Applicability and Limitations: `PAT` targets OLTP workloads that rely on clustered indexes, which are widely adopted in production DBMSs (e.g., MySQL [66], SQL Server [46], Oracle RAC [37], CockroachDB [67], TiDB [68]), and are common in applications such as e-commerce and banking. For example, an `Order` table clustered by `order_id` supports efficient order lookups on `order_id` and range queries over recent orders. In contrast, `PAT` offers limited benefits for OLAP-style workloads dominated by full table scans, where page-level cache coherence overhead is minimal. It is also unsuitable when clustered indexes are not feasible, such as workloads driven by predicates on secondary indexes or append-only log tables.

VI. IMPLEMENTATION

We build `PAT` on ScaleStore [1], a shared-cache prototype that decouples compute and storage nodes. ScaleStore ensures strong consistency with a distributed cache-coherence protocol that invalidates stale cached copies on updates. We extend ScaleStore with a router for transaction routing and a page reorganization module on each compute node.

Router. To update the routing table without blocking transaction routing, we use copy-on-write. We apply changes to a shadow copy and atomically swap it in after the update completes. In the graph, we assign each vertex a weight equal to the number of transactions accessing it, since key ranges have similar sizes in our experiments. We build the initial routing table by partitioning the key range co-access graph with METIS [51]. We incrementally update and re-partition the graph. We set $\alpha = 0$, $\beta = 2/3$, and $\gamma = 1.5$, following prior work [53], [55].

Page Reorganization. To guide page reorganization, each compute node keeps a local replica of the routing table and periodically synchronizes it with the router. Each node also runs a page reorganization module that reorganizes pages based on the routing table. To reduce cache coherence overhead from concurrent reorganizations, we allow only one node to reorganize pages. A background thread pulls pages from remote nodes, reorganizes them in the local cache, and flushes them to storage via LRU eviction.

VII. EVALUATION

A. Setup

Physical environment. We run our experiments on a 6-node Aliyun ECS cluster. Each node (ecs.c8i.12xlarge instance) is equipped with 24 CPU cores, 96GB DRAM, and runs Ubuntu 20.04 with the Linux kernel version 5.15.0-101-generic. We use eRDMA [10] for inter-node communication, with a network bandwidth of 25 Gbps between nodes. One node serves as a client and router. We set the number of clients to match the total number of worker threads in the cluster.

Workloads. We run TPC-C [48] and YCSB [47]. TPC-C simulates the activity of a wholesale supplier. We initialize four warehouses per node. We use unit key ranges for `WAREHOUSE` and `DISTRICT` because they are hot. For YCSB, we use a single table with an 8-byte integer key and a 128-byte random value. To make the workload transactional, we wrap operations within transactions and let each transaction contain 10 operations. We initially partition the table across nodes, with 10*M* rows per partition. Each transaction accesses only one partition. To demonstrate the effectiveness of page reorganization, we select 30 pages as hot pages. Each hot page mixes rows from multiple partitions, and every transaction accesses at least one row from these pages. Requests follow a Zipf distribution with $\theta = 0.99$. We evaluate two workloads: (1) YCSB-A* (50% reads, 50% updates) and (2) YCSB-B* (95% reads, 5% updates).

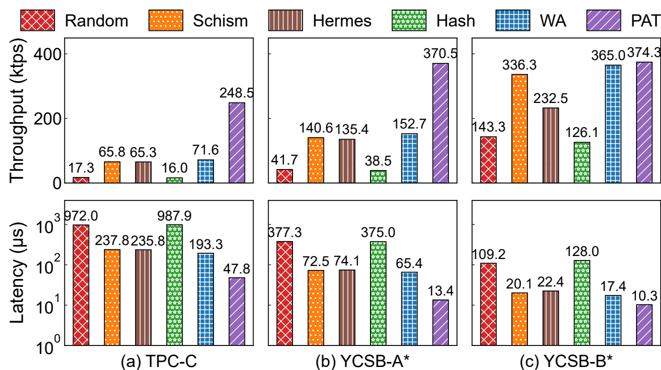


Fig. 9: Performance comparison.

Competitors. We compare five routing methods: `Random`, `Schism`, `Hash`, `Hermes`, and `Workload-Aware`.

- `Random`: Each transaction is routed to a random node. This method fails to leverage data affinity, resulting in significant local cache misses and cache coherence overhead.
- `Schism`: `Schism` [22] is an offline data partitioning scheme that represents a database as a graph, where vertices correspond to rows and edges capture the co-access frequency of rows within transactions. The graph is then partitioned using a graph partitioning algorithm. We adapted Schism to our baseline by using its graph partitioning results to guide transaction routing, rather than performing data partitioning in the storage layer.
- `Hash`: `Hash` analyzes the set of primary keys accessed by a transaction and uses a hash function to map the primary keys to node IDs. The transaction is routed to the node whose node ID appears the most. `Hash` can focus accesses to a row on a specific node as much as possible, thereby taking advantage of the local cache.
- `Hermes`: `Hermes` [41] is a *prescient* transaction routing method for deterministic DBMSs. It uses a heuristic algorithm to route a batch of transactions. Since it requires pre-known read/write sets, we adapted our baseline to expose read/write sets for each transaction ahead of execution and then invoke `Hermes` to route transactions.
- `Workload-Aware` (`WA`): `WA` routes transactions based on access patterns to improve cache locality. In TPC-C, it routes transactions that access the same warehouse to the same node, since transactions typically access data of a single warehouse. In YCSB, it routes transactions to the node that hosts the accessed partition.
- `PAT`: `PAT` produces a routing table offline and then incrementally updates the routing table every 20,000 transactions. At the same time, the compute nodes will periodically reorganize the pages according to the routing table.

B. Overall Performance

Figure 9 shows throughput and average latency for TPC-C, YCSB-A*, and YCSB-B* on a 5-node cluster. `PAT` consistently delivers the best performance, achieving the highest throughput and the lowest latency across all workloads. The key reason is that `PAT` jointly optimizes: (i) transaction routing based on *page affinity* and (ii) route-aware page organization, which together concentrate accesses to strongly co-accessed pages on the same node and thus reduce remote page accesses, and cache coherence traffic.

`PAT` improves throughput by 14.36×, 3.78×, 3.81×, 15.53×, and 3.47× over `Random`, `Schism`, `Hermes`, `Hash`, and `WA`, respectively, and achieves the lowest latency (47.8 μs). `Hermes` makes routing decisions for each transaction based on the accessed rows. While this can reduce remote reads at the row level, it does not explicitly optimize page-level co-location and introduces non-trivial routing overhead, making it less effective than `PAT`. `Schism` leverages row-level affinity, which is insufficient under page-granular caching and can still induce heavy coherence overhead. `Random` and `Hash` ignore page-level co-access and therefore suffer from frequent remote page accesses. Finally, `WA` benefits from workload-aware routing, but without route-aware page organization, it leaves avoidable page contention. For the read-heavy YCSB-B* workload (Figure 9c), all systems improve due to effective caching in the shared-cache architecture. However, other methods (e.g., `Random`) still incur significant cache coherence overhead, reducing performance.

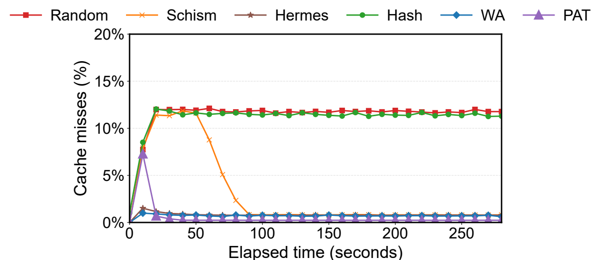
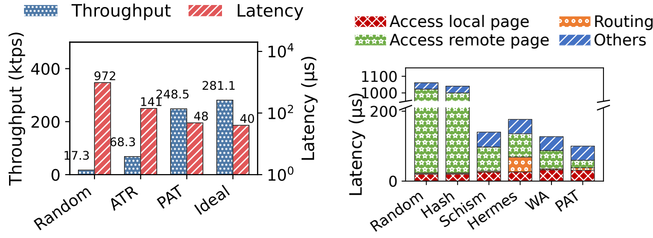


Fig. 10: The comparison of local cache misses under TPC-C.

C. Local Cache Misses

Figure 10 shows the local cache misses under the TPC-C workload. `PAT` substantially improves cache locality across all competitors. Specifically, it reduces local cache misses by 3.06×, 3.61×, 3.42×, 52.68×, and 50.65× compared to `WA`, `Schism`, `Hermes`, `Random`, and `Hash`, respectively. Both `Random` and `Hash` suffer from frequent local cache misses, as they cannot capture the co-access patterns of workloads. Although `Schism`, `Hermes`, and `WA` outperform `Random` and `Hash`, they still have many local cache misses. This is because they rely on row affinity to route transactions, which causes pages to be shared across multiple nodes. When a shared page is updated, its copies on other compute nodes are invalidated to maintain consistency. This introduces remote page accesses if a node subsequently accesses the invalidated page. Furthermore, since the page layout is workload-agnostic, a single page may contain rows from multiple warehouses, making it inevitably shared by multiple nodes and increasing local cache misses. When such pages are hot, this effect significantly amplifies local cache miss rates.



(a) Ablation results.

(b) Latency breakdown of New-Order.

Fig. 11: The performance breakdown under TPC-C.

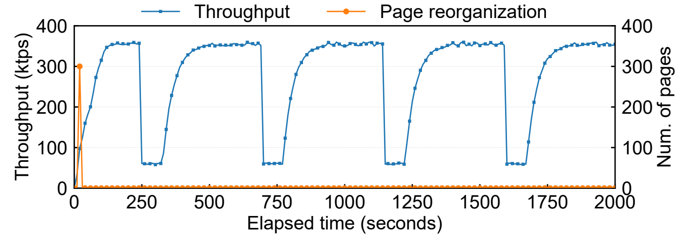


Fig. 12: Throughput when changing the workload.

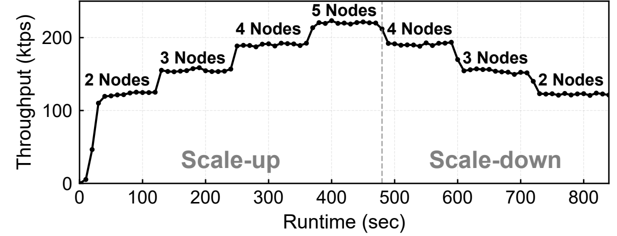


Fig. 13: Elasticity of decoupling graph partitioning and placement on compute nodes.

D. Performance Breakdown

Figure 11 shows the results of the performance breakdown. `ATR` employs only page affinity-based transaction routing, isolating the benefit of route-aware page reorganization. `Ideal` applies `WA` routing and page reorganization. It represents the best performance for the TPC-C workload.

As shown in Figure 11a, `ATR` achieves a 3.95× higher throughput compared to `Random` routing. This is because `ATR` can leverage page affinity for transaction routing, highlighting the effectiveness of identifying affinity at the granularity of key ranges. In addition, `PAT` outperforms `ATR` by 3.64× in throughput. This shows the substantial impact of cache coherence on system performance and demonstrates that route-aware page organization can effectively reduce cache coherence overhead, especially for hot pages. Finally, `PAT` achieves throughput comparable to `Ideal`.

To analyze how the transaction routing algorithm and route-aware page organization improve system performance, we also track the time spent on the New-Order transaction. Figure 11b shows the average latency breakdown. We make two observations. First, the routing latency of `PAT` is about 7 μs per transaction, accounting for about 7% of the total latency. This includes analyzing the transaction, looking up the routing table, and forwarding the transaction. Second, most of the latency in `PAT` is spent on local page accesses and interactions with other components, such as index traversal and SQL parsing. This is because page affinity-based routing significantly improves cache locality. Notably, `Hermes` spends about 24% of the end-to-end latency on routing, as it computes routing decisions at runtime, incurring non-trivial scheduling overhead. In contrast, the other methods only perform a lightweight lookup on a precomputed routing table.

We adjust the weights of read and write operations on edges to examine their effect on transaction routing. For example, when both vertices of an edge are read-only in the transaction, the edge weight is 1. If one of the vertices is written, the edge weight is 10. If both of the vertices are written, the edge weight is 100. We do not observe significant performance gains or losses in our results. This is because the TPC-C workload is well-partitioned, and the weights of reads and writes do not significantly affect partitioning results.

E. Workload Change

To evaluate the performance and reorganization overhead of `PAT` under dynamic workloads, we create a changing workload using YCSB-A*. We uniformly partition 50M rows into 25 partitions and distribute them on a 5-node cluster, with five partitions per node. Initially, transactions access only local partitions. To create a dynamic workload, we periodically change the partitions that each transaction accesses. For example, transaction 0 switches from accessing partitions {0–4} to {0–3, 5}. The workload switches between the two access patterns every 450 s. Figure 12 shows that throughput drops after each shift, then recovers as `PAT` incrementally updates key range co-access graph and refines the partitioning. Page reorganization adds little overhead after the first reorganization. This is because the asymmetric split/merge rules prevent recently split pages from being merged back immediately, reducing oscillating reorganization overhead.

F. Elasticity

To evaluate elasticity, we scale out compute nodes at runtime every 120 s until reaching five nodes. We then remove nodes one by one at the same interval. Figure 13 shows that decoupling graph partitioning from partition placement enables `PAT` to utilize the additional computing resources. The adaptation occurs within a few seconds after a new compute node has been added. The throughput scaling is not linear (e.g., between 2 and 4 nodes) due to page-level cache coherence. `PAT` routes transactions using range affinity to improve locality, but key ranges do not always align with physical pages, especially for cold pages that are not reorganized. As a result, some pages are still accessed by multiple nodes, which increases coherence traffic and limits scaling.

G. Page Reorganization Cost

Figure 14 shows the impact of page reorganization on remote page accesses and throughput under the TPC-C work-

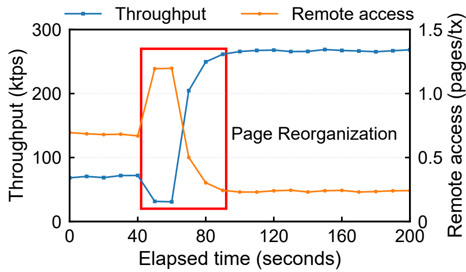


Fig. 14: Impact of page reorganization on remote accesses and throughput under TPC-C.

load. The page reorganization can improve throughput by 3.71×, while remote page accesses decrease by 2.96×. As discussed in Section VII-C, the reduction in remote accesses also reflects a decrease in cache coherence overhead. However, remote access increases during reorganization. This is because our current implementation performs reorganization on a single node, which must fetch pages from other nodes. To address this, we can partition the reorganization task across all compute nodes. Throughput also drops during reorganization due to increased remote accesses and the reduced number of worker threads available for transaction execution.

H. Varying the Key Range Size

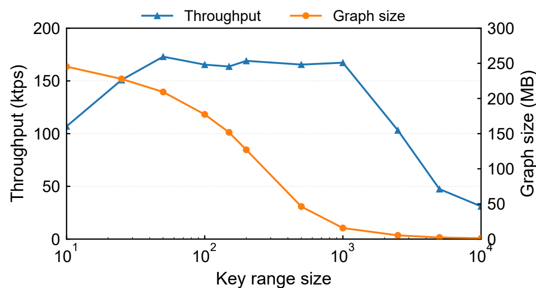


Fig. 15: Varying key range.

We evaluate the throughput and graph size of `PAT` under different key range sizes using the TPC-C benchmark on a 5-node cluster. The key range size is fixed at 1 for the `WAREHOUSE` and `DISTRICT` tables, while the remaining tables use varying key range sizes. Figure 15 shows that finer-grained key ranges produce larger graphs and routing tables, increasing memory consumption. In particular, when the key range size is 1, the router runs out of memory. Conversely, overly coarse key ranges reduce throughput, as a single key range may span many pages and limit the effectiveness of page affinity-based routing. Overall, a key range size between 50 and 1000 achieves the best throughput.

I. Scalability

We evaluate scalability by increasing the dataset size. Each node is equipped with a 16 GB cache. Figure 16 reports throughput under TPC-C and YCSB-A*. In TPC-C (Fig. 16a), the throughput of `WA`, `Schism`, and `Hermes` increases from 50 to 200 warehouses, as a larger database spreads accesses

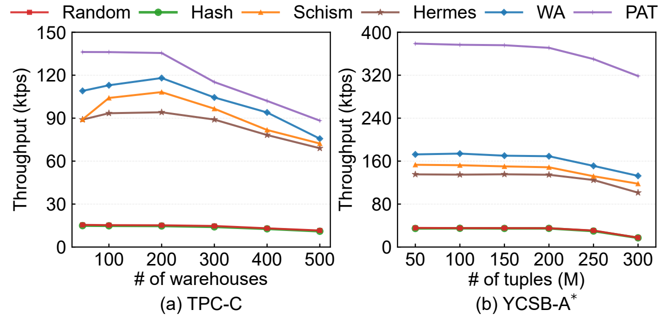


Fig. 16: The scaling performance.

to hot data across more pages and reduces cache coherence overhead. Throughput then declines once the dataset exceeds cache capacity (200 warehouses), making storage I/O the bottleneck. Nevertheless, `PAT` consistently outperforms other approaches by routing requests for each page to a designated node, thereby reducing accesses to shared storage and cache coherence overhead for hot pages. In YCSB-A* (Fig. 16b), hot data remain concentrated on a small set of pages. Thus, increasing the dataset size does not reduce cache-coherence overhead, and throughput remains stable until the dataset exceeds cache capacity (200*M*–300*M* rows), after which storage I/O becomes the bottleneck.

We evaluate the memory consumption of the co-access graph and routing table under TPC-C with 500 warehouses (∼100 GB). We set the key range size so that each range corresponds to roughly one page (4 KB). The co-access graph and routing table occupy ∼2.6 GB and ∼400 MB DRAM, respectively. Notably, the graph and routing table size do not scale with the database size. It is mainly determined by the key range size. Large ranges can significantly reduce the graph and routing table size. To reduce the DRAM consumption, `PAT` can cache only hot entries in memory and evict cold entries to shared storage using a replacement policy. These methods ensure that `PAT` is practically feasible for large databases.

VIII. Conclusion

This paper presents a shared-cache database prototype that routes transactions accessing frequently co-accessed pages to the same compute node. We observe that key ranges serve as an effective intermediate abstraction for pages in clustered indexes. Based on this observation, we design a page affinity–based routing algorithm that reduces page coherence overhead and local cache misses. To align key ranges with physical pages, we introduce route-aware page reorganization. Our experiments show that `PAT` improves throughput by 1.03×-14.36× compared with state-of-the-art methods.

Acknowledgment

This work was supported by the National Key R&D Program of China (2023YFB4503601), the National Natural Science Foundation of China (U2241212, 62461146205), the Distinguished Youth Foundation of Liaoning Province (2024021148-JH3/501), and Huawei. Yanfeng Zhang and Zeshun Peng are the corresponding authors.

AI-Generated Content Acknowledgement

Sections I and VII were polished with the assistance of OpenAI's ChatGPT models (GPT-4 and GPT-5). The AI tools were used solely for language polishing, grammar correction, and improving readability. The authors carefully reviewed and edited all technical contents, and are responsible for all analyses and conclusions.

References

[1] T. Ziegler, C. Binnig, and V. Leis, "Scalestore: A fast and cost-efficient storage engine using dram, nvme, and rdma," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 685–699.

[2] S. Chandrasekaran and R. Bamford, "Shared cache-the future of parallel databases," in *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*. IEEE, 2003, pp. 840–850.

[3] X. Yang, Y. Zhang, H. Chen, F. Li, B. Wang, J. Fang, C. Sun, and Y. Wang, "Polardb-mp: A multi-primary cloud-native database via disaggregated shared memory," in *Companion of the 2024 International Conference on Management of Data*, 2024, pp. 295–308.

[4] H. Dong, C. Zhang, G. Li, and H. Zhang, "Cloud-native databases: A survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 36, no. 12, pp. 7772–7791, 2024.

[5] T. Ziegler, P. A. Bernstein, V. Leis, and C. Binnig, "Is scalable oltp in the cloud a solved problem?" in *CIDR*, 2023.

[6] T. Lahiri, V. Srihari, W. Chan, N. Macnaughton, and S. Chandrasekaran, "Cache fusion: Extending shared-disk clusters with shared caches," in *VLDB*, vol. 1, 2001, pp. 683–686.

[7] E. Zamanian, C. Binnig, T. Kraska, and T. Harris, "The end of a myth: Distributed transactions can scale," *arXiv preprint arXiv:1607.00655*, 2016.

[8] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "{FaRM}: Fast remote memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 401–414.

[9] A. Depoutovitch, C. Chen, P.-A. Larson, J. Ng, S. Lin, G. Xiong, P. Lee, E. Boctor, S. Ren, L. Wu *et al.*, "Taurus mm: bringing multi-master to the cloud," *Proceedings of the VLDB Endowment*, vol. 16, no. 12, pp. 3488–3500, 2023.

[10] H. Cao, C. Xu, Y. Han, M. Lin, K. Shen, G. Wang, J. Li, X. Sun, R. He, L. You *et al.*, "An efficient cloud-based elastic rdma protocol for hpc applications," *CCF Transactions on High Performance Computing*, vol. 6, no. 1, pp. 45–53, 2024.

[11] F. Li, S. Das, M. Syamala, and V. R. Narasayya, "Accelerating relational databases by leveraging remote memory and rdma," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 355–370.

[12] Compute Express Link™, "About cxl™," https://www.computeexpresslink.org/about-cxl, 2022, accessed: 2024-10-24.

[13] B. Lu, K. Huang, C.-J. M. Liang, T. Wang, and E. Lo, "Dex: Scalable range indexing on disaggregated memory," *Proceedings of the VLDB Endowment*, vol. 17, no. 10, pp. 2603–2616, 2024.

[14] I. Pandis, P. Tözün, F. R. Johnson, and A. Ailamaki, "Plp: page latch-free shared-everything oltp," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 610–621, 2011.

[15] B. Hilprecht, C. Binnig, and U. Röhm, "Learning a partitioning advisor for cloud databases," in *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, 2020, pp. 143–157.

[16] X. Zhou, G. Li, J. Feng, L. Liu, and W. Guo, "Grep: A graph learning based database partitioning system," *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–24, 2023.

[17] P. S. Yu, A. Leff, and Y.-H. Lee, "On robust transaction routing and load sharing," *ACM Transactions on Database Systems (TODS)*, vol. 16, no. 3, pp. 476–512, 1991.

[18] U. Röhm, K. Böhm, and H.-J. Schek, "Olap query routing and physical design in a database cluster," in *Advances in Database Technology—EDBT 2000: 7th International Conference on Extending Database Technology Konstanz, Germany, March 27–31, 2000 Proceedings 7*. Springer, 2000, pp. 254–268.

[19] C. Nikolaou, A. Labrinidis, V. Bohn, D. Ferguson, M. Artavanis, C. Kloukinas, and M. Marazakis, "The impact of workload clustering on transaction routing," *FORTH, Institute of Computer Science, Technical Report*, vol. 238, 1998.

[20] S. Gançarski, H. Naacke, E. Pacitti, and P. Valduriez, "The leganet system: Freshness-aware transaction routing in a database cluster," *Information Systems*, vol. 32, no. 2, pp. 320–343, 2007.

[21] U. Rohm, K. Bohm, and H.-J. Schek, "Cache-aware query routing in a cluster of databases," in *Proceedings 17th International Conference on Data Engineering*. IEEE, 2001, pp. 641–650.

[22] C. Curino, E. P. C. Jones, Y. Zhang, and S. R. Madden, "Schism: a workload-driven approach to database replication and partitioning," *Proceedings of the VLDB Endowment*, 2010.

[23] E. Zamanian, C. Binnig, and A. Salama, "Locality-aware partitioning in parallel database systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 17–30.

[24] A. Quamar, K. A. Kumar, and A. Deshpande, "Sword: scalable workload-aware data placement for transactional workloads," in *Proceedings of the 16th international conference on extending database technology*, 2013, pp. 430–441.

[25] E. Zamanian, J. Shun, C. Binnig, and T. Kraska, "Chiller: Contention-centric transaction execution and data partitioning for modern networks," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 511–526.

[26] A. Pavlo, C. Curino, and S. Zdonik, "Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 61–72.

[27] K. Q. Tran, J. F. Naughton, B. Sundarmurthy, and D. Tsirogiannis, "Jecb: A join-extension, code-based approach to oltp data partitioning," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014, pp. 39–50.

[28] M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker, "Clay: Fine-grained adaptive partitioning for general database schemas," *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 445–456, 2016.

[29] S.-H. Wu, T.-Y. Feng, M.-K. Liao, S.-K. Pi, and Y.-S. Lin, "T-part: Partitioning of transactions for forward-pushing in deterministic database systems," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1553–1565.

[30] U. Cubukcu, O. Erdogan, S. Pathak, S. Sannakkayala, and M. Slot, "Citus: Distributed postgresql for data-intensive applications," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2490–2502.

[31] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[32] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang *et al.*, "The snowflake elastic data warehouse," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 215–226.

[33] Q. Lin, P. Chang, G. Chen, B. C. Ooi, K.-L. Tan, and Z. Wang, "Towards a non-2pc transaction management in distributed database systems," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1659–1674.

[34] S. Das, D. Agrawal, and A. El Abbadi, "G-store: a scalable data store for transactional multi key access in the cloud," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 163–174.

[35] M. Liroz-Gistau, R. Akbarinia, E. Pacitti, F. Porto, and P. Valduriez, "Dynamic workload-based partitioning algorithms for continuously growing databases," in *Transactions on Large-Scale Data-and Knowledge-Centered Systems XII*. Springer, 2013, pp. 105–128.

[36] D. Kuhn, S. R. Alapati, B. Padfield, D. Kuhn, S. R. Alapati, and B. Padfield, "Index-organized tables," *Expert Oracle Indexing and Access Paths: Maximum Performance for Your Database*, pp. 77–91, 2016.

[37] J. Srinivasan, S. Das, C. Freiwald, E. I. Chong, M. Jagannath, A. Yalamanchi, R. Krishnan, A.-T. Tran, S. DeFazio, and J. Banerjee, "Oracle8i index-organized table and its application to new domains." in *VLDB*, 2000, pp. 285–296.

[38] A. Katsarakis, Y. Ma, Z. Tan, A. Bainbridge, M. Balkwill, A. Dragojevic, B. Grot, B. Radunovic, and Y. Zhang, "Zeus: locality-aware distributed transactions," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 145–161.

[39] M. Serafini, E. Mansour, A. Aboulnaga, K. Salem, T. Rafiq, and U. F. Minhas, "Accordion: Elastic scalability for database systems supporting

distributed transactions," *Proceedings of the VLDB Endowment*, vol. 7, no. 12, pp. 1035–1046, 2014.

[40] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker, "E-store: Fine-grained elastic partitioning for distributed transaction processing systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 3, pp. 245–256, 2014.

[41] Y.-S. Lin, C. Tsai, T.-Y. Lin, Y.-S. Chang, and S.-H. Wu, "Don't look back, look into the future: Prescient data partitioning and migration for deterministic database systems," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 1156–1168.

[42] S. Loesing, M. Pilman, T. Etter, and D. Kossmann, "On the design and scalability of distributed shared-data databases," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 663–676.

[43] J. Hennessy, M. Heinrich, and A. Gupta, "Cache-coherent distributed shared memory: perspectives on its development and future challenges," *Proceedings of the IEEE*, vol. 87, no. 3, pp. 418–429, 1999.

[44] B. Bhattacharjee, L. Lim, T. Malkemus, G. Mihaila, K. Ross, S. Lau, C. McArthur, Z. Toth, and R. Sherkat, "Efficient index compression in db2 luw," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1462–1473, 2009.

[45] E. I. Chong, J. Srinivasan, S. Das, C. Freiwald, A. Yalamanchi, M. Jagannath, A.-T. Tran, R. Krishnan, and R. Jiang, "A mapping mechanism to support bitmap index and other auxiliary structures on tables stored as primary b+-trees," *ACM SIGMOD Record*, vol. 32, no. 2, pp. 78–88, 2003.

[46] Microsoft, "Clustered and nonclustered indexes described," 2025, accessed: 2025-08-28. [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-ver17

[47] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[48] T. P. Council, "Tpc-c benchmark," 2024, accessed: 2024-10-17. [Online]. Available: https://www.tpc.org/tpcc/

[49] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, 1979, pp. 23–34.

[50] W. Wu, Y. Chi, H. Hacígümüş, and J. F. Naughton, "Towards predicting query execution time for concurrent and dynamic database workloads," *Proceedings of the VLDB Endowment*, vol. 6, no. 10, pp. 925–936, 2013.

[51] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[52] W. Fan, M. Liu, C. Tian, R. Xu, and J. Zhou, "Incrementalization of graph partitioning algorithms," *Proceedings of the VLDB Endowment*, vol. 13, no. 8, pp. 1261–1274, 2020.

[53] Z. Abbas, V. Kalavri, P. Carbone, and V. Vlassov, "Streaming graph partitioning: an experimental study," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1590–1603, 2018.

[54] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni, "Hdrf: Stream-based partitioning for power-law graphs," in *Proceedings of the 24th ACM international on conference on information and knowledge management*, 2015, pp. 243–252.

[55] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "Fennel: Streaming graph partitioning for massive scale graphs," in *Proceedings of the 7th ACM international conference on Web search and data mining*, 2014, pp. 333–342.

[56] C.-W. Ou and S. Ranka, "Parallel incremental graph partitioning using linear programming," in *Supercomputing '94:Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, 1994, pp. 458–467.

[57] A. Uta, S. Au, A. Ilyushkin, and A. Iosup, "Elasticity in graph analytics? a benchmarking framework for elastic graph processing," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 381–391.

[58] S. Yang, X. Yan, B. Zong, and A. Khan, "Towards effective partition management for large graphs," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 517–528.

[59] D. Dai, W. Zhang, and Y. Chen, "Iogp: An incremental online graph partitioning algorithm for distributed graph databases," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, 2017, pp. 219–230.

[60] M. Pundir, M. Kumar, L. M. Leslie, I. Gupta, and R. H. Campbell, "Supporting on-demand elasticity in distributed graph processing," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2016, pp. 12–21.

[61] R. Dindokar and Y. Simmhan, "Adaptive partition migration for irregular graph algorithms on elastic resources," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 281–290.

[62] G. Urdaneta, G. Pierre, and M. Van Steen, "Wikipedia workload analysis for decentralized hosting," *Computer Networks*, vol. 53, no. 11, pp. 1830–1845, 2009.

[63] G. Karypis and V. Kumar, "A parallel algorithm for multilevel graph partitioning and sparse matrix ordering," *Journal of parallel and distributed computing*, vol. 48, no. 1, pp. 71–95, 1998.

[64] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri, "Partitioning trillion-edge graphs in minutes," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 646–655.

[65] C. Martella, D. Logothetis, A. Loukas, and G. Siganos, "Spinner: Scalable graph partitioning in the cloud," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 2017, pp. 1083–1094.

[66] MySQL, "MySQL 8.4 Reference Manual: Introduction to InnoDB," Online, 2024, accessed: 2026-01-25. [Online]. Available: https://dev.mysql.com/doc/refman/8.4/en/innodb-introduction.html

[67] R. Taft and O. Tan, "How We Built Easy Row-Level Data Homing in CockroachDB with REGIONAL BY ROW," Cockroach Labs Blog, Mar. 2024, accessed: 2026-01-25. [Online]. Available: https://www.cockroachlabs.com/blog/regional-by-row/

[68] PingCAP, "Clustered indexes," TiDB Documentation, accessed: 2026-01-25. [Online]. Available: https://docs.pingcap.com/tidb/stable/clustered-indexes/