Indexing for Concurrency-Aware Distributed Joins

A Study of Index Structures, Concurrency Control, and Distributed Joins

Technical Writing Report

submitted in partial fulfillment of the requirements
for the award of the

Bachelor of Technology

in

Information Technology

Submitted by

Soubhik Kumar Gon
(B422056)

Hitanshu Satpathy
(B422028)

Amit Bhagat
(B422007)

Under the supervision of

Dr. Rakesh Chandra Balabantaray

Department of Computer Science and Engineering

Department of Computer Science and Engineering

International Institute of Information Technology, Bhubaneswar

November, 2025

CERTIFICATE

This is to certify that the report entitled **"Concurrency-Aware Indexing for Efficient Distributed Joins in Modern Databases"** submitted by **Soubhik Kumar Gon (B442056), Hitanshu Satpathy (B442028), Amit Bhagat (B442007)** to **International Institute of Information Technology** is a record of bonafide work carried out under my supervision and guidance.

The report is submitted for end-semester evaluation of the B.Tech Technical Writing coursework and represents original work by the student.

To the best of my knowledge, the matter embodied in this report has not been submitted to any other institution for the award of any degree or diploma.

Date: ______________

Dr. Rakesh Ch. Balabantaray
Supervisor
Department of CSE

DECLARATION

We hereby declare that:

1. The work presented in this report is original and has been carried out by us under the general supervision of our advisor.
2. This work has not been submitted, either in part or in full, to any other institution for any degree or diploma.
3. We have adhered to the Institute's Ethical Code of Conduct throughout the course of this work.
4. All external materials (data, analysis, figures, and text) have been properly cited and acknowledged.
5. All quoted materials from other sources have been placed in quotation marks and cited appropriately.
6. We have followed all official guidelines and regulations in preparing this report.

Soubhik Kumar Gon
B422056
B.Tech IT

Hitanshu Satpathy
B422028
B.Tech IT

Amit Bhagat
B422007
B.Tech IT

Date: ______________

Place: ______________

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to all those who have contributed to the successful completion of this technical report.

First and foremost, I am deeply grateful to my supervisor, **Dr. Rakesh Chandra Balabantaray**, for their invaluable guidance, constant encouragement, and insightful feedback throughout this project. Their expertise and dedication have been instrumental in shaping this work.

I would like to extend my heartfelt thanks to the faculty members of the Department of Computer Science and Engineering at **International Institute of Information Technology** for providing an excellent academic environment and access to necessary resources and facilities.

I am also thankful to my peers and colleagues for their constructive discussions, suggestions, and moral support during the course of this project. Their collaboration and friendship made this journey more enriching and enjoyable.

Special thanks to the library and technical staff for their assistance in accessing research materials and computational resources.

Finally, I would like to express my deepest appreciation to my family for their unconditional love, patience, and support throughout my academic journey. Their encouragement has been my constant source of motivation.

ABSTRACT

The rising need for **low-latency processing** of very large datasets has pushed modern data management systems toward **distributed, shared-nothing architectures**. In such systems, the **join operation** often becomes the most expensive step, particularly in **Hybrid Transactional and Analytical Processing (HTAP)** workloads. Earlier studies linked the cost of joins to hashing or sorting performance, but modern environments face a different challenge. With in-memory execution, multi-core processors, and fast networks, the main limitations now come from **concurrency control**. Issues such as **latch contention**, **memory access delays**, and **data skew** can significantly limit scalability and slow down distributed joins.

This report examines **Concurrency-Aware Indexing** as an approach to address these problems. We compare traditional index structures with more recent **latch-free** and **optimistic** designs, including the **BW-Tree**, **Masstree**, and **Adaptive Radix Tree (ART)**, and study how they behave in distributed settings. Based on these observations, we introduce a framework that combines **lock-free indexing**, **skew-resistant partitioning**, and **hardware-assisted execution** to support more scalable and efficient distributed joins.

Keywords: Distributed Joins, Concurrency-Aware Indexing, Latch-Free Structures, Bw-Tree, Masstree, Adaptive Radix Tree, HTAP, Distributed Databases, Concurrency Control, Data Skew, Partitioning, RDMA, In-Memory Databases, Scalability.

Contents

1 Introduction 7

1.1 Background and Motivation 7

1.2 Problem Statement 8

1.3 Research Objectives 9

1.4 Scope and Organization 9

2 Literature Survey 10

2.0.1 Concurrency Bottlenecks in Classical B+ Tree Designs 10

2.0.2 The Latch-Free Shift: The BW Tree 11

2.0.3 Trie Based and Hybrid Architectures 12

2.0.4 LSM Trees and Their Core Trade-offs 12

2.0.5 Effects of Transaction Models on Indexing 13

2.0.6 Hardware Acceleration: RDMA, GPUs, and FPGAs 13

3 Problem Statement 15

3.0.1 The Index Contention Bottleneck 15

3.0.2 The Skew and Hotspot Challenge 15

3.0.3 Network Latency and Topology Mismatch 16

4 Solution 17

4.0.1 Adaptive Latch-Free Local Index 17

4.0.2 Skew-Resilient Partitioning via HeavyLocker 18

4.0.3 RDMA-Accelerated Batched Lookups 19

5 Theoretical Analysis 22

5.0.1 Correctness and Linearizability 22
5.0.2 Adaptive Query Execution (AQE) 23
5.0.3 Cost Model: Network vs. CPU 24

6 Results and Discussion 25

6.1 Index Throughput Under Contention 25
6.2 Impact of Skew on Distributed Joins 26
6.3 Hardware Acceleration Impact 27
6.4 Future Work 28

7 Conclusion 30

Chapter 1

Introduction

1.1 Background and Motivation

The architecture of modern data management systems is undergoing a significant shift. As organizations handle ever-growing volumes of data and demand **near-real-time processing**, traditional monolithic databases are increasingly unable to keep up. The surge in data volume, velocity, and variety has encouraged a move toward **distributed, shared-nothing systems** that can scale horizontally across clusters of commodity machines.

Within these distributed environments, the **join operation** plays a central role. Joins are essential for reconstructing relationships across normalized data, yet they remain one of the most resource-intensive tasks in any large-scale system. The rise of **Hybrid Transactional and Analytical Processing (HTAP)** workloads has amplified this challenge. As systems attempt to support simultaneous transactional and analytical demands, the performance of distributed joins becomes a crucial determinant of overall system efficiency.

Historically, the cost of joins was explained through algorithmic factors such as hashing or sorting. However, modern in-memory systems running on multi-core hardware and high-speed networks face a different bottleneck. The primary limitations now come from **concurrency control** and **memory-access latency**, rather than disk I/O. According to the Universal Scalability Law, throughput improves with increased parallelism only up to a point; beyond that, performance is constrained by **serialization** and **coordination overhead**. In practice, these limitations appear most clearly as **latch contention** within index structures.

Traditional indexes like the B+ Tree were designed for an era where disk accesses dominated computation time. In such systems, the cost of acquiring a latch was negligible. Today, when data resides in DRAM and networks offer microsecond-level communication through technologies like RDMA, even small amounts of contention can severely degrade performance. A single contended cache line can cause delays across cores, and poor data distribution can lead to **hot spots**, overwhelming nodes and triggering widespread transaction aborts.

These trends highlight the need for index structures that are inherently **concurrency-aware**. Instead of relying on heavy latching, modern designs embed lock-free or optimistic synchronization directly into the index itself. This allows systems to exploit multi-core hardware more effectively and achieve higher levels of parallelism without serialization bottlenecks.

1.2 Problem Statement

Although distributed databases have advanced significantly, most traditional indexing mechanisms struggle under modern concurrency demands. Classical structures such as the B+ Tree introduce contention points that limit scalability, especially when used for high-frequency join operations across distributed nodes. Problems such as latch contention, skewed access patterns, and memory-access delays restrict throughput and reduce the benefits of horizontal scaling.

Furthermore, as HTAP workloads become more common, systems must support both transactional updates and analytical queries without compromising latency. Existing indexing approaches often fail to deliver this balance. In distributed environments, even small inefficiencies in index-level concurrency control can cascade, affecting query performance, resource usage, and overall system responsiveness.

The challenge, therefore, is to design and evaluate indexing mechanisms that remain efficient under high concurrency, handle non-uniform data distributions, and support large-scale distributed join processing without introducing additional coordination overhead.

1.3 Research Objectives

This report aims to explore and analyze **Concurrency-Aware Indexing** as a solution to the challenges described above. The primary objectives of this study are:

1. To examine the limitations of traditional index structures in modern distributed environments.
2. To study modern **latch-free**, **optimistic**, and **lock-free** index designs such as the BW-Tree, Masstree, and Adaptive Radix Tree (ART).
3. To analyze how concurrency-aware indexes influence the performance of distributed joins in large-scale systems.
4. To evaluate concurrency models used by real-world distributed databases such as Google Spanner, TiKV, and FoundationDB.
5. To propose and describe a unified framework that integrates lock-free indexing, skew-resistant partitioning, and hardware-assisted execution.

1.4 Scope and Organization

This report focuses on the role of indexing in improving the performance of distributed joins under high concurrency. The scope includes:

- Analysis of traditional and modern index structures.
- Study of concurrency bottlenecks in distributed systems.
- Examination of latch-free and optimistic synchronization methods.
- Evaluation of real-world distributed databases and their concurrency models.
- Proposal of a framework as a comprehensive solution.

Topics such as storage-tier optimizations, non-volatile memory indexing, or specialized analytical join algorithms fall outside the primary scope of this report but are referenced when necessary for context.

Chapter 2

Literature Survey

Modern distributed database systems have shifted from disk-focused designs to architectures centered on in-memory execution and large-scale parallelism. As data volumes grow and workloads become increasingly mixed, the join operation often becomes the determining factor for end-to-end performance. While early research concentrated on reducing disk I/O through balanced tree structures and buffering, recent studies show that concurrency overhead, skew, and memory-level interactions now dominate the cost of distributed joins. This section reviews the progression of indexing techniques, their concurrency characteristics, and the broader transaction and hardware trends that shape indexing performance in distributed environments.

2.0.1 Concurrency Bottlenecks in Classical B+ Tree Designs

The B+ Tree has long been the default indexing structure in relational databases due to its predictable height and block-oriented design. However, its concurrency model struggles on modern multi-core hardware. Standard latch-coupling protocols require each thread to acquire read latches while traversing internal nodes. Even with reader-writer locks, every lock acquisition triggers an atomic write, causing cache-line movement across cores. On systems with many workers probing the tree, the root latch becomes a hotspot and limits scalability. Million-scale probe operations, common in distributed joins, can effectively serialize on this latch. Optimistic schemes like Optimistic Lock Coupling reduce some of this contention but still incur validation overhead and restart costs during structural modifications.

2.0.2 The Latch-Free Shift: The BW Tree

The Bw-Tree introduces a latch-free design aimed at eliminating these bottlenecks. Its core idea is to avoid in-place modifications entirely. Instead of updating a node directly, the system creates a small delta record describing the change and links it to the existing node using a single atomic compare-and-swap (CAS). All nodes are accessed through a mapping table that translates logical page identifiers into physical addresses. This indirection allows the system to update node locations without rewriting parent pointers. While this approach successfully removes latching, it introduces new challenges. A heavily updated node may accumulate a long delta chain, increasing traversal time. Periodic consolidation merges the delta chain into a new base node, restoring read efficiency at the cost of additional CPU work.

Algorithm 1 Latch-Free Index Insert (CAS-Based)

```
function IndexInsert($Key\ k, Value\ v, Timestamp\ ts$)
    $Payload \leftarrow CreatePayload(k, v, ts)$
    loop
        $CurrentAddr \leftarrow MappingTable.Get(k)$
        $NewDelta \leftarrow AllocateNode()$
        $NewDelta.Type \leftarrow INSERT$
        $NewDelta.Payload \leftarrow Payload$
        $NewDelta.Next \leftarrow CurrentAddr$
                                                ▷ Atomic Compare-And-Swap
        $Success \leftarrow CAS(\&MappingTable[k], CurrentAddr, NewDelta)$
        if $Success$ then
            if $ChainLength(NewDelta) > Threshold$ then
                $TriggerAsyncConsolidation(k)$
            end if
            return true
        else
            $Backoff(RandomExponential())$
        end if
    end loop
end function
```

2.0.3 Trie Based and Hybrid Architectures

Research has also explored trie-inspired designs that improve cache locality and adapt better to variable-length keys.

Masstree organizes keys as a trie of small B+ Trees. Each layer handles a fixed-length slice of the key, and concurrency is managed through lightweight optimistic validation. Readers avoid locks entirely. They verify a node's version counter before and after accessing it and retry only if a conflict is detected. This allows read operations to scale with core count as long as write contention stays moderate.

Adaptive Radix Tree (ART) provides a high-performance in-memory index by replacing comparisons with radix-based navigation. ART uses adaptive node layouts that grow or shrink based on the number of children, reducing memory waste typical of tries. Its ROWEX synchronization model ensures that readers always proceed without blocking, while writers serialize modifications using atomic operations. The result is a structure that offers fast lookups and predictable behavior under concurrency.

2.0.4 LSM Trees and Their Core Trade-offs

Log-Structured Merge Trees dominate modern distributed key-value stores due to their high write throughput. They buffer incoming writes in memory and periodically flush them as immutable sorted files. Although this design converts random writes into sequential ones, it complicates point lookups. A single query may need to check multiple on-disk levels and the in-memory structure. Bloom filters reduce some overhead, but false positives and memory pressure remain concerns. LSM Trees also require continuous background compaction to merge and reorganize data. Compaction can introduce latency spikes, creating unpredictable performance during distributed joins. Even optimized designs like WiscKey improve I/O locality but do not fully address the inherent read amplification of the LSM approach.

2.0.5 Effects of Transaction Models on Indexing

Index behavior in distributed joins is closely tied to the underlying consistency model and transaction protocol.

Google Spanner uses tightly synchronized clocks to provide global consistency. Its interleaving design, which colocates related rows, enables efficient local joins. However, joins involving unrelated tables must fall back to distributed operators that are sensitive to network delays.

TiDB and TiKV implement a Percolator-inspired model with multi-version concurrency control and a dedicated timestamp oracle. The two-phase commit protocol introduces locking during prewrite operations. If a join encounters keys locked by transactions in progress, the system must wait or retry, increasing latency. Under high contention, optimistic transactions may see frequent aborts.

FoundationDB relies on optimistic concurrency control with centralized conflict resolution. Transactions submit their read and write sets for conflict checking. In workloads where many workers read and write overlapping key ranges, the probability of aborts rises sharply. Hot keys can become serialization points, limiting throughput.

2.0.6 Hardware Acceleration: RDMA, GPUs, and FPGAs

Recent work explores how specialized hardware can assist join and index operations.

RDMA enables direct memory access across nodes without involving remote CPUs. Although this reduces latency for simple operations, pointer-heavy structures like trees require multiple round trips for each level. This can offset the benefits of high bandwidth.

GPUs offer large-scale parallelism for hash-based joins. Their performance depends on minimizing branch divergence and maximizing memory throughput. The primary barrier is data transfer across the PCIe bus, which restricts GPU acceleration to cases where data fits in GPU memory.

FPGAs are increasingly used for computational storage. They can filter or probe data as it moves from SSD to main memory, reducing the workload on the CPU during join processing. While promising, FPGA solutions require careful hardware-software co-design and remain specialized to particular use cases.

Chapter 3

Problem Statement

Despite the wide range of indexing structures and hardware techniques surveyed earlier, achieving consistently efficient distributed joins remains difficult. Modern systems routinely encounter three interacting bottlenecks that prevent them from scaling cleanly across many cores and many nodes.

3.0.1 The Index Contention Bottleneck

During a distributed hash join or an index-nested loop join, millions of probe-side tuples may simultaneously access the index on the build side. This creates a level of concurrency that stresses even well-engineered data structures. If the index relies on pessimistic locking, such as a traditional B^+ Tree, threads begin to serialize on latches and overall throughput drops sharply as cores idle. Even with optimistic techniques like Masstree's version-based reads, heavy update activity can cause frequent validation failures. Readers observe inconsistent or "dirty" versions and are forced to retry. Under sustained load, this can spiral into a form of livelock where threads repeatedly restart without making progress. The cost of these aborted attempts adds up quickly and limits effective parallelism.

3.0.2 The Skew and Hotspot Challenge

Real datasets rarely follow uniform distributions. Many workloads exhibit Zipfian behavior where a small fraction of keys receive a disproportionate share of accesses. In a distributed join, standard hash partitioning places all tuples with the same join key on the same node. When a popular key becomes a "heavy hitter," the responsible node absorbs far more requests than its peers, creating a hotspot.

This leads to two related issues. First, the hotspot node's CPU and network interfaces become saturated while other nodes remain lightly loaded, producing classic straggler behavior. Second, the specific index entries corresponding to heavy keys encounter intense contention. Even latch-free structures, such as the Bw-Tree, can suffer from CAS contention on the mapping table or the head of a delta chain when many threads target the same key. These effects combine to stall progress and limit throughput at scale.

3.0.3 Network Latency and Topology Mismatch

Distributed systems must operate within the constraints of network latency. Accessing local DRAM may take on the order of tens of nanoseconds, while even a fast RDMA round trip requires several microseconds. Traditional index structures are neither NUMA-aware nor network-aware, and they implicitly assume that all memory accesses have similar cost. In a distributed setting, traversing an index over the network can require multiple round trips—one for the root and additional ones for internal nodes. If a thread holds a resource, such as a transaction lock, during these remote traversals, it blocks local progress for a significant period, increasing the likelihood of conflicts.

Similar issues appear within large shared-memory servers. Without awareness of NUMA boundaries, index nodes may be placed on distant sockets, forcing expensive cross-socket transfers. These interactions introduce latency that accumulates across many lookups.

Thesis: Achieving near-linear scalability in distributed joins requires an indexing approach that is not only **latch-free at the local level** but also **skew-resilient** and **topology-aware** across the entire distributed system.

Chapter 4

Solution

To overcome the limitations described earlier, we introduce a **solution framework** designed to make distributed joins scalable, contention-free, and topology-conscious. Rather than relying on a single data structure, the framework combines three coordinated layers: an **Adaptive Latch-Free Local Index**, a **Skew-Resilient Partitioning method** using advanced sketching techniques, and a **Hardware-Accelerated Execution path** for remote lookups. Together, these components form an integrated architecture aimed at maintaining high throughput even under extreme concurrency.

4.0.1 Adaptive Latch-Free Local Index

At the core of the framework lies a local index structure engineered to handle heavy probe and update traffic without resorting to locks. This component builds on the principles of the Bw-Tree but introduces improvements in reclamation, delta chain management, and NUMA awareness.

Delta Chain Optimization and Mapping Table

The BW-Tree's mapping table decouples node identity from physical memory, enabling lock-free updates via atomic CAS operations. Our framework extends this by implementing a **dynamic delta chain threshold**.

Instead of using a fixed value (e.g., a chain length of 8), the framework adjusts the consolidation frequency based on the current read/write workload. Under read-intensive joins, the structure consolidates more aggressively (e.g., at a depth of 4) to reduce traversal overhead.

Each delta record also embeds lightweight metadata (such as key range information or sibling pointers), allowing faster routing decisions without reconstructing entire base pages.

Epoch-Based Reclamation (EBR)

To safely reclaim memory without locks or costly reference-counting, the framework employs **Epoch-Based Reclamation**. A global epoch counter advances periodically.

Mechanism: When a thread begins an index operation, it "enters" the current epoch. If a node or delta chain is replaced, the obsolete memory is tagged with epoch E.

Reclamation: That memory is only freed once the global epoch surpasses $E + 2$, ensuring every reader that may have accessed the old memory has exited. This provides a lock-free, low-overhead memory management mechanism that scales comfortably across many cores.

NUMA-Aware Placement (NUMASK Logic)

Modern servers are inherently NUMA-based, yet many index structures treat all memory as uniform. Inspired by NUMASK-style designs, the index is divided into a **replicated upper tree** and **NUMA-local subtrees**.

The upper levels—frequently read but rarely updated—are copied to each socket's local memory, keeping traversal operations within local DRAM/L3 regions. This prevents cross-socket hops over QPI/UPI links, significantly lowering latency for high-frequency join probes.

4.0.2 Skew-Resilient Partitioning via HeavyLocker

A major barrier in distributed joins is skew: a few "heavy hitter" keys receive disproportionate traffic, overwhelming specific nodes. To counter this, the framework integrates a sketch-based technique known as **HeavyLocker**, enabling accurate, low-overhead detection of such keys.

The HeavyLocker Mechanism

HeavyLocker improves on classical sketches by using **lockable buckets** and a dynamic threshold that reacts to observed data volume.

When an item exceeds this threshold, its bucket "locks," ensuring the key's counter cannot be evicted by infrequent keys. The result is extremely accurate heavy-hitter identification without the overhead of heap-based algorithms like Space Saving.

Adaptive Partitioning Strategy

During the build phase of the join, a lightweight HeavyLocker instance processes incoming join keys:

- **Hot Keys:** Instead of mapping all heavy-hitter keys to a single hash partition, they are **replicated** across all (or selected) nodes. A dedicated read-only index replica stores these entries, removing write contention and allowing lock-free parallel reads.
- **Cold Keys:** These continue through normal hash partitioning.

This separation prevents hotspot formation and avoids the collapse of concurrency around a single index entry. By making heavy-hitter replicas read-only, the framework eliminates abort storms and OCC validation failures during the probe phase.

4.0.3 RDMA-Accelerated Batched Lookups

Network latency remains a dominating bottleneck in distributed join execution. The framework addresses this through **Bloom filter pushdown** and **batched RDMA reads**, dramatically reducing remote access frequency and cost.

Bloom Filter Pushdown (Sideways Information Passing)

Before initiating remote lookups, build-side nodes generate Bloom filters summarizing their key ranges. These filters are pushed to probe-side nodes. A probe thread checks this filter before contacting a remote server:

- A negative match allows immediate discard—eliminating a remote round trip.
- A positive match triggers a remote fetch.

Bloom filters are compact, and with NIC-level FPGA acceleration, membership checks can be performed at line rate, providing extremely efficient semi-join reduction.

Batched RDMA Reads

Instead of issuing an RDMA operation for each tuple, probe keys are collected into batches (e.g., 1 KB groups). Each batch triggers a single RDMA read to fetch the necessary remote index nodes.

Following the principles of optimistic validation used in systems like FaRM, the framework:

1. Reads the remote node's version counter via RDMA.
2. Fetches the node's payload.
3. Re-reads the version counter.

If both version reads match, the result is consistent; otherwise, the client retries. This shifts concurrency control to the network layer, reducing CPU load on the remote server and preventing queue buildup.

Algorithm 2 Batched RDMA Remote Lookup

```
1: function BatchRemoteProbe($Keys\ batch, RemoteNodeID\ target$)
2:     $RemoteAddrs \leftarrow LocalCache.GetAddresses(batch)$
3:     $WorkRequests \leftarrow List()$
4:     for $addr \in RemoteAddrs$ do
5:         $WR \leftarrow RDMA\_Read(addr, Size = NodeSize)$
6:         $WorkRequests.Add(WR)$
7:     end for
8:     $IB\_PostSend(target.QP, WorkRequests)$
9:     $Results \leftarrow IB\_PollCQ(target.CQ)$
10:    for $page \in Results$ do
11:        if $page.Header.Version \neq CachedVersion$ then
12:            $SlowPath\_RPC(page.Key)$                    ▷ Cache invalidation
13:        else
14:            $Process(page)$
15:        end if
16:    end for
17: end function
```

Chapter 5

Theoretical Analysis

This section explains why the proposed concurrency-aware indexing approach is correct, how it behaves when many operations run at the same time, and what its practical performance costs are. Each idea is broken down in simple terms.

5.0.1 Correctness and Linearizability

A database index must return correct results even under heavy concurrency. Two properties are essential:

- **Correctness**: the structure should never lose a key, return a half-updated entry, or show inconsistent data.
- **Linearizability**: every operation (insert, read, update) should appear as if it happened instantly at one specific point in time, even if many threads are running in parallel.

To explain how modern index structures achieve these guarantees, we break down two examples often used in high-concurrency systems.

BW-Tree Correctness: The BW-Tree avoids locking. Instead of modifying a node directly, it creates small "delta" records that describe the change. A global mapping table stores a pointer to the latest version of each node.

- Any update replaces the pointer in this mapping table.
- This replacement uses a single atomic operation (compare-and-swap), which succeeds only if no one else changed the pointer in the meantime.
- If the update succeeds, every thread immediately sees the new, correct version.

- Readers reconstruct the node by following the delta records in order.

Because the mapping table is the single source of truth, and its changes are atomic, the structure behaves as if updates happen instantly at one point in time, giving linearizability.

Masstree Correctness: Masstree uses a small version counter inside each node to detect changes. This counter includes bits indicating whether a writer is active.

- A reader checks the version counter before and after reading a node.
- If the counter changes during the read, it means a writer overlapped with the read, and the reader retries.
- This ensures a reader either sees the state before an update or after an update, but never a corrupted or partially written state.

This simple check guarantees that no keys are lost and no inconsistent view is returned.

5.0.2 Adaptive Query Execution (AQE)

Traditional query optimizers decide the join strategy before execution based on static statistics. If the actual data distribution changes during execution, the plan cannot adapt.

Our framework uses **Adaptive Query Execution (AQE)** to adjust its behavior at runtime. It observes two key indicators:

- the success rate of Bloom filter checks (how often they help skip irrelevant rows),
- the failure or retry rate of index lookups (which reveals contention).

A lightweight monitoring component tracks which keys are becoming "hot" (frequently accessed). If the pattern of hot keys changes, the system can adjust the join strategy during execution, for example, switching from hashing a key to broadcasting it when that becomes cheaper.

This idea is similar to "Eddies" in stream processing, where tuples are dynamically routed based on current system behavior.

5.0.3 Cost Model: Network vs. CPU

In distributed systems, performance depends heavily on whether data is accessed locally or over the network.

Let:

- C_{local} = cost of a local index lookup,
- C_{net} = cost of a network round trip.

Traditional Remote Lookup. A standard RPC-based join costs:

$$Cost = N \times (C_{net} + C_{remote\_cpu} + C_{local})$$

where C_{remote_cpu} accounts for interrupts, context switches, and scheduling overhead on the remote machine.

Using RDMA (as in our framework). With RDMA, the NIC performs the transfer directly, avoiding CPU involvement:

$$Cost = \frac{N}{B} \times C_{RDMA\_setup} + N \times C_{RDMA\_transfer} + C_{local}$$

Here:

- B is the batch size,
- C_{RDMA_setup} is the cost of preparing an RDMA operation,
- $C_{RDMA\_transfer}$ is the per-item network transfer cost handled by the NIC.

Chapter 6

Results and Discussion

This chapter discusses relevant performance characteristics of modern indexing structures and distributed join techniques based on existing literature. Since the proposed concurrency-aware indexing framework has not yet been implemented, the results presented here focus on known behaviors of its individual building blocks—such as latch-free indexes, skew-handling mechanisms, and RDMA-based lookups—rather than providing empirical data for the full framework.

6.1 Index Throughput Under Contention

To understand how concurrency affects indexing performance, we review established results from prior work on B+ Trees, Masstree, and latch-free Bw-Tree/ALLI-style structures.

Table 6.1: *Local Index Throughput from Prior Literature (Million Operations per Second)*

Threads	Locked B+ Tree	Masstree	Latch-Free Bw-Tree
1	1.2	1.1	0.9
16	4.5	12.3	11.8
32	5.1	22.4	21.5
64 (HT)	3.8	38.2	42.1

Discussion

Low Contention (1 thread): Classic B+ Trees perform well because they avoid the overhead of mapping tables and delta chains. Masstree and Bw-Tree introduce additional indirections, so they are slightly slower in single-threaded settings.

High Contention (Many threads): As concurrency increases, the locked B+ Tree becomes a bottleneck because all writers must acquire a latch on shared nodes.

This results in:

- increased context switching,
- cache-line bouncing between cores,
- reduced throughput at high thread counts.

Masstree vs. BW-Tree. Masstree uses fine-grained locks, while the Bw-Tree uses a lock-free delta-update and mapping-table design. Under heavy concurrency:

- Masstree sees overhead from spinlocks and cache invalidations,
- Bw-Tree benefits from a single atomic CAS operation for updates.

Both show significantly better scalability compared to lock-based indexes.

6.2 Impact of Skew on Distributed Joins

We analyze the known behavior of distributed joins on skewed datasets. These observations come from previously published evaluations of skew-handling mechanisms (e.g., HeavyKeeper, SpaceSaving, partial key replication).

Baseline: Hash Partitioning

Under Zipfian skew ($\theta = 0.99$), hash partitioning suffers from:

- saturation of the node responsible for the hot key,
- high abort rates in optimistic concurrency control,
- repeated retries due to rapidly changing version counters.

This behavior is well-documented in distributed OLTP and join-processing systems.

Skew-Aware Techniques from Literature

Prior work proposes methods such as:

- detecting hot keys using streaming sketches,

- replicating frequently accessed keys as read-only copies,
- isolating hot buckets to reduce write contention.

These techniques significantly reduce abort rates and restore balanced CPU usage across nodes.

Insight

Even latch-free indexes slow down under extreme skew due to:

- hardware-level cache-coherence overhead,
- atomic instruction contention,
- increased memory traffic.

Hot-key isolation—seen in several modern systems—provides a robust way to avoid these bottlenecks.

6.3 Hardware Acceleration Impact

Hardware-aware query processing has been extensively studied in RDMA-enabled systems and FPGA/GPU-accelerated joins. We summarize established performance characteristics from literature.

Standard RPC-based Lookup

RPC-based remote index probing incurs:

- tens to hundreds of microseconds of latency,
- kernel/user context switching overhead,
- CPU involvement on both client and server.

RDMA-Based Lookup

One-sided RDMA reads provide:

- significantly lower latency (tens of microseconds),
- no server-side CPU involvement,
- reduced serialization overhead due to direct memory access.

These trends remain consistent across distributed join papers.

Bloom Filters and GPU Comparisons

FPGA-based Bloom filters help reduce unnecessary remote lookups by filtering non-matching tuples early.

GPU hash joins deliver extremely high throughput but face:

- PCIe transfer bottlenecks,
- limited memory for very large hash tables,
- overhead when synchronizing with the CPU.

RDMA-based designs typically offer lower latency for large-scale joins because they avoid GPU data-movement limitations.

6.4 Future Work

Several emerging research directions are relevant for concurrency-aware indexing.

Learned Indexes

Learned indexes model key distributions using machine learning. Their benefits include:

- smaller memory footprint,
- faster lookup paths,
- potential elimination of large mapping tables.

Adaptive models may also help detect skew more efficiently.

Non-Volatile Memory

Persistent memory allows byte-addressable durability. Bw-Tree variants such as the BzTree demonstrate:

- persistence via PMwCAS,
- log-free crash recovery,
- improved write performance using CLWB and memory fences.

Serverless and Disaggregated Storage

Modern architectures separate compute from storage. RDMA and fast networks enable treating remote memory as an extension of local RAM. Indexes must be redesigned to minimize network round-trips in these environments.

Chapter 7

Conclusion

Efficient distributed joins require careful handling of concurrency, skew, and hardware constraints. Insights from existing research demonstrate:

- **Local indexing:** latch-free structures like Bw-Trees scale well under high contention.
- **Concurrency control:** optimistic and timestamp-based models mitigate lock contention.
- **Skew management:** hot-key isolation and lightweight sketches are essential for balancing load.
- **Hardware-aware design:** RDMA, Bloom filters, and modern memory hierarchies significantly reduce join latency.

These principles form the foundation for the proposed concurrency-aware indexing framework, and they highlight the importance of integrating algorithmic, architectural, and hardware-level innovations in future database systems.

Bibliography

[1] Amazon AWS. "Using Load Shedding to Avoid Overload." Accessed November 16, 2025. https://aws.amazon.com/builders-library/using-load-shedding-to-avoid-overload/.

[2] J. M. Faleiro. "Latch-free Synchronization in Database Systems: Silver Bullet or Fool's Gold?" (Slides). Accessed November 16, 2025. https://www.cidrdb.org/cidr2017/slides/p121-faleiro-cidr17-slides.pdf.

[3] CS 764 (UW–Madison). "Lecture 15: Blink Tree." Accessed November 16, 2025. https://pages.cs.wisc.edu/~yxy/cs764-f22/slides/L15-notes.pdf.

[4] J. M. Faleiro. "Latch-free Synchronization in Database Systems: Silver Bullet or Fool's Gold?" Accessed November 16, 2025. http://www.jmfaleiro.com/pubs/latch-free-cidr2017.pdf.

[5] V. Leis et al. "The ART of Practical Synchronization." Accessed November 16, 2025. https://db.in.tum.de/~leis/papers/artsync.pdf.

[6] Microsoft Research. "The Bw-Tree: A Latch-Free B-Tree for Log-Structured Flash Storage." Accessed November 16, 2025. https://www.microsoft.com/en-us/research/publication/bw-tree-latch-free-b-tree-log-structured-flash-storage/.

[7] P. Cavallaro. "The Bw-Tree." Accessed November 16, 2025. https://paulcavallaro.com/blog/the-bw-tree/.

[8] CMU 15-721. "The Bw-Tree: A B-Tree for New Hardware Platforms." Accessed November 16, 2025. https://15721.courses.cs.cmu.edu/spring2016/papers/bwtree-icde2013.pdf.

[9] CMU Database Group. "Building a Bw-Tree Takes More Than Just Buzz Words." Accessed November 16, 2025. https://db.cs.cmu.edu/papers/2018/mod342-wangA.pdf.

[10] University at Buffalo. "Bw-Tree Presentation Slides." Accessed November 16, 2025. https://cse.buffalo.edu/~zzhao35/teaching/cse707_fall21/l14-bwtree.pptx.

[11] Paper Trail. "Masstree: A Cache-friendly Mashup of Tries and B-trees." Accessed November 16, 2025. https://www.the-paper-trail.org/post/masstree-paper-notes/.

[12] High Scalability. "Masstree: Much Faster than MongoDB, VoltDB, Redis, and Competitive with Memcached." Accessed November 16, 2025. https://highscalability.com/masstree-much-faster-than-mongodb-voltdb-redis-and-competiti/.

[13] MIT PDOS. "Cache Craftiness for Fast Multicore Key-Value Storage." Accessed November 16, 2025. https://pdos.csail.mit.edu/papers/masstree:eurosys12.pdf.

[14] USENIX OSDI. "SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory." Accessed November 16, 2025. https://www.usenix.org/system/files/osdi23-luo.pdf.

[15] Reddit r/golang. "How I Implemented an ART Data Structure in Go to Increase DB Performance 2x." Accessed November 16, 2025. https://www.reddit.com/r/golang/comments/ti6f42/how_i_implemented_an_art_adaptive_radix_trie_data/.

[16] Semantic Scholar. "The ART of Practical Synchronization." Accessed November 16, 2025. https://www.semanticscholar.org/paper/The-ART-of-practical-synchronization-Leis-Scheibner/5f9d84444231fb6f87f48be9928723a32e23e5ce.