algorithms

MDPI

Review

Hash Tables as Engines of Randomness at the Limits of Computation: A Unified Review of Algorithms

Paul A. Gagniuc [1] **and Mihai Togan** [2,*]

[1] Faculty of Engineering in Foreign Languages, National University of Science and Technology Politehnica Bucharest, 060042 Bucharest, Romania; paul.gagniuc@upb.ro
[2] Faculty of Information Systems and Cyber Security, Military Technical Academy "Ferdinand I", 050141 Bucharest, Romania
* Correspondence: mihai.togan@mta.ro

Abstract

Hash tables embody a paradox of deterministic structure that emerges from controlled randomness. They have evolved from simple associative arrays into algorithmic engines that operate near the physical and probabilistic limits of computation. This review unifies five decades of developments across universal and perfect hashing, collision-resolution strategies, and concurrent and hardware-aware architectures. The synthesis shows that modern hash tables act as thermodynamic regulators of entropy, able to transform stochastic mappings into predictable constant-time access. Recent advances in GPU and NUMA-aware designs, lock-free and persistent variants, and neural or quantum-assisted approaches further expand their capabilities. The analysis presents hash tables as models that evolve order within randomness and expand their relevance from classical computation to quantum and neuromorphic frontiers.

Keywords: hash tables; collision resolution; consistent hashing; concurrent data structures; hardware-aware hashing; GPU-based hashing; learned hashing; quantum-safe hashing; neuromorphic computation; entropy regulation

1. Introduction

Hash tables represent one of the most influential abstractions in computer science, that serve as the cornerstone of efficient data retrieval mechanisms since their formal introduction by D. Knuth in the early 1970s [1]. Their conceptual simplicity, namely the association of keys with values through a deterministic hashing function, conceals the immense engineering sophistication that underlies their modern evolution [1,2]. From compiler design, where symbol tables rely on constant-time lookups [3], to database indexing and key-value stores that define the scalability of cloud infrastructures [4], hash tables are omnipresent. Their role in cryptographic constructions, such as hash-based signatures and commitment schemes, extends their influence beyond data structures into the mathematical foundations of digital trust [5].

In high-performance computing, hash tables are the silent engines behind memory caching and load-balancing schemes, particularly in distributed systems where consistent hashing has redefined partition tolerance and fault resilience [4,6]. Frameworks such as *Redis* and *Cassandra* have built entire paradigms of real-time data management around these principles [7]. Yet, far from being a "solved problem", hashing remains in a dynamical evolution. The advent of multicore architectures has catalyzed a shift toward

check for updates

Academic Editor: Frank Werner

Received: 10 October 2025
Revised: 27 November 2025
Accepted: 13 December 2025
Published: 18 December 2025

lock-free and concurrent hash tables (see Supplementary Materials S2 for glossary of terms), which exploit atomic primitives and memory models to eliminate contention and maximize throughput [8]. Similarly, GPU-accelerated and parallel hashing techniques exploit massive data-level parallelism to achieve performance gains orders of magnitude beyond traditional CPU implementations [9]. Emergent architectural awareness has introduced cache-conscious and cache-oblivious designs that exploit spatial locality, which in turn leads to a minimisation of latency with improvement in power efficiency [10]. Memory-safe and space-optimized variants, such as *Robin Hood* and *Hopscotch* hashing, embody a new generation of algorithmic refinements, that offer deterministic performance and resilience against adversarial key distributions [11]. Furthermore, persistent and flash-optimized hash tables redefine data durability in non-volatile memory environments, that bridge the semantic gap between DRAM and secondary storage [12]. Beyond systems engineering, hashing has penetrated AI and bioinformatics, which power high-dimensional nearest neighbor searches and k-mer indexing in genomic pipelines [13]. Historically, hashing arose in the late 1950s and 1960s as a practical method for associative tables in file systems and early compilers, and the work of Knuth in the 1970s formalized collision resolution, load-factor thresholds, and expected probe lengths [1]. Subsequent developments in universal and perfect hashing extended the paradigm from ad hoc strategies to probabilistic and information-theoretic constructions. Modern variants such as linear, Robin Hood [11], hopscotch, and cuckoo hashing thus appear as stages in a longer trajectory that leads from simple associative arrays to entropy-aware, architecture-aligned designs. Therefore, this review provides four main contributions. First, it unifies theoretical results on universal, perfect, and collision-resilient hashing methods within a single conceptual framework. Second, it surveys architectural variants that range from classical CPU tables to GPU, NUMA-aware, FPGA, and persistent memory structures, with emphasis on hardware effects on performance. Third, it outlines the strategic role of hash tables across database engines, distributed systems, AI search, and genomic analytics. Fourth, it formulates a forward-looking perspective that links hashing theory to quantum and neuromorphic computation, which reveals how the data structure evolves toward adaptive and entropy regulated forms.

2. Theoretical Foundations of Hashing

The theoretical foundations of hashing rest upon the statistical behavior of hash functions as mappings from a key space K to a table of fixed size m, typically modeled as a random function to ensure uniform distribution and minimal collision probability [1,2,14]. The ideal hash function approximates a uniform random oracle, where each key is equally likely to map to any slot, a property essential for the preservation of constant time expectations in search and insertion operations [1,15]. Uniformity and randomness together define the core performance envelope of a hash table, while the avalanche effect, the sensitivity of the output to minute variations in input bits, ensures diffusion and unpredictability, especially in cryptographic contexts [16]. The probabilistic model of collisions follows the classical occupancy problem, where n keys are inserted into m bins; the expected number of collisions C is given by

$$C = n - m + m\left(1 - \frac{1}{m}\right)^{n}, \tag{1}$$

which implies that for $n \approx m$, collision frequency grows quadratically with load factor $\alpha = n/m$ [17]. The load factor is formally defined as $\alpha = n/m$, where n is the number of stored elements and m is the table capacity (i.e., the total number of slots). The *load factor* thus serves as a statistical control variable governing throughput and memory trade-

offs, with empirical thresholds, typically $\alpha < 0.75$, that balances density against access latency [18]. The expected probe sequence length in open addressing,

$$E[P] = \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}\right), \tag{2}$$

establishes the amortized constant-time performance of successful searches under the assumption of random probing [19]. Deletion in open addressing usually uses tombstones (i.e., markers that indicate that the slot formerly held an element and must be treated as occupied during probing). If an element is removed by clearing its slot, later searches may terminate too early and miss keys deeper in the probe sequence. Tombstones avoid this error because probe chains remain intact, yet they accumulate and raise the effective load factor [1,2,18,19]. Dense stretches of tombstones increase average probe length and, in long runs, the cost approaches that of an overfull table. Practical implementations either rebuild the table when the tombstone fraction crosses a threshold or use backward-shift deletion in schemes such as linear or Robin Hood hashing, which restores compact clusters while it preserves the invariants of the probe policy [11,18,19]. Thus, the estimate in Equation (2) assumes a uniform hash function, independence between probes, and the absence of deleted slots (i.e., no tombstones). Under these standard occupancy assumptions, probe lengths follow a geometric distribution that yields the expression in Equation (2). Universal hashing, introduced by Carter and Wegman, formalizes the construction of hash families H such that for any two distinct keys $x, y \in K$, the probability that $h(x) = h(y)$ for a randomly chosen $h \in H$ is bounded by $1/m$ [20]. This property counters adversarial key selection and forms the theoretical underpinning of randomized hashing algorithms. A hash function h is said to be perfect with respect to a finite key set S if it maps each key to a unique slot without collisions:

$$\forall x, y \in S, x \neq y \Longrightarrow h(x) \neq h(y).$$

If the mapping additionally uses exactly $|S|$ slots, then h is a *minimal perfect hash function*. Perfect hashing extends this paradigm, targeting zero-collision mappings for static sets via minimal perfect hash functions (MPHFs), which guarantee deterministic constant-time lookups at the cost of preprocessing complexity [21]. Techniques like CHD (Compress, Hash, Displace) and BDZ algorithms achieve near-optimal space utilization approaching $2.07n$ bits per key [22]. Cryptographic hash functions, such as SHA-256 or BLAKE3, assign priority to collision resistance and preimage hardness rather than table uniformity [23]. In contrast, non-cryptographic hashes like *MurmurHash* and *CityHash* are optimized for speed and low entropy environments, which trade security for computational efficiency [24]. The avalanche characteristics of cryptographic hashes are quantified through bit independence and correlation coefficients, while non-cryptographic variants are evaluated via dispersion tests [16,25].

3. Collision Resolution Strategies

Collision resolution stands at the core of hashing efficiency, which translates probabilistic collisions into deterministic recovery strategies that define practical performance (Table 1) [26]. Among these, separate chaining remains the archetypal approach, wherein each slot of the hash table maintains a secondary structure, typically a linked list or dynamic array, to accommodate colliding keys (Table 1) [27]. Its simplicity and robustness against clustering make it particularly resilient under high load factors, yet the pointer overhead and cache inefficiency of linked structures degrade spatial locality and throughput on modern architectures [28]. Amortized insertion complexity remains $O(1)$ for uniform

distributions, but degradation to $O(n)$ under pathological key sets necessitates adaptive rehashing and dynamic resizing mechanisms [29].

Table 1. Classification of hash functions by construction method, functional properties, and deployment context. This taxonomy outlines principal hash function classes, each defined by a distinct construction paradigm (e.g., modular arithmetic, multiplicative schemes, static mapping). Representative algorithms are listed for each class, alongside their dominant operational property (e.g., collision resistance, dispersion uniformity) and common deployment domains, which include compiler internals, cryptographic systems, and high-throughput data indexing. Note: CHD = Compress-Hash-Displace; BDZ = Bipartite Double-Zero; FCH = Fast Compress-Hash. Non-cryptographic functions (e.g., MurmurHash) are unsuitable for adversarial settings due to low collision resistance.

Class	Representative Algorithms	Primary Property	Typical Use
Multiplicative	Knuth method, Fibonacci hashing	Uniform key spreading via multiplication by irrational constants	Compiler symbol tables
Modular	Division and prime modulus hashing	Fast integer modular arithmetic	General-purpose hashing
Universal	Carter-Wegman families	Controlled collision probability	Randomized algorithms
Perfect	CHD, BDZ, FCH	Zero-collision static mapping	Databases, compilers
Cryptographic	SHA-256, BLAKE3	Collision and preimage resistance	Security, integrity checking
Non-cryptographic	*MurmurHash*, *CityHash*	Speed and practical uniformity	Data indexing, hash tables

Open addressing resolves collisions in situ by probing alternative slots following deterministic or pseudo-random sequences. Linear probing, the earliest and most intuitive variant, exhibits favorable cache behavior but suffers from primary clustering (i.e., the formation of long contiguous probe regions caused by collisions that pull subsequent keys into the same expanding cluster), where consecutive probe chains amplify collision zones [1,2,19,30]. *Quadratic probing* solves primary but not secondary clustering (i.e., collisions among keys that hash to different positions but follow identical probe sequences), as keys hashing to identical initial positions still share probe sequences [31]. *Double hashing* introduces a second independent hash function to compute the probe increment, which produces a near-uniform permutation of indices and achieves the best trade-off between randomness and predictability [32]. The theoretical analysis of open addressing is able to model probe lengths as geometric random variables, with expected search cost under uniform assumptions [17,33]. The expected probe cost follows the same expression as in Equation (2). A more egalitarian principle is introduced in Robin Hood hashing, which enforces balance by stealing positions from keys with shorter probe distances, thereby minimizing variance in search lengths that in turn allows for an improvement in worst-case guarantees [11,34]. Its deterministic redistribution policy leads to more uniform probe distributions, which reduces cache misses and achieves a tighter latency bounds in steady state [35]. *Hopscotch hashing*, a modern evolution, introduces neighborhood constraints, that allow for displaced elements to remain within a bounded proximity of their ideal bucket via localized swaps and bitmaps, which combines concurrency-friendliness with spatial clustering [36].

Cuckoo hashing revolutionizes collision handling through multiple independent hash functions and relocation cascades. Each key is placed in one of two (or more) possible positions, and upon conflict, an existing occupant is kicked out and reinserted into its alternate position [37]. This process guarantees $O(1)$ lookups and bounded insertions under load factors up to 50–90%, depending on hash independence [38]. Its deterministic lookup path (two probes) and predictable memory access patterns have made it foundational for

high-speed networking tables and cache-resident key-value stores [39]. Extensions such as *d*-ary cuckoo hashing and partial-key cuckoo hashing enhance resilience to insertion loops (i.e., cycles of repeated evictions that prevent successful placement of a new key) and memory fragmentation [37,40].

Moreover, hybrid and adaptive collision resolution mechanisms integrate the strengths of these paradigms. Techniques like *two-choice hashing*, where each element selects the less-loaded of two candidate buckets, minimize maximum bucket size logarithmically relative to table size [41]. Adaptive rehashing frameworks, dynamically adjust hash functions or probing policies based on runtime distribution entropy, which, in turn, enables stability under non-uniform workloads [42].

4. Architectural and Implementation Advances

The architectural evolution of hash tables mirrors the broader trajectory of computer architecture itself, which shifted from abstract data structures to deeply hardware-coupled mechanisms that exploit memory hierarchies, atomic primitives, and parallel execution models [8–10,12,28,43]. *Dynamic hash tables* embody this evolution with the introduction of elasticity in the structure and the load management. *Classical resizing*, once a disruptive operation that involved complete rehashing, has been refined into incremental rehashing, which migrates entries gradually across epochs, with the maintenance of service continuity under concurrent access [44]. This technique amortizes the reallocation cost, and it transforms a linear-time rehash into a bounded background task governed by the insertion rate [45]. In contrast, consistent hashing, first formalized for distributed caching, partitions the key space along a virtual ring to minimize remapping during table expansion or node churn (i.e., frequent join/leave events in a distributed cluster); thereby it allows for a stabilisation of distributed hash tables (DHTs) under dynamic conditions [46]. Such strategies underpin scalable infrastructures like *Amazon Dynamo* and *Apache Cassandra*, where node-level elasticity is essential to availability and fault tolerance [47].

The rise of multicore architectures redefined concurrency as the dominant bottleneck in hash table performance [8]. Early coarse-grained lock (i.e., one global lock for the entire table) designs used global mutexes to ensure correctness but caused throughput serialization [48]. As expected, fine-grained locks improve scalability by segmentation of the table into independent shards, which reduces contention but adds lock-management overhead [49]. Modern designs therefore adopt lock-free or wait-free approaches that rely on atomic compare-and-swap (CAS) primitives (i.e., a single-step hardware instruction that updates a memory location only if it still holds an expected value), memory fences, and hazard pointers (i.e., a technique that prevents the release of nodes still in use by other threads) to maintain consistency without mutual exclusion [50]. The Michael concurrent hash table exemplifies this paradigm, which achieves near-linear scalability across cores through non-blocking operations [51]. Concurrency correctness is typically established under the linearizability model, that ensures each operation appears instantaneous and globally ordered [52]. Further optimizations leverage read-copy-update (RCU) and epoch-based reclamation, which reduces the cost of memory reclamation under high contention workloads [53].

Hardware-conscious hashing represents the fusion of algorithmic design and microarchitectural awareness [9,10,12,28]. Cache-aware schemes structure tables to align with cache lines and prefetch patterns, optimize spatial locality, and reduce translation lookaside buffer (TLB) pressure [54]. Cache-oblivious variants remove explicit cache parameters and reach near-optimal performance across memory hierarchy levels through recursive layout principles similar to *van Emde Boas* trees [55]. On parallel hardware, SIMD-accelerated probing exploits vectorized comparisons to resolve multiple keys simultaneously, which produces

order-of-magnitude speedups in probing-heavy workloads [9,10,28,56]. GPU-based hashing, through massive thread-level parallelism, transforms latency-bound operations into throughput-bound computation; structures such as *SlabHash* and *MegaKV* demonstrate billions of lookups per second using warp-cooperative insertions and fine-grained atomic synchronization [57]. On NUMA systems, memory locality dictates access performance; NUMA-aware hash tables allocate buckets per node and restrict threads to local memory, which reduces the penalties of cross-socket access [58]. Specialized hardware realizations on FPGA and ASIC architectures extend cuckoo-based hashing with pipelined and bucketized designs that deliver deterministic, low-latency lookup at multi-gigabit rates [59].

Memory efficiency has emerged as a parallel frontier. Compact hashing reduces pointer and metadata overhead via quotienting (i.e., retains only the high-order portion of the hash after the remainder selects the bucket) and fingerprinting, and stores partial hashes to reconstruct positions dynamically [60]. *Quotient* filters, a probabilistic cousin of *Bloom* filters, achieve efficient membership queries and deletions at low false-positive rates while it preserves locality [61]. Minimal perfect hash functions, optimized for static datasets, approach the theoretical entropy bound of key representation, and recent succinct codes achieve compression below 2 bits per element. [62]. The advent of non-volatile memory (NVM) redefined persistence semantics; NVM-resident hash tables utilize log-structured allocation, transactional persistence, and fine-grained journaling to guarantee crash consistency without full serialization [63]. Designs such as *RECIPE* and *Level* hashing exploit byte-addressable persistence to reach near-DRAM performance and secure durability guarantees [64]. Empirical studies confirm that architectural awareness yields order-of-magnitude performance differentials. GPU-accelerated key-value stores such as Mega-KV already process over 160 million key-value operations per second on a dual-CPU, dual-GPU server and scale to about 6.2×10^8 operations per second on an 8-CPU/8-GPU cluster [65]. The synthesis of algorithmic ingenuity and hardware intimacy thus defines the modern hash table, not merely as a theoretical abstraction but as a living system optimized for the memory hierarchies and concurrency semantics of 21st-century computing [8–10,28].

5. Applications Across Domains

Hash tables have transcended their algorithmic origin to become a universal substrate in data-centric computation, where their structural adaptability aligns with the architectural and statistical constraints of various application domains [47,66]. In databases, they underpin hash joins and in-memory indexing, which serves as the backbone of query execution engines. The classical *Grace* and *Hybrid Hash Join* algorithms exploit partitioned hashing to handle data that exceeds main memory capacity, optimizing for I/O-bound workloads through cache-conscious partitioning [67]. Modern relational engines such as *PostgreSQL* and *Oracle* deploy dynamic hash indexes with *adaptive rehashing*, which align bucket allocation with cardinality estimations derived from query plans [68]. In distributed key-value stores like *Redis* and *LevelDB*, the combination of in-memory hash indexing and log-structured merge-trees establishes the latency-durability equilibrium critical to real-time analytics [69]. Furthermore, hash partitioning in parallel query execution systems like *Spark* SQL and *Flink*, provides deterministic load balancing and minimizes shuffle overhead because it keeps data uniformly distributed across worker nodes [70].

In networking and distributed systems, hashing dictates scalability and fault tolerance. Consistent hashing remains the canonical mechanism for node allocation in distributed hash tables (DHTs), as seen in Chord and Kademlia, where it minimizes data migration during topology changes [71]. Network routers employ hardware-accelerated cuckoo and bucketized hashing for packet classification and flow tracking, which allows for a line-rate performance on programmable data planes [72]. Hash-based load balancing (i.e., uniform

spread of work across nodes), embodied by ECMP (*Equal-Cost Multi-Path*) and flow hashing, secures deterministic routing symmetry and congestion avoidance [73]. These principles extend naturally into large-scale distributed caches, where hash ring (i.e., the circular identifier space created by consistent hashing) stabilization minimizes cross-node state synchronization (i.e., to keep shared data consistent across nodes) under churn conditions (i.e., continuous node arrivals, and departures, that force dynamic rebalancing of key ownership) [74].

In cryptography and security, the duality of hash functions, namely fast non-cryptographic variants and collision-resistant cryptographic ones, creates a nuanced space of applications [16,23]. Hash tables serve as the structural basis for *Bloom* filters, which probabilistically compress membership information with built-in privacy safeguards for deduplication and intrusion detection systems [75]. In password management, salted hashing tables (i.e., hash tables that incorporate a random per-instance salt value into the hash function to prevent predictable key placement and collision attacks) counter rainbow table attacks (i.e., precomputed lookup attacks that store chains of hashed values to reverse cryptographic hashes and recover original plaintexts such as passwords) through the introduction of entropy at the key level, while modern secure deduplication frameworks integrate cryptographic hashing with homomorphic encryption to achieve content addressability without plaintext exposure [76]. Hash-based signatures, derived from Merkle tree constructions, formalize a lineage from simple lookup tables to quantum-resistant cryptographic primitives [77].

In bioinformatics, hashing provides a combinatorial scaffold for the high-throughput processing of genomic data. The concept of *k*-mer hashing, namely mapping fixed-length DNA substrings into integer key spaces, forms the basis for tools such as *Jellyfish*, *Kraken*, and *MinHash*-based aligners [78]. These systems exploit compact and cache-aware hash tables to store billions of *k*-mers with minimal overhead, which enables sublinear sequence similarity computations [79]. The probabilistic *MinHash* technique, by approximating *Jaccard* similarity through hash sampling, revolutionized metagenomic clustering and read classification, which allowed a reduction in memory consumption by several orders of magnitude [80]. Parallel implementations employing GPU-resident *cuckoo* tables now perform *k*-mer counting at terabase scales, which exploits warp divergence minimization for high-throughput alignment tasks [81].

In artificial intelligence and big data analytics, hashing reemerges as a geometric tool for similarity search and embedding compression (i.e., reduction in vector dimensions to save memory and computation). *Locality-sensitive hashing* (LSH) partitions high-dimensional feature spaces into probabilistic buckets that preserve neighborhood proximity under cosine or Euclidean metrics, which allows for approximate nearest-neighbor search in large-scale vector databases [82]. Modern AI frameworks integrate LSH variants for accelerated semantic retrieval in multimodal embeddings, which powers systems such as *FAISS* and *Annoy* [83]. In deep learning, hash-based parameter servers compress gradient updates via sparse hash maps, making a reduction in communication overhead in distributed training [84]. Even transformer architectures now employ hashing in *HashFormer* and *Reformer* variants, where hashed attention reduces quadratic complexity (i.e., computational cost that grows proportionally to the square of the input size, denoted $O(n^2)$) in sequence length to sublinear scales [85].

6. Comparative Evaluation and Taxonomy

A comparative taxonomy (i.e., a structured classification that contrasts algorithmic families across multiple design dimensions to reveal relationships, trade-offs, and evolutionary trends) of hash tables demands a multidimensional synthesis that unifies algorithmic

strategy, spatial organization, concurrency model, and architectural affinity into a single analytical fabric (Table 2) [86]. A taxonomy of the principal families and subfamilies of hash-table designs is illustrated in Figure 1. Across these axes, one observes that collision resolution techniques define the semantic layer of hashing (i.e., the algorithmic dimension that defines how meaning or behavior emerges from collision-resolution rules, insertion policies, and key-value associations, independent of physical storage layout), while memory layout and hardware mapping define its physical layer. Separate chaining structures (i.e., hash table designs in which each slot holds a secondary list or bucket of colliding keys instead of a single entry), despite their resilience and simplicity, exhibit poor cache locality (i.e., how often data is already in cache) due to pointer indirection (i.e., extra pointer hops along the chain; ex., table[i] = data vs. table[i] = pointer → data), which produces measurable throughput decay as the load factor (i.e., the ratio of stored keys to available buckets; $\alpha = n/m$) surpasses 0.8 [87]. In contrast, open addressing families such as linear, quadratic, and double hashing, achieve tighter memory packing and lower pointer overhead but become increasingly sensitive to clustering and memory latency variance under contention (i.e., many threads competing for the same table locations) [28,31,32,88].

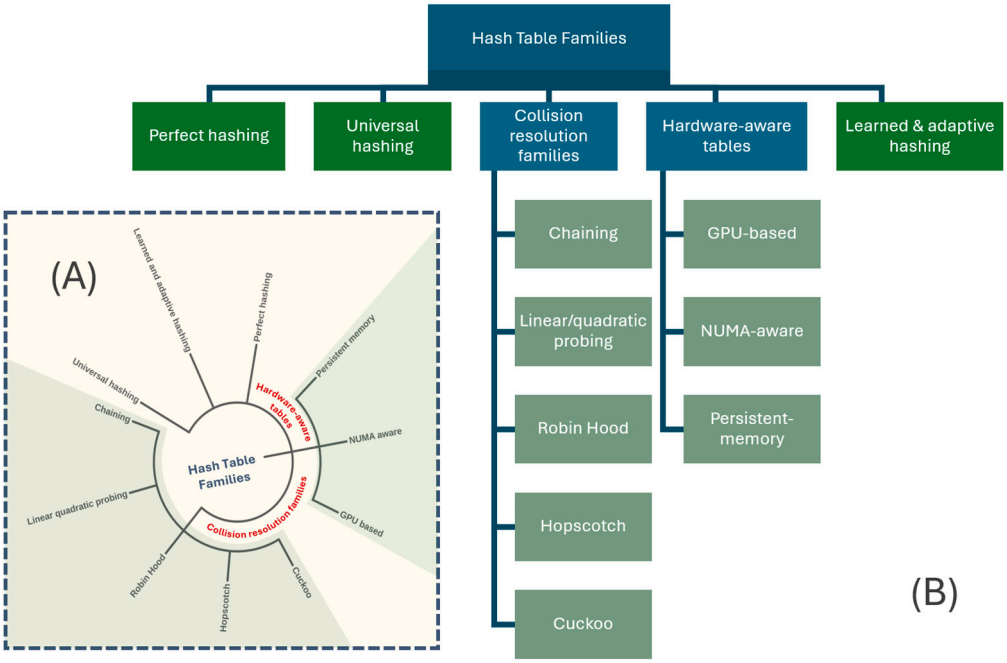

Figure 1. Taxonomy of major hash-table families. The diagram distinguishes perfect and universal hashing, classical collision-resolution strategies (chaining, probing, Robin Hood, Hopscotch, cuckoo), hardware-aware designs (GPU-based, NUMA-aware, and persistent-memory variants), and learned or adaptive hashing. Panel (**A**) shows the taxonomy in a radial layout, while panel (**B**) presents the same structure in a hierarchical tree form.

Cuckoo and *Robin Hood* hashing occupy an intermediate stratum in this taxonomy, characterized by bounded probe variance, predictable memory access, and deterministic lookup paths [34,35,37,40,89]. Their structural determinism enables efficient vectorization and lock-free extensions, yet their relocation or displacement mechanisms impose nontrivial insertion costs beyond certain load thresholds [28,34,37,90]. From a concurrency perspective, coarse-grained lock-based schemes (i.e., designs that use one software lock to guard the whole table) retain algorithmic clarity but collapse under contention, which exhibits sublinear scaling as the number of threads grows beyond core count [91]. Fine-grained partitioning restores partial scalability but introduces synchronization noise that dominates under short-lived operations [49,92]. Lock-free tables achieve near-ideal

scalability but at the cost of higher implementation complexity and hardware dependency on atomic primitives such as compare-and-swap (CAS) and load-linked/store-conditional (LL/SC) [50–52,91,93].

The asymptotic complexity remains $O(1)$ for expected operations, yet empirical latency distributions under concurrent access display long-tailed deviations due to cache invalidations and memory fences [94]. Moreover, hardware taxonomy introduces further stratification. Cache-aware designs that prioritize frequently accessed blocks in the last-level cache reduce miss penalties and improve performance, as demonstrated by the Scavenger architecture [95]. Cache-oblivious variants, while theoretically elegant, exhibit unpredictable real-world behavior due to interference from hardware prefetchers and TLB behavior [96]. GPU-based tables dominate throughput metrics, which offer up to $10\times$ acceleration over CPU counterparts through warp-parallel insertions and coalesced memory transactions but exhibit high build cost and restricted mutability [97]. FPGA and ASIC implementations occupy the deterministic extreme of the taxonomy, engineered for pipeline uniformity, zero branching, and submicrosecond latency, at the expense of flexibility and memory dynamism [59,98]. The trade-off surface across these implementations can be conceptualized as a Pareto frontier (i.e., the set of optimal trade-off points where improvement in one metric necessarily degrades another) between memory overhead M, throughput T, and concurrency C. Empirical synthesis suggests that:

$$T \propto f\left(\frac{1}{M} \times \alpha^{-1}\right),$$

where α denotes the load factor, which further implies an exponential sensitivity of throughput to space compression in high-density tables [11,31,34,99]. Note that symbol $\propto$ means "is proportional to". This performance relation follows directly from the expected probe-length model introduced in Section 2. Under uniform hashing and independence between probe steps, the expected cost grows as (2), which increases sharply as $\alpha \to 1$. When memory overhead M acts as the dominant amortization term for metadata and displacement work, the throughput contribution reduces to a proportional dependence on $M^{-1}\alpha^{-1}$. The expression therefore summarizes the scaling trend encoded in the classical occupancy-based model rather than introducing a new equation. The term $M^{-1}\alpha^{-1}$ in the proportionality reflects the effective entropy of a hash table under dense configurations. This proportional model presumes uniform hashing and random-access independence across probes. It captures how probe-sequence variability grows as $\alpha \to 1$, which we describe metaphorically as increased algorithmic entropy. More concretely, in this review 'entropy' refers to the measurable increase in probe-length variance and the emergence of heavy-tailed probe-length distributions that arise in open-addressing schemes under high load. Here, "entropy" is used metaphorically to denote the algorithmic disorder arising from increased probe-sequence variability under high load factors; it does not refer to Shannon entropy. Thus, as memory overhead M decreases or the load factor α approaches 1, the distribution of probe lengths becomes increasingly disordered, raising the algorithmic entropy of the system [33]. We therefore interpret hash tables as entropy-regulating structures, in which collision-resolution and memory layout mechanisms act to compress stochastic key distributions into deterministic, constant-time behavior [86]. This formalizes the perspective introduced in the abstract and frames the unification of theory, architecture, and application domains throughout the review.

Experimental meta-analyses reveal that while open addressing excels in read-dominated workloads, chaining remains superior under write-intensive conditions and environments that tolerate high cache-miss latency [100]. In parallel systems, lock-free cuckoo hash tables (i.e., concurrent designs using atomic primitives rather than locks to

ensure progress under parallel updates) can sustain high throughput and maintain performance stability at load factors exceeding 0.95, as demonstrated in recent bucketized variants [101].

Table 2. Comparative summary of principal hash-table approaches, contrasting their performance regime, concurrency behavior, and efficiency characteristics. "Performance regime" refers to expected lookup and insertion behavior under typical and high-load conditions. "Concurrency behavior" indicates how each design tolerates parallel operations under contention. "Efficiency profile" covers memory overhead, locality properties, and stability of probe-length distributions.

Approach	Performance Regime	Concurrency Behavior	Space/Efficiency Profile
Chaining [68]	Stable average-case constant-time lookup; cost rises with long chains	Strong concurrency; independent bucket chains	Higher pointer overhead; predictable degradation
Linear probing [32]	Excellent cache locality at moderate load	Weak concurrency; clustering amplifies contention	Space-efficient; suffers from primary clustering
Quadratic probing [31]	Reduced primary clustering; similar to linear probing	Moderate concurrency	Space-efficient; secondary clustering persists
Robin Hood [11]	Tight distribution of probe distances at high load	Good concurrency with controlled relocation	Slightly costlier insertion due to displacement
Hopscotch [36]	Near-constant lookup under high load	Very strong concurrency; bounded relocation	Requires neighborhood metadata; excellent locality
Cuckoo [37]	Worst-case O(1) lookup	Moderate concurrency; relocation cascades	Requires multiple tables; occasional rehash
Bucketed/multi-hash cuckoo [39,40,101]	More stable insertion bounds	Good concurrency; bounded-relocation buckets	Higher memory footprint; predictable updates
Cache-/hardware-aware designs [10,28,56]	Extreme throughput; architecture-aligned	Varies by platform; often excellent	Memory-layout-optimized; overhead amortized

A unified classification emerges when visualized as a graph, where vertices represent design families and edges encode transitions through optimization parameters: *chaining* to *open addressing* through elimination of indirection (i.e., pointer-based access structures such as linked lists used in separate chaining), *open addressing* to *cuckoo* via an increase in hash multiplicity, *cuckoo* to *Robin Hood* through distance regularization, and *Robin Hood* to *hopscotch* through restriction of displacements under concurrency. Such a taxonomy reveals that the evolutionary trajectory of hash tables reflects a progressive compression of entropy, spatial, temporal, and architectural, toward deterministic constant-time behavior under non-ideal physical conditions [10,36,37,86,102]. The result is not a single optimal design but a dynamic equilibrium of trade-offs, where the optimality frontier is continually redrawn by the hardware substrate itself.

7. Open Challenges and Future Directions

The frontier of hashing research has shifted from algorithmic optimization toward adaptation, which, in turn, reflects the transformation of the computational fabric itself [10,28,56,65,87,103]. In quantum computing, for instance, classical assumptions of randomness and irreversibility are destabilized by quantum superposition and entanglement, compelling a redefinition of what it means for a hash function to be collision-resistant [104,105]. *Grover* algorithm, which quadratically accelerates brute-force search, effectively halves the bit security of conventional cryptographic hashes, which necessitates the design of quantum-safe or amplitude-balanced hash families that remain resilient under superposed queries [105]. In principle, lookup operations in quantum-accessible hash tables could exploit quantum parallelism over superposed keys to attain Grover-style

sublinear retrieval time $O(\sqrt{n})$, a possibility grounded in quantum search theory rather than demonstrated data-structure implementations [104–106]. The intersection of hashing and quantum memory architectures thus remains both theoretically rich and practically distant, a paradox of attainable complexity without stable hardware substrates [107].

Concurrently, energy efficiency has emerged as a defining constraint in exascale (i.e., high-performance supercomputing at 10^{18} operations per second, where power efficiency becomes a critical design limit) and edge computing (i.e., decentralized data processing performed close to the data source to minimize latency and energy cost). The asymptotic uniformity of hash functions masks their thermodynamic inefficiency: cache thrashing, redundant bit toggling, and contention-driven synchronization collectively inflate dynamic energy consumption [108]. Modern low-power designs pursue green hashing, which align bucket placement with cache and power domains to minimize cross-domain memory accesses [109]. Moreover, FPGA-based dynamic reconfiguration improves energy efficiency because hash-function behavior can adapt to workload patterns, as observed in *FASTHash* and related FPGA designs [110]. Thermal-aware rehashing strategies, that use temperature feedback to modulate load factors and memory allocation, represent a nascent but promising direction for sustainable data structures [111]. Another active axis is the integration of machine learning into hashing itself. Adaptive or learned hash tables use reinforcement learning and online optimization to predict key distributions and dynamically tune hash functions or probing policies [112]. Most learned hashing schemes train their predictive models offline on representative workloads, while adaptive variants rely on lightweight online updates; in both cases, the training cost is amortized over many operations and remains decoupled from the constant-time lookup path. Neural-assisted hash tables model the input entropy landscape via lightweight predictors, achieving better locality preservation and lower collision variance than static designs [113]. Deep hashing networks, initially developed for image retrieval, are now influence general-purpose key-value indexing, where embeddings generated by neural encoders (i.e., neural network models that transform inputs into dense vector representations that capture semantic or structural similarity) replace random hash functions entirely [114]. This convergence marks a shift from static to cognitive hashing, where structure adapts autonomously to data dynamics, thus a transition parallel to that seen in self-tuning databases [115].

Security, however, remains a challenge defined by endurance and evolutionary processes. Hash flooding attacks, exploiting adversarial key distributions to induce degenerate $O(n)$, insertion time, have prompted the adoption of randomized seeding and *SipHash*-like keyed functions for defense [116]. In high-throughput servers, cryptographic-strength hashing is often infeasible due to latency constraints, so designers employ probabilistic hardening, which adds low-overhead entropy to the hash-function space without providing full cryptographic guarantees [117]. Recent research explores hybrid defenses that combine adaptive bucket resizing with statistical anomaly detection, thus adding to the identification of malicious collision patterns at runtime [118]. Looking further, the boundary between determinism and probability in hashing is dissolving. Probabilistic hash functions, inspired by randomized quantization and stochastic computation, inject controlled uncertainty into the mapping process, which balances load adaptively across dynamic workloads [119]. Even more radical are neural hash functions, mentioned above, which learn latent representations (i.e., numerical descriptions of the input) that preserve entropy (i.e., retain the variability and structural properties of the original keys) from data directly and bypass analytical uniformity assumptions, which in turn provide latent-space locality that is superior to regular handcrafted algorithms [120]. These approaches, when combined with neuromorphic hardware (i.e., computing architectures that emulate the structure and dynamics of biological neural systems using analog or event-driven circuits

to achieve brain-like efficiency and parallelism), suggest a new computational ontology where hashing ceases to be a rigid transformation and becomes instead a continuously learned projection of data geometry [121].

Related Work and Survey Literature

While this review aims to provide a unified and comprehensive perspective on the evolution of hash tables, several prior studies have examined narrower aspects of the field. Tapia-Fernández (2022) provides a comparative benchmark of classical hash table implementations across programming languages, that focuses on educational and empirical performance analysis [122]. Yusuf et al. offer a descriptive overview of collision resolution techniques, but limit the discussion to chaining, linear probing, and related policies [123]. Lehmann (2025) surveys minimal perfect hashing techniques in detail, including recent compression strategies and graph-based constructions, yet without broader architectural or application-level concerns [124]. Misto (2024) explores implementation optimizations and compares a subset of collision handling strategies, particularly double hashing and Robin Hood hashing, through empirical evaluation [125].

Space-efficient membership structures related to hashing, such as *Bloom filters*, *Quotient filters*, and *Cuckoo filters*, have been reviewed in a variety of systems contexts [126–128]. In networking, Zink (2011) surveys hash tables in connection with IP lookup in packet processing pipelines, with emphasis on pipelined and multi-bank hardware [129]. Distributed hash tables (DHTs), a distinct subdomain of hashing, are extensively covered in surveys by Wang et al. and by Lua et al., which outline theory, platforms, and fault-tolerant architectures for peer-to-peer applications [130,131]. Recent developments in AI-related hashing, such as deep and learned hashing, have also been reviewed extensively in the context of approximate nearest neighbor search, semantic similarity, and deep embedding compression [132–134]. In cryptography, the structural role of hash functions in authentication, digital signatures, and commitment schemes is addressed in several foundational handbooks and surveys [135,136]. Hash-based memory allocators, persistence under non-volatile memory, and NUMA-aware hash tables have also been analyzed from a systems architecture point of view [137–140]. Distributed hash tables in large-scale environments have also been examined through broader systems perspectives, including cloud-, fog-, and edge-oriented architectures, as surveyed in Algorithmic Aspects of Distributed Hash Tables on Cloud, Fog, and Edge Computing Applications [141]. Complementary advances appear in domain-specific peer-to-peer ecosystems, exemplified by the WiCHORD+ scalable Chord-based framework for smart-agriculture infrastructures [142]. Thus, despite the depth of these topical surveys, none of the existing works integrate the full theoretical, algorithmic, architectural, and domain-specific landscape presented in this review. As such, the current synthesis addresses a structural gap in the literature by an alignment of the algorithmic evolution with modern computing substrates and cross-domain utility.

8. Conclusions

Hash tables are no longer static key-value abstractions but dynamic computational entities whose structural evolution reflects the shifting landscape of hardware architectures, memory models, and probabilistic computation. This review has demonstrated that modern hash tables function as systems of entropy regulation, which transforms stochastic inputs into deterministic behavior across multiple operational layers. Their algorithmic diversity, spanning collision resolution strategies such as cuckoo, Robin Hood, and hopscotch hashing, encapsulates a broader principle of statistical stabilization, where insertion variance, probe distribution, and access latency are shaped by adaptive rebalancing and probabilistic redistribution mechanisms. Architecturally, contemporary implementations

engage directly with the microphysical substrate of computation, and exploit memory hierarchies, atomic synchronization, cache line alignment, and parallel execution units to convert theoretical efficiency into empirical performance. The integration of lock-free concurrency, NUMA-aware memory placement, and SIMD or GPU acceleration exemplifies this alignment, which reveals hash tables as concrete forms of algorithm-hardware co-design. Beyond structural optimization, recent advances introduce cognitive and adaptive behavior into hashing systems, where learned functions and neural-assisted mappings respond dynamically to data distributions, optimize entropy profiles, and enforce locality constraints. Such methods redefine hashing not as a fixed transformation but as a data-driven projection function, continuously modulated by feedback and workload properties. This marks a departure from static uniformity assumptions and repositions hash tables within a class of self-organizing data structures.

Supplementary Materials: The following supporting information can be downloaded at: https://www.mdpi.com/article/10.3390/a18120804/s1, Supplementary Materials S1: Minimal Python 3.14.2 implementation of a Hash Table. A minimal educational implementation of a hash table is available here: https://github.com/gagniuc/hash-table (accessed on 16 December 2025). The code demonstrates the core mechanics of modular hashing and open addressing, using linear probing for collision resolution [143]. It provides a pedagogical example of constant-time average insertion and lookup under uniform key distribution; Supplementary Materials S2: Glossary of specialized terms. A glossary of specialized terminology referenced throughout the manuscript. This document provides concise definitions and contextual explanations for key technical expressions appearing in the main text, including algorithmic, architectural, and concurrency-related terms. It serves as a reference aid for readers outside the immediate domain of computer architecture or data structures, thus pushing for conceptual clarity across interdisciplinary contexts.

Author Contributions: Conceptualization, P.A.G. and M.T.; methodology, P.A.G. and M.T.; software, P.A.G. and M.T.; validation, P.A.G. and M.T.; formal analysis, P.A.G. and M.T.; investigation, P.A.G. and M.T.; resources, P.A.G. and M.T.; data curation, P.A.G. and M.T.; writing-original draft preparation, P.A.G. and M.T.; writing-review and editing, P.A.G. and M.T.; visualization, P.A.G. and M.T.; supervision, P.A.G. and M.T.; project administration, P.A.G. and M.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Executive Unit for Financing Higher Education, Research, Development and Innovation (UEFISCDI), Romania, grant number 39PTE/2025.

Data Availability Statement: No new data were created or analyzed in this study.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

HT	Hash Table
CHT	Chained Hash Table
OAH	Open Addressing Hashing
RH	Robin Hood Hashing
HH	Hopscotch Hashing
CH	Cuckoo Hashing
DHT	Distributed Hash Table
NVM	Non-Volatile Memory
GPU	Graphics Processing Unit
FPGA	Field-Programmable Gate Array
NUMA	Non-Uniform Memory Access
CAS	Compare-and-Swap

LL/SC	Load-Link/Store-Conditional
RCU	Read-Copy-Update
LSH	Locality-Sensitive Hashing
MPHF	Minimal Perfect Hash Function

References

1. Knuth, D.E. *The Art of Computer Programming, Volume 3: Sorting and Searching*; Addison-Wesley: Boston, MA, USA, 1973.
2. Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. *Introduction to Algorithms*; MIT Press: Cambridge, MA, USA, 2009.
3. Aho, A.; Lam, M.; Sethi, R.; Ullman, J. *Compilers: Principles, Techniques, and Tools*; Pearson: London, UK, 2006.
4. Dean, J.; Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Commun. ACM* **2008**, *51*, 107–113. [CrossRef]
5. Merkle, R.C. A certified digital signature. In *Advances in Cryptology—CRYPTO '89 Proceedings*; Springer: Berlin/Heidelberg, Germany, 1989; pp. 218–238.
6. Mirrokni, A.V.; Thorup, M.; Zadimoghaddam, M. Consistent hashing with bounded loads. In Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '18), New Orleans, LA, USA, 7–10 January 2018; pp. 587–604.
7. Lakshman, A.; Malik, P. Cassandra: A decentralized structured storage system. *ACM SIGOPS Oper. Syst. Rev.* **2010**, *44*, 35–40. [CrossRef]
8. Herlihy, M.; Shavit, N. *The Art of Multiprocessor Programming*; Morgan Kaufmann: San Francisco, CA, USA, 2012.
9. Ashkiani, M.; Farach-Colton, M.; Owens, J.D. A dynamic hash table for the GPU. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium, Vancouver, BC, Canada, 21–25 May 2017; pp. 583–592.
10. Vitter, J.S. Algorithms and Data Structures for External Memory. *Found. Trends Theor. Comput. Sci.* **2008**, *2*, 305–474. [CrossRef]
11. Celis, P.; Larson, P.; Munro, J.I. Robin Hood hashing. In Proceedings of the IEEE Annual Symposium on Foundations of Computer Science, Portland, OR, USA, 21–23 October 1985; pp. 281–288.
12. Oukid, I.; Lasperas, J.; Nica, A.; Willhalm, T.; Lehner, W. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16), San Francisco, CA, USA, 26 June–1 July 2016; Association for Computing Machinery: New York, NY, USA, 2016; pp. 371–386.
13. Langmead, B.; Salzberg, S.L. Fast gapped-read alignment with Bowtie 2. *Nat. Methods* **2012**, *9*, 357–359. [CrossRef]
14. Ramakrishna, M.V.; Zobel, J. Performance in practice of string hashing functions. In Proceedings of the ACM SIGMOD International Conference on Management of Data, Tucson, AZ, USA, 13–15 May 1997; pp. 215–226.
15. Pagh, A. Uniform Hashing in Constant Time and Optimal Space. *SIAM J. Comput.* **2006**, *35*, 302–313.
16. Rogaway, P.; Shrimpton, T. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *Fast Software Encryption*; Roy, B., Meier, W., Eds.; Lecture Notes in Computer Science; Springer: Berlin, Germany, 2004; Volume 3017, pp. 371–388.
17. Motwani, R.; Raghavan, P. *Randomized Algorithms*; Cambridge University Press: Cambridge, UK, 1995.
18. Mehta, D.P.; Sahni, S. *Handbook of Data Structures and Applications*; Chapman & Hall/CRC: Boca Raton, FL, USA, 2018.
19. Skiena, S.S. *The Algorithm Design Manual*, 3rd ed.; Springer: Berlin/Heidelberg, Germany, 2020.
20. Carter, J.L.; Wegman, M.N. Universal classes of hash functions. *J. Comput. Syst. Sci.* **1979**, *18*, 143–154. [CrossRef]
21. Botelho, F.C.; Pagh, R.; Ziviani, N. Simple and space-efficient minimal perfect hash functions. In Proceedings of the ACM WADS, Halifax, NS, Canada, 15–17 August 2007; pp. 139–150.
22. Belazzougui, M.; Botelho, F.C.; Dietzfelbinger, M. Hash, displace, and compress. In Proceedings of the 17th Annual European Symposium, Copenhagen, Denmark, 7–9 September 2009; pp. 682–693.
23. Krawczyk, N.; Bellare, M.; Canetti, R. *HMAC: Keyed-Hashing for Message Authentication*; IETF RFC 2104; Internet Engineering Task Force (IETF): Fremont, CA, USA, 1997.
24. Dahlgaard, S.; Knudsen, M.B.T.; Thorup, M. Practical hash functions for similarity estimation and dimensionality reduction. In Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17), Long Beach, CA, USA, 4–9 December 2017; Curran Associates Inc.: Red Hook, NY, USA, 2017; pp. 6618–6628.
25. Lemire, D.; Kaser, O. Faster 64-bit universal hashing using carry-less multiplications. *J. Cryptogr. Eng.* **2016**, *6*, 171–185. [CrossRef]
26. Østlin, A.; Pagh, R. *Simulating Uniform Hashing in Constant Time and Optimal Space*; BRICS Report Series, RS-02-27; University of Aarhus: Aarhus, Denmark, 2002; pp. 1–17.
27. Cooper, K.; Torczon, L. *Engineering a Compiler*, 2nd ed.; Morgan Kaufmann: Burlington, MA, USA, 2011.
28. Böther, M.; Benson, L.; Klimovic, A.; Rabl, T. Analyzing Vectorized Hash Tables Across CPU Architectures. In Proceedings of the VLDB 2023, Vancouver, BC, Canada, 28 August–1 September 2023; Volume 16, pp. 2755–2768.
29. Tucker, A.B. (Ed.) *Computer Science Handbook*, 3rd ed.; Chapman & Hall/CRC: Boca Raton, FL, USA, 2014.
30. Robert, C.P.; Casella, G. *Monte Carlo Statistical Methods*; Springer: Berlin/Heidelberg, Germany, 2004.
31. Kuszmaul, W.; Xi, Z. A partial analysis of quadratic probing. In Proceedings of the 2024 International Colloquium on Automata, Languages, and Programming (ICALP), Tallinn, Estonia, 8–12 July 2024; Schloss Dagstuhl–Leibniz Center for Informatics: Dagstuhl, Germany, 2024; pp. 103:1–103:19.

32. Sedgewick, R.; Wayne, K. *Algorithms*, 4th ed.; Addison-Wesley: Boston, MA, USA, 2011.
33. Mitzenmacher, M.; Upfal, E. *Probability and Computing*; Cambridge University Press: Cambridge, UK, 2017.
34. Poblete, P.V.; Viola, A. Analysis of Robin Hood and other hashing algorithms under the random probing model, with and without deletions. *Comb. Probab. Comput.* **2019**, *28*, 600–617. [CrossRef]
35. Kirsch, A.; Mitzenmacher, M. The power of Robin Hood hashing revisited. In Proceedings of the 18th Annual European Symposium, Liverpool, UK, 6–8 September 2010; pp. 684–695.
36. Herlihy, M.; Shavit, N.; Tzafrir, M. Hopscotch hashing. In Proceedings of the ACM Symposium on Principles of Distributed Computing, Toronto, ON, Canada, 18–21 August 2008; pp. 350–359.
37. Pagh, R.; Rodler, F.F. Cuckoo hashing. *J. Algorithms* **2004**, *51*, 122–144. [CrossRef]
38. Fotakis, D. Space efficient hash tables with worst case constant access time. *Theory Comput. Syst.* **2005**, *38*, 229–248. [CrossRef]
39. Le Scouarnec, N. Cuckoo++ Hash Tables: High-Performance Hash Tables for Networking Applications. In Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems (ANCS '18), Ithaca, NY, USA, 23–24 July 2018; ACM/IEEE: Red Hook, NY, USA, 2018; pp. 1–12.
40. Kirsch, A.; Mitzenmacher, M.; Wieder, U. More robust hashing: Cuckoo hashing with a stash. *SIAM J. Comput.* **2009**, *39*, 1543–1561. [CrossRef]
41. Mitzenmacher, M. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.* **2001**, *12*, 1094–1104. [CrossRef]
42. Bentley, J.L.; Sedgewick, R. Fast algorithms for adaptive rehashing. *Softw. Pract. Exp.* **2014**, *44*, 1475–1493.
43. Harris, T.L.; Fraser, K.; Pratt, I.A. A Practical Multi-Word Compare-and-Swap Operation. In Proceedings of the 16th International Symposium on Distributed Computing (DISC '02), Toulouse, France, 28–30 October 2002; pp. 265–279.
44. Enbody, R.J.; Du, H.C. Dynamic hashing schemes. *ACM Comput. Surv.* **1988**, *20*, 85–113. [CrossRef]
45. Fatourou, P.; Kallimanis, N.D.; Ropars, T. An efficient wait-free resizable hash table. In Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), Vienna, Austria, 16–18 July 2018; ACM: New York, NY, USA, 2018; pp. 111–120.
46. Lamping, J.; Veach, E. A fast, minimal memory, consistent hash algorithm. *arXiv* **2014**, arXiv:1406.2294. [CrossRef]
47. DeCandia, G.; Hastorun, D.; Jampani, M.; Kakulapati, G.; Lakshman, A.; Pilchin, A.; Sivasubramanian, S.; Vosshall, P.; Vogels, W. Dynamo: Amazon's highly available key-value store. In Proceedings of the ACM Symposium on Operating Systems Principles, Stevenson, WA, USA, 14–17 October 2007; pp. 205–220.
48. Cooper, B.F.; Silberstein, A. Benchmarking cloud serving systems with YCSB. In Proceedings of the ACM Symposium on Cloud Computing, Indianapolis, IN, USA, 10–11 June 2010; pp. 143–154.
49. Michael, M.M. High performance dynamic lock-free hash tables and list-based sets. In Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), Winnipeg, MB, Canada, 11–13 August 2002; ACM: New York, NY, USA, 2002; pp. 73–82.
50. Harris, T.L. A pragmatic implementation of non-blocking linked-lists. In Proceedings of the International Symposium on Distributed Computing, Lisbon, Portugal, 3–5 October 2001; pp. 300–314.
51. Purcell, C.; Harris, T. Non-blocking hashtables with open addressing. In *Distributed Computing*; Fraigniaud, P., Ed.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2005; Volume 3724, pp. 108–122. [CrossRef]
52. Herlihy, M.; Wing, J.M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* **1990**, *12*, 463–492. [CrossRef]
53. McKenney, P.E. RCU usage in the Linux kernel: One decade later. *ACM Queue* **2017**, *15*, 30–46.
54. Sanchez, D.; Kozyrakis, C. The ZCache: Decoupling ways and associativity. In Proceedings of the IEEE International Symposium on Microarchitecture, Atlanta, GA, USA, 4–8 December 2010; pp. 187–198.
55. Frigo, M.; Leiserson, C.E.; Prokop, H.; Ramachandran, S. Cache-oblivious algorithms. In Proceedings of the IEEE Symposium on Foundations of Computer Science, New York, NY, USA, 17–19 October 1999; pp. 285–298.
56. Shahrokhi, H.; Shaikhha, A. An Efficient Vectorized Hash Table for Batch Computations. In Proceedings of the 37th European Conference on Object-Oriented Programming (ECOOP 2023), Seattle, WA, USA, 17–21 July 2023; Volume 263, pp. 27:1–27:27.
57. Qureshi, M.K.; Thompson, D.; Patt, Y.N. The V-Way cache: Demand-based associativity via global replacement. In Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05), Madison, WI, USA, 4–8 June 2005; pp. 544–555.
58. Gaud, F.; Lepers, B.; Funston, J.; Dashti, M.; Fedorova, A.; Quéma, V.; Lachaize, R.; Roth, M. Challenges of memory management on modern NUMA systems. *Commun. ACM* **2015**, *58*, 59–66. [CrossRef]
59. Tran, T.N.; Kittitornkun, S. FPGA-Based Cuckoo Hashing for Pattern Matching in NIDS/NIPS. In *Managing Next Generation Networks and Services; Lecture Notes in Computer Science*; Springer: Berlin/Heidelberg, Germany, 2007; LNCS; Volume 4773, pp. 334–343.
60. Fan, B.; Andersen, D.G.; Kaminsky, M.; Mitzenmacher, M.D. Cuckoo Filter: Practically Better Than Bloom. In Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT '14), Sydney, Australia, 2–5 December 2014; Association for Computing Machinery: New York, NY, USA, 2014; pp. 75–88.

61. Bender, M.A.; Farach-Colton, M.; Johnson, R.; Kuszmaul, B.C.; Medjedovic, D.; Montes, P.; Shetty, P.; Spillane, R.P.; Zadok, E. Don't thrash: How to cache your hash on flash. In Proceedings of the ACM VLDB, Istanbul, Turkey, 27–31 August 2012; pp. 1627–1637.
62. Genuzio, M.; Ottaviano, G.; Vigna, S. Fast scalable construction of (minimal perfect) hash functions. *Comput. J.* **2022**, *65*, 358–371.
63. Izraelevitz, J.; Yang, J.; Swanson, S. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In Proceedings of the 21st International Conference Architectural Support for Programming Languages and Operating Systems (ASPLOS '16), Atlanta, GA, USA, 2–6 April 2016; pp. 427–442.
64. Lee, S.K.; Mohan, J.; Kashyap, S.; Kim, T.; Chidambaram, V. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19), Huntsville, ON, Canada, 27–30 October 2019; pp. 462–477.
65. Zhang, K.; Wang, K.; Yuan, Y.; Guo, L.; Lee, R.; Zhang, X. Mega-KV: A Case for GPUs to Maximize the Throughput of In-Memory Key-Value Stores. *Proc. VLDB Endow.* **2015**, *8*, 1226–1237. [CrossRef]
66. Gray, J.; Reuter, A. *Transaction Processing: Concepts and Techniques*; Morgan Kaufmann: San Francisco, CA, USA, 1993.
67. DeWitt, D.J.; Gerber, R. Multiprocessor hash-based join algorithms. In Proceedings of the VLDB, Stockholm, Sweden, 21–23 August 1985; pp. 151–164.
68. Garcia-Molina, H.; Ullman, J.D.; Widom, J. *Database Systems: The Complete Book*; Pearson: London, UK, 2008.
69. Zhang, B.; Du, D.H. NVLSM: A Persistent Memory Key-Value Store Using Log-Structured Merge Trees. In Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC '21), Santa Clara, CA, USA, 14–16 July 2021; pp. 385–398.
70. Zaharia, M.; Xin, R.S.; Wendell, P.; Das, T.; Armbrust, M.; Dave, A.; Meng, X.; Rosen, J.; Venkataraman, S.; Franklin, M.J.; et al. Apache Spark: A unified engine for big data processing. *Commun. ACM* **2016**, *59*, 56–65. [CrossRef]
71. Stoica, I.; Morris, R.; Karger, D.; Kaashoek, M.F.; Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In Proceedings of the ACM SIGCOMM, San Diego, CA, USA, 27–31 August 2001; pp. 149–160.
72. Bosshart, P.; Daly, D.; Gibb, G.; Izzard, M.; McKeown, N.; Rexford, J.; Schlesinger, C.; Talayco, D.; Vahdat, A.; Varghese, G.; et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Comput. Commun. Rev.* **2014**, *44*, 87–95. [CrossRef]
73. McKeown, N.; Anderson, T.; Balakrishnan, H.; Parulkar, G.; Peterson, L.; Rexford, J.; Shenker, S.; Turner, J. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Comput. Commun. Rev.* **2008**, *38*, 69–74. [CrossRef]
74. Chang, F.; Dean, J.; Ghemawat, S.; Hsieh, W.C.; Wallach, D.A.; Burrows, M.; Chandra, T.; Fikes, A.; Gruber, R.E. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* **2008**, *26*, 1–26. [CrossRef]
75. Geravand, S.; Ahmadi, M. Bloom filter applications in network security: A state-of-the-art survey. *Comput. Netw.* **2013**, *57*, 4047–4064.
76. Anderson, P.; Zhang, L. Fast and secure laptop backups with encrypted de-duplication. In Proceedings of the USENIX Conference on Large Installation System Administration, San Jose, CA, USA, 7–12 November 2010; pp. 1–8.
77. Rivest, R.L.; Shamir, A.; Adleman, L. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM* **1978**, *21*, 120–126. [CrossRef]
78. Dobin, A.; Davis, C.A.; Schlesinger, F.; Drenkow, J.; Zaleski, C.; Jha, S.; Batut, P.; Chaisson, M.; Gingeras, T.R. STAR: Ultrafast universal RNA-seq aligner. *Bioinformatics* **2013**, *29*, 15–21.
79. Marçais, G.; Kingsford, C. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics* **2011**, *27*, 764–770. [CrossRef] [PubMed]
80. Broder, A. On the resemblance and containment of documents. In Proceedings of the IEEE Compression and Complexity of SEQUENCES, Salerno, Italy, 13 June 1997; pp. 21–29.
81. Erbert, M.; Rechner, S.; Müller-Hannemann, M. Gerbil: A fast and memory-efficient k-mer counter with GPU support. *Algorithms Mol. Biol.* **2017**, *12*, 9.
82. Indyk, P.; Motwani, R. Approximate nearest neighbors: Towards removing the curse of dimensionality. In Proceedings of the ACM STOC, Dallas, TX, USA, 23–26 May 1998; pp. 604–613.
83. Johnson, J.; Douze, M.; Jégou, H. Billion-scale similarity search with FAISS. *IEEE Trans. Big Data* **2021**, *7*, 535–547.
84. Li, M.; Zhou, L.; Yang, Z.; Li, A.; Xia, F.; Andersen, D.G.; Smola, A. Parameter server for distributed machine learning. In Proceedings of the USENIX OSDI, Broomfield, CO, USA, 6–8 October 2014; pp. 583–598.
85. Kitaev, N.; Kaiser, L.; Levskaya, A. Reformer: The efficient transformer. In Proceedings of the ICLR, Virtual, 26–30 April 2020.
86. Mitzenmacher, M.; Vadhan, S. Why simple hash functions work: Exploiting the entropy in data. In Proceedings of the ACM-SIAM Symposium on Discrete Algorithms, San Fracisco, CA, USA, 20–22 January 2008; pp. 746–755.
87. Zhang, G.; Sanchez, D. Leveraging Caches to Accelerate Hash Tables and Memoization. In Proceedings of the MICRO '19: 52nd Annual IEEE/ACM International Symposium on Microarchitecture, Columbus, OH, USA, 12–16 October 2019; pp. 805–818.
88. Patrascu, M.; Thorup, M. The Power of Simple Tabulation Hashing. *J. ACM* **2012**, *59*, 1–50. [CrossRef]
89. Richter, S.; Alvarez, V.; Dittrich, J. A seven-dimensional analysis of hashing methods and its implications on query processing. *Proc. VLDB Endow* **2015**, *9*, 96–107. [CrossRef]
90. Mehlhorn, K.; Meyer, U. External-Memory Breadth-First Search with Sublinear I/O. In *Algorithms-ESA 2002*; Möhring, R., Raman, R., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2002; Volume 2461.

91. Herlihy, M. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.* **1991**, *13*, 124–149. [CrossRef]
92. Shalev, O.; Shavit, N. Split-Ordered Lists: Lock-Free Extensible Hash Tables. *J. ACM* **2006**, *54*, 13. [CrossRef]
93. Sundell, H.; Tsigas, P. Lock-free and practical doubly linked list-based deques using single-word compare-and-swap. In Proceedings of the OPODIS, Grenoble, France, 15–17 December 2004.
94. Michael, M.M. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.* **2004**, *15*, 491–504. [CrossRef]
95. Basu, A.; Kirman, N.; Kirman, M.; Chaudhuri, M.; Martinez, J. Scavenger: A New Last Level Cache Architecture with Global Block Priority. In Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007), Chicago, IL, USA, 1–5 December 2007; pp. 421–432.
96. Chowdhury, R.A.; Ramachandran, V. The Cache-Oblivious Gaussian Elimination Paradigm: Theoretical Framework and Parallel Implementation. In Proceedings of the SPAA, Cambridge, MA, USA, 30 July–2 August 2006; pp. 171–180.
97. Jünger, A.; Kobus, R.; Müller, A.; Hundt, C.; Xu, K.; Liu, W.; Schmidt, B. WarpCore: A library for fast hash tables on GPUs. *arXiv* **2020**, arXiv:2009.07914. [CrossRef]
98. Tata, N.T. MicroCuckoo Hash Engine for High-Speed IP Lookup. Master's Thesis, Department of Computer Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA, USA, 2017; pp. 1–102.
99. Pagh, A.; Pagh, R.; Ruzic, M. Linear Probing with Constant Independence. *SIAM J. Comput.* **2009**, *39*, 1107–1120. [CrossRef]
100. Fujimoto, R.M. Parallel and distributed simulation systems. In Proceedings of the 2001 Winter Simulation Conference, Arlington, VA, USA, 9–12 December 2001; Volume 1, pp. 147–157.
101. Li, W.; Cheng, Z.; Chen, Y.; Li, A.; Deng, L. Lock-Free Bucketized Cuckoo Hashing. In *Euro-Par 2023: Parallel Processing*; Cano, J., Dikaiakos, M.D., Papadopoulos, G.A., Pericàs, M., Sakellariou, R., Eds.; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2023; Volume 14100, pp. 275–288.
102. Eppstein, D. Cuckoo Filter: Simplification and Analysis. In Proceedings of the 15th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2016), Reykjavík, Iceland, 22–24 June 2016; Volume 53, pp. 8:1–8:12.
103. Dietzfelbinger, M.; Schellbach, U. On Risks of Using Cuckoo Hashing with Simple Universal Hash Classes. In Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA), New York, NY, USA, 4–6 January 2009; pp. 795–804.
104. Shor, P.W. Algorithms for quantum computation: Discrete logarithms and factoring. In Proceedings of the IEEE Annual Symposium on Foundations of Computer Science, Santa Fe, NM, USA, 20–22 November 1994; pp. 124–134.
105. Grover, L.K. A fast quantum mechanical algorithm for database search. In Proceedings of the ACM STOC, Philadelphia, PA, USA, 22–24 May 1996; pp. 212–219.
106. Ambainis, A. Quantum search algorithms. *SIGACT News* **2004**, *35*, 22–35. [CrossRef]
107. Giovannetti, V.; Lloyd, S.; Maccone, L. Quantum random access memory. *Phys. Rev. Lett.* **2008**, *100*, 160501. [CrossRef]
108. Eyerman, S.; Eeckhout, L. Fine-grained power modeling for modern processors. *ACM SIGARCH Comput. Archit. News* **2010**, *38*, 143–154.
109. Ma, Z.; Sha, E.H.-M.; Zhuge, Q.; Jiang, W.; Zhang, R.; Gu, S. Towards the design of efficient hash-based indexing scheme for growing databases on non-volatile memory. *Future Gener. Comput. Syst.* **2020**, *105*, 1–12. [CrossRef]
110. Yang, Y.; Wijeratne, S.; Zheng, D.; Prasanna, V.K. FASTHash: FPGA-Based High Throughput Parallel Hash Table. In Proceedings of the 2020 International Supercomputing Conference (ISC) Workshops, Virtual Event, 22–25 June 2020; pp. 1–9.
111. Li, J.; Deng, Y.; Zhou, Y.; Zhang, Z.; Min, G.; Qin, X. Towards Thermal-Aware Workload Distribution in Cloud Data Centers Based on Failure Models. *IEEE Trans. Comput.* **2023**, *72*, 586–599.
112. Kraska, A.; Beutel, A.; Chi, E.H.; Dean, J.; Polyzotis, N. The case for learned index structures. In Proceedings of the ACM SIGMOD, Houston, TX, USA, 10–15 June 2018; pp. 489–504.
113. Licks, G.P.; Meneguzzi, F. Automated Database Indexing Using Model-Free Reinforcement Learning. In Proceedings of the 2020 International Conference on Automated Planning and Scheduling (ICAPS 2020), Virtual, 26–30 October 2020; pp. 329–337.
114. Liong, V.E.; Lu, J.; Wang, G.; Moulin, P.; Zhou, J. Deep hashing for compact binary codes learning. In Proceedings of the IEEE Computer Vision and Pattern Recognition Conference, Boston, MA, USA, 7–12 June 2015; pp. 2475–2482.
115. Marcus, R.; Negi, P.; Mao, H.; Zhang, C.; Alizadeh, M.; Kraska, T.; Papaemmanouil, O.; Tatbul, N. Neo: A Learned Query Optimizer. *Proc. VLDB Endow* **2019**, *12*, 1705–1718.
116. Bellare, M.; Canetti, R.; Krawczyk, H. Keying Hash Functions for Message Authentication. In *Advances in Cryptology-CRYPTO '96*; Koblitz, N., Ed.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 1996; Volume 1109, pp. 1–15.
117. Bar-Yosef, N.; Wool, A. Remote Algorithmic Complexity Attacks Against Randomized Hash Tables. In Proceedings of the Second International Conference on Security and Cryptography, Barcelona, Spain, 28–31 July 2007.
118. Chakraborty, T.; Saia, J.; Young, M. Defending hash tables from algorithmic complexity attacks with resource burning. *Theor. Comput. Sci.* **2024**, *2014*, 114762.
119. Karger, D.; Lehman, E.; Leighton, T.; Levine, M.; Lewin, D.; Panigrahy, R. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC '97), El Paso, TX, USA, 4–6 May 1997; pp. 654–663.

120. Cakir, F.; He, K.; Bargal, S.A.; Sclaroff, S. MIHash: Online Hashing with Mutual Information. *IEEE Trans. Pattern Anal. Mach. Intell.* **2017**, *41*, 1572–1585.
121. Misra, S.; Bland, L.C.; Cardwell, S.G.; Incorvia, J.A.C.; James, C.D.; Kent, A.D.; Schuman, C.D.; Smith, J.D.; Aimone, J.B. Probabilistic Neural Computing with Stochastic Devices. *Adv. Mater.* **2023**, *35*, 2204569. [CrossRef]
122. Tapia-Fernández, S.; García-García, D.; García-Hernandez, P. Key Concepts, Weakness and Benchmark on Hash Table Data Structures. *Algorithms* **2022**, *15*, 100. [CrossRef]
123. Yusuf, A.D.; Abdullahi, S.; Boukar, M.M.; Yusuf, S.I. Collision Resolution Techniques in Hash Table: A Review. *Int. J. Adv. Comput. Sci. Appl.* **2021**, *12*, 152–161. [CrossRef]
124. Lehmann, H.P.; Mueller, T.; Pagh, R.; Pibiri, G.E.; Sanders, P.; Vigna, S.; Walzer, S. Modern Minimal Perfect Hashing: A Survey. *arXiv* **2025**, arXiv:2506.06536. [CrossRef]
125. Misto, S. Efficiency in Hash Table Design: A Study of Collision Resolution Strategies and Performance. Bachelor's Thesis, Malmö University, Malmö, Sweden, 2024.
126. Broder, A.; Mitzenmacher, M. Network Applications of Bloom Filters: A Survey. *Internet Math.* **2004**, *1*, 485–509. [CrossRef]
127. Tarkoma, S.; Rothenberg, C.E.; Lagerspetz, E. Theory and Practice of Bloom Filters for Distributed Systems. *IEEE Commun. Surv. Tutor.* **2012**, *14*, 131–155. [CrossRef]
128. Zhao, Y.; Dai, W.; Wang, S.; Xi, L.; Wang, S.; Zhang, F. A Review of Cuckoo Filters for Privacy Protection and Their Applications. *Electronics* **2023**, *12*, 2809. [CrossRef]
129. Zink, T. *A Survey of Hash Tables with Summaries for IP Lookup Applications*; Technical Report; University of Konstanz: Konstanz, Germany, 2011.
130. Wang, J.; Zhang, T.; Song, J.; Sebe, N.; Shen, H.T. A Survey on Learning to Hash. *IEEE Trans. Pattern Anal. Mach. Intell.* **2018**, *40*, 769–790. [CrossRef]
131. Lua, E.K.; Crowcroft, J.; Pias, M.; Sharma, R.; Lim, S. A Survey and Comparison of Peer-to-Peer Overlay Network Schemes. *IEEE Commun. Surv. Tutor.* **2005**, *7*, 72–93.
132. Luo, X.; Wang, H.; Wu, D.; Chen, C.; Deng, M.; Huang, J.; Hua, X.S. A Survey on Deep Hashing Methods. *ACM Trans. Intell. Syst. Technol.* **2023**, *14*, 3. [CrossRef]
133. Wang, J.; Liu, W.; Kumar, S.; Chang, S.F. Learning to Hash for Indexing Big Data-A Survey. *Proc. IEEE* **2016**, *104*, 34–57. [CrossRef]
134. Manna, A.; Dewan, D.; Sheet, D. Structured hashing with deep learning for modality, organ, and disease content sensitive medical image retrieval. *Sci. Rep.* **2025**, *15*, 8912. [CrossRef] [PubMed]
135. Menezes, A.J.; van Oorschot, P.C.; Vanstone, S.A. *Handbook of Applied Cryptography*; CRC Press: Boca Raton, FL, USA, 1996.
136. Bellare, M.; Rogaway, P. Random Oracles Are Practical: A Paradigm for Designing Efficient Protocols. In Proceedings of the ACM Conference Computer and Communications Security (CCS '93), Fairfax, VA, USA, 3–5 November 1993; pp. 62–73.
137. Wang, Q.; Lu, Y.; Li, J.; Xie, M.; Shu, J. Nap: Persistent Memory Indexes for NUMA Architectures. *ACM Trans. Storage* **2022**, *18*, 2. [CrossRef]
138. Jamil, S.; Salam, A.; Khan, A.; Burgstaller, B.; Park, S.S.; Kim, Y. Scalable NUMA-aware persistent B^+-tree for non-volatile memory devices. *Clust. Comput.* **2023**, *26*, 2865–2881. [CrossRef]
139. Coburn, J.; Caulfield, A.M.; Akel, A.; Grupp, L.M.; Gupta, R.K.; Jhala, R.; Swanson, S. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11), Newport Beach, CA, USA, 5–11 March 2011; pp. 105–118.
140. Chang, C.; Zhang, Z.; Zhou, Y.; Li, J. Uncertainty-Aware Discrete Hashing for Low-Risk Binary Code Generation. *Inf. Sci.* **2025**, *660*, 2671.
141. Karras, A.; Karras, C.; Schizas, N.; Sioutas, S.; Zaroliagis, C. Algorithmic Aspects of Distributed Hash Tables on Cloud, Fog, and Edge Computing Applications: A Survey. In *Algorithmic Aspects of Cloud Computing. ALGOCLOUD 2023*; Chatzigiannakis, I., Karydis, I., Eds.; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2024; Volume 14053.
142. Balatsouras, C.-P.; Karras, A.; Karras, C.; Karydis, I.; Sioutas, S. WiCHORD+: A Scalable, Sustainable, and P2P Chord-Based Ecosystem for Smart Agriculture Applications. *Sensors* **2023**, *23*, 9486.
143. Gagniuc, P.A. *Antivirus Engines: From Methods to Innovations, Design, and Applications*; Elsevier Syngress: Cambridge, MA, USA, 2024; pp. 1–656.