Enhancing Transaction Processing through Indirection Skipping

Riki Otaki
University of Chicago
rotaki@uchicago.edu

Jun Hyuk Chang
University of Chicago
junhyukc@uchicago.edu

Aaron J. Elmore
University of Chicago
aelmore@cs.uchicago.edu

Goetz Graefe
Google
goetzg@google.com

ABSTRACT

In modern database management systems (DBMS), data retrieval typically requires traversing multiple layers—such as secondary indexes, primary indexes, and buffer pools—which introduces significant overhead and creates performance bottlenecks. In this paper, we propose a novel method that minimizes this overhead by establishing more direct access paths during data retrieval. Our experimental results demonstrate substantial efficiency gains across various DBMS components, including secondary indexing and concurrency control mechanisms. Specifically, we observe that implementing direct access paths can boost the throughput of transaction processing systems by up to 19.7× when executing the TPC-C-like benchmark with 40 threads. Furthermore, our approach holds promise for broader applications, potentially transforming data retrieval practices by enabling efficient handling of data movements with minimal overhead.

PVLDB Reference Format:
Riki Otaki, Jun Hyuk Chang, Aaron J. Elmore, and Goetz Graefe. Enhancing Transaction Processing through Indirection Skipping. PVLDB, 18(11): 4104 - 4116, 2025.
doi:10.14778/3749646.3749680

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/rotaki/LIPAH/tree/vldb2025.

1 INTRODUCTION

A well-known maxim in computer science attributed to David Wheeler suggests that "all problems in computer science can be solved by another level of indirection" [1]. DBMSs exemplify this principle through decoupling physical storage from logical data retrieval [9, 10, 38]. However, the caveat "except for the problem of too many layers of indirection" cautions against potential performance drawbacks from overusing this technique. A DBMS internally employs multiple layers of indirection to organize the physical records. Each indirection adds new functionality to the system, but also introduces overhead to the data retrieval process. In this paper, we explore strategies to minimize the performance penalties associated with the indirection layers in the data retrieval processes within DBMSs.

An indirection can be defined as a translation mechanism that maps one address to another. There are multiple layers of indirection in the data retrieval process in a DBMS. Figure 1 illustrates these layers of indirection in the data retrieval process performed by a transaction in a DBMS. As an example, a transaction might consult a secondary index, which maps secondary keys to primary keys. The primary index then maps primary keys to the addresses of the pages containing the data records. When accessing the pages during the traversal of the indexes, the buffer pool manager translates those page IDs into frame IDs, locating the corresponding data in memory. Each of these indirections is necessary to retrieve physical objects using logical identifiers, since objects are often physically organized in ways different from their logical usage. These indirection layers enable rapid access to target objects without full scans of the data storage.

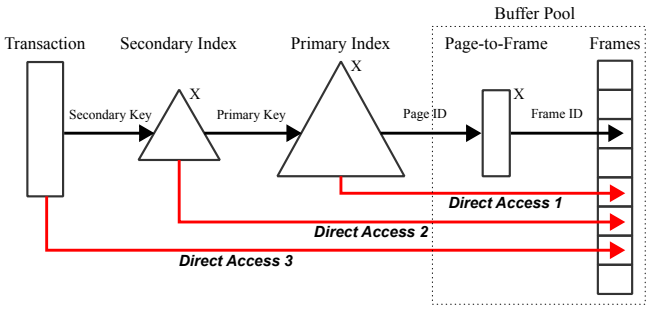


Figure 1: Layers of Indirection in Data Retrieval — Pages containing data records are accessed through secondary indexes, primary indexes, and the page-to-frame mapping in the buffer pool in the DBMS. Note that although the diagram implies that the buffer pool interaction occurs only at the end of the index traversal, in practice each node access during the traversal results in a separate invocation of the buffer pool manager. Points of contention are marked with X. Direct accesses enable skipping of the indirection layers.

Although these indirections make data access faster, they do not come for free. For example, with two five-level indexes, accessing a record requires 10 page accesses and 10 page-to-frame mappings. A transaction with 100 index accesses incurs 1,000 page accesses and 1,000 mapping operations—significant overhead [20]. Additionally, there are points of contention in the indirection layers. For example, the root node of a B$^+$-tree primary index will be accessed by all record accesses. If the root node is protected by a reader-writer latch, a cache-line invalidation due to the read-latch modifying the latch word will impede scalability in multi-core systems [28, 29, 31].

Pointer swizzling [14] is one technique devised to reduce the overhead of the indirection layer in the data retrieval process. This technique has been explored in B$^+$-tree indexes residing in the buffer pool of the DBMS [18, 28, 32]. The key idea is to embed physical frame addresses or raw pointers of child pages in their parent page of the B$^+$-tree to directly access a child page without going

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749680

through the page-to-frame mapping in the buffer pool when it is in memory. The page-to-frame mapping of the buffer pool is one of the points with high contention in database systems with high concurrency [18, 28]. With pointer swizzling, the contention is reduced because the page-to-frame mapping is bypassed. A reference of a page is implemented as a 64-bit union type which can store either a logical page ID or a physical frame address. If a page moves to memory from disk, it is *swizzled* from the logical page ID to the physical frame address. On the other hand, if the page moves to disk from memory, the word is *unswizzled* from the physical frame address to the logical page ID. Although this technique is effective in reducing the overhead of the indirection layer, it is not easy to apply to many systems. The primary challenge lies in invalidating the physical frame address when the page is offloaded to disk. When evicting a page from the buffer pool, all the physical frame addresses pointing to the evicted page must be invalidated to prevent the use of the stale physical frame addresses. This requires ownership tracking of the references. Furthermore, before evicting the page, all the physical frame addresses in the page must be invalidated. Due to these complexities, it is difficult to incrementally adopt the pointer swizzling technique and apply it to the existing systems. Strategies for mitigating the overhead of this invalidation process have been proposed, such as limiting the number of owners of the physical frame addresses [18, 28] but it is still not easy to implement and maintain.

In this paper, we generalize the idea of pointer swizzling to a broader context within database systems, such as secondary indexes and transaction read/write sets. Additionally, we simplify the implementation of the pointer swizzling technique by only using the physical address as a hint to locate the data record, while using the logical ID as a fallback when the hint is no longer valid. This is achieved by maintaining both the logical ID and the physical address of the object as a reference to the object, which we call Logical ID with Physical Address Hinting (LIPAH). Figure 1 shows the applicability of LIPAH to the components of the database system. Direct access 1 skips page-to-frame mapping in the buffer pool, direct access 2 skips the primary index and the page-to-frame mapping, and direct access 3 skips the secondary index, primary index, and the page-to-frame mapping. We believe that the applicability of LIPAH is not limited to the database systems, but it can be applied to many systems where objects dynamically change locations.

The main contributions of this paper are as follows:

- We demonstrate that indirection skipping demonstrated by pointer swizzling can be generalized to a broader context within database systems, such as secondary indexes and transaction read/write sets.
- We provide a general technique called Logical ID with Physical Address Hinting (LIPAH) to support indirection skipping without eager invalidation of the physical addresses of objects (introduced in our vision paper on a resource-efficient DBMS where all temporary and persistent state reside in paged structures [34]).
- We illustrate concrete implementation techniques to support LIPAH effectively in concurrency control protocols in transaction processing systems.
- We evaluate the performance of LIPAH using TPC-C-like benchmark [4] and show that LIPAH can improve the performance of transaction processing systems, achieving up to 19.7× throughput when executed with 40 threads.

2 BACKGROUND AND RELATED WORK

This section provides background information on how data is retrieved in a DBMS. Specifically, the section illustrates how data is organized in on-disk systems, how data modifications are handled in transaction processing, and how certain techniques in the buffer pool manager can reduce contention and overhead in the data retrieval process.

2.1 Retrieving Data in On-Disk DBMS

In on-disk systems, records are typically stored in pages stored in files. There are several ways to organize the records in a file such as a heap file, a sequential file, a hash partitioned file, a B^+-tree organized file [37]. Records are identified either by physical addresses or by logical identifiers. A physical address typically consists of a page ID and a slot ID in a page. With the physical address, a record can be retrieved directly from a file. In contrast, a logical identifier is independent of the physical organization of the records. A logical identifier is either a system-generated unique integer or a schema-defined primary key. With the logical identifier, records are retrieved through an indirection layer that maps the logical identifier to the record such as a hash table or B^+-tree.

Logical identifiers are essential because a record's physical address can frequently change during updates, especially in OLTP workloads. Frequent operations—insertions, updates, and deletions—can force records to move due to page overflows, underflows, or data reorganization (like hash re-partitioning or B^+-tree re-balancing) caused by page size limits. Similarly, in multi-version systems, updating a record creates a new version that is stored separately [11, 26, 35] (or through in-place updates that may change its location due to size differences [8, 13, 33]) to support concurrency control. In contrast, logical identifiers remain constant regardless of these physical relocations.

Frequent updates to the physical addresses of records are problematic if secondary data structures reference them by their physical addresses. For example, in PostgreSQL, indexes point to the physical address of a record, which is kept in a heap-organized file. Updating a record normally requires inserting a new entry in every index that references it [24, 35]. The *Heap-Only Tuple* (HOT) optimization skips this step when (i) none of the modified columns are indexed and (ii) the new tuple still fits on the same page as the old one [6]. This is a significant overhead in the data retrieval process. MySQL, on the other hand, stores records in a clustered index (B^+-tree organized file) with logical identifiers [2]. The secondary indexes point to the logical identifiers of the records. Hence, the secondary indexes do not need to be updated when the record is updated unless the search key of the secondary index is modified.

Although using the logical identifiers reduces the overhead of updating the secondary indexes, there are several advantages of using the physical addresses in the secondary indexes. First, accessing by a physical address is faster than accessing by a logical identifier. If secondary indexes point to the search key of the primary

index, then the search time will roughly be doubled because two lookups are required to retrieve a record. On the other hand, if the secondary index points to the physical address of the record, the record can be retrieved directly from the file after the secondary index lookup. Second, using physical addresses reduces the contention in the indirection layer. If multiple secondary indexes point to the same primary index, then the primary index will be a point of contention that throttles the scalability of the system if a search must be performed in the primary index for each secondary index lookup. For example, if the primary index is a B^+-tree, then the root node of the B^+-tree will be contended because all the lookups pass through the root node. In contrast, secondary indexes that point to the physical address of a record exhibit less centralized contention because the records are distributed across multiple pages in a file.

Oracle uses a hybrid approach where secondary index entries contain both the logical key and 10-byte physical address information (file ID, page ID, slot ID) [19, 22]. If the physical address is accurate, records are accessed directly. If stale due to record movement, the logical key is used to consult the primary index.

2.2 Installing Updates To Shared Database

Concurrency control protocols that ensure serializability can be categorized into two types based on how they install modifications to records during transaction execution.

Immediate Modifications. The first type is called immediate modification [37]. In this approach, during the execution of a transaction, records in a shared database are updated in-place (or, in multi-version systems, a new version of a record is created in the shared database). To avoid cascading aborts, the transaction must protect the installed updates from being read or written to by other transactions until the transaction commits.

To protect the installed updates, systems employ locks, transaction IDs, timestamps, or a combination thereof [7, 11, 13, 23, 26, 33]. With locks, other transactions cannot read or write the record until the transaction releases the lock. With transaction IDs, other transactions must check the transaction status table to determine if the record is produced by a committed transaction. With timestamps, other transactions check the visibility of the record by comparing the timestamp of the record with the timestamp of the transaction. When aborting a transaction, the installed updates must be rolled back. This is done by restoring the before-image of the record as the latest version of the record. With the transaction ID, the system may simply mark the transaction as aborted in the transaction status table so that other transactions can ignore the updates. The actual removal of the installed updates can be done eagerly right after the transaction aborts, or lazily during the next access to the record, if the system maintains the old versions of the record accessible in the shared database.

Deferred Modifications. The second category installs updates only after the serial order of the transactions is determined. This is called deferred modifications [37]. Deferred modifications are often used in single-version optimistic concurrency control protocols [21, 25, 40, 41] but can also be used in pessimistic and multi-version systems [30]. In the optimistic approach, the transaction stores the updates in a private workspace until the transaction commits. The transaction validates its read and write set during the commit phase. If the validation is successful, the transaction will install the updates to the records in the shared database. If the validation fails, the transaction aborts. In this approach, aborting transactions does not require rolling back because the updates are not installed to the shared database.

With deferred modifications, a read-modify-write transaction will access a record at least twice during the transaction processing. The first access will be during the execution phase to read the record, to check its existence, or to verify its non-existence of the record. The second access will be after the commit phase to actually install the updates. In in-memory database systems, second access is often done through a record pointer kept in the private workspace of the transaction [21, 40, 41]. Hence, the second access is fast. In on-disk systems, the second access cannot be done through a record pointer because the record might have moved to a different location due to memory-disk swapping or page reorganizations [25]. Records are accessed through a logical key which goes through the index structure to locate the record, which may incur additional disk I/O by the buffer pool manager. Hence, many on-disk systems employ the first approach of immediate modifications to avoid a second disk access [7]. Note that some in-memory immediate modification systems also require accessing the record twice if concurrency control primitives such as locks are co-located with the record representation in the shared database [15, 36].

2.3 Reducing Contention In Buffer Pool

The buffer pool manager caches frequently accessed pages in memory to reduce disk overhead in a data retrieval process. The buffer pool is implemented as a fully associative cache where a page can be placed anywhere in a set of page frames. Therefore, the buffer pool manager must have a mechanism to locate a page in the buffer pool. This is typically done by maintaining a page-to-frame mapping table that maps a page ID to a frame ID in memory [12]. When a page is requested, the buffer pool manager consults the mapping table to locate a page in the buffer pool. If the page is not in the buffer pool, the buffer pool manager must load it from the disk.

The page-to-frame mapping table is a point of contention and thus a bottleneck in scaling the system [18, 28]. The mapping table must ensure that no two pages are mapped to the same frame and that no page is mapped to more than one frame. Latches are often used to ensure the consistency of the mapping table when multiple threads insert or remove an entry from the table [3]. This limits the scalability of the system. To address this problem, several techniques have been proposed.

Pointer Swizzling. The idea of pointer swizzling in the buffer pool is to use frame pointers to directly access a page in the buffer pool without consulting the page-to-frame mapping table [18, 28, 32]. A page reference is implemented using a union type that stores either a logical page ID or a physical frame address. The reference is *swizzled* to the frame address when a page is loaded into the buffer pool and *un-swizzled* to the page ID when a page is evicted from the buffer pool. If all pages are in memory, pages are accessed with zero overhead from the mapping table because physical addresses enable direct access to the pages. However, careful management of page references is required when evicting a page from the buffer pool. For example, when evicting a page, the buffer pool manager must

identify all the owners of the page reference with a physical frame address, and it must un-swizzle them before evicting it in order to avoid dangling pointers. Additionally, the buffer pool manager must ensure that frame pointers do not get written to disk. Thus, un-swizzling physical pointers residing in the evicting page is also necessary. This increases complexity in the buffer pool management. Consequently, past research has limited the applicability of pointer swizzling to data structures that do not have more than one incoming reference to a page [18, 28]. Hence, data structures with intricate pointer dependencies, such as graphs and B^+-trees with sibling pointers, secondary indexes with pointers to primary index pages, are not supported because they make it difficult to determine which page to evict first.

Virtual-Memory Assisted Buffer Management. *vmcache* is an approach to reduce the overhead of converting the page ID to the physical frame pointer by eliminating the page-to-frame mapping table entirely [27]. The page-to-frame mapping table in the conventional buffer pool manager serves two purposes: to track which pages are in memory and to locate a frame of a page. To keep track of which pages are in memory, vmcache uses an array of 64-bit words with each word representing the status of a page on the underlying storage device. The word is not only used to determine whether the corresponding page is in memory or on disk but also for latching the page. To remove the necessity for finding the frame of a page, vmcache uses a preallocated virtual memory address that is initialized with the same size as the storage device. This virtual memory address is mapped one-to-one with the underlying storage device, which eliminates the need to map the page ID to the frame address. The buffer pool manager reads and writes pages to and from the pre-determined virtual memory addresses to serve the page requests and evictions. By employing these techniques, vmcache removes the page-to-frame mapping table entirely and reduces the overhead of the buffer pool management.

3 INDIRECTION SKIPPING WITH LIPAH

In our earlier vision paper on building a resource-efficient DBMS fully based on paged memory, including query operator state [34], we introduced the concept of **Logical ID with Physical Address Hinting (LIPAH)** as a novel approach to reduce the contention and overhead in buffer pool management. In that paper, we discussed how using a pair—a logical identifier coupled with a physical address—can improve data retrieval efficiency compared to traditional pointer swizzling techniques. Building on that vision, this section provides a detailed discussion of LIPAH and its applications not only in buffer pool management but also in secondary indexes and transaction read/write sets.

3.1 Buffer Pool Management

LIPAH reduces contention in the buffer pool manager [34] by using a "fat-pointer" to access a frame in the buffer pool. The fat-pointer here is a pair of page ID (logical identifier) and a frame ID (physical address hint). Using the pair, a frame is directly accessed by its frame ID, and if it does not contain the expected page ID, the mapping table is consulted. The logical key serves two purposes—to validate the physical address hint and to locate the object when the hint is invalid. Since the physical address is only a hint, it can be updated lazily. With pointer swizzling, ownership of pointers must be tracked because they must be updated when the object transfers between memory and disk. In contrast, LIPAH does not require ownership tracking because the physical address can be stale—the logical part of the pair can be used to locate the object when the physical address is invalid. By using LIPAH, data structures with complex pointer dependencies can be stored in the buffer pool, which was difficult to achieve with pointer swizzling.

3.2 Secondary Indexes

Secondary indexes are used to locate records when records are not necessarily physically organized by attribute keys used in a search query. Secondary indexes map a secondary key to either a physical address of a record or a key in a primary index. In the case of the former, the physical addresses must be updated when records are moved due to data modification or reorganization in the primary index. In the case of the latter, updates to the primary index do not always propagate to the secondary index but a lookup in the primary index is required to locate a record after locating the primary key in the secondary index.

Similar to Oracle's database [19, 22] that uses a combination of a logical key and a physical row ID to locate a record in the primary index, we propose a middle ground between these two approaches by storing both the logical ID and the physical address of a record in the secondary index. We use the physical address as a hint to locate a record first, and if the hint is not valid, the logical key is used to locate the record in the primary index. Here, the physical address is not limited to the page ID and slot ID but may also include the frame ID in the buffer pool, which makes it different from the Oracle's approach. Moreover, we aim to identify the optimal combination of these physical address components because there is a potential trade-off between the size of the physical address and the speedup of a record access gained by the hint.

In order to utilize LIPAH, a validation step must ensure that a physical location hint is still valid. The validation process depends on the elements included in the hint. For example, if the hint is a page ID, validation must ensure that a record is still in the page. If the frame ID is included in the hint, we can bypass the page-to-frame mapping table in the buffer pool but the additional validation must ensure that the hinted page is still in that frame. If the hint consists of a page ID and a slot ID within the page, the validation must ensure that a record is still in the slot. A detailed validation process is described in Section 4.

If the validation fails, the search must be performed with the logical key from the point where the stale hint was used. After locating the record, the physical location hint is updated with the new physical location of the record. Since the physical location is only used as a hint, the hint can be updated lazily. For example, if repairing the hint is expensive due to contention on a page, it can be delayed until the contention is resolved.

3.3 Read and Write Sets in Transactions

In transaction processing systems, read and write sets of a transaction are often maintained in the private workspace of the transaction. They are used for validating operations and installing modifications in optimistic concurrency control protocols, and for cleaning

up the transaction after the transaction is finished such as releasing locks, and rolling back the transaction in case of abort. In in-memory systems, the read and write sets accommodate keys and pointers to records that are accessed by the transaction as described in Section 2.2. During a transaction's execution phase, a record is accessed by a key and its pointer will be cached in the read and write set. When the record needs to be accessed again during the validation phase or during rollback, the pointer is used to access the record directly. However, in on-disk systems, a memory pointer cannot be used to locate the record because the pages that contain the record can be moved from memory to disk due to eviction. Moreover, identifiers such as page ID and slot ID cannot be used, since data reorganization operations—such as page compaction, splits, and merges—can alter the physical location of the record.

Similar to the approach we described for the secondary index in the previous subsection, we propose to store the physical location of a record in the read and write sets as a hint. If the hint does not match the actual physical location of the record, indexes are consulted to locate the record. The physical hint can potentially bypass multiple page accesses when lookups for both secondary and primary indexes are skipped. This also prevents transactions from accessing the root node of the indexes (if the indexes are B$^+$-trees) multiple times, which can become a bottleneck in multi-core systems [29, 31].

Concurrency control protocols without validation phases such as 2PL can also benefit from the physical location hint in case of abort. When a transaction with immediate modification is aborted, the transaction must roll back its changes. The physical location hint can be used to locate the records that were modified by the transaction. Additionally, queries such as `SELECT ... FOR UPDATE`, which use a read-modify-write pattern, can naturally benefit from the physical location hint because the record must be located again during the update phase after the read phase, eliminating redundant index traversals.

4 IMPLEMENTATION DETAILS

In this section, we describe the implementation of LIPAH—from the buffer pool at the system's foundation to its use in index structures (e.g., trees and hash tables) and integration into transaction processing. Notably, the buffer pool manager and index structures share key similarities. Both store objects accessible via logical identifiers (page IDs for the buffer pool, keys for indexes) and offer a conceptually similar `Get` function. This function not only takes in the logical key but also an optional physical address hint to bypass standard lookup steps (e.g., the page-to-frame mapping or tree traversal). If the hint is invalid, the system retrieves the object using the logical key and updates the hint with the current physical location.

4.1 Buffer Pool Manager

We implement a standard buffer pool with a fixed number of frames. Each frame contains a fixed size page and in-memory metadata such as a dirty flag, a page ID, and a frame latch. The buffer pool is essentially a fully associative cache of pages. Hence, in order to locate a frame that contains a page, a hash table is maintained that maps a page ID to a frame ID.

Data Types and Addressable Space. Page and frame IDs are 32-bit unsigned integers; with 16 KiB pages this yields 64 TiB of addressable space per ID. Because a page ID is an offset within its file, a single file is capped at the same 64 TiB. To grow beyond that limit we add a 16-bit *container ID* that maps to a file, so an on-disk block is uniquely identified by the pair of container ID and page ID. Because every index structure keeps its container ID in a metadata page, embedded physical-address hints do not need to store the container ID.

Applying LIPAH to Paged Data Structures. A page-based data structure on the buffer pool is a good candidate for LIPAH. These data structures often have a parent-child relationship between pages and they are linked together by page IDs. For example, B$^+$-tree indexes and hash indexes with buckets implemented as a linked list can benefit from LIPAH. By integrating a physical address, which in this case is the frame ID of the page, into a page reference, we can directly access a frame that potentially contains a desired page without going through the page-to-frame mapping table. This is especially useful when the working set fits in memory because there will be zero accesses to the mapping table after the initial access, thanks to the cached hint. When the working set does not fit in memory, pages will be evicted from the buffer pool and the hint will be naturally expired. During a traversal of the data structure, if a child page is not found in the expected frame, the frame hint will be repaired. Our system currently repairs the hint opportunistically (i.e., when a write-latch can be acquired without waiting), which is described in detail in Section 4.4.

4.2 Indexes

Foster B-tree. To evaluate LIPAH, we implement a Foster B-tree [17], which is a variant of the write-optimized B$^+$-tree [16]. This design ensures that each node has only one incoming edge, allowing for lazy reorganization. The key advantage is that leaf nodes can split without updating their parent nodes. When a leaf node splits, it creates a foster child page linked to the original leaf. This simplifies transaction processing under LIPAH by avoiding the need to cache an address of a parent node of the leaf in the transaction's read/write set, in case of a leaf page split when installing updates from the transaction's private workspace. The foster relationship is resolved eventually by other transactions traversing the tree from the root to the leaf, connecting the foster child to the parent node [17]. Although our prototype employs a Foster B-tree, LIPAH is not limited to that structure. Indexes that lack Foster-style splits can still use LIPAH by recording the parent page's physical address in the transaction's read/write set whenever a split or merge might occur. More broadly, LIPAH integrates cleanly with conventional B$^+$-trees, hash indexes, and other common index structures.

Node Page. Each node of a tree is a fixed-size page on a buffer pool frame. A slotted page enhanced with metadata (e.g., tombstone, height, node location, record count) is used to store records. Each page incorporates fence keys that delineate a key range of a subtree rooted at that page. Slots, arranged in sorted order by key, reference index entries containing both the key and the corresponding record in a contiguous memory region. To retrieve a record, a binary search is conducted on the slots, after which the record is accessed from the memory region. With slot IDs—implemented as 32-bit unsigned

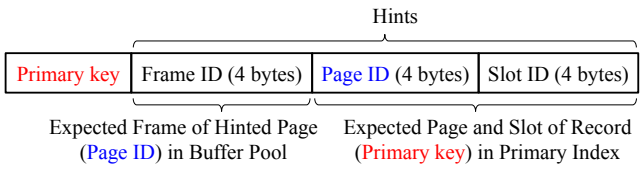


Figure 2: An Entry of a Secondary Index With Complete Hints

integers—a slot can be directly accessed without a binary search. Concurrent access to a node page is controlled by a frame-latch implemented as a readers-writer lock.

Applying LIPAH to Secondary Indexes. An entry of a secondary index with LIPAH is a pair consisting of a primary key and a combination of page ID, frame ID and slot ID of the record in the primary index as depicted in Figure 2. Any element in the superset of the ⟨Page ID, Frame ID, Slot ID⟩ can be used as a hint. However, using the frame ID or slot ID as a hint is not useful without the page ID. Therefore, hints for secondary index are limited to ⟨Page ID⟩ and ⟨Page ID, Frame ID⟩ ⟨Page ID, Slot ID⟩, and ⟨Page ID, Frame ID, Slot ID⟩. After traversing the secondary index, the primary key and the hint are fetched. The hint is used to directly access the record in the primary index. Different hints require different validations.

- **Frame ID Validation.** The frame ID hint is used to directly access a frame, with the expectation that the frame contains a page corresponding to the page ID hint. It is validated by checking that the page ID in the frame matches the page ID hint. Even though the frame ID validation succeeds, it may get repaired when the page ID hint is pointing to an incorrect page.
- **Page ID Validation.** The page read using the page ID hint is validated by checking whether (a) the page is not deleted (i.e., connected to the tree), (b) the page is a leaf page, and (c) the primary key is in the range of the page's fence keys. Additionally, we check that the primary key is not in the range of the foster child, which is a necessary step for a Foster B-tree.
- **Slot ID Validation.** After running the frame and/or page validation, the slot ID hint is used to directly access the slot in the page. The slot is validated by checking whether the number of slots in the page can accommodate that slot ID, and whether the slot points to a record with the primary key. If the page ID validation fails, the slot ID hint is disregarded, given the low probability that the slot hint is correct when the page hint is erroneous.

If the validation fails, the search is continued from where the hint failed. For example, if the slot validation fails, the search begins on the page which was found by the frame and page hints. The system then opportunistically repairs the slot ID with the new location of the record. As we show in Section 5.3, the slot ID hint is invalidated frequently, so we disable it by default in our implementation. More details on the use of hints in secondary indexes are provided in Section 4.4.

Preventing Accesses to Removed Pages. Reorganization operations, such as page merges, may remove a page referenced by the hint from the tree. Accessing a removed page can yield incorrect results; therefore, a tombstone bit is employed to invalidate the page ID during validation if it is set. Since setting the tombstone bit marks the page as dirty and causes write amplification, deleted pages are instead tracked in an in-memory lock-free data structure that records their page and frame IDs. These pages are reused when a new page is required, reducing write amplification. When a deleted page is reused, its tombstone bit is reset, and it is repopulated with new records and fence keys. Thus, if there is no overlap between the old and new fence key ranges, outdated references are invalidated during page ID validation, thereby preserving system correctness.

4.3 Concurrency Control Protocols

We implement pessimistic concurrency control with deferred modification using two-phase locking. During transaction execution, each accessed key is locked, and the locks are only released upon transaction commit or abort. We adopt a *NOWAIT* strategy for lock acquisition, which causes an immediate abort if the requested lock is not available. An in-memory concurrent hash table maps each key to its lock object. The lock object is a signed integer representing the lock state.

Each transaction maintains a read set and a write set to track both lock acquisition history and in-flight data modifications. In the deferred modification approach, any changes made by a transaction are written to a private workspace during its execution phase. The transaction installs these changes into the shared database after all locks are acquired and the commit is assured. This strategy avoids rolling back partially installed changes on abort. To protect against phantoms, we employ next-key locking. Before inserting a new record, the transaction locks the next key, then locks the to-be-inserted key. After installing the new record in the shared database, the transaction unlocks the next key while keeping the newly inserted key locked until commit. This ensures that concurrent range scans either include the new record entirely (post-commit) or not at all (pre-insert), thereby preventing phantoms.

Potential Extensions. Although our current implementation focuses on deferred modification with locking, it is straightforward to adapt the system for:

(1) Immediate Modification (Pessimistic). A transaction would modify the shared database during its execution phase, storing old values in its write set for rollback. This approach ensures minimal commit overhead but necessitates undo operations on abort or deadlock.
(2) Optimistic Concurrency Control (Similar to Silo [40]). Each record would carry a version number to track changes. A transaction would gather read and write sets during execution, then lock its write set and validate all version numbers before installing changes. A mismatch in any version would trigger an abort and retry. Phantoms would be prevented by page-level or range-based versioning.

4.4 Balancing Hint Benefits and Repair Costs

This section explains how each hint can become obsolete, weighs the cost of refreshing it against the performance benefit, and sketches an adaptive mechanism that throttles repairs during periods of high churn.

Slot ID Hints (Tuple Position within a Leaf). Every insert, delete, or in-place update inside the leaf shifts tuple offsets and therefore immediately invalidates the hint. As Section 5.3 shows, slot ID hints expire so frequently that the only benefit—skipping a binary search in the leaf—is rarely worth the maintenance overhead. Accordingly, our implementation disables slot ID hints by default.

Frame ID Hints (Buffer Pool Frame). A frame ID becomes stale only when its page is evicted. While the working set fits in DRAM, the hint remains valid; once the workload exceeds memory capacity, the system is I/O-bound and an incorrect hint merely adds a single in-memory lookup. We therefore refresh the hint *opportunistically*: the frame is latched in exclusive mode to repair the hint only if the latch can be acquired without waiting, and we do not set the page's dirty bit, because a frame hint is unlikely to survive the next eviction anyway.

Page ID Hints (Page in the Primary Index). Page ID hints survive evictions and break only on structural actions (splits and merges), which are much rarer than evictions as shown in Section 5.3. When the page latch is uncontended we correct the hint and mark the page dirty so that the fix persists across future evictions. Before following a page ID hint, the system first checks whether the hinted leaf is memory resident; if it is not, we fall back to a normal tree traversal. This check is based on the observation that inner nodes of the primary index are typically stored in memory, as they are orders of magnitude fewer than leaves. Without this check, a stale page ID could trigger an unnecessary I/O to fetch the wrong leaf, negating the benefit of the hint.

Adaptive Repairing. A practical extension would monitor each hint's hit rate—e.g., in rolling ten-second windows—and suspend opportunistic repairs when the hit rate drops below a threshold, as happens during bulk loads or other high-churn phases. Although we have not yet implemented this mechanism, it offers a lightweight avenue to further reduce contention. More details on the cost of repairing hints are provided in Section 5.4.

Taken together, these policies delineate *when* to initiate repairs and illustrate how adaptive strategies can steer LIPAH under dynamic, skew-heavy, or rapidly evolving workloads.

5 EVALUATION

All experiments were conducted on a server equipped with dual-socket Intel® Xeon® Silver 4116 CPUs. Each socket features 12 physical cores operating at a clock speed of 2.10 GHz, with hyper-threading enabled (i.e., two threads per core) and 192 GB of RAM. The external storage is an Intel 760p NVMe SSD rated at 340 kIOPS on random-read and 275 kIOPS on random-write for 4 KiB blocks. Pages are sized at 16 KiB. Thus, the theoretical maxima are 85 kIOPS on read and 69 kIOPS on write. We employ the `io_uring` [39] interface with Direct I/O for all disk operations, ensuring that the pages are not cached in the operating system's page cache. For the purpose of these experiments, the term "index" refers to a Foster B-tree, regardless of whether it is a primary or secondary index. Additionally, unless specified, the key size is 10 bytes and the record size is 100 bytes. The database system employed in our experiments is implemented in Rust. Experiments are single-threaded except for Sections 5.1 and 5.4. This evaluation seeks to address the following research questions:

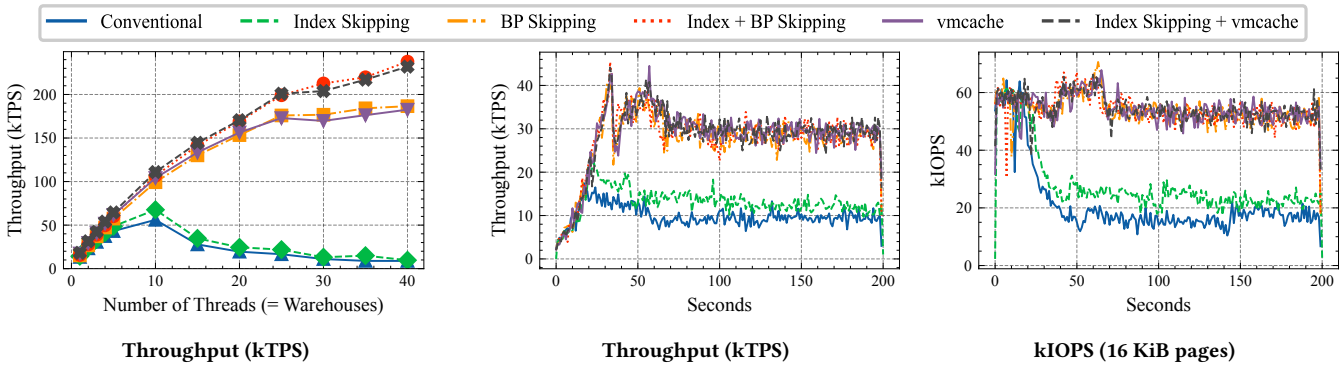
(1) **Performance** — How effective are hints in enhancing the performance of transaction processing and index access?
(2) **Robustness** — How resilient are hints to modifications of data? How do the hints perform under various data modifications?
(3) **Overhead** — What is the cost of repairing hints when they are invalidated? What are the storage overheads of maintaining hints?

5.1 Transaction Processing Performance

To evaluate the performance of LIPAH, we compare the performance of transaction processing with and without hints. We use the TPC-C benchmark [4], excluding the *keying* and *thinking* times, for this evaluation. The TPC-C benchmark simulates an online transaction processing (OLTP) workload, where multiple transactions concurrently access the shared database. We ran two sets of experiments: one with the entire database in memory and another with a larger-than-memory dataset. For the in-memory experiment, we varied the number of threads and warehouses from 1 to 40, and recorded the average throughput of 60 seconds for each configuration after a warm-up phase. The experiment was repeated three times for each configuration. For the larger-than-memory experiment, we used a dataset of 500 warehouses, which is approximately 80 GiB in size, and a buffer pool of 32 GiB. We used 40 threads to run the benchmark, where each thread accesses a random warehouse. We started with an empty buffer pool and measured the throughput and IOPS for 200 seconds.

We use 2PL with deferred modification as a proxy for protocols that access the shared database twice (read then write), such as OCC. Both approaches write to a private workspace then apply changes after acquiring locks (2PL) or validation (OCC). While some OCC variants store versioning information in indexes for validation [40, 41], we focus on the common case where each write requires two index accesses, assuming read/write sets fit in memory. To assess the impact of hints, we evaluated six distinct configurations:

(1) **Baseline (Conventional):** No hints are used; page-to-frame look-ups are resolved via a concurrent hash table [5], similar to the approach in PostgreSQL [3].
(2) **Buffer Pool (BP) Skipping:** Frame ID hints bypass the buffer pool's page-to-frame mapping.
(3) **Index Skipping:** Page ID hints bypass the traversal from the index root to the target leaf.
(4) **Index + BP Skipping:** Both Frame ID and Page ID hints are enabled.
(5) **vmcache [27]**: Pages are addressed directly through the operating system's page table, removing the application-level indirection.
(6) **Index Skipping + vmcache:** LIPAH's index-skipping applies to vmcache-based systems, as it is independent of page-to-frame mapping. This novel combination skips root-to-leaf traversal in large, frequently accessed indexes.



Throughput (kTPS) **Throughput (kTPS)** **kIOPS (16 KiB pages)**

(a) In-Memory TPC-C — Fixed one warehouse per thread.

(b) Larger-than-memory TPC-C — 500 warehouses ≈ 80 GiB, 32 GiB buffer pool, 40 threads. Warehouses chosen at random per thread.

Figure 3: TPC-C Experiments: (a) In-Memory and (b) Larger-than-Memory.

To isolate the impact of the indirection mechanism on performance, we integrated vmcache into our own prototype so that all other components remain identical (clock eviction, Foster B-tree index [18], 2PL, `io_uring` [39] backend with Direct I/O and the same TPC-C driver). Logging was disabled to isolate indirection skipping performance.

In-Memory TPC-C Performance. Figure 3a shows the TPC-C throughput under different hint configurations when the entire database fits in memory. Overall, enabling skipping in the buffer pool and the index improves the throughput by up to 19.7× compared to the conventional buffer pool when the number of threads is 40. Due to hyperthreading, the scalability growth is reduced when more than 24 threads are utilized. Although the index skipping reduces the cost of traversing the index tree, it provides little benefit without the BP skipping. This shows that BP skipping plays a more critical role in terms of scalability. It applies to every page access in the buffer pool, where a single page-to-frame mapping for all pages becomes a bottleneck. Once BP skipping is enabled, index skipping can further enhance the throughput because it cuts down the number of page accesses required per transaction. Index + BP Skipping outperforms plain vmcache by ≈ 1.3×, as vmcache does not skip the index traversal and thus incurs the overhead of traversing the index tree from root to leaf. Adding our index-skipping optimization to vmcache enhances the throughput by ≈ 1.3×, confirming that our technique is complementary to OS-level indirections.

For memory-resident transactional workloads, our results show that LIPAH significantly improves performance through both BP skipping and index skipping. Importantly, it achieves these gains without requiring tracking of the ownership of the references, which is a common challenge in pointer swizzling.

Larger-than-Memory TPC-C Performance. Figure 3b shows the TPC-C transaction-processing throughput and IOPS for the larger-than-memory experiment. All variants that avoid the page-to-frame mapping—BP Skipping, Index + BP Skipping, vmcache, and Index Skipping + vmcache—converge at roughly 53 kIOPS (≈ 30 kTPS). Although frame IDs are frequently invalidated by page evictions, checking an invalid frame has little impact on throughput because the workload is I/O-bound. Adding our index-skipping optimization to either BP Skipping or vmcache does not improve throughput, as most internal nodes of the Foster B-tree remain in memory and traversing from root to leaf costs far less than reading leaf pages from the SSD. The benchmark defines two secondary indexes, on the customer and order tables. We confirmed that the customer secondary index experiences no page ID invalidations because the customer table sees neither inserts nor deletes in TPC-C. The order secondary index does see invalidations, caused by the `NewOrder` transaction inserting new rows, which affects around 20% of all rows accessed through the index. In contrast, variants that consult the page-to-frame mapping on every page access (Conventional and Index Skipping) cap throughput at 10-15 kIOPS: their transactional throughput is limited by contention on the mapping table, as also observed in the in-memory TPC-C experiment (Fig. 3a).

In summary, under memory pressure the system becomes I/O-bound, and eliminating latch contention on the page-to-frame mapping table is essential to fully exploit SSD bandwidth. An invalid frame hint has negligible impact because throughput is I/O-bound, and an invalid page hint rarely matters because page-ID invalidations are infrequent even when insertions are common on the primary index. The rate of hint invalidations is examined in more detail in Section 5.3.

LIPAH vs vmcache. LIPAH translates logical page IDs to physical frame addresses entirely in user space, whereas vmcache relies on the operating-system page table and thus removes one layer of indirection. Each design choice has practical consequences:

- **Portability and Operational Effort.** Because vmcache has a stronger dependency on OS page-table, large deployments may need to load or maintain the *exmap* kernel module when page-table scalability limits are reached [27]. Installing and tracking such modules typically requires administrative privileges and re-compilation after kernel upgrades. LIPAH, by contrast, incurs no kernel changes: the page table lives in user space and the buffer-pool pages are never unmapped, so remote TLB shoot-downs are avoided. This user-space approach can simplify adoption on managed or cloud platforms where kernel extensions are difficult.

- **Metadata Footprint.** Both approaches keep per-page metadata (dirty bit, latch state, and so on), but the allocation strategy differs. vmcache pre-allocates metadata for every on-disk page, leading to a footprint proportional to disk size; LIPAH allocates metadata only for frames that actually reside in the buffer pool and therefore scales with buffer-pool size.

In short, LIPAH sidesteps OS-level dependencies and keeps metadata proportional to the resident working set, whereas vmcache offers direct address translation at the cost of tighter coupling to the kernel and a larger metadata reservation. LIPAH is a viable solution for eliminating the overhead of accessing the page-to-frame mapping in the buffer pool, while also being more portable and flexible than vmcache.

5.2 Secondary Index Performance

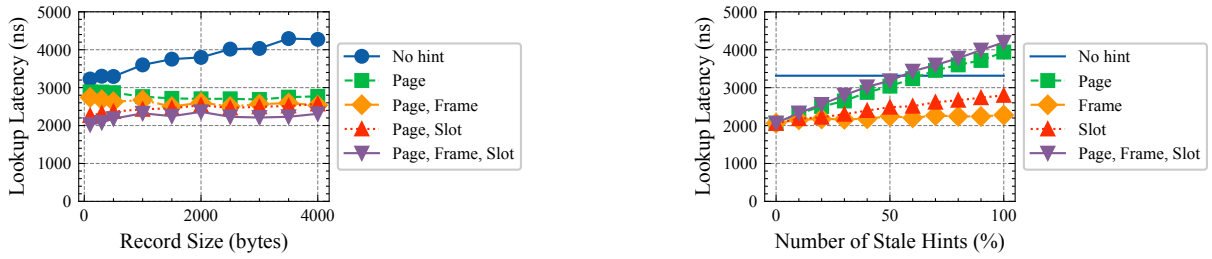
Lookup with Correct Hints. We evaluate the performance of index lookups under different hint types. Here, we aim to understand which types of hints are useful when the hints are all correct. We constructed a primary index with 1, 000, 000 entries and a corresponding secondary index with one-to-one correspondence. Here and in the next experiment, the buffer pool was sized to hold both the primary and secondary indexes.

The lookup performance was quantified by measuring the average time required to retrieve all keys from the secondary index in a random order, without any updates to the primary index. A warm-up phase preloaded the indexes into memory, after which the experiment was performed 20 times to determine the average lookup time for each hint configuration. Five different hint types were evaluated: No-hint, ⟨Page ID⟩, ⟨Page ID, Frame ID⟩, ⟨Page ID, Slot ID⟩, and ⟨Page ID, Frame ID, Slot ID⟩. The hints were stored alongside the secondary index entries, pointing to records in the primary index. Here, frame hints for root-to-leaf traversals of each index were enabled by default (i.e, the internal nodes of the Foster B-trees were connected with the page and frame ID). Only the information stored in the leaf nodes of the secondary index were varied between the different hint configurations. All the hints are initially set to the correct address and they do not change during the measurement. We changed the size of the records in the primary index to evaluate the effectiveness of the hints.

Figure 4a shows the lookup performance with different hint types. The lookup time with complete hints (⟨Page ID, Frame ID, Slot ID⟩) is the lowest configuration across all record sizes. In this scenario, the lookup time remains nearly constant, irrespective of the record size, as the organization of the primary index does not influence the retrieval time. Comparing ⟨Page ID, Slot ID⟩ and ⟨Page ID, Frame ID⟩, we can see that with small record sizes, the slot hint is more effective than the frame hint, while with larger record sizes, their performance differences diminish. This is because with small record sizes, binary searching the records in the primary index leaf page is expensive, and slot hints can reduce the overhead of binary search. However, with larger record sizes, the overhead of binary search is less significant, and the overhead of accessing the page-to-frame mapping is more significant because the number of pages active in the buffer pool increases. In the No-hint case, the lookup time increases with the record size because primary index grows with the record size. From record size 500 to 1000, the level of the tree increases from 3 to 4, adding one page access per lookup. Although the overhead associated with the binary search in the primary index leaf page diminishes as record size increases, the total number of pages in the primary index grows, thereby elevating the fill factor of its internal nodes. This increased fill factor, in turn, raises the overhead of performing binary searches at the internal nodes, ultimately prolonging the lookup time slightly in the No-hint case as record size increases. Overall, these experiments show that in the best case, LIPAH offers significant improvements to index lookups when the hints are accurate.

Lookup with Stale Hints. The performance benefits of LIPAH are predicated on the hints being correct. However, in practice, hints may become stale as described in Section 4.4. Therefore, we evaluate the performance of lookup operations in the secondary index where hints are intentionally rendered incorrect. Analogous to the preceding experiment, both a primary index and a corresponding secondary index were constructed, which contains 1, 000, 000 entries. Here, complete hints (⟨Page ID, Frame ID, Slot ID⟩) are embedded within the leaf nodes of the secondary index. Prior to executing the lookup operations, a predetermined percentage of the hints was deliberately invalidated by subtracting 1 from the correct hints. The aggregate time required to perform lookup operations on all keys in a random order was recorded. Each key was accessed only once during the experiment, thereby eliminating any potential benefits from subsequent hint repair.

Figure 4b illustrates the lookup performance when utilizing stale hints. As the proportion of stale hints increases, the overhead associated with the lookup operation correspondingly rises. A comparative analysis of page, frame, and slot hints reveals that the page hint has the most significant impact on lookup performance. This effect is attributed to the reduction in page accesses achieved by bypassing the root-to-leaf traversals in the primary index; each access to an internal node mandates a binary search within that node. In contrast, a stale frame hint imposes minimal overhead relative to the other hint types. Here, a single thread is employed, thereby eliminating any contention within the buffer pool. The overhead incurred by a stale frame hint is limited to a single lookup operation on the page-to-frame mapping, which is computationally inexpensive. Conversely, a stale slot hint introduces greater overhead compared to a stale frame hint, as it necessitates a binary search on the leaf node of the primary index. When all elements of the hint are incorrect, the performance degradation is most pronounced. In this scenario, the overall cost is primarily attributable to the outdated page ID and frame ID. Notably, the cost associated with the slot hint is subsumed under the cost of the stale page ID because, in our implementation, if a stale page is detected, the slot hint is disregarded, given the low probability that the slot hint is correct when the page hint is erroneous. This experiment is designed to show performance degradation, as the percentage of read operations uses an invalid hint where any repair is not used in the future. We believe that this case is extremely unlikely for most workloads, but still demonstrates that the performance penalty is graceful and does not dominate the operation cost.



(a) Lookup Latency with Different Hint Types Under Varying Record Size — Varying the record size affects the overhead of binary search in the leaf page of the primary index, number of pages in the page-to-frame mapping in the buffer pool, and the fill factor of certain pages in the primary index.

(b) Lookup Latency with Complete Hints (⟨Page ID, Frame ID, Slot ID⟩) Under Varying Number of Stale Hints. — The number of stale hints is set before the lookup operations. Hints not indicated in the legend were remained as correct.

Figure 4: Secondary Index Lookup Performance: (a) Correct Hints Under Varying Record Sizes and (b) Complete Hints Under Increasing Staleness.

5.3 Robustness of Hints to Data Modifications

We now assess the robustness of each hint type in the presence of various data modifications. Hints that remain stable despite insertions, updates, and deletions are more likely to preserve their performance benefits over time. To quantify this stability, we simulate data modifications within the primary index and subsequently evaluate their impact on the hints stored in the secondary index. Initially, a primary index and a corresponding secondary index were constructed with $1,000,000$ entries. Each secondary index entry contained complete hints (⟨Page ID, Frame ID, Slot ID⟩) with no stale hints, ensuring that all hints accurately referenced their corresponding records in the primary index. To evaluate the robustness of these hints under modifications to the primary index, a designated number of records were inserted, updated, and deleted from both indexes, and the number of valid hints in the secondary index was subsequently recorded. Specifically, for insertions, a percentage of the original number of records was added to both indexes; for updates, a percentage of the original records in the primary index was modified to values ranging from 0 to twice the original size; and for deletions, a percentage of the original records was removed from both indexes. These operations induce splits and merges in the primary index nodes that potentially invalidate some hints in the secondary index.

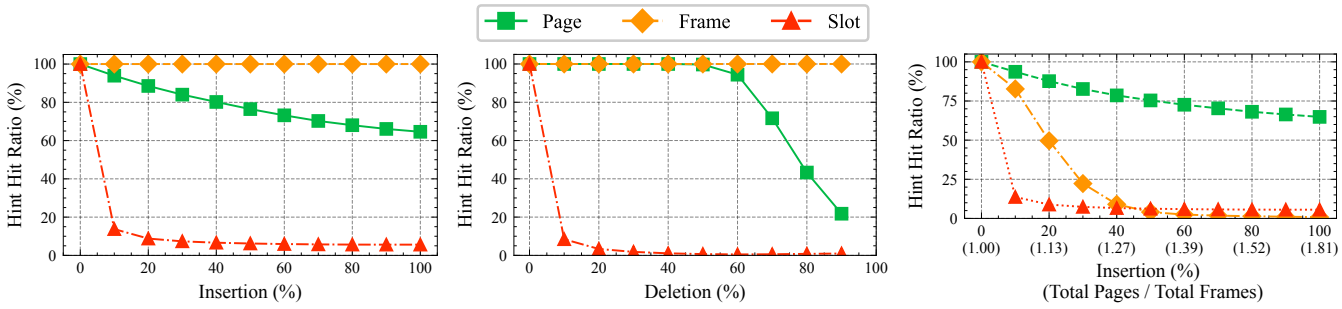
For these experiments, we evaluate the **hint hit ratio** which is defined as the number of valid hints divided by the total number of entries in the secondary index. We ran the experiment in two scenarios: (1) an in-memory scenario, where the buffer pool is large enough to accommodate both the primary and secondary indexes, and (2) a larger-than-memory scenario, where the buffer pool is sized to fit only the initial primary and secondary indexes, causing pages to be swapped in and out of memory as modifications occur. It is worth noting that these experiments cause a high number of invalidations due to the use of a primary tree index with a high percentage of modifications.

In-Memory Scenario. Figures 5a and 5b show the hint hit ratio of the secondary index under the aforementioned data modifications. Here, frame hints do not become invalid, as all the pages reside in memory and the frame contains the hinted page as pages are not evicted. As the number of records inserted increases, the number of valid page and slot hints decreases. When the insertion percentage reaches 100% (i.e., the number of records is doubled in the primary index), the number of valid page hints decreases by 35%. The number of valid slot hints declines more rapidly than that of valid page hint. This discrepancy arises because page hints become invalid only when the primary index expands sufficiently to trigger a leaf page split, whereas slot hints are invalidated upon the insertion of a new record into a leaf page, which necessitates a reorganization of the slot array to preserve sorted order.

An alternative approach could be to use the record's offset from the beginning of the page as the hint instead of its slot ID, rendering the hint less susceptible to invalidation. This approach sounds viable, as the record's offset remains stable even if the slot array is modified due to insertions—except in cases where a page compaction or reorganization operation (e.g., a page merge or split) occurs. However, employing the offset introduces a new challenge: if an operation alters a record's offset and another value subsequently occupies the old offset position, it becomes unsafe to read the data at the offset speculatively. In such cases, the value at the offset may appear to be valid, but it might correspond to an entirely different data entry. We therefore do not consider this approach.

In terms of deletions, the number of valid page hints does not drop until around 60% of the records are deleted. This is because page merges are triggered lazily. As with insertions, the slot hints are more susceptible to deletions than the page and frame hints.

Larger-than-Memory Scenario. Figure 5c presents the stability of hints under insertions in a larger-than-memory scenario. The experiment demonstrates that page hints remain robust under modifications, except when the primary index is significantly altered. In contrast, slot hints are vulnerable to invalidation even with minimal insertions, and frame hints become invalid when the working set exceeds the buffer pool. Nonetheless, stale slot and frame hints pose minimal issues since validation is performed in memory-slot validation merely compares keys, while frame validation confirms the presence of a page. The potential cost lies in repairing hints after invalidation, and the primary consideration is the impact on performance with modifications. We evaluate both next.



(a) Insertion (In-Memory) **(b) Deletion (In-Memory)** **(c) Insertion (Larger-than-Memory)**

Figure 5: Hint Hit Ratio of the Secondary Index Under Various Data Modifications: (a, b) In-Memory and (c) Larger-than-Memory — The hint hit ratio is defined as the number of valid hints divided by the total number of keys in the secondary index. Update operation results on primary index records are omitted from this figure, as their impact was negligible, with the hint hit ratio remaining above 98% after these modifications.

5.4 Overhead of Repairing Stale Hints

When hints become stale due to data modifications, the system can either repair invalid hints when encountered or ignore them. The repair strategy updates hints during lookup operations, benefiting future accesses but incurring immediate overhead. The no-repair strategy avoids this overhead but leads to cumulative performance degradation.

To evaluate the robustness of LIPAH in the presence of stale hints and the subsequent repair of these hints, we measure the impact on secondary index lookup latency. Here we compare three configurations: repairing stale hints, ignoring them (leaving stale for the next read), and no hint. Our goal is to quantify the cost and benefit of accessing a stale hint and repairing it. A primary index and a corresponding secondary index containing $1,000,000$ entries are constructed. The secondary index includes complete hints ($\langle$Page ID, Frame ID, Slot ID$\rangle$) for each entry. Our workload consists of key lookups in the secondary index that retrieve records from the primary index and an operation deliberately invalidates a hint by decrementing a random hint value (page, frame, or slot) by one. Note that an invalid slot results in fresh search for the given page and the 'locality' of the hint is not a factor—with the same applied for the page and frame hints. Here, the buffer pool is large enough to accommodate both the primary and secondary indexes.

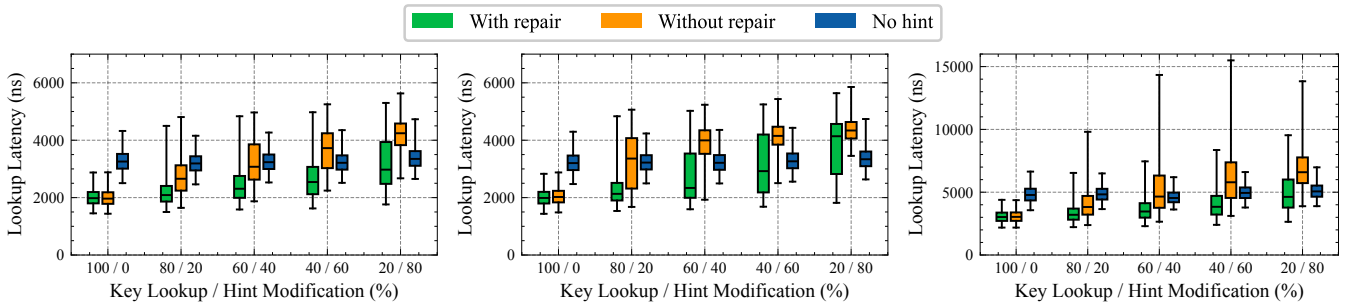
Figure 6 illustrates the lookup latency when hint modifications are interleaved with key lookups. A thread probabilistically selects an action: when a hint modification is chosen, the hint is invalidated according to a predefined ratio (e.g., 10% for page, 10% for frame, etc.). With repair, the lookup operation corrects any invalid hints, and valid hints are retained for future accesses. Without repair, invalid hints are left and subsequent reads follow the same invalid hint. We measured latency of 10 million key lookups, with latencies summarized in a box-and-whisker plot (whiskers at the 2nd and 98th percentiles).

Single Thread Scenario. Figure 6a presents latency results for a hint modification ratio of page 10%, frame 10%, and slot 80% using a single thread. On average, the repair strategy outperformed both the non-repair and baseline cases; however, at high hint modification rates, the interquartile range (25-75%) increased due to the latency difference between accessing records with valid and invalid hints. In the absence of repair, the accumulation of invalid hints resulted in progressively degraded performance. Under a 20/80 key lookup to hint modification ratio, performance disparity between with repair and without repair arises primarily from accesses using incorrect page IDs. At the end of the experiment, only 1.3% of page hints remained correct without repair, compared to 71.7% with repair. Thus, even when slot IDs are frequently invalidated, the repair approach maintains superior performance provided a sufficient number of page hints remain valid.

Figure 6b demonstrates lookup latency for a hint modification ratio of page 50%, frame 10%, and slot 40%. Here, latency for both with repair and without repair increased rapidly due to frequent page ID invalidations. Although the repair strategy maintained an advantage over the no-hint baseline in terms of the median value up to a 40/60 lookup-to-modification ratio, it exhibited a larger interquartile range when executed with hint modifications. At the end of 20/80 experiment, 0% of page hints remained correct without repair, whereas 33.6% were valid with repair; the repair approach incurred additional latency when over 60% of reads involved invalid page hints.

Multi-Threaded Scenario. Figure 6c shows the lookup latency with a hint modification ratio of page 10%, frame 10%, and slot 80% using 10 threads. The trends observed in the single-threaded experiment (Fig. 6a) are also evident here, with the repair approach outperforming both the no-repair and no-hint configurations for median latency. However, due to contention among threads, such as accessing the root page of the index or the page-to-frame mapping, the latency increases. Without repair becomes increasingly detrimental as following a stale hint leads to more costly lookups. The repair approach, while incurring some additional overhead, provides a more stable performance profile than without repair and generally maintains better performance than the no-hint configuration.

In practical workloads—where significant numbers of inserts or deletes (necessary for page splits and merges) are infrequent—the negative impact of invalid hints is reduced. However, if such modifications become prevalent, the use of page hints may not



(a) Page 10 %, Frame 10 %, Slot 80 %
1 thread

(b) Page 50 %, Frame 10 %, Slot 40 %
1 thread

(c) Page 10 %, Frame 10 %, Slot 80 %
10 threads

Figure 6: Latency of Key Lookup Operations Mixed With Hint Modification Operations Comparing With Repair, No Repair, and No Hint — A hint modification operation invalidates the hint based on a predefined ratio, denoted in the caption of each figure. This operation is mixed with the key lookup operations. With repair, an invalid hint is fixed when it is encountered. Without repair, it is ignored and left invalid for the next access. No hint configuration does not use any hints.

Table 1: Secondary Index Size Analysis – All values are in pages except for the last column (MiB).

Hint Type	Root	Inner	Leaf	Total	MiB
No hint	1	19	4058	4068	63.56
Page	1	21	4545	4567	71.36
Page, Frame	1	23	5025	5049	78.89
Page, Slot	1	23	5025	5049	78.89
Page, Frame, Slot	1	26	5524	5551	86.73

be advisable. Overall, our experiments demonstrate that while the with-repair approach can exhibit a larger interquartile range than without-repair and no-hint, they generally preserve superior lookup performance to no hints when the proportion of invalid page hints remains moderate.

5.5 Storage Overhead of Hints

One cost of LIPAH is the additional space consumed by the hint. Here we evaluate the storage overhead of embedding hints into a secondary index by comparing a baseline configuration with no hints against with hints (⟨Page ID⟩, ⟨Page ID, Frame ID⟩, ⟨Page ID, Slot ID⟩, and ⟨Page ID, Frame ID, Slot ID⟩.) For all the configurations, a primary and a secondary index using 1,000,000 entries with a one-to-one correspondence between the indexes were built.

Table 1 summarizes the result. For all hint types, the secondary index has three levels: root, inner, and leaf. In the No-hint configuration, the secondary index occupied 4068 pages (63.56 MiB). With complete hints (⟨Page ID, Frame ID, Slot ID⟩), each secondary index entry grew from a payload of 10 bytes (just the primary key) to 22 bytes (the primary key plus three 4-byte hints), and the page count increased to 5551 (86.73 MiB). This represents an overall size increase by a factor of about 1.36—even though the per-entry payload more than doubled.

6 CONCLUSION

In this paper, we propose and evaluate the potential for indirection skipping in transaction processing systems through the use of LIPAH. LIPAH utilizes a composite reference consisting of a logical ID and a physical address to identify objects. The physical address serves as an initial hint for object location, while the logical ID functions as a fallback when the physical address becomes invalid. This design renders LIPAH resilient to physical address invalidations and facilitates efficient indirection skipping with minimal overhead in reference maintenance.

Our implementation of LIPAH across primary indexes, secondary indexes, and transaction read/write sets demonstrates that the approach can enhance the performance of TPC-C-like workloads by up to 19.7× compared to baseline systems when using 40 threads. Unlike pointer swizzling, LIPAH requires no ownership tracking of references; unlike vmcache, it has fewer kernel dependencies and maintains metadata proportional to buffer pool size rather than disk size. Its simplicity also supports incremental adoption in existing systems.

We believe that LIPAH is applicable to many systems in which objects dynamically change locations, as opposed to remaining fixed in a direct-mapped cache. In such systems, the physical location of the object is tracked by a table that maps the object's logical name to its corresponding physical location, which is likely to become a bottleneck during system scaling. With LIPAH in place, the system can bypass the indirection layer and directly access the object, thereby reducing the overhead associated with determining the object's location.

ACKNOWLEDGMENTS

This work was in part supported by NSF Award IIS-2048088 and a Google Data Analytics and Insights (DANI) Award. The authors would also like to thank the anonymous reviewers for their valuable feedback.

REFERENCES

[1] 2024. Indirection. *Wikipedia* (July 2024). https://en.wikipedia.org/w/index.php?title=Indirection&oldid=1232446174

[2] 2024. MySQL :: MySQL 8.4 Reference Manual :: 17.6.2.1 Clustered and Secondary Indexes. https://dev.mysql.com/doc/refman/8.4/en/innodb-index-types.html

[3] 2024. Postgres/Src/Backend/Storage/Buffer/README at Master · Postgres/Postgres. https://github.com/postgres/postgres/blob/master/src/backend/storage/buffer/README

[4] 2024. TPC-C Homepage. https://www.tpc.org/tpcc/

[5] 2024. Xacrimon/Dashmap: Blazing Fast Concurrent HashMap for Rust. https://github.com/xacrimon/dashmap

[6] The PostgreSQL Global Development Group . 2024. 65.7. Heap-Only Tuples (HOT). https://www.postgresql.org/docs/17/storage-hot.html

[7] Adnan Alhomssi and Viktor Leis. 2023. Scalable and Robust Snapshot Isolation for High-Performance Storage Engines. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1426–1438.

[8] Panagiotis Antonopoulos, Peter Byrne, Wayne Chen, Cristian Diaconu, Raghavendra Thallam Kodandaramaih, Hanuma Kodavalla, Prashanth Purnananda, Adrian-Leonard Radu, Chaitanya Sreenivas Ravella, and Girish Mittur Venkataramanappa. 2019. Constant Time Recovery in Azure SQL Database. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2143–2154.

[9] Donald D. Chamberlin and Raymond F. Boyce. 1974. SEQUEL: A Structured English Query Language. In *SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. 249–264.

[10] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Communications of The Acm* 13, 6 (June 1970), 377–387.

[11] Cristian Diaconu, Craig Freedman, Erik Ismert, Paul Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD*.

[12] Wolfgang Effelsberg and Theo Haerder. 1984. Principles of Database Buffer Management. *ACM Transactions on Database Systems* 9, 4 (Dec. 1984), 560–595.

[13] Michael Freitag, Alfons Kemper, and Thomas Neumann. 2022. Memory-Optimized Multi-Version Concurrency Control for Disk-Based Database Systems. *Proc. VLDB Endow.* 15, 11 (July 2022), 2797–2810.

[14] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2008. *Database Systems: The Complete Book* (2 ed.).

[15] Vibby Gottemukkala and Tobin J. Lehman. 1992. Locking and Latching in a Memory-Resident Database System. In *VLDB*. 533–544.

[16] Goetz Graefe. 2004. Write-Optimized b-Trees. In *VLDB*. 672–683.

[17] Goetz Graefe, Hideaki Kimura, and Harumi Kuno. 2012. Foster B-Trees. *ACM Transactions on Database Systems* 37, 3, Article 17 (Sept. 2012).

[18] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi Kuno, Joseph Tucek, Mark Lillibridge, and Alistair Veitch. 2014. In-Memory Performance for Big Data. *Proc. VLDB Endow.* 8, 1 (Sept. 2014), 37–48.

[19] Janis Greenberg and et al. 2024. Design Considerations for REFs. https://docs.oracle.com/en/database/oracle/oracle-database/23/adobj/design-considerations-for-REFs.html#GUID-79DD95A2-3080-47BC-95FB-7FD42D1E1BBF

[20] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP through the Looking Glass, and What We Found There. In *SIGMOD*. 981–992.

[21] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2022. Opportunities for Optimism in Contended Main-Memory Multicore Transactions. *The VLDB Journal* 31, 6 (Nov. 2022), 1239–1261.

[22] Donna Keesling and et al. 2024. Indexes and Index-Organized Tables. https://docs.oracle.com/en/database/oracle/oracle-database/21/cncpt/indexes-and-index-organized-tables.html#GUID-1A9D370B-12F0-4161-875E-3121C8DEF2AD

[23] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *SIGMOD*. 1675–1687.

[24] Evan Klitzke. 2016. Why Uber Engineering Switched from Postgres to MySQL. https://www.uber.com/en-BR/blog/postgres-to-mysql-migration/

[25] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems* 6, 2 (June 1981), 213–226.

[26] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *Proc. VLDB Endow.* 5, 4 (Dec. 2011), 298–309.

[27] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-Memory Assisted Buffer Management. *Proc. ACM Manag. Data* 1, 1, Article 7 (May 2023).

[28] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-memory Data Management beyond Main Memory. In *ICDE*. 185–196.

[29] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of Practical Synchronization. In *DaMoN*. Article 3.

[30] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core in-Memory Transactions. In *SIGMOD*. 21–35.

[31] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-Value Storage. In *EuroSys*. 183–196.

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

[33] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD*. 677–689.

[34] Riki Otaki, Jun Hyuk Chang, Charles Benello, Aaron J. Elmore, and Goetz Graefe. 2025. Resource-Adaptive Query Execution with Paged Memory Management. In *CIDR*.

[35] Andrew Pavlo and Bohan Zhang. 2023. The Part of PostgreSQL We Hate the Most // Blog // Andy Pavlo - Carnegie Mellon University. https://www.cs.cmu.edu/~pavlo/blog/2023/04/the-part-of-postgresql-we-hate-the-most.html

[36] Kun Ren, Alexander Thomson, and Daniel J. Abadi. 2015. VLL: A Lock Manager Redesign for Main Memory Database Systems. *The VLDB Journal* 24, 5 (Oct. 2015), 681–705.

[37] A. Silberschatz, H.F. Korth, and S. Sudarshan. 2020. *Database System Concepts*.

[38] Michael Stonebraker and Joseph M. Hellerstein. 2005. What Goes Around And Comes Around. In *Readings in Database Systems* (4th ed.). 2–41.

[39] Linux Kernel Team. 2024. Efficient IO with Io_uring.

[40] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *SOSP*. 18–32.

[41] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *SIGMOD*. 1629–1642.