MASTER'S THESIS 2026

High-performance, High-Throughput memory-efficient lookup tables for schemaless relational data

Hugo Persson, Yazan Al-Aswad

ISSN 1650-2884
LU-CS-EX: 2026-26

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2026-26

High-performance, High-Throughput memory-efficient lookup tables for schemaless relational data

Högpresterande, höggenomströmning, minneseffektiv uppslagstabell för schemalösa relationella data

Hugo Persson, Yazan Al-Aswad

High-performance, High-Throughput memory-efficient lookup tables for schemaless relational data

Hugo Persson
hugo.e.persson@gmail.com

Yazan Al-Aswad
yazanalaswad0@gmail.com

June 16, 2026

Master's thesis work carried out at Neo4j.

Supervisors: Lukas Gustavsson, lukas.gustavsson@neo4j.com
Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

Examiner: Per Andersson, per.andersson@cs.lth.se

Abstract

Bulk importing data into Neo4j requires a temporary mapping from external identifiers to internal 64-bit identifiers to resolve relationships before the graph is materialized. At a billion-key scale, this lookup becomes a major bottleneck, yet no prior work has empirically compared modern retrieval structures in this setting.

This thesis benchmarks state-of-the-art lookup structures for the externalId-to-internalId mapping. Four architectures were implemented on top of BuRR and PtrHash, including hybrid designs that combine a coarse-grained well function with sharded per-well minimal perfect hash functions. They were evaluated against the `BinarySearchLookup` baseline at 10^9 and 10^{10} keys on a Lenovo ThinkPad.

At 10^9 keys, `Direct_PTR` reduced end-to-end import time by 84 % (6.1× speedup). At 10^{10} keys, `Hybrid_BuRR_PTR` was 11 % faster than Direct_PTR and the BinarySearchLookup is too slow to benchmark because of I/O page faults.

Keywords: Neo4j, graph databases, bulk import, minimal perfect hash functions, PtrHash, BuRR, lookup structures, retrieval data structures, identifier mapping, succinct data structures, external memory algorithms, cache efficiency, benchmarking

Acknowledgements

We would like to thank Lukas Gustavsson and Anton Klarén for always being available and guiding us during this thesis. We also thank Simon Priisalu for his help with benchmarking and testing and our supervisor Jonas Skeppstedt for his responsiveness and pragmatism.

Finally, we thank everyone at Neo4j. From the beginning, we felt welcome at the company, and we are grateful for the supportive engineering environment.

Contents

Chapter 1
Introduction

The adoption of graph databases has accelerated rapidly in recent years, driven by increasingly connected and relationship-centric data in modern applications. Use cases such as fraud detection, knowledge graphs, recommendation systems, supply chain analytics, cybersecurity, and generative AI pipelines rely heavily on efficient graph representations to model complex relationships [11]. Neo4j has positioned itself as a leading graph-native database vendor [5].

For some use cases Graph Databases are not constructed from scratch, instead, they are built from massive volumes of pre-existing, non-graph data imported from files of different formats, the formats and reading of them is not covered by this thesis[4]. This data is provided as nodes and edges where structure is unknown before import, duplicate nodes may exist and need to be handled gracefully. A special case occurs in analytical workflows where another system serves as the primary source of truth. In these scenarios, Neo4j is used transiently: data is imported, analytical queries are executed, and the database is subsequently discarded. This places additional requirements on the import process being efficient.

During the import process, a fundamental step is the construction of a mapping from external identifiers (e.g., strings or long integers originating from source systems) to internal identifiers used by the database engine. This temporary externalId-to-internalId lookup structure is required to resolve relationships between nodes before the graph is fully materialized. Once the import is complete, this mapping is discarded.

While individual data structures such as hash maps and minimal perfect hash functions (MPHFs) have been studied extensively in isolation [28], no existing research compares these data retrieval structures against each other in the context of graph-modeled data imports. This gap leaves database engineers without empirical guidance on which structures best balance throughput, memory consumption, and correctness during bulk graph construction.

This thesis systematically benchmarks state-of-the-art lookup tables for the externalId-to-internalId mapping during bulk graph import into Neo4j. It evaluates throughput and memory consumption.

1.1 Research questions

This thesis aims to answer the following research questions:

- **RQ1** How do different state-of-the-art lookup table data structures perform in terms of optimizing the import process for graph databases?
- **RQ2** Can modification/combinations of different state-of-the-art lookup table data structures be more efficient at performing graph database imports?

Chapter 2
Background

This chapter gives background for the problem Neo4j is facing and how it relates to Neo4j as a product. Introduction of theoretical concepts needed to understand this thesis is given.

2.1 Problem context

2.1.1 Databases

An important background for understanding why this work is useful is to understand what databases are. In computer systems, a way to store data and later retrieve it is needed. Data may also need to be transformed in various ways; for example, storing transactions performed by a user and then calculating the total sum of those transactions. There are different types of databases, each suited to unique use cases.

Neo4j develops a specific type of database called a graph database.

The two most common database categories are relational and non-relational [9]. Relational databases model data as structured tables that can be associated with one another through defined relationships [12]. Each table consists of a set of columns, and relationships are established by linking columns across different tables.

In contrast, non-relational databases model data as collections of documents. Each collection contains multiple documents, which are typically characterized by a nested and flexible data structure.

Graph databases model data differently from relational and non-relational databases. In graph databases, data is represented as nodes and edges. Nodes represent entities, while edges represent relationships between these entities. Each node can have labels and properties. Labels act as a set of tags that can be used to group nodes together, while properties are key-value pairs used to store additional information about a node. Edges have *types*, which are similar to labels, and describe the relationship between nodes.

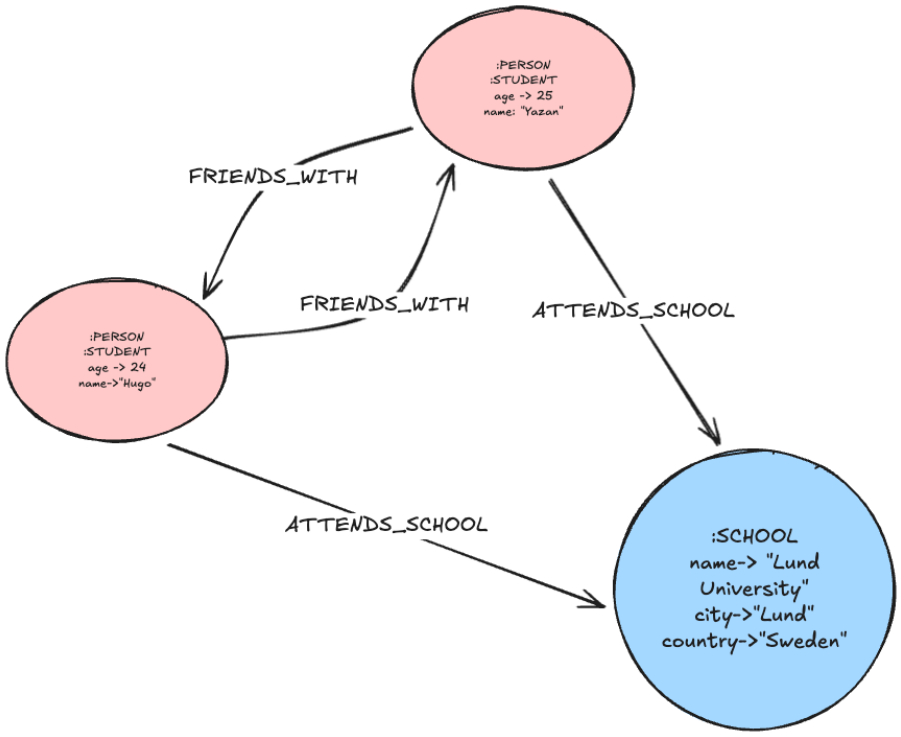

Figure 2.1: Example of a graph database with nodes and edges

2.1.2 Neo4J

Neo4j, an industry-leading graph database and software company, is where this master's thesis is conducted. The goal of this work is to apply the findings to optimize the import process, which involves creating and populating a database by reading input data from generic files such as CSV or Parquet files [11]. To understand the design choices and challenges addressed in this thesis, it is important to have a background in Neo4j, both as a business and as a database system.

As a business, Neo4j develops and maintains the Neo4j database and offers it in two main ways [11]. Customers can use Neo4j through a Database-as-a-Service (DBaaS) model, where Neo4j hosts and manages the database. This product is called AuraDB [1]. Alternatively, users can run the database on their own infrastructure, referred to as on-prem hosting [2].

This difference in deployment environments affects how the problem is approached, since the environment in which the database will run is unknown. There are trade-offs between memory consumption and CPU performance. If the available system resources were known, these trade-offs could be optimized more precisely. However, because Neo4j must accommodate enterprise customers who might run Neo4j in their own data centers on hardware configurations unknown to Neo4j, optimizations must be made for unknown configurations while remaining performant.

2.1.3 Importing data into Neo4j

Importing data into Neo4j can be done using the *neo4j-admin database import* tool. This is performed by providing a set of *nodes* CSV files and a set of *edges* CSV files [4]. In this report,

these are referred to as two files, `Nodes.csv` and `Edges.csv`, although in practice they may consist of multiple files that are concatenated.

2.1.4 X1 store

The X1 data store is the primary structure for storing nodes and their relationships in Neo4j. A basic understanding of this layout is necessary to motivate the implementation decisions made in this work.

The store is organized as a contiguous sequence of fixed-size X1 blocks, where each block represents a single node together with a portion of its associated data. Blocks are addressed by a unique identifier corresponding to their position in the store (e.g., id $= 0$ refers to the first block). This direct indexing enables efficient lookup and predictable memory access patterns.

Each X1 block is 128 bytes in size and is divided into two equally sized sub-blocks of 64 bytes each. One sub-block is reserved for node properties (e.g., labels and key–value pairs), while the other is dedicated to relationship data. This design enforces a strict upper bound on the amount of data that can be stored inline per node, ensuring compact, cache-friendly storage for small nodes, as illustrated in Figure 2.2.

Relationships in Neo4j are directional; however, they are stored redundantly at both the start and end nodes. As a result, each relationship is represented twice within the X1 store. This duplication enables efficient traversal from either endpoint without requiring additional indirection, at the cost of increased storage usage and write amplification during insertion.

The fixed-size layout improves data locality and simplifies memory management, but also introduces limitations. When a node's properties or relationships exceed the 64-byte allocation limit, the data must be moved to external storage (e.g., XD), which introduces additional pointer indirection and potential cache misses.

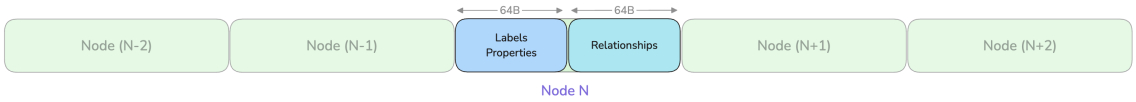

Figure 2.2: Block storage. Adapted from [30].

2.2 Theoretical foundation

2.2.1 Memory Accesses

Modern processors execute instructions far faster than main memory can supply data. To bridge this gap, hardware designers interpose a hierarchy of progressively larger but slower memories between the CPU and main storage [3]. Table 2.1 summarizes the typical capacity and access latency at each level. Each step down the hierarchy is roughly an order of magnitude slower than the one above: accessing DRAM is approximately 100 times slower than the L1 cache, and accessing an SSD is roughly 1000 times slower than DRAM [23].

Table 2.1: Typical capacity and access latency for each level of the memory hierarchy on modern server-class processors [3, 23].

Level	Typical size	Access latency
L1 cache	32–64 KB per core	~ 1 ns
L2 cache	256 KB–1 MB per core	~ 3–5 ns
L3 cache	8–32 MB (shared)	~ 10–20 ns
DRAM	16–512 GB	~ 50–100 ns
SSD	256 GB–8 TB	~ 10 000–100 000 ns
HDD	1–20 TB	~ 5 000 000–10 000 000 ns

Cache lines and data transfer Data is not fetched byte-by-byte from lower levels of the hierarchy. Instead, the hardware transfers data in fixed-size blocks called *cache lines*, typically 64 bytes on modern x86-64 and ARM processors [23]. When the CPU accesses a memory address whose cache line is not present in the L1 cache, a *cache miss* occurs, and the processor stalls until the entire 64-byte line has been fetched from a lower level. Consequently, if a data structure's working set for a single query fits within one or two cache lines, the query can be answered with minimal stalling; if the query requires data scattered across many cache lines, performance degrades regardless of algorithmic complexity. Modern CPUs also perform *hardware prefetching*: when sequential access patterns are detected, the hardware speculatively loads adjacent cache lines before they are explicitly requested [23].

Locality of reference Caches are effective because real programs exhibit two forms of locality [3]. *Temporal locality* means that a recently accessed memory location is likely to be accessed again soon. Caches exploit this by retaining recently used lines. *Spatial locality* means that when a location is accessed, nearby locations are likely to be accessed shortly afterward. The cache line mechanism exploits this by loading a contiguous block around the requested address. Algorithms and data structures that exhibit strong locality achieve high cache hit rates and thus approach the throughput of the fastest available cache level. Conversely, poor locality forces frequent fetches from DRAM or slower storage, negating any algorithmic improvements [23].

Implications for compact data structures For data structures queried billions of times, such as when importing 10^{14} keys, the number of cache misses per query can dominate wall-clock time. A structure requiring one random memory access per query runs at DRAM throughput; a structure requiring two random accesses approximately halves effective throughput [28]. Compactness, measured in bits per key, is therefore not merely a matter of saving memory it directly determines whether the structure fits in cache. This motivates designs such as PtrHash, which stores 8-bit pilots in contiguous arrays to maximize spatial locality and uses a cache line-aligned Elias-Fano encoding 2.2.9 for its remap table so that each remapping query touches at most one additional cache line [27]. Similarly, the BuRR retrieval structure concentrates its query's memory accesses into a narrow band, achieving high locality [20].

Storage beyond DRAM When the working set exceeds available DRAM, data must be moved between main memory and secondary storage. As 2.1 shows, SSDs are roughly

1000 times slower than DRAM, and HDDs another 100 times slower still. An I/O round-trip to secondary storage therefore far exceeds the cost of a cache miss. For the Neo4j import use case, the X1 store (128-byte blocks per node, 2.1.4 may not fit entirely in memory, making the number of distinct pages touched during edge insertion a critical performance factor. Neo4j manages this through its own *page cache*, discussed in Section 2.2.2.

2.2.2 Paging

Rather than relying on the operating system's virtual-memory subsystem to decide which data resides in DRAM, Neo4j implements its own user-space *page cache* [29].

Pages and page files The page cache divides every on-disk store file into fixed-size *pages* of 8 KB by default. A bounded pool of in-memory page frames backs this abstraction: when a store file is opened, it is *mapped* into the cache and represented by a `PagedFile` handle. Application code accesses a page by acquiring a `PageCursor`, which pins the page in memory for the duration of the access. Reads use a shared lock, while writes use a shared write lock that marks the page as *dirty*, ensuring it will be flushed to disk before eviction.

Page faults and eviction When a cursor requests a page that is not currently resident in the cache, a *page fault* occurs: the page cache reads the 8 KB block from the underlying file into a free frame. If no free frames are available, the cache must first *evict* an existing page. The eviction policy is a clock-based replacement algorithm: each page maintains a small usage counter that is incremented on access and decremented by a background sweep. Pages whose counter reaches zero are eligible for eviction. If a page is dirty, it is flushed to disk before its frame is reclaimed [29]. The I/O latency of the secondary storage device dominates the cost of a page fault. As 2.1 shows, a single SSD access takes roughly 10–100 µs and an HDD access 5-10 ms, orders of magnitude slower than a DRAM access.

Implications for this work In the Neo4j import use case, the X1 store consists of 128-byte records, one per node (2.1.4) and may contain billions of entries. A single 8 KB page therefore holds approximately 64 node records. When the store exceeds the page cache's capacity, every access to an evicted page triggers a page fault and an expensive I/O round-trip. Because multiple node records reside on the same page, batching updates that target the same page distributes the cost of a single read-write pair across many updates. This observation motivates the *wells* partitioning strategy described in 2.3.2, by grouping insertions so that all updates to a given page range are performed together, the total number of page faults is minimized.

2.2.3 Hash Functions

A *hash function* $h : U \to \{0, \ldots, m-1\}$ maps keys drawn from a large universe U to a fixed-size integer range. Three properties are desirable for data-structure applications [7]:

- *determinism* - the same key always produces the same output,
- *uniformity* - outputs are distributed as evenly as possible across the range,

- *avalanche effect* - a small change in the input produces a large, unpredictable change in the output, which helps avoid systematic collisions.

Scope in this thesis This thesis does not include a separate empirical comparison of hash functions. Instead, hashing is treated as an implementation choice, and a single primitive is shared between the BuRR and PtrHash implementations evaluated in this work. This keeps the evaluation focused on the behaviour of the studied data structures and their integration into Neo4j, rather than on benchmarking alternative hashing primitives in isolation.

Master Hash Code (MHC) Both BuRR and PtrHash are driven by a 64-bit *master hash code* (MHC). The MHC is computed once per key from the original input and then reused as the only "key" the data structures see, so that internal layers can derive cheap secondary hashes by multiplying the MHC with layer-specific constants without ever rehashing the original input. Storing $(MHC, rowId)$ pairs as the canonical fixture format also lets the same precomputed input drive every benchmark configuration.

Mixing function The mixing function used to produce the MHC, denoted `BuRRMasterHashFunction`, is a single-round rotate-multiply mixer in the style of FxHash, ported from the constant used in the Rust PtrHash reference implementation [27]. Given a 64-bit input value v and a 64-bit seed s, the MHC is computed as

$$\mathrm{MHC}(v, s) \;=\; \mathrm{rotl}_{17}(v \cdot C) \;\oplus\; s, \tag{2.1}$$

where C = `0x517cc1b727220a95` is a fixed odd 64-bit multiplier and rotl_{17} denotes a 17-bit left rotation on a 64-bit word.

2.2.4 Minimum Perfect Hashing Functions

A Minimum Perfect Hash Function (MPHF) is the first data structure examined in this work. This section defines MPHFs, the problem they solve, and how they are constructed. MPHFs build on two related primitives, hash functions (HF) and perfect hash functions (PHF), which are introduced first.

The first type of data structure is a *hash table*, which stores key-value pairs in an array-like structure of fixed size m. A *hash function* 2.2.3 h maps a key from a (typically very large) key space K to an integer index in $\{0, \ldots, m-1\}$, which is then used to access the corresponding slot in the table [7].

Because $|K| \gg m$ in most realistic settings, different keys may map to the same slot, creating a *collision* [7]. The frequency and cost of collisions depend strongly on the *load factor* $\alpha = n/m$, where n is the number of stored elements.

When collisions occur, the hash table must apply a collision-resolution strategy [7]. There exist different ways to handle this, two common families are (i) *separate chaining*, where each slot stores a secondary structure (e.g., a list) of colliding items, see 2.3 for example, and (ii) *open addressing*, where colliding keys are placed into alternative slots following a probe sequence (e.g., linear, quadratic, or double hashing).

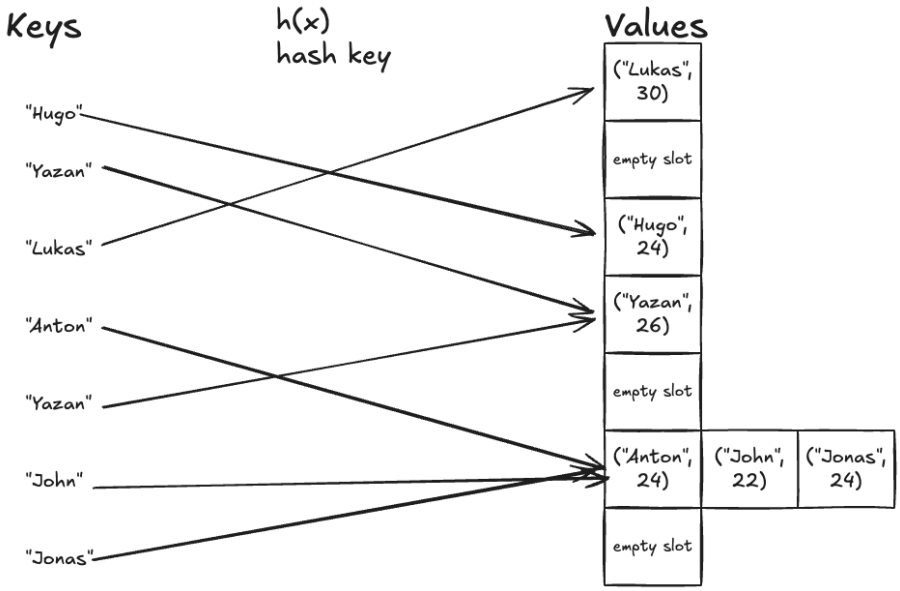

Figure 2.3: Example showing separate chaining hashing function

Perfect Hashing Functions If the restriction is placed on the Hashing Function that no collisions may occur, the result is a Perfect Hash Function (PHF) [28]. More formally a PHF maps keys in K to $m \geq n$ without collisions. This allows for constant-time lookups without the overhead of collision resolution, but constructing such a function can be computationally expensive and may require more space than a standard hash table, especially if m is much larger than n.

Minimum Perfect Hashing Functions (MPHF) A subset of PHF are *Minimum Perfect Hash Functions* (MPHF), which place the additional constraint that $m = n$ [28]. This creates a bijective mapping between keys and slots. The information-theoretic lower bound on the space required to represent an MPHF is $\log_2(e) \approx 1.44$ bits per key [28]; state-of-the-art constructions come within 0.1% of this bound [28]. Queries can be answered in $O(1)$ time, with the fastest practical schemes requiring only a single memory access [28]. Construction, however, is more expensive than for a general PHF: typical algorithms run in $O(n)$ expected time but with non-trivial constant factors, and may require sophisticated techniques to scale to billions of keys while maintaining fast query times [28].
To be clear, the space cost mentioned prior maps a key to a unique arbitrary integer $i \in \mathbb{N}_0$ such that $i \in [0, m)$. That integer can then be used as an index to an array, within which one can store any value one might want to store. Clearly, mapping the keys to arbitrary values incurs an extra cost of vn bits, where v is the bit size of the values.

2.2.5 Approximate Membership Structures

Filters, also called approximate membership query (AMQ) data structures, are a class of data structures used to determine whether a key belongs to a particular set. Formally, an AMQ represents a set $S \subseteq U$ over some universe U and supports membership queries of the form "$x \in S$?" It returns either **true** (indicating that x is possibly in S) or **false** (indicating that x is

definitely not in S), with a controllable probability of false positives but no false negatives.

The primary reason for using AMQs instead of traditional hash tables is their superior space efficiency and, often, faster query time. AMQs exploit the fact that the value itself does not need to be stored or retrieved. The keys need not be stored explicitly either. Instead, AMQs generally store a compact, deterministically queryable representation of set membership, aiming to balance strong memory savings, fast query and construction speed, and a controllable false-positive rate.

2.2.6 Retrieval Data Structures

Retrieval data structures solve a restricted version of the dictionary problem. They associate a value with each key in a set S, but may return arbitrary answers for keys not in S. More formally, given a set $S \subseteq U$ and a function $f : S \to \{0,1\}^r$, a retrieval data structure stores a representation of f such that, for any query $x \in S$, it returns $f(x)$, while for any $x \notin S$ it may return an arbitrary value [19].

Space and time bounds

Retrieval data structures can be implemented using space $nv + o(n)$ bits while supporting $O(1)$-time queries when the value size v is sufficiently small. However, for larger values (e.g., $v = \Theta(\log n)$), there is an inherent time–space trade-off: in particular, achieving constant query time with space $nv + o(n)$ bits is impossible, and any such data structure must incur an additional space cost [26].

Reduction

Importantly, under the assumption of free access to fully random hash functions, every static retrieval data structure yields an approximate membership data structure whose false positive rate is 2^{-r}, with no additional space and only constant extra evaluation time. This transformation is straightforward: one can use any retrieval structure as a filter by storing, for each key, a fingerprint that is derivable from the key itself (e.g., via hashing) during construction. Upon querying, the fingerprint is recomputed directly from the key and also recovered through the retrieval structure; a mismatch then guarantees non-membership [19].

Retrieval data structures can be viewed as systems of linear equations over $\mathbb{F}_2$, where each key k_i is mapped to a row in a matrix A, with a corresponding value row in a matrix B. Using

$$AX = B, \tag{2.2}$$

X can be viewed as a solution matrix that maps each row in A to a row in B. Importantly, the matrix A does not contain the original keys themselves, rather, each key is hashed to create a row in A. Thus, the original keys can be of any type as long as they can all be hashed consistently to a fixed-width bit string. The associated values, however, can only be fixed-width bit strings [18]. In most retrieval constructions, the system is defined over $\mathbb{F}_2$, so the coefficient matrix is binary and addition reduces to bitwise XOR. This makes both the analysis and the implementation especially simple and efficient on modern hardware.

Clearly, not all equation systems of this form are solvable. The key requirement is that A has full row rank over $\mathbb{F}_2$. The hashes derived from the keys can naturally cause linear

dependencies, which are discovered at construction time. A standard remedy is to rehash all keys with a different hash function or, in practice, a different seed. This naturally introduces another quantity of interest: the *construction success rate*.

Furthermore, even when solvability holds, dense linear systems can be prohibitively expensive to solve. Straightforward construction via Gaussian elimination can require expected $O(n^3)$ time and $O(n^2)$ scratch space [19]. In practice, sparse or otherwise structured systems are preferred because they make both lookup and construction much more efficient. In the basic construction of Dietzfelbinger and Pagh, each row has exactly k ones, so evaluating a query requires XORing only k table entries, resulting in $O(k)$ lookup time [19].

Most later work, therefore, focuses on creating constraints that make the system cheaper to solve and faster to query.

2.2.7 Ribbon

Ribbon is a data structure that can be used both as an AMQ and as a retrieval data structure. It is designed to improve locality compared to earlier xor-based approaches. As the name suggests, the idea is to create a matrix that visually resembles a ribbon running along the diagonal. More formally, Ribbon constructs a banded linear system over $\mathbb{F}_2$, where each key x is assigned a starting position $s(x)$ and a width-w coefficient vector $c(x)$. The corresponding row can have nonzero entries only in a contiguous region of width w starting at $s(x)$. For efficient elimination, the first bit of $c(x)$ is fixed to 1. When the rows are ordered by starting position, the nonzero entries form a ribbon-shaped band through the matrix, as illustrated in Figure 2.4 [22, 21].

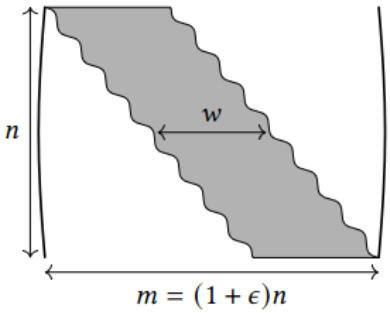

Figure 2.4: Figure 2 in [22] shows a visual representation of the banded diagonal matrix.

Unlike many modern linear-system-based retrieval structures and filters that use hypergraph peeling, Ribbon uses on-the-fly Gaussian elimination. In Ribbon construction, key-value equations are inserted incrementally into a linear system using the procedure shown in Figure 2.5. While the on-the-fly aspect of construction dates back to 2009 [17], the added banding yields several further advantages. Dillinger and Walzer call this *Rapid Incremental Boolean Banding ON the fly* [22].

Ribbon's primary advantage is its increased locality. Because all operations remain confined to a small contiguous window of size w, both construction and queries access memory in a highly localized and cache-friendly manner. This contrasts with earlier approaches that rely on a small number of randomly distributed accesses, which significantly reduce the cache hit rate.

A significant limitation of Ribbon is that the required ribbon width grows quickly with the key count when very low overhead is targeted. For use cases with billions or even trillions of keys, as in our setting, minimizing overhead is essential. Ribbon struggles in these scenarios: for $\varepsilon \leq 3.5\%$ and a construction success rate of at least 50%, standard Ribbon with $w = 64$ scales only to about $n \leq 10^4$ keys, while $w = 128$ scales only to about $n \leq 10^6$

Algorithm 1: Adding a key's equation to the linear system M.

```
1  i ← s(x)
2  c ← c(x)
3  b ← b(x)
4  loop
5      if M.c[i] = 0 then // row i of M is empty
6          M.c[i] ← c
7          M.b[i] ← b
8          return SUCCESS (inserted)
9      c ← c ⊕ M.c[i]
10     b ← b ⊕ M.b[i]
11     if c = 0 then
12         if b = 0 then return SUCCESS (redundant)
13         else return FAILURE (inconsistent)
14     j ← findFirstSet(c) // a.k.a. BitScanForward
15     i ← i + j
16     c ← c >> j // logical shift last toward first
```

Figure 2.5: Ribbon insertion algorithm [21].

keys. This limitation is one of the main motivations for the bumped multi-layer design of BuRR [21].

The result is a data structure that can be constructed with high probability, using space arbitrarily close to the information-theoretic minimum, with near-linear construction time and constant-time queries for small r. However, it scales poorly to very large data sets [21].

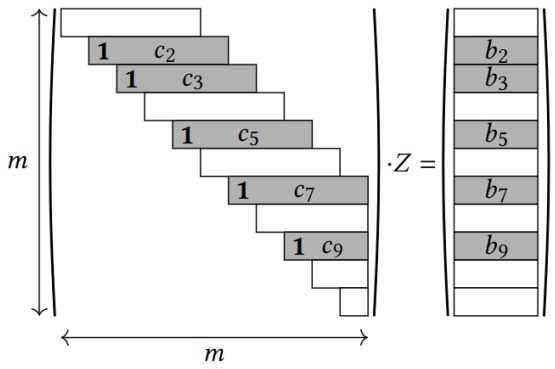

Figure 2.6: Visual representation of Ribbon construction matrices [21].

ICML

To maximally exploit Ribbon's locality, Dillinger and Walzer recommend storing the solution in an interleaved column-major layout (ICML). This minimizes cache misses at query time. The details are omitted here for brevity [22].

To address the size limitation of Ribbon, Dillinger et al. introduced a simple multi-layer extension. The idea is to remove the few rows that cause linear dependencies in a Ribbon construction and move them to a different Ribbon system. Some auxiliary metadata is stored to indicate which rows were moved ("bumped") to subsequent layers. This approach is referred to as *BuRR* (Bumped Ribbon Retrieval).

More formally, given a system $AX = B$, a Ribbon construction can insert most keys successfully, even when the ribbon width w is small. The remaining rows that would introduce linear dependencies during the construction of layer i are removed and handled in layer $i+1$ after rehashing. It is shown that only $O\left(\frac{n \log w}{w}\right)$ keys need to be bumped in expectation [21]. Therefore, after a constant number of layers (four in the original paper), the remaining keys can be handled efficiently using an auxiliary data structure such as a traditional hash table.

Bumping individual keys would be extremely expensive in terms of space, as it would require at least one extra bit per key. This, combined with the fact that most linear dependencies arise from local conflicts, motivates the use of *buckets* for bumping. Buckets are a conceptual grouping of columns in the matrix A, each of width

$$b = O\left(\frac{w^2}{\log w}\right). \tag{2.3}$$

Each bucket is allocated a small amount of metadata (e.g., 2 bits) to specify which subsection of the bucket gets bumped. Additional bits give finer bumping granularity at the cost of additional space. If a linear dependency is detected during insertion, the entire subsection of the bucket where it was detected is bumped, and this choice is reflected in the metadata bits. The borders of the intra-bucket subdivisions are defined using a set of threshold values. In the paper, a 2-bit variant with values $\{0, \ell, u, b\}$ is used, where ℓ and u are chosen empirically [21].

For this bumping algorithm to be space efficient, insertion must be performed in a specific order: buckets are processed in ascending order of starting position $s(x)$, while the entries within each bucket are processed in descending order of starting position. The reason for this insertion order is that insertions from bucket i can spill over into bucket $i+1$, creating additional load on the left side of bucket $i+1$. By forcing a right-to-left intra-bucket order, the rows that get bumped are the most recently inserted ones, thereby avoiding unnecessary rollbacks and minimizing space usage [21].

Another optimization implemented in BuRR is *overloading*, i.e., choosing a negative slack parameter $\varepsilon < 0$ in

$$m = (1 + \varepsilon)n. \tag{2.4}$$

The idea is to intentionally overfill the table in anticipation of bumping. This reduces unused space and shifts the space overhead toward the bucket metadata, yielding better overall space efficiency. It is shown that, by choosing $\varepsilon < 0$, the excess table slots can be almost eliminated, so that the metadata becomes the dominant source of overhead after construction [21].

The resulting data structure is formally analyzed in Theorem 2 of Dillinger et al., restated as Theorem 2.2.1 [21].

Theorem 2.2.1 (BuRR construction and query bounds, restated from [21])**.** *An r-bit BuRR data structure with ribbon width $w = O(\log n)$ and $r = O(w)$ has expected construction time $O(nw)$, space overhead $O\left(\frac{\log w}{rw^2}\right)$, and query time $O\left(1 + \frac{rw}{\log n}\right)$.*

The main downside compared to Ribbon is the additional metadata overhead. BuRR also introduces additional preprocessing steps, including two sorting passes: one intra-bucket and one inter-bucket sort, both of which can be implemented in $O(n)$ time using radix sort [21].

2.2.8 Encodings

An encoding is a method for representing information in a different form, often to reduce space usage while preserving the ability to reconstruct or query the original data. In the context of this thesis, encodings are used to store integers and monotone sequences in a compact representation that remains efficient to access. This is particularly important for memory-sensitive data structures, where the number of bits required per key directly affects whether the structure fits in cache or main memory.

Universal Integer Codes

Universal integer codes are prefix-free variable-length encodings of positive integers that do not assume prior knowledge of the source distribution. They are particularly useful when encoding sequences of integers of unknown magnitude, such as upper-bit differences in Elias-Fano representations [15].

Unary Encoding

Unary encoding represents a non-negative integer k using k zero bits followed by a one:

$$\text{Unary}(k) = 0^k 1. \tag{2.5}$$

0	2	5	10
1	001	000001	00000000001

Table 2.2: Unary encoding of increasing integers.

As seen in Table 2.2, the code length grows linearly with the encoded value. While this makes unary encoding inefficient for large integers, its simplicity and structural properties make it suitable in contexts such as Elias-Fano encoding, where unary-coded gaps correspond to positions of 1-bits in a sparse bit vector.

Elias Gamma and Delta Encoding

Elias gamma and delta codes were introduced by Elias [24] as universal codes for positive integers.

Elias gamma code. The gamma code of a positive integer k is formed by writing $\lfloor \log_2 k \rfloor$ zeros followed by the $(\lfloor \log_2 k \rfloor + 1)$-bit binary representation of k:

$$\text{Gamma}(k) = 0^{\lfloor \log_2 k \rfloor} \, \text{bin}(k). \tag{2.6}$$

The total length is $2\lfloor \log_2 k \rfloor + 1$ bits.

Elias delta code. The delta code replaces the unary-coded length prefix of the gamma code with a gamma-coded length, yielding shorter codes for larger integers:

$$\text{Delta}(k) = \text{Gamma}(\lfloor \log_2 k \rfloor + 1) \,\|\, \text{bin}(k) \setminus \text{MSB}, \tag{2.7}$$

where $\text{bin}(k) \setminus \text{MSB}$ denotes the binary representation of k with the leading 1-bit removed, and $\|$ denotes concatenation. The total length is $\lfloor \log_2(\lfloor \log_2 k \rfloor + 1) \rfloor + 2\lfloor \log_2 k \rfloor + 1$ bits.

k	Gamma	Delta
1	1	1
3	011	0101
6	00110	01110
11	0001011	00100011

Table 2.3: Elias gamma and delta encodings of selected integers.

As Table 2.3 illustrates, the delta code grows as $\log_2 k + 2\log_2 \log_2 k + O(1)$, making it substantially more compact than unary for large values. When the gaps in an Elias-Fano upper-bit sequence exhibit high variance, replacing the unary code with the delta code can reduce space significantly [15].

Fibonacci Encoding

Fibonacci encoding is based on Zeckendorf's theorem, which states that every positive integer has a unique representation as a sum of non-consecutive Fibonacci numbers.

Let $F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5, \ldots$ denote the Fibonacci sequence starting from index 2. Every positive integer k can be written as

$$k = \sum_i F_{c_i}, \quad c_1 > c_2 > \cdots \geq 2, \tag{2.8}$$

where no two consecutive Fibonacci numbers appear. The Fibonacci code writes the indicator bits of this representation in order of increasing index, followed by an extra 1-bit as a terminator:

$$\text{Fib}(k) = b_2\, b_3 \cdots b_m\, 1, \tag{2.9}$$

where $b_i = 1$ if F_i is used in the representation and $b_i = 0$ otherwise, and m is the index of the largest Fibonacci number in the sum. Because Zeckendorf representations never contain two consecutive 1-bits, the terminating 11 pattern uniquely marks the end of each codeword.

k	Fibonacci code
1	11
3	0011
6	10011
11	001011

Table 2.4: Fibonacci encoding of selected integers.

The code length for k is approximately $\log_\varphi k + O(1)$ bits, where $\varphi = (1 + \sqrt{5})/2$ is the golden ratio [14]. Like the delta code, Fibonacci encoding is a viable alternative to unary for

encoding upper-bit gaps in Elias-Fano when the gap distribution has high variance [15]. An additional practical advantage is that the **11** terminator allows byte-aligned scanning without maintaining a bit-level position pointer.

Gap Encoding of Monotone Sequences

Given a non-decreasing integer sequence

$$0 \le x_1 \le \cdots \le x_n \le U, \tag{2.10}$$

gap encoding represents the sequence using differences:

$$d_1 = x_1, \quad d_i = x_i - x_{i-1},\ i \ge 2. \tag{2.11}$$

If the original sequence is monotone, all gaps are non-negative. The sequence of gaps can then be encoded using any universal integer code. Gap encoding is used in PtrHash 2.2.9. The theoretical space lower bound is

$$n \log\left(\frac{u+n}{n}\right) \tag{2.12}$$

bits [31].

Elias-Fano Representation

Elias-Fano encoding is commonly described as a *quasi-succinct* representation [31]. A data structure is succinct if it uses space equal to the information-theoretic lower bound plus lower-order terms.

Elias-Fano uses

$$n \log_2 \frac{U}{n} + O(n) \tag{2.13}$$

bits of space, which matches the leading term of the entropy bound but incurs an additive $O(n)$ [31].

Because $O(n)$ is linear rather than lower-order, the structure is not strictly succinct, but *quasi-succinct* [31].

Implementation Elias-Fano's representation splits each value into lower and upper parts.

Let

$$\ell = \left\lfloor \log_2 \frac{U}{n} \right\rfloor. \tag{2.14}$$

Each value is written as

$$x_i = y_i \cdot 2^{\ell} + z_i \tag{2.15}$$

where

$$z_i = x_i \bmod 2^{\ell} \tag{2.16}$$

are the lower ℓ bits, stored consecutively in an array $L(\ell)$, and

$$y_i = \left\lfloor \frac{x_i}{2^{\ell}} \right\rfloor \tag{2.17}$$

form a non-decreasing sequence.

The upper sequence is stored using gap encoding:

$$\Delta_i = y_i - y_{i-1}, \quad y_0 = 0, \tag{2.18}$$

which in the original formulation is encoded using unary code.

The total space usage is bounded by

$$n\ell + O(n) \tag{2.19}$$

bits, making the representation close to the information-theoretic lower bound.

CacheLineEF

CacheLineEF is a cache-aware variant of Elias-Fano style encoding introduced in PtrHash for storing the remapping structure used when a perfect hash function is first constructed over a range larger than the final key set size [27].

The problem CacheLineEF solves is how to store a monotonically increasing array. Standard Elias-Fano 2.2.8 encoding reduces the memory footprint, but may require multiple memory accesses during lookup. CacheLineEF was designed to improve this trade-off by storing the remap data in a format that fits within a single cache line, thereby favoring high query throughput.

The main idea is to partition the non-decreasing remap sequence into chunks of $C = 44$ values. Each chunk is encoded into exactly 64 bytes. As described in the PtrHash paper, the layout consists of a 4-byte offset, a 16-byte bitvector encoding the relative high parts, and 44 trailing bytes storing the low 8 bits of each value. Thus, the complete representation of one chunk fits in one 64-byte cache line.

Let the values in one chunk be $v_0, \ldots, v_{43}$. Each value is split into low and high parts, and the high part is further decomposed into a shared offset and a relative high component:

$$v_i = 2^8 \left\lfloor \frac{v_0}{2^8} \right\rfloor + 2^8 \left(\left\lfloor \frac{v_i}{2^8} \right\rfloor - \left\lfloor \frac{v_0}{2^8} \right\rfloor \right) + (v_i \bmod 2^8). \tag{2.20}$$

The first term is stored once as the chunk offset. The final term is stored explicitly as one byte per value. The middle term is encoded using a 128-bit unary-style bit vector, for each $i \in [0, 43]$, bit

$$i + \left\lfloor \frac{v_i}{2^8} \right\rfloor - \left\lfloor \frac{v_0}{2^8} \right\rfloor \tag{2.21}$$

is set to 1. Since the values are non-decreasing, these bit positions are distinct, yielding exactly 44 set bits in the 128-bit region

Lookup is then performed by reconstructing the three components. The offset and low bits are read directly, while the relative high part is recovered through a select operation on the 128-bit bit vector

$$2^8 \cdot (\text{select}(i) - i). \tag{2.22}$$

PtrHash notes that this can be implemented efficiently using bit-manipulation primitives such as popcount and `PDEP`, making the structure suitable for fast random access while still retaining most of the compression benefit of Elias-Fano style.

The main advantage of CacheLineEF is therefore not that it minimizes space as aggressively as standard Elias-Fano, but that it provides a compact representation whose access pattern is aligned with the hardware cache hierarchy. This is especially relevant in lookup-intensive data structures, where an additional memory access can dominate query time. In the PtrHash evaluation, CacheLineEF had a remap overhead that was 2.75× smaller than a plain vector and only modestly slower in lookup, while remaining substantially faster than a conventional Elias-Fano remap structure in several settings.

CacheLineEF does, however, impose practical constraints. It uses

$$\frac{64}{44} \cdot 8 \approx 11.6 \tag{2.23}$$

bits per stored value, which is somewhat larger than standard Elias-Fano for comparable gap distributions.

2.2.9 PtrHash

PtrHash is the first MPHF examined in this work.

The algorithm can be broken into two main parts (i) construction 2.2.9 and (ii) lookup 2.2.9. In the construction phase, PtrHash builds the MPHF from the input set of keys, while in the lookup phase, it uses the constructed MPHF to efficiently retrieve values associated with keys.

Construction

This section explains how PtrHash is constructed, its components, and how they work together to create an MPHF. The construction process and the steps involved are also described. To understand the construction, the concept of a hash function is required and is explained in 2.2.3.

Buckets

Buckets are the most fundamental component of PtrHash construction. Hashes are placed into different buckets by a bucket assignment function $\gamma : [0, 1) \to [0, 1)$, the function to use is a hyperparameter. PtrHash paper suggest three different functions to use, the PHOBIC paper goes into more details how the functions are decided [25]. The most important part is that the desired distribution creates large early buckets and small later buckets, as the goal is to minimize the number of evictions, see 2.2.9 for more details about evictions. The different functions that can be used are:

- $\gamma_1(x) = x$
  Linear (uniform) bucket assignment.
- $\gamma_p(x) = x + (1 - x)\ln(1 - x)$
  Optimal bucket assignment function used in PHOBIC for load factor $\alpha = 1$.
- $\gamma_{p,\varepsilon}(x) = x + (1 - \varepsilon)(1 - x)\ln(1 - x)$
  Modified PHOBIC function enforcing minimum slope ε to prevent excessively large buckets.

- $\gamma_2(x) = x^2$
  Quadratic approximation
- $\gamma_3(x) = \frac{2^8-1}{2^8} \cdot \frac{x^2+x^3}{2} + \frac{1}{2^8}x$
  Practical cubic variant used in PtrHash to enforce a minimum slope and avoid very large buckets.

The functions affect different performance characteristics of the MPHF, as shown in 4 and discussed in 5.

Parts

To optimize construction time and cache locality, see 2.2.1. The hashes inside the shard are divided into smaller parts. Each part has its own buckets. The number of buckets per part is $B = \lfloor(\alpha \cdot S)/\lambda\rfloor$, where λ is a hyperparameter representing the expected size of each bucket, usually between 3 and 4. The parts are mirror images of each other, the size of bucket 0 in part 0 equals the size of bucket 0 in part 1, and so on. The number of parts is $P = \lceil n/(\alpha S)\rceil$, see 2.2.9 for more details about α.

Pilots

Pilots are what make the PtrHash a perfect hash function; they are 8-bit values stored for each bucket and used as a seed for the hash function to ensure each key maps to a unique value. Initially, a hash collision might occur between keys within the same bucket and those outside the bucket. When a lookup is performed for a key, it is first hashed once to determine the shard, part, and bucket. When a bucket is determined, an index inside the part is determined by combining the hash and the pilot. This means that a different pilot yields a completely new assignment for all keys in the bucket for the slots in its part.

The goal of the construction is to assign pilots to all buckets so there exist no hash collisions. This is done first by a brute-force approach, in which all pilot values in the range $[0, 255]$ are tried for a bucket. If no value can be found that avoids all hash collisions inside the part, the algorithm progresses to eviction 2.2.9.

The algorithm maintains a priority queue of buckets that need to find a pilot, sorted by the tuple (bucketLen, bucketIdx).

Eviction

Because pilots are limited to values in range $[0, 255]$ a situation where no pilot can be found that avoids collision might occur. The algorithm implements eviction logic in which an eviction score is calculated for all pilots. We denote all buckets where some hash collision exists with some key in this bucket with the assigned pilot p, B_p, and the length of bucket i as L_i. If any bucket in B_p has been evicted among the last 16 times, this pilot is skipped to avoid evicting in a loop.

Then the eviction score for one pilot is $S(p) = \sum_{i \in B_p} L_i^2$. The pilot p that is chosen is $p^* = \arg\min_p S(p)$.

For this pilot p, all buckets in B_p are pushed on the priority queue, and the pilot is chosen. This allows the algorithm to always progress.

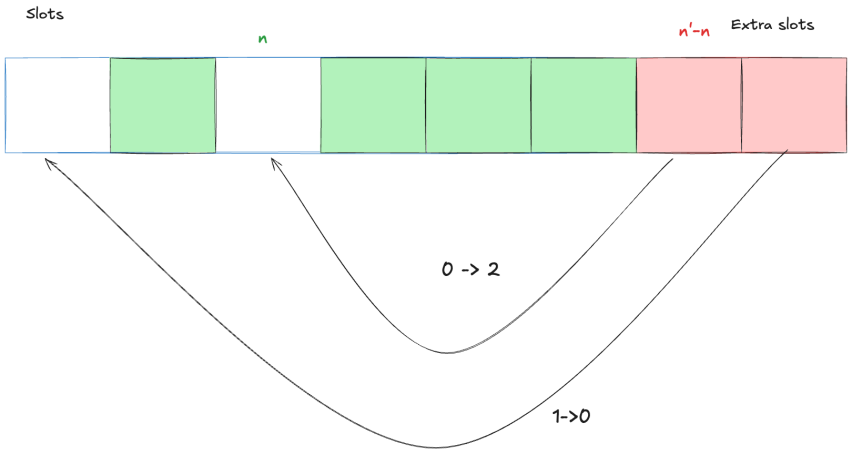

Figure 2.7: Remapping process

Load-factor

To speed up construction `PtrHash` introduces a hyper parameter load factor that is denoted α in range $[0, 1]$. The load factor is a trick to make the rest of the algorithm think we have more slots than there actually are. The algorithm assumes there are $n' = n/\alpha$ slots and maps keys to the range $[0, s]$. When α decreases, the probability of hash collisions goes down because we have the same number of keys mapping to a larger output range. Because the probability of hash collisions is lower, the construction speed is faster, as we have fewer evictions and find pilots more quickly.

The load factor introduces a new problem, `PtrHash` is not minimal. This is addressed by a remap process. The slots greater than n are remapped to the free slots in the range $[0, n]$. This remapping is either stored in a plain array or in a CacheLineEF encoded structure 2.2.8.

CacheLineEF requires that stored values fit within 40 bits and that consecutive values within each 44-element chunk do not span more than the 128-bit bitvector can encode. With typical PtrHash parameters, this corresponds to an average stride — the gap between consecutive values in the monotonic remap sequence — of roughly 500. When the load factor α approaches 1, the few remaining free slots become widely and unevenly spaced, pushing the stride beyond what the encoding supports. PtrHash therefore uses CacheLineEF only for $\alpha \leq 0.99$; at higher load factors, a plain array suffices, since remapping is rarely needed and the simpler structure adds negligible overhead.

Our Java implementation uses a plain width-packed array (`byte[]`, `short[]`, or `int[]`, chosen by the maximum value) for the remap structure rather than CacheLineEF. While we implemented and benchmarked CacheLineEF with proper 64-byte-aligned off-heap allocation, it provided no measurable query improvement at the scales tested (up to 10^9 keys). This aligns with the PtrHash paper, that states that for fast configurations they used plain arrays instead of CacheLineEF [27]. This was not investigated any further.

Lookup

Lookup is the second part of PtrHash, and is much simpler than construction. In this phase, the MPHF is queried for a given key, and the corresponding value is returned.

In the lookup process, the goal is to compute the index in the slot array for a given key. The high-level process is as follows:

1. Hash the key
2. Determine its part
3. Determine its bucket
4. Load that bucket's 8-bit pilot
5. Compute the slot
6. Remappning

The part is determined by

$$\text{part}(k) = \text{hi}(P \cdot h(k))$$

where P is the number of parts, h is the hash function, and hi is the high 64 bits of 128-bit multiplication. To determine a bucket let:

$$x = \text{lo}(P \cdot h), \tag{2.24}$$

where $\text{lo}(\cdot)$ extracts the lower 64 bits. The bucket index inside the part is

$$\text{bucket}(k) = \text{hi}\Big(B \cdot (2^{64} \cdot \gamma(x/2^{64}))\Big), \tag{2.25}$$

where B is the number of buckets per part and γ is defined in 2.2.9.

The global bucket index is

$$b = \text{part}(k) \cdot B + \text{bucket}(k). \tag{2.26}$$

Then the pilot p = pilots[globalBucket] is loaded, and the slot is computed as $\text{slot}(k) = \text{part}(k) \cdot S +$ offset.. If slot $\geq n$, it is remapped using *CacheLineEF*, see 2.2.9. After this, the value is loaded from the slot array, completing the lookup.

2.3 Implementation consideration

2.3.1 Duplicate keys

Neo4j specifies that duplicate nodes in `Nodes.csv` are not allowed. Duplicate nodes are defined as two distinct rows in `Nodes.csv` that share the same external key. This restriction exists because such duplicates would introduce ambiguity when referencing nodes while processing `Edges.csv`. In that case, a node is referenced by its external key, but multiple candidate nodes may exist.

According to the specification, this situation should be handled by skipping one of the duplicate nodes and issuing a warning to the user.

In the existing Neo4j architecture, all keys are first hashed, then sorted by their hash values. If two identical hash values are encountered, an additional comparison is performed to determine whether the corresponding keys are identical. While the details of this implementation are outside the scope of this thesis, it is important to note that the input to the system proposed in this work is a file in which each row represents a key.

Each row in this file contains the following information: the `hash` of the key, the original row identifier from `Nodes.csv`, and a cursor pointing to the corresponding row in `Nodes.csv`, enabling retrieval of the original key. The rows in this file are sorted by their hash values. Both the hashing function and the sorting algorithm are configurable.

We refer to this file as *IdMapperKeyProvider*.

2.3.2 Problem understanding

In this section, we describe the problem that Neo4j aims to solve and how this thesis fits into the broader import pipeline and the work of the Kernel team. The general import process consists of the following steps:

- Reading all nodes and inserting them into the X1 store.
- Reading all edges and inserting both endpoints into the X1 store.

To perform these operations efficiently, paging must be taken into account. Due to memory constraints, only a limited portion of the X1 store can be loaded into memory at any given time. For example, when modifying node 1, a page containing nodes 0–8 may be loaded into memory. After updating node 1, the entire page is written back to disk, resulting in two costly I/O operations (one read and one write). In practice, pages are significantly larger and depend on the available RAM.

If multiple nodes within the same page (e.g., nodes 0–8) are modified before writing back to disk, only a single read and write operation is required for all updates. Consequently, it is beneficial to group updates that target the same memory region. This observation motivates batching operations based on data locality.

In this thesis, groups of nodes that are updated together are referred to as *wells*. This introduces the first requirement of our implementation: insertions and updates to the X1 store must be batched and executed within one well at a time. However, this is nontrivial because `Nodes.csv` and `Edges.csv` are not consistently ordered, and each edge may reference nodes from different wells.

To address this, the import process does not directly insert nodes during the first pass over `Nodes.csv`. Instead, it constructs an *IdMapperKeyProvider*, builds a lookup table (LUT), and partitions the data into wells based on key ranges, as illustrated in Figure 2.8.

Subsequently, `Edges.csv` is processed, and each row is assigned to the appropriate well using the LUT. After this step, each well contains all nodes and edges that must be inserted together, enabling batched updates within a single page. This significantly reduces the number of I/O operations.

The primary objective of this thesis is therefore to design an efficient LUT that satisfies the following requirements:

- Translate an external identifier to a well index.

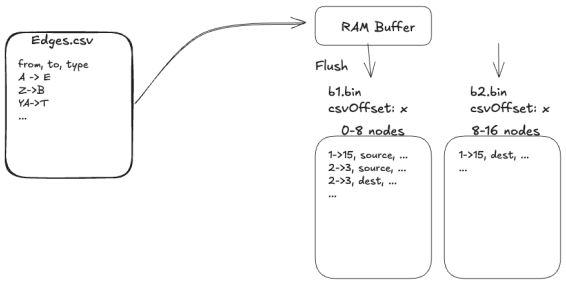

Figure 2.8: Illustration of wells and their mapping.

- Given a well index, bijectively map an external identifier to an internal identifier within the corresponding well range.

2.3.3 BitPackedArray

To store a series of numbers with a known number of values, indexed by an integer, the usual solution is an array. This works in most cases, but has the downside that we cannot adjust the size of each value. We are limited to using 1, 2, 4, 8 bytes, each with an increasing range of numbers being supported. If the desired range of values is known, we can use a BitPackedArray to represent values using any number of bytes, allowing us to save space compared to a normal array. This is used for the value array for `PtrHash`.

Chapter 3
Method

3.1 Literature Review

The starting point for the literature review was the problem itself: mapping external identifiers to internal long integers during bulk graph import. This framing naturally directed the search toward hash tables and lookup structures as the primary category of interest. Sources used throughout the review include ACM Digital Library, Google Scholar, and arXiv. The search strategy was exploratory rather than systematic; after identifying an initial set of relevant papers, further work was conducted by following references cited within those papers, commonly referred to as snowball sampling [32].

From the problem description it was clear that that the key set in the import problem is known in its entirety before queries begin. This is because all nodes must be read before edges can be resolved. The static nature of the key set makes the problem a candidate for perfect hashing, where a collision-free mapping can be constructed once and queried many times.

The first structure studied in depth was BBHash [16], a fingerprinting-based Minimal Perfect Hash Function (MPHF). BBHash is conceptually simple and provides a concrete understanding of what an MPHF achieves: a bijective mapping from n keys to $\{0, \ldots, n-1\}$ using only a few bits per key, with $O(1)$ lookup time. This initial study confirmed that MPHFs are a promising direction for the import use case, where memory consumption is a limiting factor and lookup speed directly affects import throughput.

From BBHash, the search broadened upon discovering the survey by Lehmann et al. [28], which provides a comprehensive overview of modern MPHF constructions. The survey identifies three families of MPHF constructions: retrieval, bucket-based, and fingerprinting. This survey served as the primary roadmap for identifying candidate structures.

Based on the survey and subsequent reading, the candidate set was narrowed by the theoretical memory and speed performance. This is discussed in 3.2.

3.2 Theoretical Analysis

This section presents the candidate data structures identified during the literature review and analyzes their theoretical properties. Two architectural approaches for the externalID-to-internalID mapping are then described: a single lookup table and a hybrid approach that partitions keys into wells. Finally, the combinations are compared, and the selected structures for implementation are justified.

3.2.1 Architecture

Two architectures are considered for implementing the externalID-to-internalID mapping during import.

Single Lookup Table

The simplest architecture uses a single structure that maps each external key directly to its internal identifier. This has two alternatives, one big BuRR, see 2.2.4, implementation, or one big PtrHash, see 2.2.9, table with a value array.

With an MPHF such as PtrHash, the MPHF maps each key to a position in $\{0, \ldots, n-1\}$, and a separate bit-packed array (see Section 2.3.3) stores the internal identifier at each position. The space usage is approximately $(c + \log_2 n) \cdot n$ bits. For 10^{14} keys, which is the largest number of keys we expect to handle, this is approximately $(3 + 47) \cdot 10^{14} = 5 \cdot 10^{15}$ bits, which corresponds to about 625 TB. This is far too large to fit entirely in main memory, so we need to shard the values.

The BuRR option is much more straightforward, as it would only entail using the data structure for its intended purpose. However, it would take up nr bits, plus a minor multiplicative overhead of

$$O\left(\frac{\log w}{rw^2}\right), \tag{3.1}$$

where $r = \lceil \log_2 n \rceil$ as each node needs a unique identifier. This results in $O(n \log n)$ space and $O(\log n)$ query time. There might be a cache locality benefit to using BuRR over an MPHF with a bit-packed array, as querying BuRR would only entail querying 1 data structure that is optimized for locality, the inferior space and time complexity leave us with little hope for this approach.

Sharding For a key set of this size, it is not feasible to keep all required data in memory. As established in Section 3.2.1, our largest possible input would require approximately 625 TB of memory. Modern hardware cannot be expected to provide this capacity, and parts of the data structure must therefore be stored on disk.

Our implementation of sharding is agnostic to the data structure. It performs the sharding by first determining an arbitrary number of shards, let this number be s. We then use $\lceil \log_2 s \rceil$ bits from the hash value to assign each key to a shard. These bits are referred to as *shard bits*.

Because the hashes are sorted during construction, all keys belonging to the same shard can be processed consecutively. For each shard, the PtrHash table and lookup array are con-

structed in memory, then serialized and written to disk to free memory before processing the next shard.

During lookup, the shard bits identify the relevant shard. If that shard is already loaded in memory, the lookup proceeds directly. Otherwise, the required shard is loaded from disk and the currently loaded shard or shard group is evicted.

The drawback of this approach is that it requires $O(s)$ disk I/O operations during construction and, in the worst case, one disk I/O operation per query. This cost can be reduced by sorting hashes and grouping queries that target the same shard. Still, as discussed in Section 2.2.1, disk I/O is approximately 10^3 to $5 \cdot 10^5$ times slower than RAM access. Even so, sharding is necessary to make data structures of this scale practical.

Hybrid Approach

The hybrid approach splits the mapping into two stages. The first stage is a *Well Function* (WellFN) that maps each external key to a *well index*. The second stage is a collection of sharded MPHFs, one per well, each mapping the keys within its well to internal identifiers.

The number of wells W is a configurable parameter that depends on the available memory. The implementation details of the well processing pipeline are described in Section 3.3.2.

The well index requires $r = \lceil \log_2 W \rceil$ bits. Since W is typically a factor of 10^7 smaller than the key count n, the value of r is small. For example, with $n = 10^9$ and $W = 100$, only $r = 7$ bits are needed per key for the WellFN.

The key advantage of the hybrid approach is that the sharded MPHFs do not require a separate value array. Within each well, the MPHF maps the well's keys bijectively to $\{0, \ldots, w-1\}$, where w is the size of the well. The internal identifier is then computed as the well's starting offset plus the MPHF output. The mapping is not chosen; the MPHF construction determines it. This eliminates the 64-bit value array that dominates space usage in the single-lookup-table approach. This, of course, comes at the cost of constructing an extra MPHF per well and querying it. The details of this tradeoff are discussed in Section 3.2.2.

The WellFN itself can be implemented as either a retrieval structure or an MPHF with a value array. With BuRR, it stores the r-bit well index directly at approximately $r \cdot n$ bits. With PtrHash, it requires the MPHF plus an r-bit value array at approximately $(2.56 + r) \cdot n$ bits.

3.2.2 Comparison

A lookup table over n keys must map each key to a unique identifier in $\{0, \ldots, n-1\}$. Modern MPHFs occupy $cn + n \log_2 n$ bits, where c is a small constant—approximately 2.4 bits per key for PtrHash [27]—and $\log_2 n$ bits store the internal identifier. A retrieval data structure such as BuRR accomplishes the same goal in $m \log_2 n$ bits where $m = (1 + \varepsilon)n$ with less than $1\,\%$ overhead [21]. For the hybrid approach we reintroduce the constant W (number of wells). When $W \ll n$, a hybrid lookup table can provide significant space savings that reduce the number of I/O operations needed to complete the import, potentially making the extra construction and query time worthwhile. Assuming a retrieval data structure for the well function, a hybrid lookup table occupies $m \log_2 W + Wc(n/W) = n \log_2 W + cn$ bits, while an MPHF-based hybrid occupies $cn + n \log_2 W + Wc(n/W) = 2cn + n \log_2 W$ bits [28].

On a standard von Neumann architecture with a conventional memory hierarchy, a relationship file is traversed multiple times so that the largest possible number of shards reside in memory during each traversal. This prevents I/O thrashing and minimizes disk operations. Having smaller shards at this stage allows fewer traversals, thus improving import time. This saving must be large enough to compensate for the construction and querying of W additional MPHFs. The hybrid approach therefore has a higher chance of outperforming the direct approach with fewer, larger wells and more data to process.

3.2.3 Decision

Based on the theoretical analysis, four configurations were selected for implementation and benchmarking:

1. **Single LUT with PtrHash.** PtrHash as MPHF with a log(n)-bit value array.
2. **Single LUT with BuRR.** A straightforward retrieval datastructure.
3. **Hybrid with PtrHash for both stages.** PtrHash as MPHF with a value array for the WellFN, and sharded PtrHash instances for the per-well MPHFs.
4. **Hybrid with BuRR for WellFN and PtrHash for sharded MPHFs.**

3.3 Implementation

This section discusses key implementation details for our data structures. It focuses on high-level architecture decisions such as sharding, I/O interactions, multi-threading, and pipelining, and how these decisions affect performance.

3.3.1 Sharding

As discussed in Section 3.2.1, the in-memory representation of a single lookup table over the full key set can exceed available memory. Our implementation therefore partitions the key space into 2^b shards, where b is the number of *shard bits* extracted from the most significant bits of the Master Hash Code (MHC). Because the key provider supplies entries sorted by MHC, all keys belonging to the same shard appear consecutively in the input stream, which allows each shard to be constructed and serialized without buffering the entire data set. This sharding scheme is shared by both PtrHash and BuRR; the two differ only in what is built per shard and what artifacts are written to disk.

The number of shard bits is a user-selected hyperparameter.

File-Based Shard Construction

Neither PtrHash nor BuRR loads the entire data set into memory. Instead, both implementations stream the sorted fixture file shard by shard. The fixture file stores $(MHC, rowId)$ pairs as 16-byte records, sorted by unsigned MHC. Because the most significant bits of the

MHC determine shard assignment, all entries for a given shard occupy a contiguous byte range in the file.

Construction proceeds as follows. A main reader thread binary-searches the fixture file to locate each shard's byte boundaries, then reads the shard's entries into a contiguous in-memory block using a 64 MB I/O buffer. Once loaded, the entries are handed to a builder thread. At most T shards reside in memory simultaneously, where T is the inter-shard thread count. After a shard has been built and its artifacts written to disk, its in-memory representation is released immediately.

For **PtrHash**, the loaded MHC values are first *remixed* by multiplying each by a mixing constant (`0x517cc1b727220a95`). Without this step, MHC values within a single shard share the same high-order bit prefix and occupy only a narrow band of the 64-bit output space, leading to excessive collisions during pilot search. After remixing, the values are radix-sorted and fed to the PtrHash builder, which constructs the pilot table and remap structure. Each shard produces two on-disk artifacts: a *table file* containing the pilots and remap structure, and a *values file* containing a bit-packed array of result values.

For **BuRR**, no explicit remix step is applied. Instead, the first ribbon layer shifts each MHC left by the number of shard bits, discarding the shared prefix; this was verified empirically to be sufficient. Subsequent layers rehash each MHC with a multiplicative factor, so the shared prefix is not an issue. The loaded entries are sorted and inserted into a ribbon banding structure. The builder constructs multiple ribbon layers via back-substitution, with bumped keys cascading to the next layer. A small fallback hash map captures any keys that remain after all layers have been processed. Each shard produces a single *snapshot file* containing the serialized ribbon layers, bumping metadata, and fallback entries.

Query-Time Shard Loading

At query time, shards are managed through explicit *group-based loading*. The query runner partitions the 2^b shards into contiguous groups of size g, where

$$g = \min\left(2^b,\ \max\left(1,\ \left\lfloor \frac{\max(0,\ f \cdot \text{maxHeap} - B_{\text{well}})}{1.1 \cdot B_{\text{shard}}} \right\rfloor\right)\right), \tag{3.2}$$

where f is a configurable cache fraction (defaulting to 0.8), $B_{\text{well}} = T \cdot W \cdot 16384 \cdot 8$ is the memory reserved for per-well output buffers (T worker threads, W wells, 16 K entries of 8 bytes each), and B_{shard} is the actual on-disk shard size multiplied by 1.1 to reserve 10% extra space and avoid OOM.

Before each pass, the runner invokes `loadShardGroup`$(s_{\text{start}}, s_{\text{end}})$, which deserializes all artifacts for shards in $[s_{\text{start}}, s_{\text{end}})$ and releases any previously loaded shards outside that range. Both PtrHash and BuRR implement this interface identically: PtrHash loads its table and values files, while BuRR loads its snapshot files. A pipelined architecture then processes the unsorted input file: a single reader thread streams MHC chunks into a bounded queue, and N worker threads dequeue chunks, filter entries to the active shard group, perform the lookup, and write results to shared per-well files. If all shards fit in a single group (i.e. $g \geq 2^b$), the filter step is skipped and only a single pass is required. When multiple passes are needed, each pass re-reads the entire input file but processes only the entries belonging to the current group; the per-well files are accumulated across passes.

Multicore Construction

The sharded construction uses two levels of parallelism: *inter-shard* and *intra-shard*.

At the inter-shard level, a fixed-size thread pool with $\min(\text{numThreads}, 2^b)$ workers is managed by an `ExecutorCompletionService`. The main thread reads the sorted fixture file sequentially, accumulating entries for the current shard. When a shard boundary is reached, the collected entries are submitted as a build task to the executor while the main thread immediately begins collecting the next shard. This provides implicit pipelining between I/O (reading the fixture file) and computation (building shards). A back pressure mechanism limits the number of in-flight shard tasks to the pool size plus a small relaxation constant, preventing the main thread from reading too far ahead and exhausting memory with buffered entries.

At the intra-shard level, each shard's build is further parallelized. PtrHash partitions each shard into multiple *parts* for cache locality and builds them concurrently using one virtual thread per part. BuRR similarly uses virtual threads to parallelize layer-segment construction within each shard when the segment count exceeds a minimum threshold.

Each worker thread independently performs the following steps for its assigned shard:

1. **Preprocessing.** MHC values are remixed (PtrHash) or left as-is (BuRR), then sorted. PtrHash uses unsigned LSD radix sort in 8 byte-level passes; BuRR uses its own radix sorter that co-sorts MHC, start-position, and value arrays together.

2. **Structure construction.** PtrHash builds the pilot table with eviction fallback and remap structure (see Section 2.2.9), and packs result values into a bit-packed array. BuRR constructs ribbon layers via back-substitution across multiple bumping iterations, with values encoded directly into the ribbon structure.

3. **Serialization.** Artifacts are written to disk. For PtrHash, the table file is written first; for BuRR, a single snapshot file is produced. Once written, in-memory build data is released.

Construction progress is reported after each shard completes.

3.3.2 Hybrid Well Processing

In the hybrid architecture, the query phase proceeds in two stages. In the first stage, the unsorted input file is streamed and each MHC is queried against the WellFN to obtain a well index. For the direct architecture, the MHC is resolved to a rowId and written to the corresponding per-well file. For the hybrid architecture, the MHC itself is written to the per-well file, deferring the final resolution.

In the second stage, each well is processed independently by a thread from a fixed pool. For the direct architecture, the well file is read sequentially and a checksum over the rowIds is computed. For the hybrid architecture, the processor seeks into the unsorted nodes file at the well's positional offset, reads exactly the well's unique MHCs, radix-sorts them, and builds a per-well PtrHash MPHF in memory. It then reads the full well file and queries each entry through the MPHF to obtain the final internal identifier.

3.4 Benchmarks

This work simulates only a subset of the full import workflow in Neo4j. This is partly because the new architecture is not yet fully implemented, and partly because the goal is to isolate and evaluate only those components that are within scope.

The benchmarks are designed as a single comprehensive run per architecture, where different execution regions are measured individually. The following architectures are evaluated:

- SinglePtrBenchmark
- SingleBuRRBenchmark
- HybridPtrPtrBenchmark
- HybridBuRRPtrBenchmark
- BinarySearchLookup - the currently deployed architecture at Neo4j

For each implementation, the following phases are measured:

- Construction
- Partitioning
- Lookup

Each benchmark execution consists of three cold-start measured runs, with no prior warm-up, from which the mean, median, and standard deviation are computed. The results are written to CSV files and presented in Section 4.

All benchmarks were executed on a single machine whose configuration is given in Table 3.1. To enable a third party to reproduce the experiments, the table lists the model, CPU, memory, operating system, JVM, and JVM heap setting.

Table 3.1: Hardware and software configuration of the benchmark host.

Component	Value
Model	Lenovo ThinkPad P1 Gen 6
CPU	Intel Core i9 vPro
Logical cores	24
RAM	64 GB
OS	Ubuntu Linux 24.04
JVM	OpenJDK 64-Bit Server VM 25.0.1
Max heap	40 GiB

BinarySearchLookup

The binary search lookup is a simplification of the current deployed import architecture. It uses a simple binary search in which all keys are loaded from disk into memory, then performs a classic binary search [8]. This implementation will perform better than Neo4j's actual implementation because of the extra overhead from disk sharding and other details. It still serves as a good baseline for comparing our implementation.

Construction

The construction phase encompasses two distinct stages: *fixture generation*, which produces the synthetic input data, and *data-structure building*, which constructs the index under test from that data. Fixture generation is performed once as a preprocessing step and is not measured; the build step is repeated for each measured run.

Fixture generation. The generator uses a seeded `SplittableRandom` to draw N random 64-bit keys, where N is the configured dataset size [6]. Each key is then converted into a 64-bit master hash code (MHC) with the shared mixer described in 2.2.3, evaluating Equation 2.1 with a fixed seed (`0xdeadbeefcafe`. The same hashing routine is used regardless of which data structure consumes the fixture: BuRR and PtrHash both ingest $(MHC, rowId)$ pairs and treat the MHC as the canonical key, so a single fixture can drive every benchmark configuration without re-hashing.

The resulting $(MHC, rowId)$ pairs are accumulated into fixed-size chunks and sorted within each chunk by unsigned MHC using an 8-pass LSD radix sort. Each sorted chunk is written to disk as a binary file. Once all chunks have been spilled, the runs are merged via a multi-round k-way merge. The result is a single binary file containing all entries globally sorted by unsigned MHC. Alongside it, an unsorted file containing only the MHC values in insertion order is produced; this file is used later in the query phase to simulate arriving keys.

Both files use a fixed-width binary format, N records of 16 bytes each (8 bytes for the MHC and 8 bytes for the rowID) in the sorted file, or N records of 8 bytes each (MHC only) in the unsorted file. This external-memory merge-sort strategy allows datasets that exceed available RAM to be prepared reliably. In the production pipeline, 24 bytes will be stored for each row, with an extra 8 bytes used as a pointer to the original key in the CSV file to handle hash collisions. This is omitted from the thesis to save disk space. Because of this, correctness is not guaranteed; two keys with the same hash will be treated as equal rather than rehashed. This was deemed okay to make larger benchmarks possible and to be a trivial change when running in production.

Data-structure building. At the start of each benchmark run, the sorted fixture is opened and streamed directly from the binary file through a 64 MB I/O buffer.

The provider is then passed to the appropriate builder depending on the benchmark configuration:

3.4.1 WellPartitioner

The lookup phase consists of two main steps. First, the data is partitioned using a process called `WellPartitioner`. During this step, the *IdMapperKeyProvider* is iterated over, and the data structure is queried to obtain a well index for each key. This well index determines the partition to which the key belongs.

The input data generated in Section 3.4 is iterated over using a two-level loop. The outer loop iterates over shard groups: for each group, the corresponding shards are loaded into memory. The inner loop re-reads the unsorted input file *passCount* times with that group still resident, where *passCount* is a configurable parameter that simulates the size of the relationship file. In production, the relationship file is typically 5-20 times larger than the nodes file, and must be processed against each shard group in turn. The total number of file reads is therefore shardGroups × passCount. For each entry, the `PerfectMapper` is queried, and the resulting mapping is passed to the `WellPartitioner`.

The purpose of the `WellPartitioner` is to efficiently write keys to partition files, where each file contains all keys corresponding to a specific well. This setup mimics the intermediate representation (IR) of relationship data used in the full system. Two different values can be received and written depending on the architecture being benchmarked. For Hybrid data structures, the well index and hash are received; the well index is used to identify the file, and the hash is written. For the direct perfect mapper, the well index and node ID are passed; because the node ID is received directly, the extra lookup during file reading can be avoided. The well index is used to identify the file, and the node ID is written.

Lookup

Once the partitioned data has been created, a thread pool is initialized with as many threads as was used by the construction phase. Each thread is assigned a partition and begins processing queries. To overlap I/O with computation, a pipelined readahead architecture is employed: a dedicated reader thread streams entries from the partition file in 1 MB chunks (128K entries each) and enqueues them into a bounded queue with a capacity of four chunks. Worker threads consume chunks from this queue and perform lookups against the cached shard index. The bounded queue provides natural back pressure; if the reader fills the queue, it blocks until a worker drains a slot, and conversely, workers block when the queue is empty. This ensures that the disk is kept busy reading ahead while the CPU cores are saturated with queries, without requiring the entire partition to be loaded into memory.

During execution, I/O thrashing can occur when one thread progresses faster than others and begins evicting shards that slower threads still need [13]. For example, consider a scenario with 100 shards, of which only 10 can be kept in memory. If the data is partitioned into wells such that well 2 is dominated by shards numbered 20 and above, the thread assigned to well 2 will reach shard 20 while the remaining threads are still working on shards 10–20. The well 2 thread will begin evicting shards 10–20; the threads using those shards will later reload them, evicting shards that yet another thread needs, and this thrashing cycle will continue.

Two different approaches were identified to address this problem; due to time constraints, only the *MultiPass* approach was implemented and benchmarked.

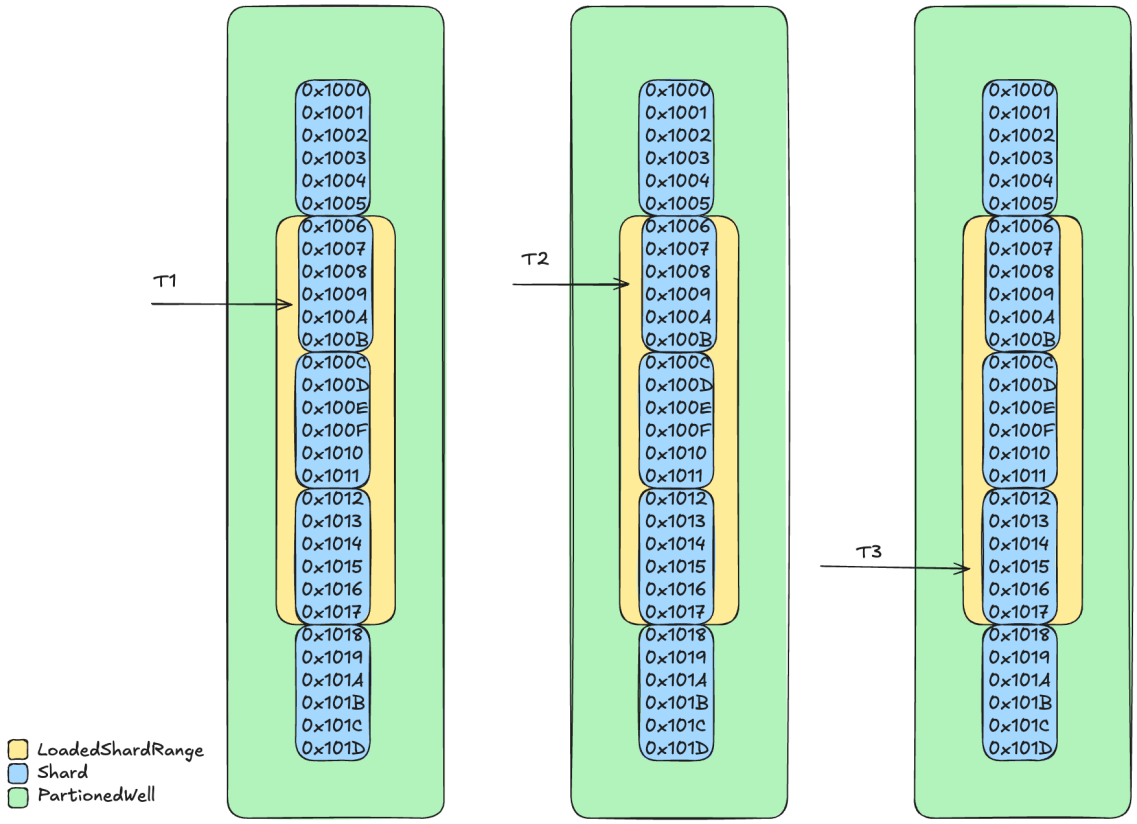

Figure 3.1: Sharded partition processing

MultiPass mitigates the issue by iterating over the IR representation multiple times. Each pass covers a designated range of shards. For each entry read, the shard bits are checked against the range permitted for the current pass; if the entry falls within range, a lookup for the well index is performed and the result is written to file.

ShardWindow mitigates the issue by first sorting the IR and then performing a single pass. During this pass, a bounded range of shards that threads are allowed to access is enforced. This mechanism acts as a sliding window that advances according to the progress of the slowest thread.

If a thread attempts to load a shard outside the allowed range, it is blocked using a semaphore until the window advances. The size of this sliding window is dynamically determined by the available free memory, see Figure 3.1 for a visual representation of this with three threads denoted T1, T2 and T3 all reading different parts.

Chapter 4
Results

This chapter presents the results obtained from the benchmarks described in Section 3.4. All experiments use a fixed PtrHash preset — the `fast` configuration for PtrHash-based architectures, and the `balanced` preset for BuRR-only architectures (which do not use PtrHash). The primary experimental variable is *passCount*, which controls how many times the unsorted input file is re-read per shard group during the query phase, simulating different relationship-file-to-node-file size ratios.

Two dataset scales were evaluated. The first uses 10^9 keys and is tested under two heap configurations: a constrained 3 GB heap with shard bits 8 only, and an unconstrained 40 GB heap with shard bits 6, 8, and 10. The second uses 10^{10} keys and is tested at a 20 GB heap with passCount = 5, as well as at a 40 GB heap with passCount = 1.

4.1 1 Billion Keys

This section presents benchmark results for all architectures using a dataset of 10^9 keys. Four architectures are evaluated: Direct_PTR, Direct_BuRR, Hybrid_BuRR_PTR, and Hybrid_PTR_PTR.)All PtrHash-based architectures use the `fast` preset; Direct_BuRR, which does not use PtrHash, uses the `balanced` preset. Two heap scenarios and two passCount values are tested. The passCount parameter controls how many times the unsorted input file is re-read per shard group during the query phase, simulating different relationship-file-to-node-file size ratios.

In each table, the fastest configuration within each architecture group (by average time) is shown in **bold**, and the overall fastest configuration across all groups is **bold and underlined**.

4.1.1 3 GB Heap, Shard Bits 8, passCount = 3

This subsection presents results under a constrained 3 GB JVM heap with shard bits fixed at 8 and passCount = 3. Higher shard-bit values (e.g. sb = 10) caused out-of-memory failures

at this heap size and are therefore excluded. Each configuration was measured over 3 runs.

Total Time

The total time is the sum of the construction, query and well partitioning, and well processing phases. As Table 4.1 shows, Direct_PTR is fastest end-to-end at 97.5 s on average, while Direct_BuRR is slowest at 149.0 s — a **1.5×** gap. Hybrid_PTR_PTR follows closely behind Direct_PTR at 100.3 s, suggesting the additional hybrid lookup imposes only a small overhead at this scale.

Table 4.1: Total time (all phases) for 1B, 3 GB heap, passCount = 3 (3 measured runs).

Architecture	shardGroups	Min (ms)	Max (ms)	Avg (ms)	Median (ms)	Std (ms)
Direct_PTR	**4**	**95 816**	**99 767**	**97 486**	**96 874**	**1 485**
Direct_BuRR	3	146 272	151 309	148 996	149 407	1 834
Hybrid_BuRR_PTR	1	120 564	146 386	132 191	129 623	8 848
Hybrid_PTR_PTR	2	93 155	106 390	100 287	101 317	4 290

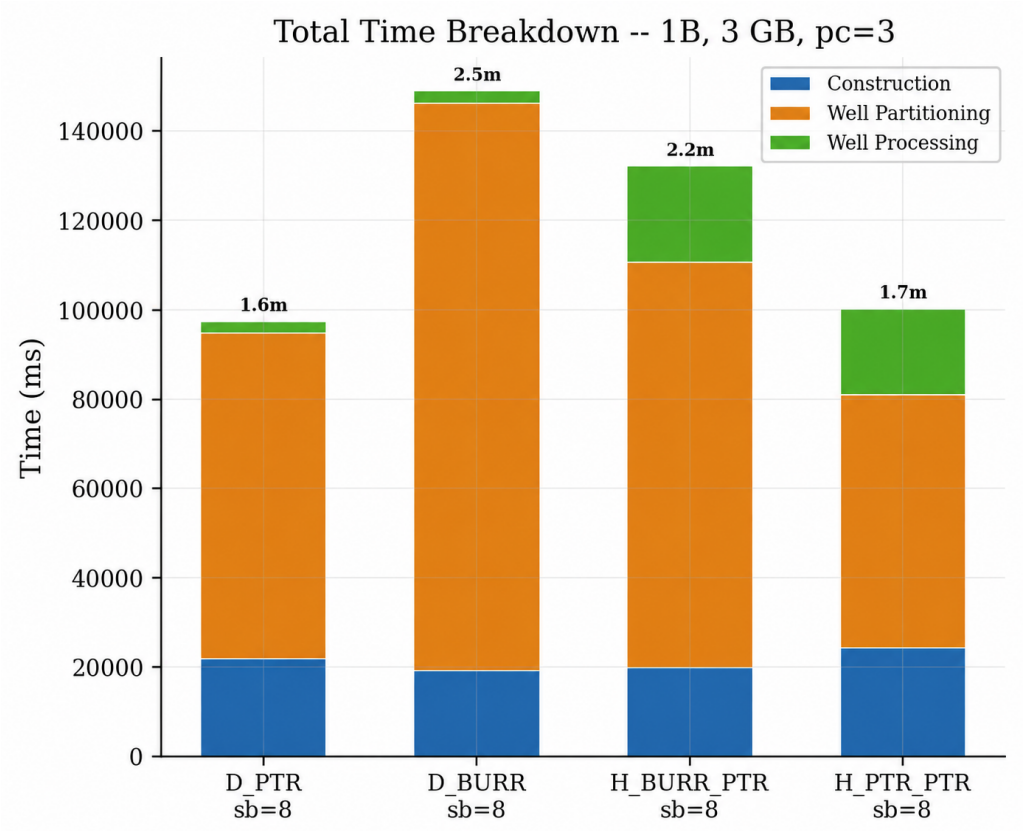

Figure 4.1: 1B, 3 GB heap: total time per configuration (focused).

Construction

From Table 4.2, Direct_BuRR has the fastest construction at 19.4 s (19.44 ns/key) and Hybrid_PTR_PTR is the slowest at 24.7 s (24.70 ns/key); the four architectures fall within a narrow 19–25 s window, indicating that the choice of MPHF backend has limited impact on construction at this dataset size. See Section 3.4 for a description of the construction phase.

Table 4.2: Construction time for 1B, 3 GB heap, passCount = 3 (3 measured runs).

Architecture	shardGroups	Min (ms)	Max (ms)	Avg (ms)	Median (ms)	Std (ms)	Avg time / key (ns)
Direct_PTR	4	21 406	22 505	21 880	21 729	565	21.88
Direct_BuRR	**3**	**18 820**	**19 982**	**19 444**	**19 531**	**586**	**19.44**
Hybrid_BuRR_PTR	1	16 394	21 897	19 964	21 602	3 096	19.96
Hybrid_PTR_PTR	2	21 983	27 595	24 704	24 534	2 810	24.70

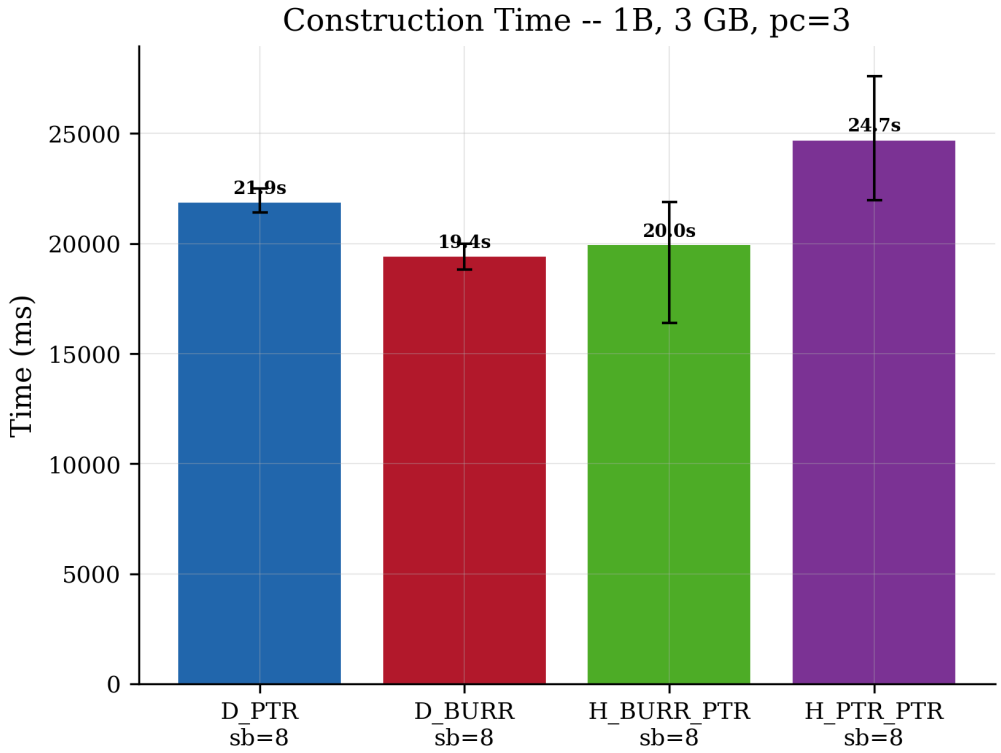

Figure 4.2: 1B, 3 GB heap: construction time per configuration (focused).

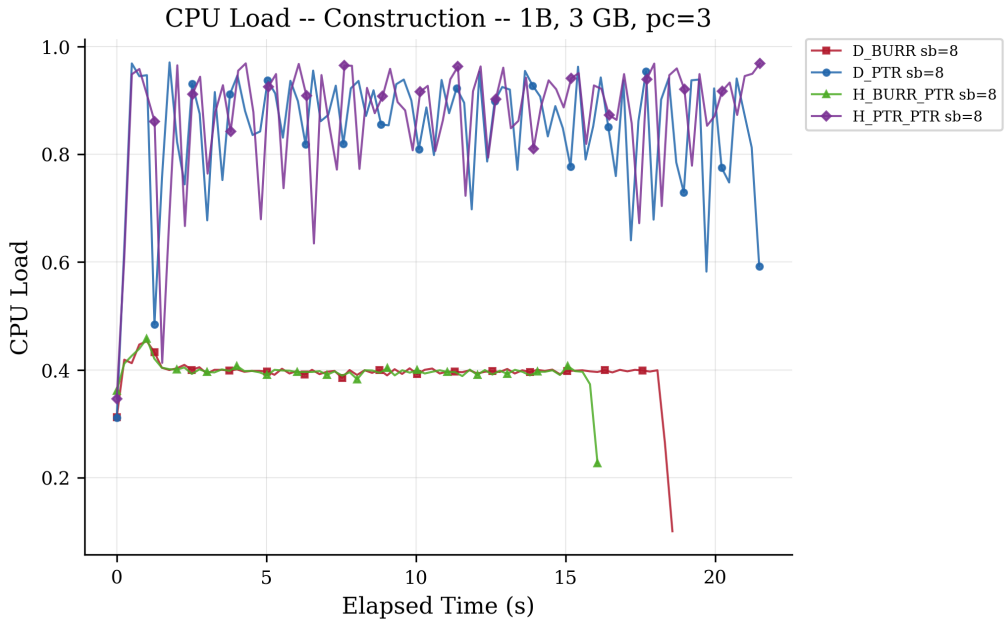

Figure 4.3: 1B, 3 GB heap: CPU load during construction (focused).

Figure 4.3 reveals a clear split in how the two MPHF backends use the machine: the PtrHash-based architectures (Direct_PTR and Hybrid_PTR_PTR) sustain a CPU load of 0.8–0.95 for the duration of construction, while the BuRR-based ones (Direct_BuRR and Hybrid_BuRR_PTR) sit flat at ≈ 0.4. Despite this ∼ 2× difference in CPU utilisation, BuRR still finishes first in wall-clock time, indicating that its construction is not CPU-bound on this machine and that the PtrHash backend's heavier parallelism does not translate into a proportional speedup at this scale.

Query and Well Partitioning

In Table 4.3, Hybrid_PTR_PTR is the fastest configuration at 56.5 s (18.83 ns/key), more than twice as fast as the slowest, Direct_BuRR, at 126.8 s (42.26 ns/key). The two direct architectures land on opposite ends of the table, while the two hybrids sit between them, indicating that the lookup choice (PtrHash vs. BuRR) dominates over the direct/hybrid distinction in this phase. See Section 3.4.1 for a description of this phase.

Table 4.3: Query and well partitioning time for 1B, 3 GB heap, passCount = 3 (3 measured runs).

Architecture	shardGroups	Min (ms)	Max (ms)	Avg (ms)	Median (ms)	Std (ms)	Avg time / key (ns)
Direct_PTR	4	71 834	74 459	72 923	72 477	1 368	24.31
Direct_BuRR	3	124 961	128 384	126 768	126 958	1 719	42.26
Hybrid_BuRR_PTR	1	85 148	99 544	90 796	87 695	7 683	30.27
Hybrid_PTR_PTR	**2**	**52 948**	**58 887**	**56 489**	**57 632**	**3 130**	**18.83**

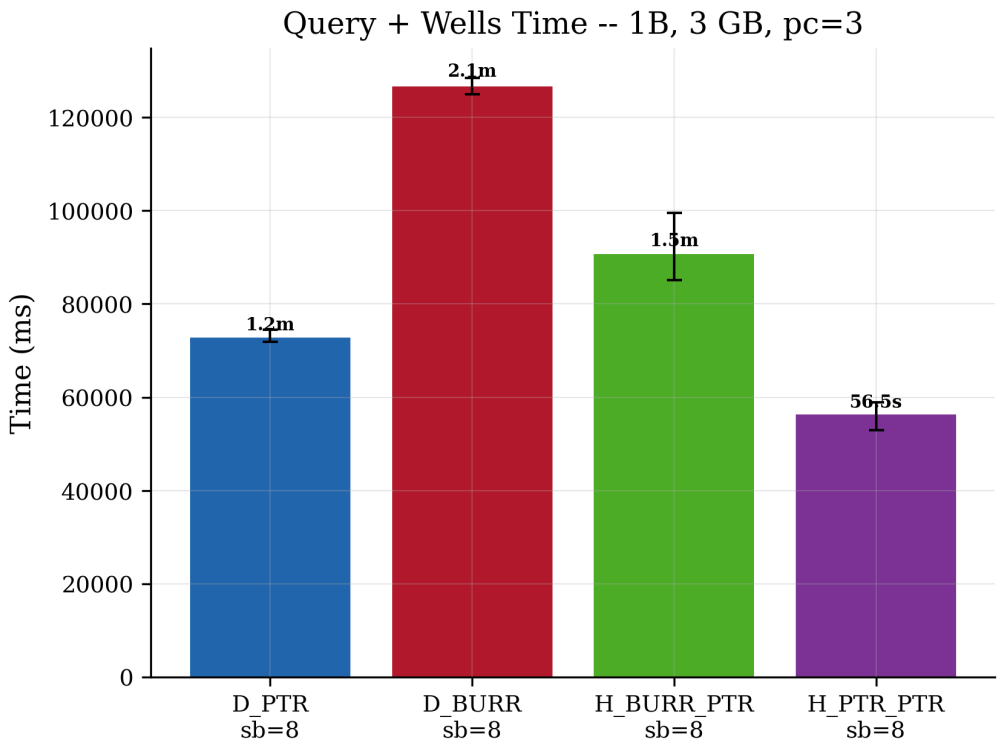

Figure 4.4: 1B, 3 GB heap: query + wells time per configuration (focused).

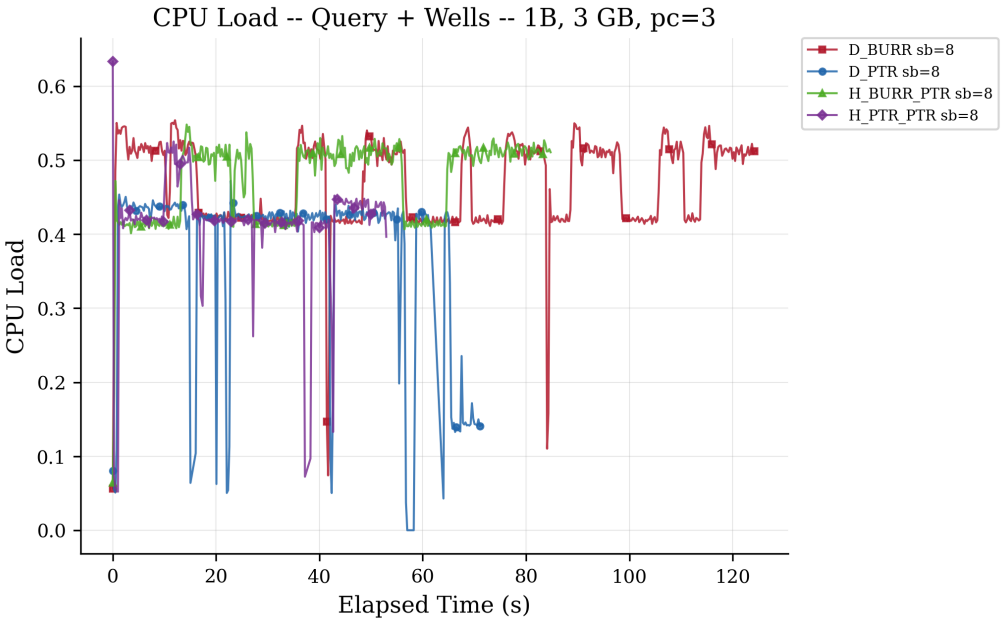

Figure 4.5: 1B, 3 GB heap: CPU load during query + wells (focused).

In Figure 4.5, the BuRR-backed configurations (Direct_BuRR and Hybrid_BuRR_PTR) sustain around 0.5 CPU load across the full phase, while the PtrHash-backed pair frequently dips toward 0.1 between bursts of activity. None of the four reaches more than ~ 0.55, indicating that the query phase is bounded by sequential reads of the input file rather than by lookup cost.

Well Processing

Table 4.4 reveals a sharp split between the direct and hybrid architectures: Direct_PTR (2.68 s) and Direct_BuRR (2.78 s) finish well processing roughly 7–8× faster than the hybrid backends (19.1 s and 21.4 s). The direct backends only stream the per-well file from disk and compute a checksum over the already-resolved row IDs, while the hybrid path also builds a per-well PtrHash MPHF and queries it for every entry to recover the final row ID — so almost the entire hybrid wall-clock budget here is the cost of that extra lookup pass.

Table 4.4: Well processing time for 1B, 3 GB heap, passCount = 3 (3 measured runs).

Architecture	shardGroups	Min (ms)	Max (ms)	Avg (ms)	Median (ms)	Std (ms)	Avg time / key (ns)
Direct_PTR	**4**	**2 576**	**2 803**	**2 682**	**2 668**	**114**	**0.89**
Direct_BuRR	3	2 491	2 943	2 784	2 918	254	0.93
Hybrid_BuRR_PTR	1	19 022	24 945	21 431	20 326	3 112	7.14
Hybrid_PTR_PTR	2	18 224	19 908	19 094	19 151	843	6.36

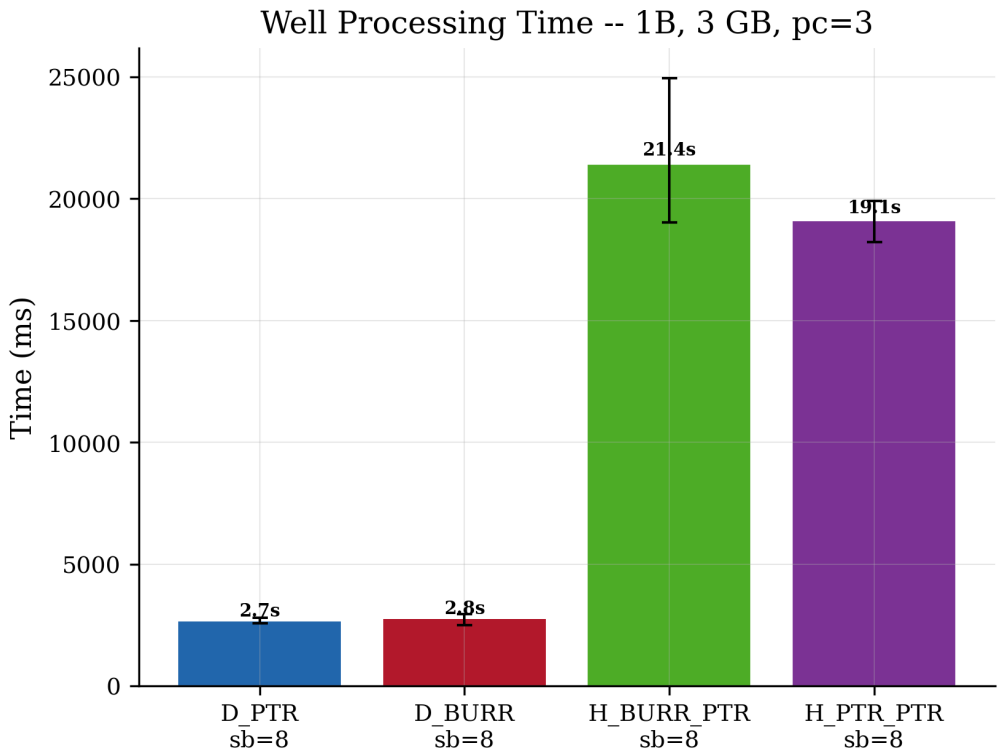

Figure 4.6: 1B, 3 GB heap: well processing time per configuration (focused).

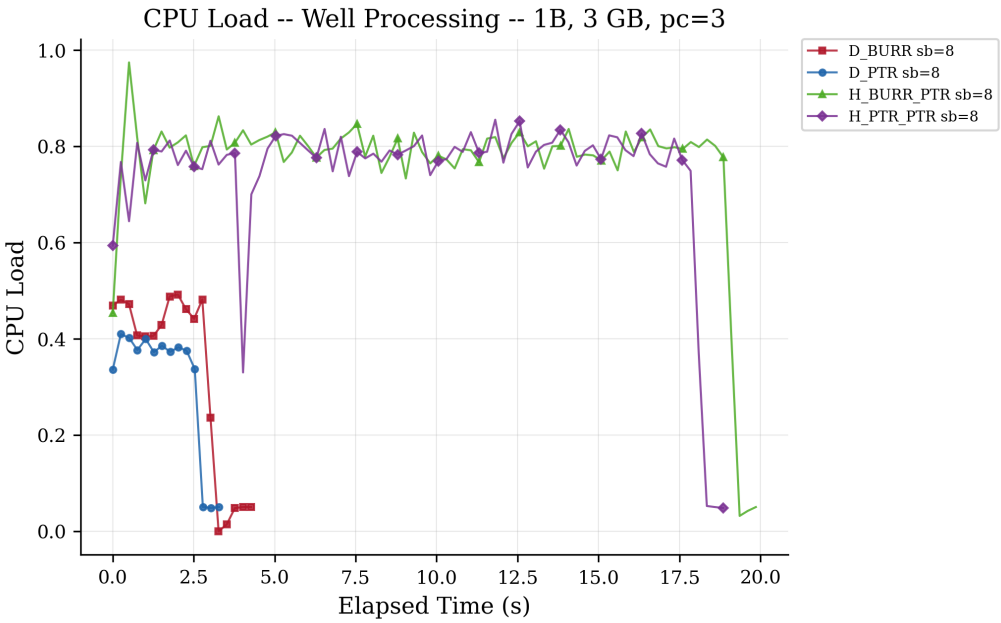

Figure 4.7: 1B, 3 GB heap: CPU load during well processing (focused).

Figure 4.7 shows the same direct/hybrid split from a different angle: the two hybrid architectures sustain a CPU load of 0.7–0.85 for the full ~ 19 s well-processing window, while the direct architectures finish in 3–4 s at a much lower 0.3–0.45. The hybrids' higher utilisation reflects the per-well MPHF construction and querying that keeps the cores busy until each well is processed.

4.1.2 3 GB Heap, Shard Bits 8, passCount = 20

This subsection uses the same 3 GB heap with passCount = 20, simulating a relationship file 20 times the size of the node file. Each configuration was measured over 3 runs.

Total Time

With the input file re-read 20 times instead of 3, end-to-end time grows by roughly 6× across the board. As Table 4.5 shows, Hybrid_PTR_PTR is now the fastest at 598 s, while Direct_BuRR is the slowest at 994 s. Direct_PTR drops from first place at passCount = 3 to roughly tied with Hybrid_BuRR_PTR (741 s and 743 s), indicating that the hybrid lookup pays off as the query phase grows.

Table 4.5: Total time (all phases) for 1B, 3 GB heap, passCount = 20 (3 measured runs; 1 run for the binary search baseline).

Architecture	shardGroups	Min (ms)	Max (ms)	Avg (ms)	Median (ms)	Std (ms)
Binary search (baseline)	—	3 015 349	3 015 349	3 015 349	3 015 349	0
Direct_PTR	4	579 953	880 612	741 377	763 567	146 410
Direct_BuRR	3	960 322	1 036 969	994 220	985 368	38 150
Hybrid_BuRR_PTR	1	692 325	788 752	743 332	748 920	36 382
Hybrid_PTR_PTR	**2**	**582 452**	**615 439**	**598 012**	**596 144**	**11 621**

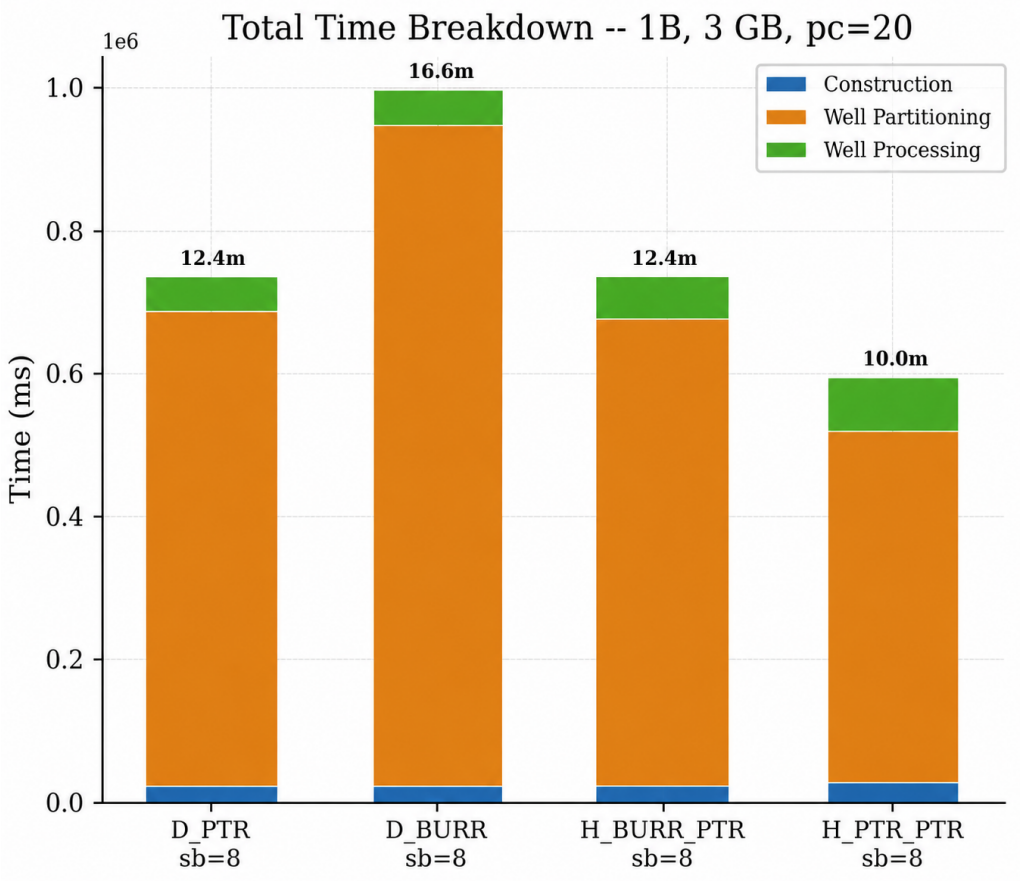

Figure 4.8: 1B, 3 GB heap: total time per configuration (focused).

Construction

Construction is independent of passCount, so Table 4.6 closely mirrors the passCount = 3 result: Direct_BuRR is the fastest at 18.9 s and Hybrid_PTR_PTR is the slowest at 24.0 s. The roughly half-second variation between the two passCount runs reflects normal run-to-run noise rather than any algorithmic difference.

Table 4.6: Construction time for 1B, 3 GB heap, passCount = 20 (3 measured runs; 1 run for the binary search baseline).

Architecture	shardGroups	Min (ms)	Max (ms)	Avg (ms)	Median (ms)	Std (ms)	Avg time / key (ns)
Binary search (baseline)	—	8 514	8 514	8 514	8 514	0	8.51
Direct_PTR	4	20 478	21 807	21 210	21 346	675	21.21
Direct_BuRR	**3**	**18 566**	**19 598**	**18 935**	**18 642**	**575**	**18.94**
Hybrid_BuRR_PTR	1	17 680	23 167	19 618	18 006	3 078	19.62
Hybrid_PTR_PTR	2	22 186	25 391	24 023	24 493	1 653	24.02

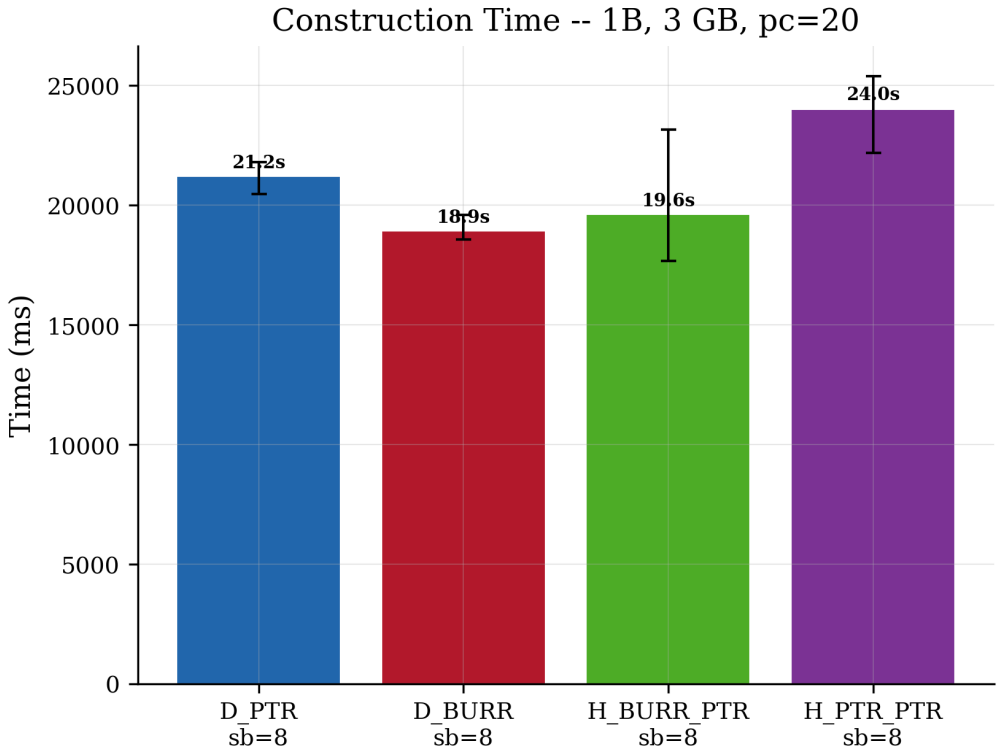

Figure 4.9: 1B, 3 GB heap: construction time per configuration (focused).

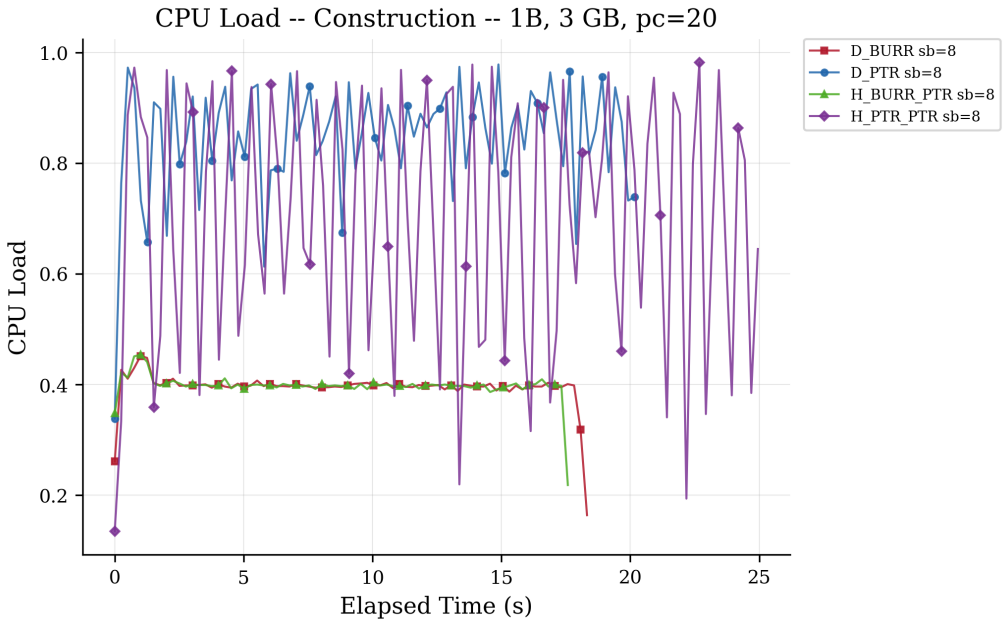

Figure 4.10: 1B, 3 GB heap: CPU load during construction (focused).

Figure 4.10 reproduces the same backend split seen at passCount = 3: PtrHash backends sustain 0.8–0.95 CPU load while BuRR backends sit flat at ≈ 0.4. This confirms that the per-backend CPU profile is independent of passCount, as expected since construction does not touch the input file again.

Query and Well Partitioning

Table 4.7 shows that Hybrid_PTR_PTR is the fastest at 500 s (25.02 ns/key), about 1.85× faster than the slowest, Direct_BuRR at 927 s (46.37 ns/key). The query phase now dominates total time, so the per-key gap between the fastest and slowest architectures translates directly into the end-to-end ranking shown above.

Table 4.7: Query and well partitioning time for 1B, 3 GB heap, passCount = 20 (3 measured runs; 1 run for the binary search baseline).

Architecture	shardGroups	Min (ms)	Max (ms)	Avg (ms)	Median (ms)	Std (ms)	Avg time / key (ns)
Binary search (baseline)	—	2 953 572	2 953 572	2 953 572	2 953 572	0	2953.57
Direct_PTR	4	514 088	803 377	671 384	696 686	146 295	33.57
Direct_BuRR	3	894 218	969 035	927 324	918 719	38 144	46.37
Hybrid_BuRR_PTR	1	611 347	679 660	647 386	651 150	34 312	32.37
Hybrid_PTR_PTR	**2**	**492 790**	**511 788**	**500 353**	**496 480**	**10 074**	**25.02**

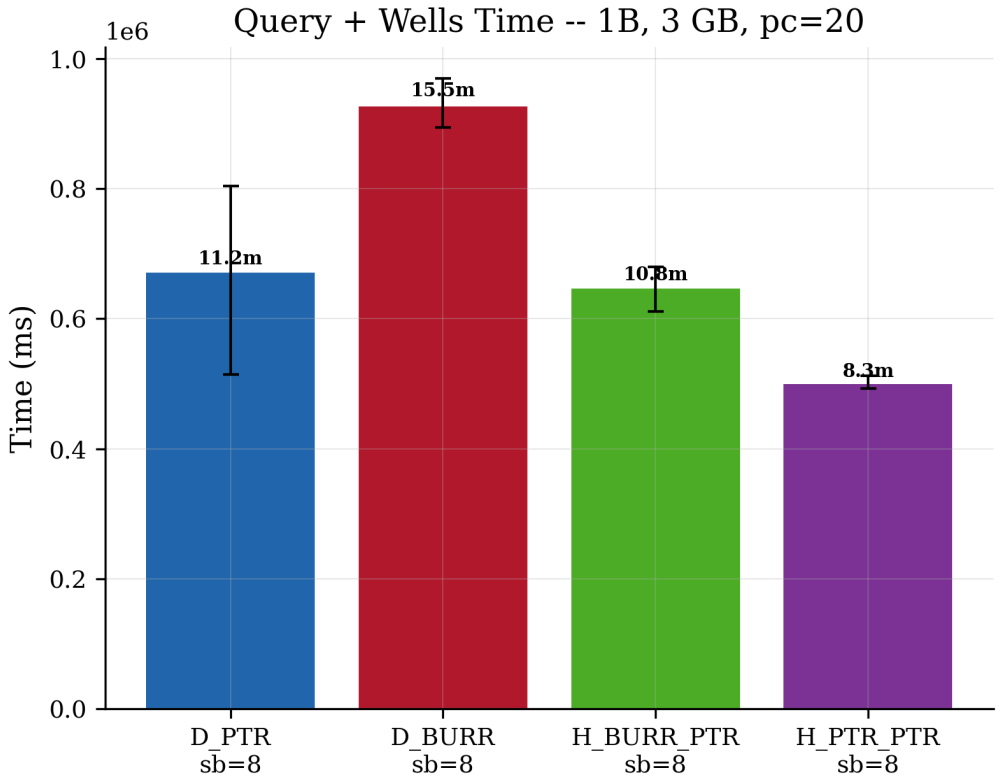

Figure 4.11: 1B, 3 GB heap: query + wells time per configuration (focused).

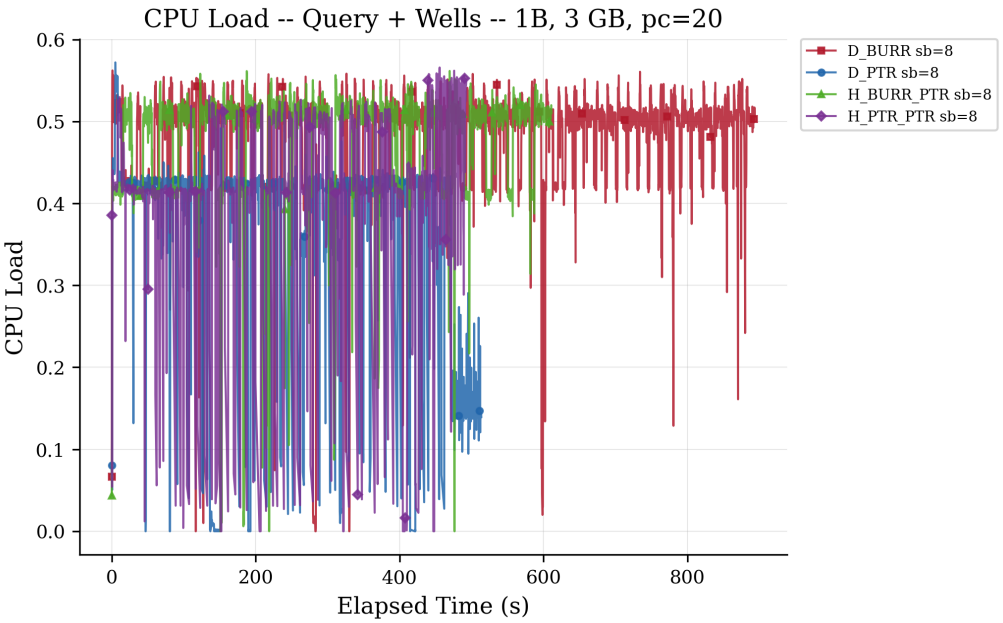

Figure 4.12: 1B, 3 GB heap: CPU load during query + wells (focused).

In Figure 4.12, all four configurations exhibit deep, high-frequency oscillations between near-zero and ~ 0.55 CPU load throughout the multi-hundred-second phase. The pattern is essentially identical across backends, reinforcing that this phase is bounded by sequential disk throughput as the input file is re-read 20 times per shard group; D_BuRR runs longest (~ 880 s) and Hybrid_PTR_PTR finishes earliest (~ 580 s).

Well Processing

In Table 4.8, Direct_BuRR is the fastest at 48.0 s (2.40 ns/key) and Hybrid_BuRR_PTR is the slowest at 76.3 s. As at passCount = 3, the direct architectures finish well processing faster than the hybrids, but the gap narrows from roughly 8× to 1.6× as the per-well work grows relative to the fixed per-well MPHF construction the hybrid path performs.

Table 4.8: Well processing time for 1B, 3 GB heap, passCount = 20 (3 measured runs; 1 run for the binary search baseline).

Architecture	shardGroups	Min (ms)	Max (ms)	Avg (ms)	Median (ms)	Std (ms)	Avg time / key (ns)
Binary search (baseline)	—	53 263	53 263	53 263	53 263	0	53.26
Direct_PTR	4	45 387	55 428	48 783	45 535	5 755	2.44
Direct_BuRR	**3**	**47 538**	**48 336**	**47 960**	**48 007**	**401**	**2.40**
Hybrid_BuRR_PTR	1	63 298	85 925	76 329	79 764	11 698	3.82
Hybrid_PTR_PTR	2	67 476	78 260	73 636	75 171	5 554	3.68

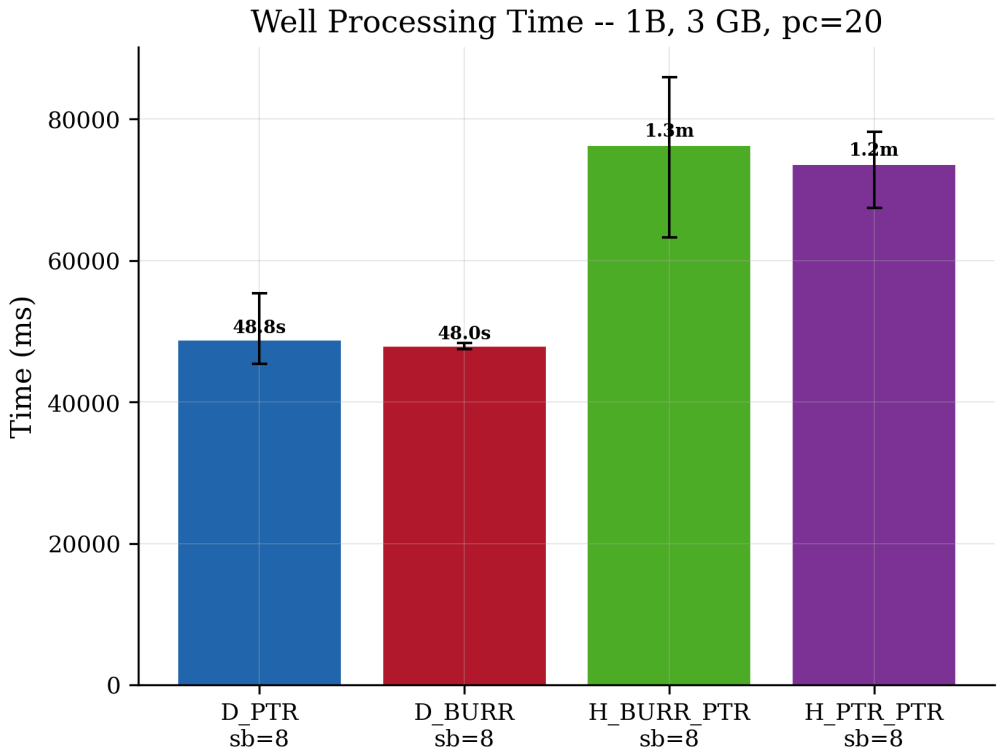

Figure 4.13: 1B, 3 GB heap: well processing time per configuration (focused).

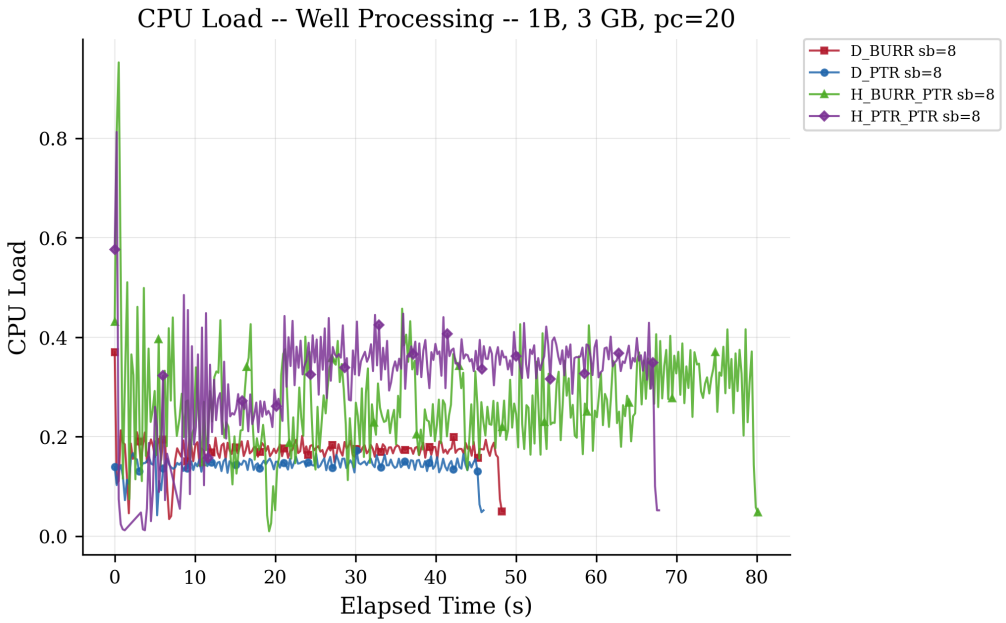

Figure 4.14: 1B, 3 GB heap: CPU load during well processing (focused).

Figure 4.14 shows that the hybrids settle into a 0.3–0.4 CPU band over their longer ~ 70–80 s well window, while the direct architectures sit lower at 0.15–0.2 and finish in ~ 40–50 s. All four are well below the saturation seen during construction, confirming that well processing remains throughput-bound at this passCount.

4.1.3 40 GB Heap, passCount = 3

This subsection presents results under an unconstrained 40 GB JVM heap with passCount = 3. All shard-bit values (6, 8, and 10) are evaluated. Each configuration was measured over 2 runs.

Total Time

Table 4.9 shows that Direct_PTR with shard bits = 8 is the fastest configuration overall at 78.6 s, while Direct_BuRR (sb = 6) is the slowest of the MPHF backends at 137 s. The Binary_Search baseline takes 480 s — roughly 6× slower than the fastest MPHF configuration, illustrating the headroom available over the currently deployed lookup. Within each architecture, sb = 8 is the best choice for the PtrHash-based ones, while Direct_BuRR favours the smallest sb = 6.

Table 4.9: Total time (all phases) for 1B, 40 GB heap, passCount = 3 (2 measured runs).

Architecture	SB	shardGroups	Min (ms)	Max (ms)	Avg (ms)	Std (ms)
Binary_Search	—	—	470 467	487 466	479 816	7 857
Direct_PTR	6	1	82 244	89 005	85 624	2 943
Direct_PTR	**8**	**1**	**77 469**	**79 818**	**78 644**	**1 076**
Direct_PTR	10	1	82 377	101 926	92 152	11 738
Direct_BuRR	**6**	**1**	**133 965**	**140 168**	**137 066**	**3 932**
Direct_BuRR	8	1	147 825	152 276	150 050	2 258
Direct_BuRR	10	1	149 951	155 494	152 722	2 471
Hybrid_BuRR_PTR	6	1	109 399	122 926	116 162	6 009
Hybrid_BuRR_PTR	**8**	**1**	**112 257**	**112 613**	**112 435**	**161**
Hybrid_BuRR_PTR	10	1	117 903	121 479	119 691	1 850
Hybrid_PTR_PTR	6	1	103 059	106 575	104 817	1 699
Hybrid_PTR_PTR	**8**	**1**	**87 857**	**94 754**	**91 306**	**3 584**
Hybrid_PTR_PTR	10	1	86 613	104 905	95 759	9 049

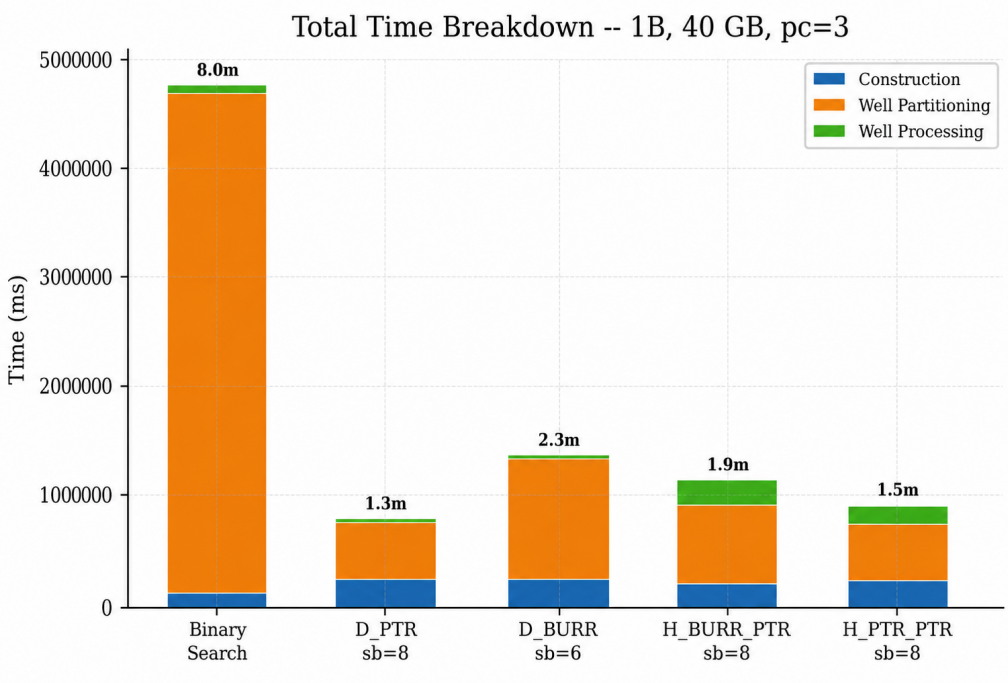

Figure 4.15: 1B, 40 GB heap: total time per configuration (focused).

Construction

In Table 4.10, Binary_Search "constructs" in 12.2 s — the fastest overall, but this number reflects only the cost of loading the sorted keys and values into memory; no MPHF or other index structure is built. Among the MPHFs, Hybrid_PTR_PTR (sb = 10) at 19.0 s and Hybrid_BuRR_PTR (sb = 10) at 19.0 s are essentially tied for fastest, while Direct_BuRR (sb = 10) at 21.9 s is the slowest. Construction time consistently improves with larger shard bits, since a larger `sb` reduces per-shard size and lets the parallel build use more workers effectively.

Table 4.10: Construction time for 1B, 40 GB heap, passCount = 3 (2 measured runs).

Architecture	SB	shardGroups	Min (ms)	Max (ms)	Avg (ms)	Std (ms)	Avg time / key (ns)
Binary_Search	—	—	**11 574**	**12 644**	**12 186**	**551**	**12.19**
Direct_PTR	6	1	26 805	30 051	28 428	2 295	28.43
Direct_PTR	8	1	23 138	24 484	23 811	952	23.81
Direct_PTR	**10**	**1**	**19 926**	**22 453**	**21 190**	**1 787**	**21.19**
Direct_BuRR	6	1	23 054	23 280	23 167	160	23.17
Direct_BuRR	8	1	22 027	23 403	22 715	973	22.71
Direct_BuRR	**10**	**1**	**20 674**	**23 033**	**21 854**	**1 668**	**21.85**
Hybrid_BuRR_PTR	6	1	20 846	25 270	23 058	3 128	23.06
Hybrid_BuRR_PTR	8	1	19 635	19 674	19 654	28	19.65
Hybrid_BuRR_PTR	**10**	**1**	**18 951**	**19 005**	**18 978**	**38**	**18.98**
Hybrid_PTR_PTR	6	1	26 983	28 857	27 920	1 325	27.92
Hybrid_PTR_PTR	8	1	22 418	22 448	22 433	21	22.43
Hybrid_PTR_PTR	**10**	**1**	**18 789**	**19 141**	**18 965**	**249**	**18.96**

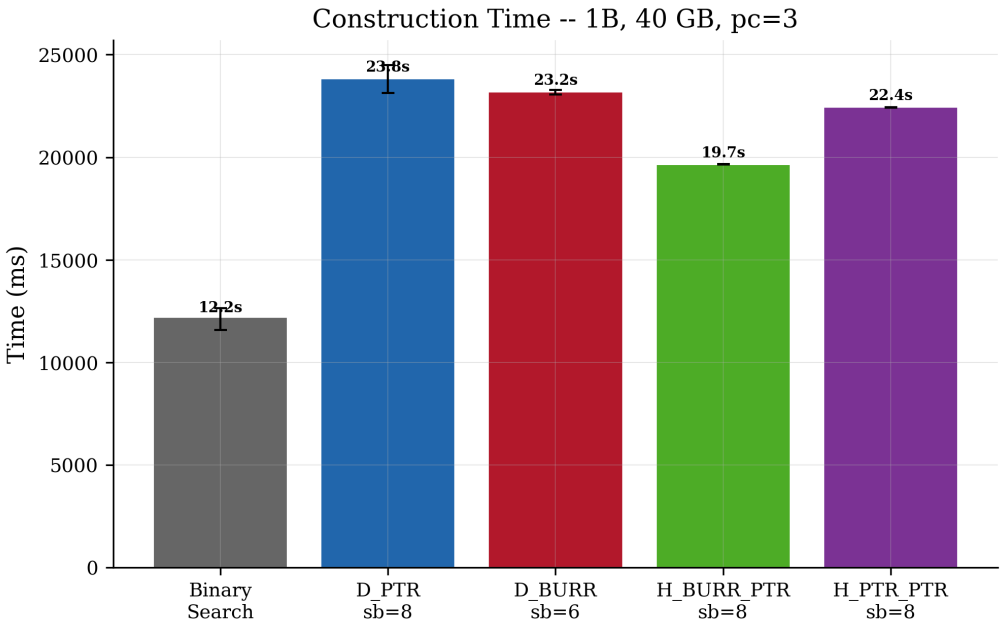

Figure 4.16: 1B, 40 GB heap: construction time per configuration (focused).

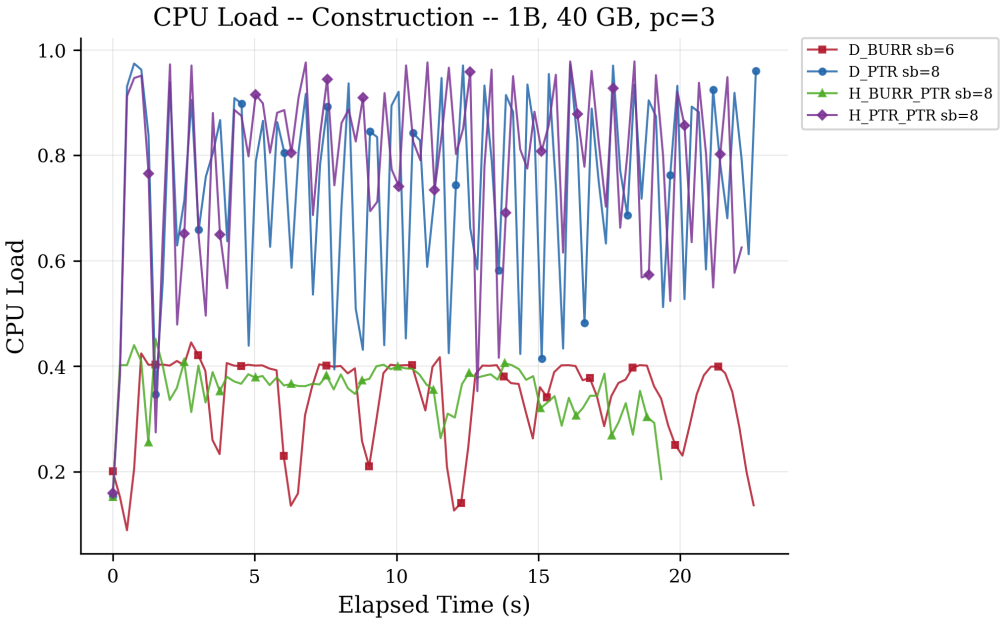

Figure 4.17: 1B, 40 GB heap: CPU load during construction (focused).

Figure 4.17 reproduces the same backend split observed at the 3 GB heap: PtrHash backends sustain ∼ 0.7–0.95 CPU load while the BuRR backends sit at ∼ 0.2–0.4. The larger heap does not change the per-backend utilisation, only the absolute timings.

Query and Well Partitioning

Table 4.11 shows Hybrid_PTR_PTR (sb = 8) as the fastest at 49.0 s (16.33 ns/key) — about 2.3× faster than the slowest MPHF configuration, Direct_BuRR (sb = 10) at 127.8 s. Binary_Search takes 460.8 s for the same lookups, around 9× slower than the fastest MPHF, confirming that the query/partitioning phase is where the new architectures yield the largest gain.

Table 4.11: Query and well partitioning time for 1B, 40 GB heap, passCount = 3 (2 measured runs).

Architecture	SB	shardGroups	Min (ms)	Max (ms)	Avg (ms)	Std (ms)	Avg time / key (ns)
Binary_Search	—	—	452 245	467 663	460 755	7 833	153.59
Direct_PTR	6	1	52 905	54 110	53 508	852	17.84
Direct_PTR	**8**	**1**	**51 948**	**52 457**	**52 202**	**360**	**17.40**
Direct_PTR	10	1	59 630	76 025	67 828	11 593	22.61
Direct_BuRR	**6**	**1**	**108 148**	**113 687**	**110 918**	**3 917**	**36.97**
Direct_BuRR	8	1	122 692	125 566	124 129	2 032	41.38
Direct_BuRR	10	1	126 600	129 081	127 840	1 754	42.61
Hybrid_BuRR_PTR	6	1	69 291	76 211	72 751	4 893	24.25
Hybrid_BuRR_PTR	**8**	**1**	**72 485**	**72 647**	**72 566**	**115**	**24.19**
Hybrid_BuRR_PTR	10	1	80 041	82 368	81 204	1 645	27.07
Hybrid_PTR_PTR	6	1	57 677	57 822	57 750	103	19.25
Hybrid_PTR_PTR	**8**	**1**	**46 769**	**51 229**	**48 999**	**3 154**	**16.33**
Hybrid_PTR_PTR	10	1	49 271	59 406	54 338	7 167	18.11

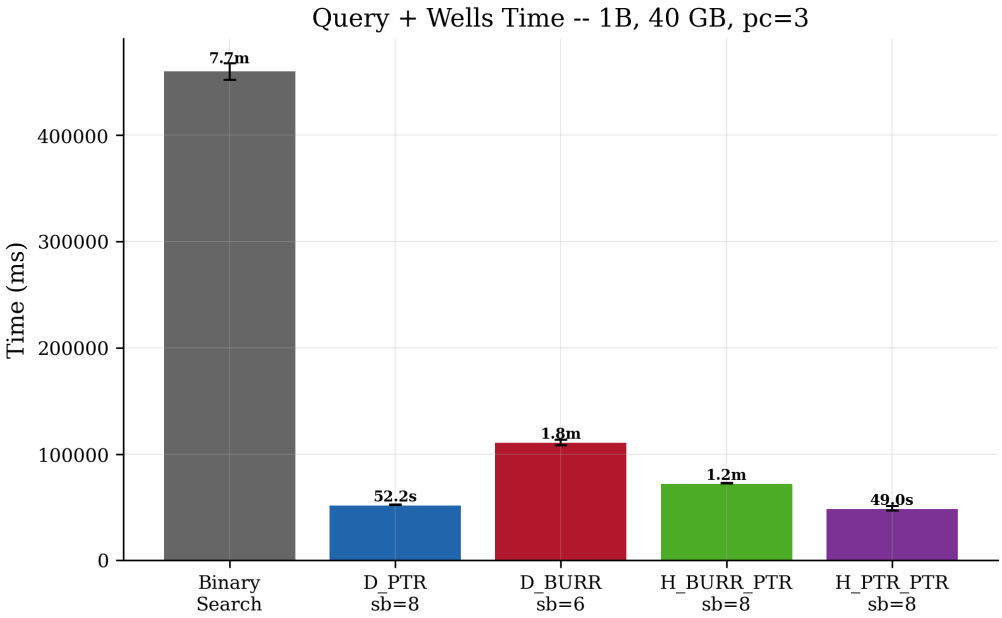

Figure 4.18: 1B, 40 GB heap: query + wells time per configuration (focused).

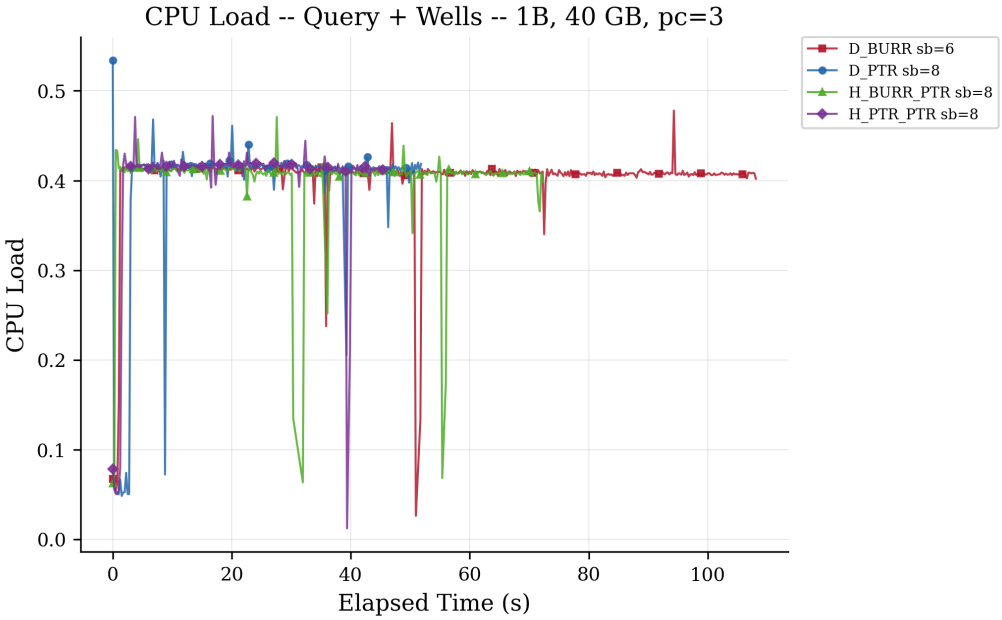

Figure 4.19: 1B, 40 GB heap: CPU load during query + wells (focused).

In Figure 4.19, all four configurations cluster in a tight band around 0.4 CPU load with brief dips toward zero where I/O briefly stalls computation. The profiles are essentially indistinguishable across backends, indicating that the phase is uniformly I/O-bound at this passCount; the larger heap budget has not changed which resource limits throughput.

Well Processing

Table 4.12 again shows the direct/hybrid split: the direct architectures finish well processing in 2.6–3.0 s, while the hybrids take 19.1–19.5 s — a roughly 7× difference. The direct backends (and Binary_Search) only stream the per-well file from disk and compute a checksum over the already-resolved row IDs, whereas the hybrid path additionally builds a per-well PtrHash MPHF and queries it for every entry to obtain the final row ID. Direct_PTR (sb = 8) is the fastest at 2.63 s; Binary_Search at 6.88 s sits in the same low band, while the hybrids' lookup work places them an order of magnitude higher.

Table 4.12: Well processing time for 1B, 40 GB heap, passCount = 3 (2 measured runs).

Architecture	SB	shardGroups	Min (ms)	Max (ms)	Avg (ms)	Std (ms)	Avg time / key (ns)
Binary_Search	—	—	6 648	7 159	6 875	260	2.29
Direct_PTR	6	1	2 534	4 844	3 689	1 633	1.23
Direct_PTR	**8**	**1**	**2 383**	**2 877**	**2 630**	**349**	**0.88**
Direct_PTR	10	1	2 821	3 448	3 134	443	1.04
Direct_BuRR	**6**	**1**	**2 763**	**3 201**	**2 982**	**310**	**0.99**
Direct_BuRR	8	1	3 106	3 307	3 206	142	1.07
Direct_BuRR	10	1	2 677	3 380	3 028	497	1.01
Hybrid_BuRR_PTR	6	1	19 262	21 445	20 354	1 544	6.78
Hybrid_BuRR_PTR	8	1	20 137	20 292	20 214	110	6.74
Hybrid_BuRR_PTR	**10**	**1**	**18 911**	**20 106**	**19 508**	**845**	**6.50**
Hybrid_PTR_PTR	**6**	**1**	**18 399**	**19 896**	**19 148**	**1 059**	**6.38**
Hybrid_PTR_PTR	8	1	18 670	21 077	19 874	1 702	6.62
Hybrid_PTR_PTR	10	1	18 553	26 358	22 456	5 519	7.49

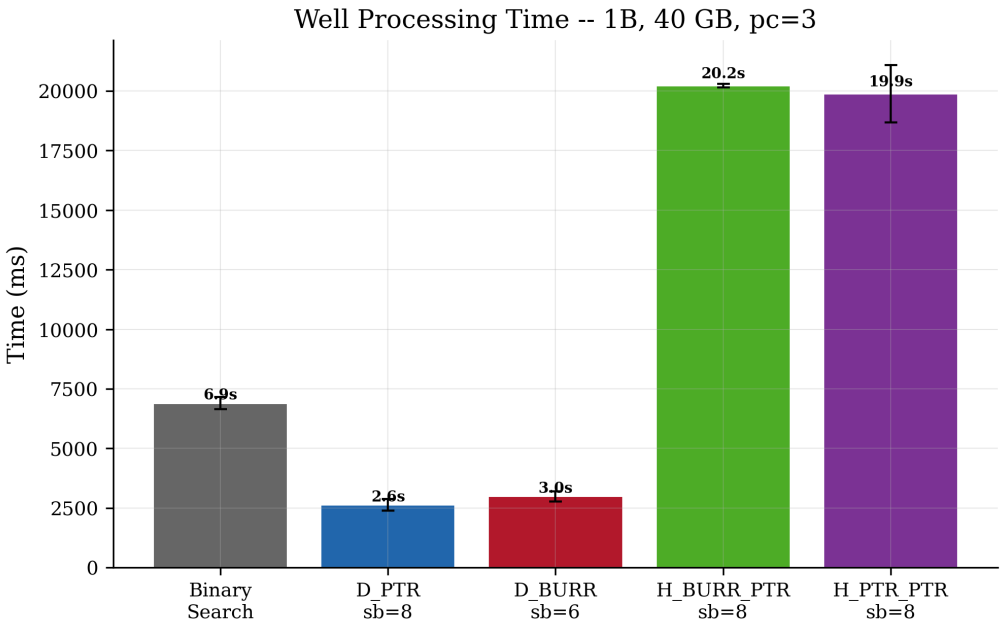

Figure 4.20: 1B, 40 GB heap: well processing time per configuration (focused).

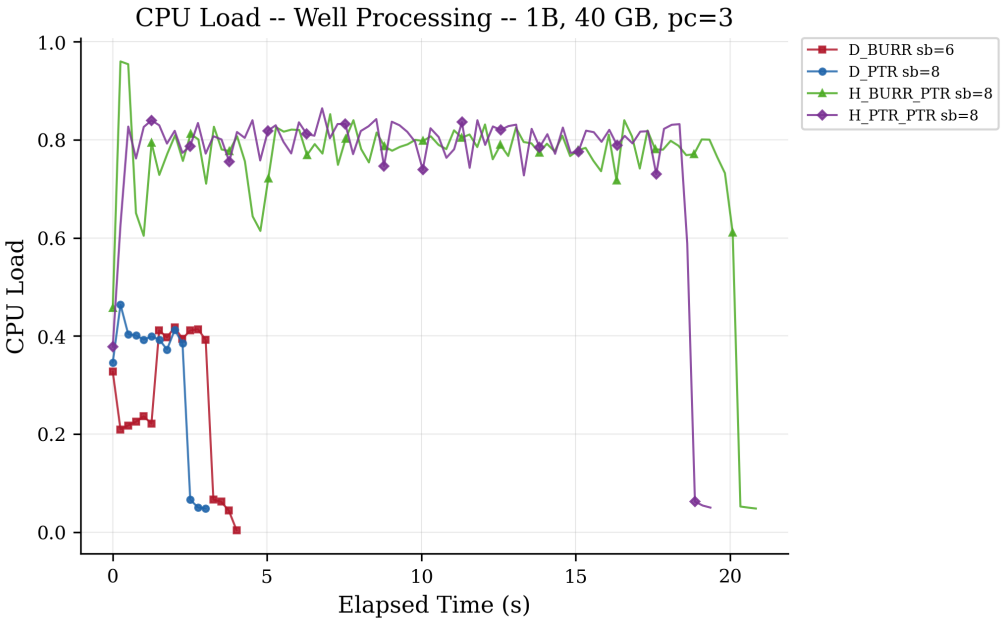

Figure 4.21: 1B, 40 GB heap: CPU load during well processing (focused).

Figure 4.21 confirms the same pattern as at 3 GB: the hybrids hold a steady **0.7–0.85** CPU load across their ∼ **19–20** s well phase, while the direct backends finish in **3–4** s with utilisation near **0.3–0.45**. The hybrid path's per-well MPHF construction translates directly into sustained CPU work.

4.1.4 40 GB Heap, passCount = 20

This subsection presents results under the same 40 GB heap with passCount = 20. Each configuration was measured over 2 runs.

Total Time

In Table 4.13, Hybrid_PTR_PTR (sb = 6) is the fastest at 513 s, while Direct_BuRR (sb = 6) is the slowest at 844 s — a **1.65×** gap. The ranking among MPHF backends matches the 3 GB / passCount = 20 case, confirming that the larger heap mainly shifts absolute times downward without changing which architecture wins. Direct_PTR's optimal shard-bit value moves to sb = 10 here, in contrast to sb = 8 at passCount = 3.

Table 4.13: Total time (all phases) for 1B, 40 GB heap, passCount = 20 (2 measured runs; 1 run for the binary search baseline).

Architecture	SB	shardGroups	Min (ms)	Max (ms)	Avg (ms)	Std (ms)
Binary search (baseline)	—	—	3 015 349	3 015 349	3 015 349	0
Direct_PTR	6	1	630 420	667 240	648 830	23 926
Direct_PTR	8	1	585 268	729 723	657 496	95 751
Direct_PTR	**10**	**1**	**531 568**	**612 448**	**572 008**	**54 268**
Direct_BuRR	**6**	**1**	**818 055**	**870 406**	**844 230**	**29 741**
Direct_BuRR	8	1	856 802	869 319	863 060	8 477
Direct_BuRR	10	1	905 418	970 284	937 851	43 199
Hybrid_BuRR_PTR	6	1	541 969	669 486	605 728	82 402
Hybrid_BuRR_PTR	**8**	**1**	**567 845**	**619 418**	**593 632**	**25 618**
Hybrid_BuRR_PTR	10	1	622 798	652 751	637 774	16 618
Hybrid_PTR_PTR	**6**	**1**	**436 751**	**588 554**	**512 652**	**88 606**
Hybrid_PTR_PTR	8	1	506 819	529 569	518 194	11 204
Hybrid_PTR_PTR	10	1	512 768	626 458	569 613	72 459

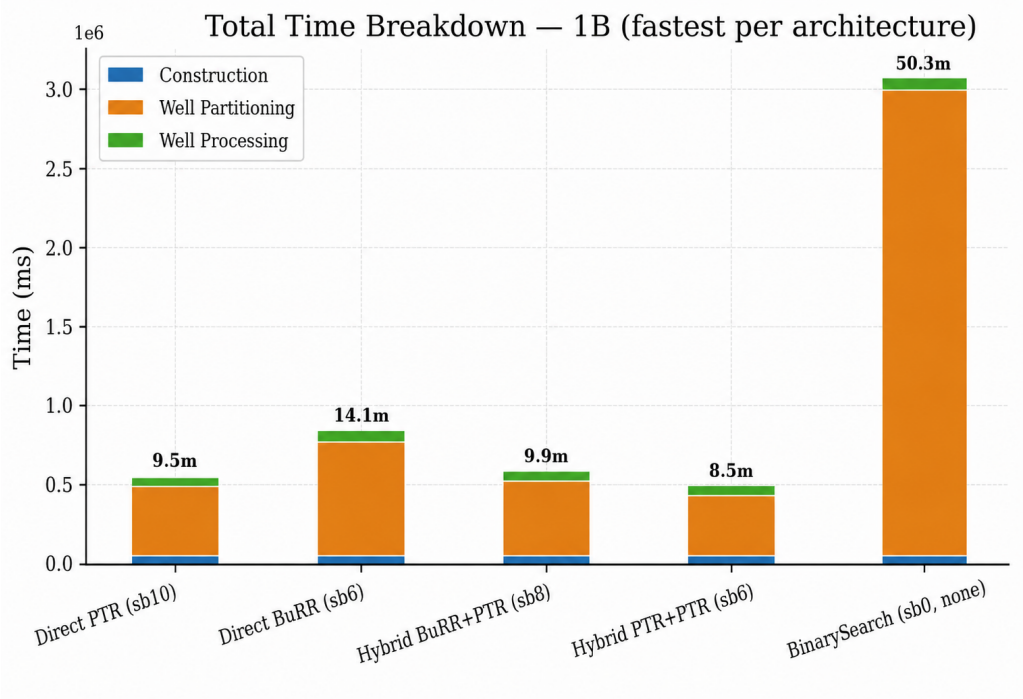

Figure 4.22: 1B, 40 GB heap: total time per configuration (focused).

Construction

Table 4.14 ranks Hybrid_BuRR_PTR (sb=10) as the fastest at 17.5 s and Direct_BuRR (sb=10) as the slowest at 20.5 s. As expected, construction times closely match those at passCount = 3 since this phase does not depend on passCount, and sb = 10 is consistently the best shard-bit choice across all four architectures.

Table 4.14: Construction time for 1B, 40 GB heap, passCount = 20 (2 measured runs; 1 run for the binary search baseline).

Architecture	SB	shardGroups	Min (ms)	Max (ms)	Avg (ms)	Std (ms)	Avg time / key (ns)
Binary search (baseline)	—	—	8 514	8 514	8 514	0	8.51
Direct_PTR	6	1	27 101	29 248	28 174	1 518	28.17
Direct_PTR	8	1	21 924	23 546	22 735	1 147	22.73
Direct_PTR	**10**	**1**	**20 016**	**20 140**	**20 078**	**88**	**20.08**
Direct_BuRR	6	1	22 815	22 867	22 841	37	22.84
Direct_BuRR	8	1	20 674	20 893	20 784	155	20.78
Direct_BuRR	**10**	**1**	**20 251**	**20 755**	**20 503**	**356**	**20.50**
Hybrid_BuRR_PTR	6	1	18 935	19 283	19 109	246	19.11
Hybrid_BuRR_PTR	8	1	18 026	18 369	18 198	243	18.20
Hybrid_BuRR_PTR	**10**	**1**	**17 358**	**17 664**	**17 511**	**216**	**17.51**
Hybrid_PTR_PTR	6	1	24 307	27 914	26 110	2 551	26.11
Hybrid_PTR_PTR	8	1	22 033	22 901	22 467	614	22.47
Hybrid_PTR_PTR	**10**	**1**	**18 611**	**19 353**	**18 982**	**525**	**18.98**

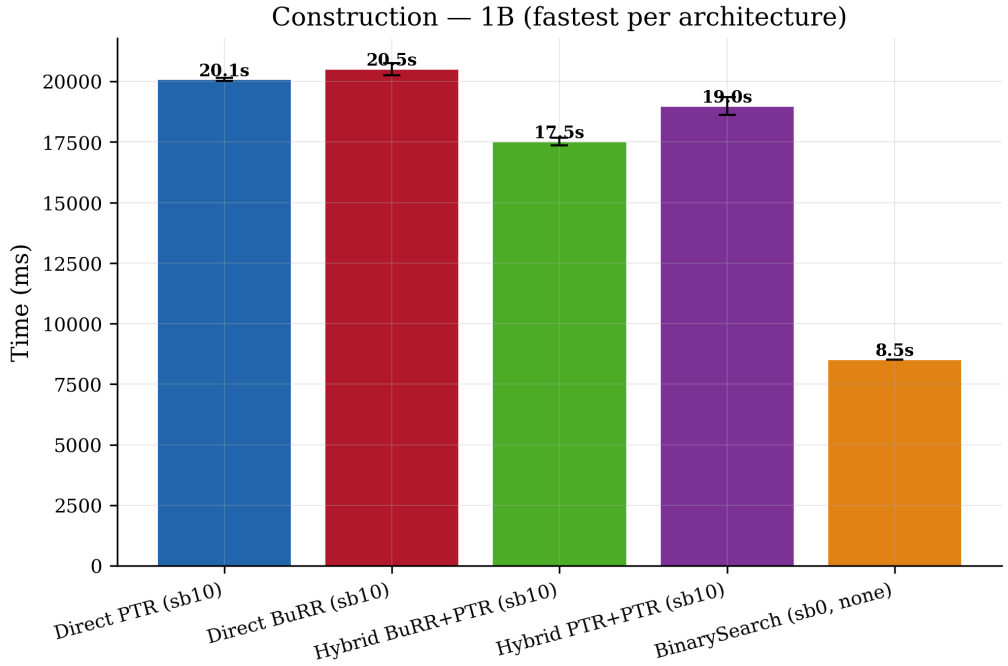

Figure 4.23: 1B, 40 GB heap: construction time per configuration (focused).

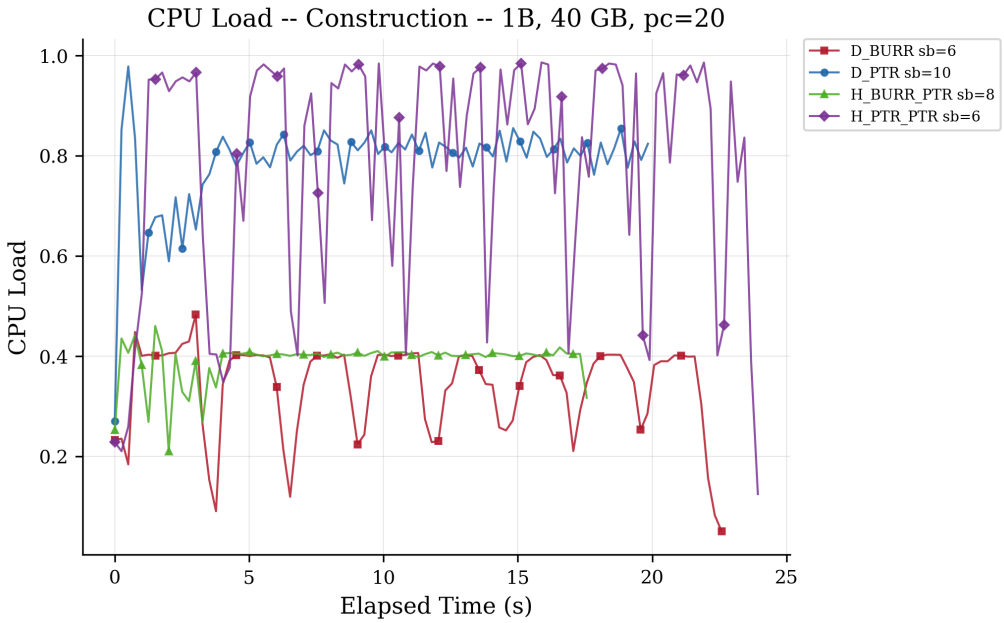

Figure 4.24: 1B, 40 GB heap: CPU load during construction (focused).

Figure 4.24 again shows the by-now-familiar split: PtrHash backends sustain ~ 0.7–0.95 CPU load, BuRR backends remain near 0.4. The construction profile is unchanged from the lower-passCount run, as expected.

Query and Well Partitioning

Table 4.15 shows that Hybrid_PTR_PTR (sb = 6) is the fastest at 416 s (20.80 ns/key), about 1.86× faster than the slowest MPHF, Direct_BuRR (sb = 6) at 773 s (38.65 ns/key). The query phase dominates total time at this passCount, so its ranking essentially determines the end-to-end ranking.

Table 4.15: Query and well partitioning time for 1B, 40 GB heap, passCount = 20 (2 measured runs; 1 run for the binary search baseline).

Architecture	SB	shardGroups	Min (ms)	Max (ms)	Avg (ms)	Std (ms)	Avg time / key (ns)
Binary search (baseline)	—	—	2 953 572	2 953 572	2 953 572	0	2953.57
Direct_PTR	6	1	528 778	562 534	545 656	23 869	27.28
Direct_PTR	8	1	493 533	628 720	561 126	95 592	28.06
Direct_PTR	**10**	**1**	**451 329**	**527 965**	**489 647**	**54 190**	**24.48**
Direct_BuRR	**6**	**1**	**752 819**	**793 137**	**772 978**	**28 509**	**38.65**
Direct_BuRR	8	1	791 867	803 849	797 858	8 473	39.89
Direct_BuRR	10	1	842 319	903 317	872 818	43 132	43.64
Hybrid_BuRR_PTR	6	1	456 767	572 764	514 766	82 022	25.74
Hybrid_BuRR_PTR	**8**	**1**	**491 130**	**516 471**	**503 800**	**17 919**	**25.19**
Hybrid_BuRR_PTR	10	1	547 739	570 071	558 905	15 791	27.95
Hybrid_PTR_PTR	**6**	**1**	**354 647**	**477 263**	**415 955**	**86 703**	**20.80**
Hybrid_PTR_PTR	8	1	427 298	440 575	433 936	9 388	21.70
Hybrid_PTR_PTR	10	1	422 306	524 175	473 240	72 032	23.66

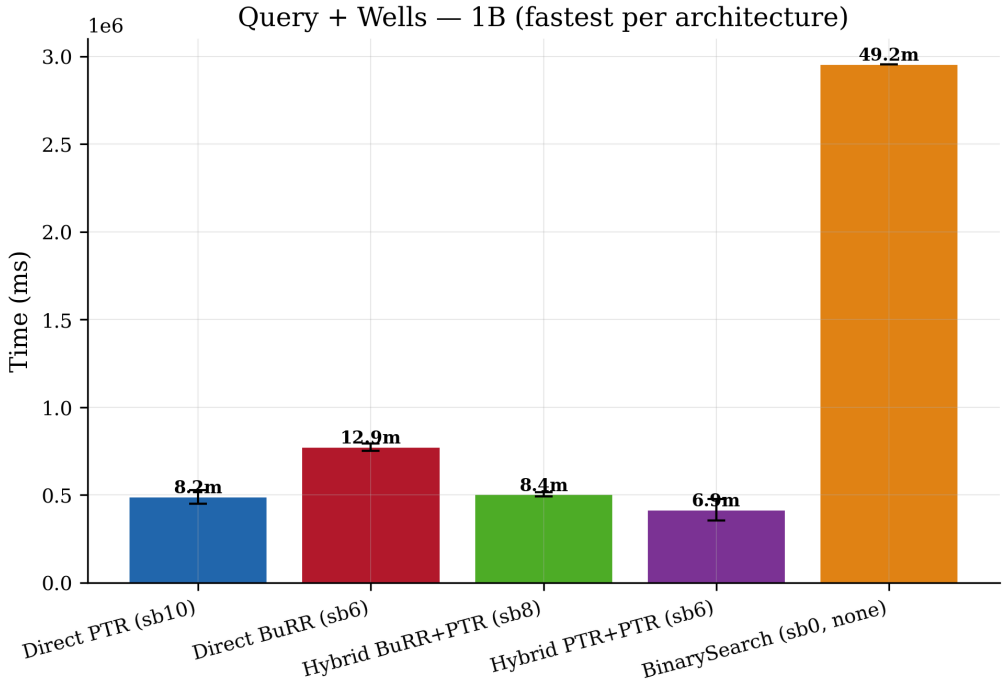

Figure 4.25: 1B, 40 GB heap: query + wells time per configuration (focused).

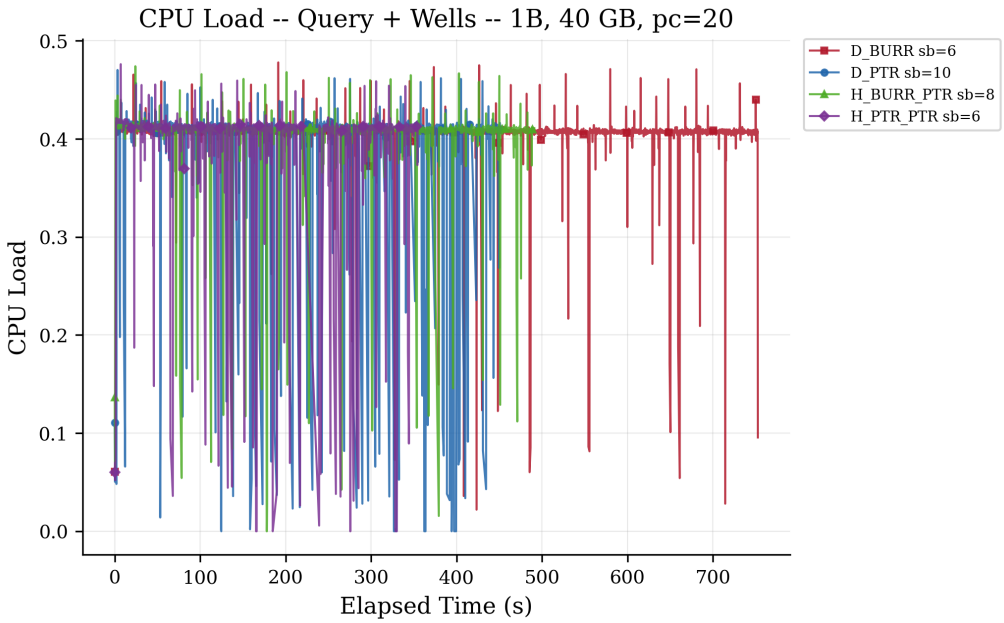

Figure 4.26: 1B, 40 GB heap: CPU load during query + wells (focused).

In Figure 4.26, all four configurations show heavy oscillation between near-zero and ~ 0.45 CPU load throughout the long query phase. As at the 3 GB heap, the high-frequency dips reflect repeated input-file passes rather than backend-specific behaviour: Direct_BuRR runs longest (~ 750 s) while Hybrid_PTR_PTR finishes earliest (~ 470 s), but the underlying utilisation profile is the same.

Well Processing

In Table 4.16, Direct_BuRR (sb = 8) is the fastest at 44.4 s (2.22 ns/key) and Direct_PTR (sb = 10) is the slowest at 62.3 s. The direct architectures still finish ahead of the hybrids (61 s and 62 s), but only by ~ 1.4× instead of 7× as at passCount = 3 — consistent with the hybrid's per-well construction overhead being amortized over more work.

Table 4.16: Well processing time for 1B, 40 GB heap, passCount = 20 (2 measured runs; 1 run for the binary search baseline).

Architecture	SB	shardGroups	Min (ms)	Max (ms)	Avg (ms)	Std (ms)	Avg time / key (ns)
Binary search (baseline)	—	—	53 263	53 263	53 263	0	53.26
Direct_PTR	6	1	74 541	75 458	75 000	648	3.75
Direct_PTR	8	1	69 811	77 457	73 634	5 407	3.68
Direct_PTR	**10**	**1**	**60 223**	**64 343**	**62 283**	**2 913**	**3.11**
Direct_BuRR	6	1	42 421	54 402	48 412	8 472	2.42
Direct_BuRR	**8**	**1**	**44 261**	**44 577**	**44 419**	**223**	**2.22**
Direct_BuRR	10	1	42 848	46 212	44 530	2 379	2.23
Hybrid_BuRR_PTR	6	1	66 267	77 439	71 853	7 900	3.59
Hybrid_BuRR_PTR	8	1	58 689	84 578	71 634	18 306	3.58
Hybrid_BuRR_PTR	**10**	**1**	**57 701**	**65 016**	**61 358**	**5 172**	**3.07**
Hybrid_PTR_PTR	6	1	57 797	83 377	70 587	18 088	3.53
Hybrid_PTR_PTR	**8**	**1**	**57 488**	**66 093**	**61 790**	**6 085**	**3.09**
Hybrid_PTR_PTR	10	1	71 851	82 930	77 390	7 834	3.87

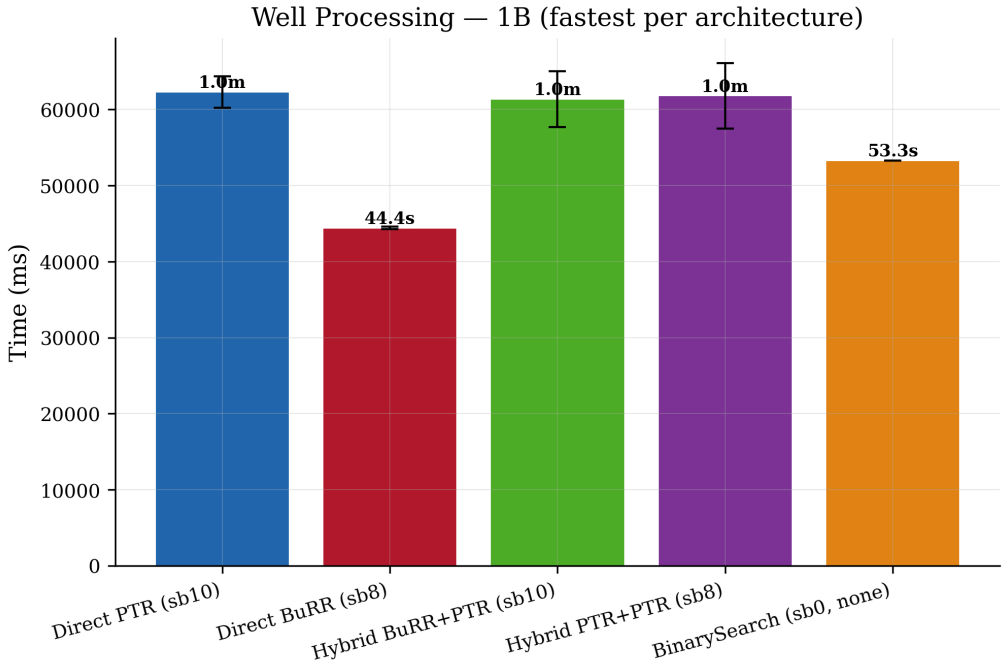

Figure 4.27: 1B, 40 GB heap: well processing time per configuration (focused).

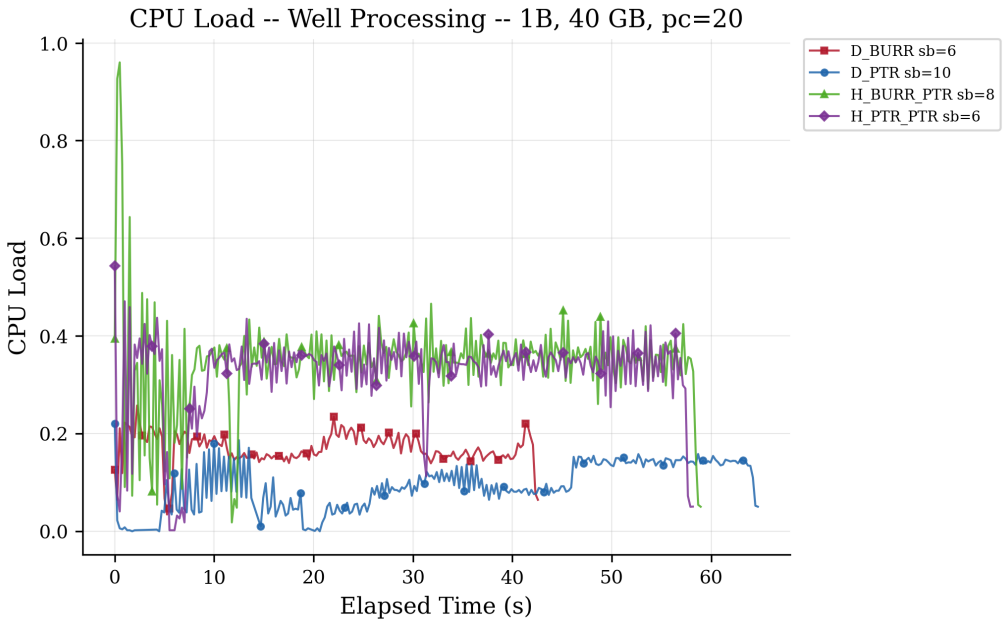

Figure 4.28: 1B, 40 GB heap: CPU load during well processing (focused).

Figure 4.28 shows that Direct_PTR sits very low at 0.05–0.15 while the other three settle in a 0.15–0.4 band. The hybrids again take the longest (~ 60 s) at the highest sustained load, mirroring the pattern seen at the smaller heap.

4.2 10 Billion Keys

This section presents benchmark results at a larger scale, using a dataset of 10^{10} keys. Two heap configurations are evaluated: a 20 GB heap with passCount = 5, and a 40 GB heap with passCount = 1. In both cases, each architecture is tested with shard bits 10 and 14, 1 024 wells, 8 threads, a cache fraction of 0.80, and the `fast` PtrHash preset (where applicable). In each table, the fastest architecture within each shard-bit group is shown in **bold**, and the overall fastest configuration is **bold and underlined**.

4.2.1 20 GB Heap, passCount = 5

This subsection presents results for all four architectures at a 20 GB JVM heap with passCount = 5. Each architecture was evaluated with shard bits 10 and 14. A single measured run was performed per configuration.

Total Time

In Table 4.17, Hybrid_BuRR_PTR (sb = 10) is the fastest end-to-end at 2 506 s (≈ 41.8 min), narrowly ahead of Hybrid_PTR_PTR (sb = 10) at 2 545 s. Direct_BuRR (sb = 10) is the slowest

at 3 725 s — a **1.5×** gap. Increasing shard bits from 10 to 14 hurts every architecture except Hybrid_PTR_PTR (which is essentially flat).

Table 4.17: Total time (all phases) for 10B, 20 GB heap, passCount = 5 (single run).

Architecture	SB	shardGroups	Time (ms)
Direct_PTR	**10**	**4**	**2 816 066**
Direct_PTR	14	4	3 346 449
Direct_BuRR	**10**	**4**	**3 724 809**
Direct_BuRR	14	4	4 233 345
Hybrid_BuRR_PTR	**10**	**1**	**2 506 321**
Hybrid_BuRR_PTR	14	1	3 551 237
Hybrid_PTR_PTR	**10**	**2**	**2 545 147**
Hybrid_PTR_PTR	14	2	2 563 739

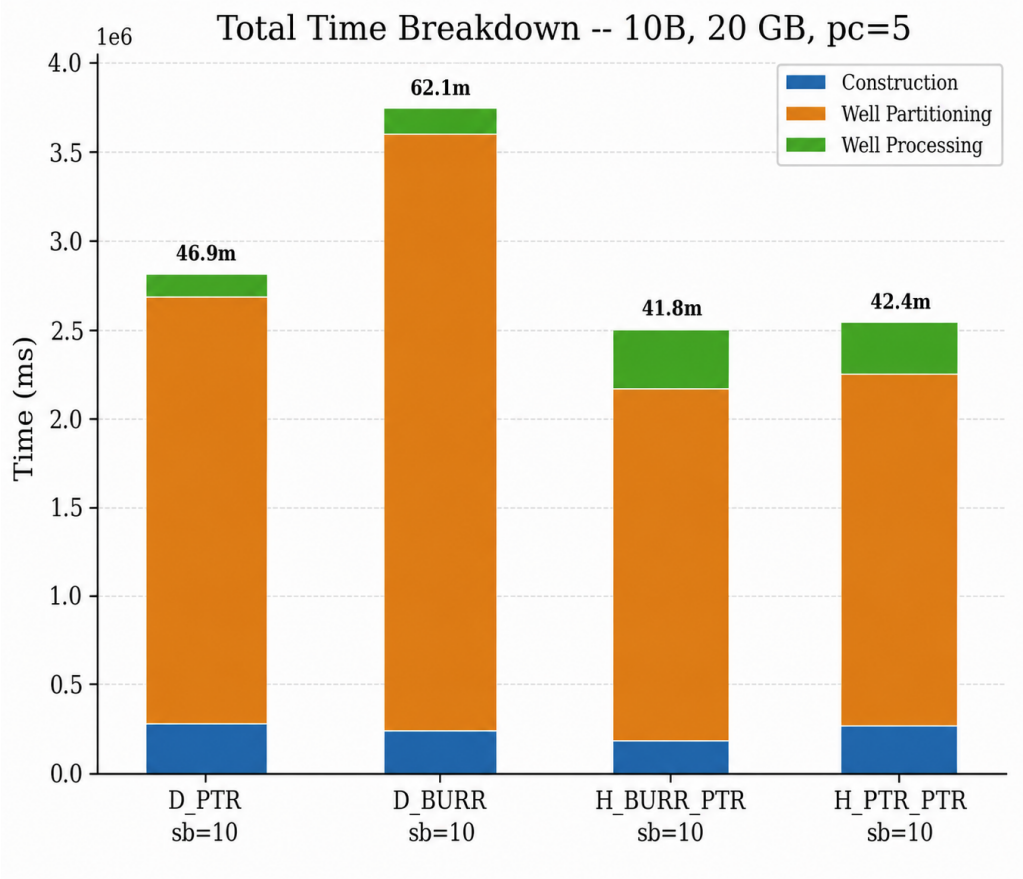

Figure 4.29: 10B, 20 GB heap: total time per configuration (focused).

Construction

Table 4.18 ranks Hybrid_BuRR_PTR (sb = 10) as the fastest at 186 s (18.64 ns/key) and Direct_PTR (sb = 10) as the slowest among per-architecture best picks at 276 s (27.64 ns/key). The hybrid backends construct noticeably faster than their direct counterparts because the well-level structure pushes most of the work into the cheaper inner hash. See Section 3.4 for a description of the construction phase.

Table 4.18: Construction time for 10B, 20 GB heap, passCount = 5 (single run).

Architecture	SB	shardGroups	Time (ms)	Avg time / key (ns)
Direct_PTR	**10**	**4**	**276 390**	**27.64**
Direct_PTR	14	4	294 604	29.46
Direct_BuRR	**10**	**4**	**240 960**	**24.10**
Direct_BuRR	14	4	281 211	28.12
Hybrid_BuRR_PTR	**10**	**1**	**186 359**	**18.64**
Hybrid_BuRR_PTR	14	1	215 682	21.57
Hybrid_PTR_PTR	10	2	273 051	27.31
Hybrid_PTR_PTR	**14**	**2**	**226 194**	**22.62**

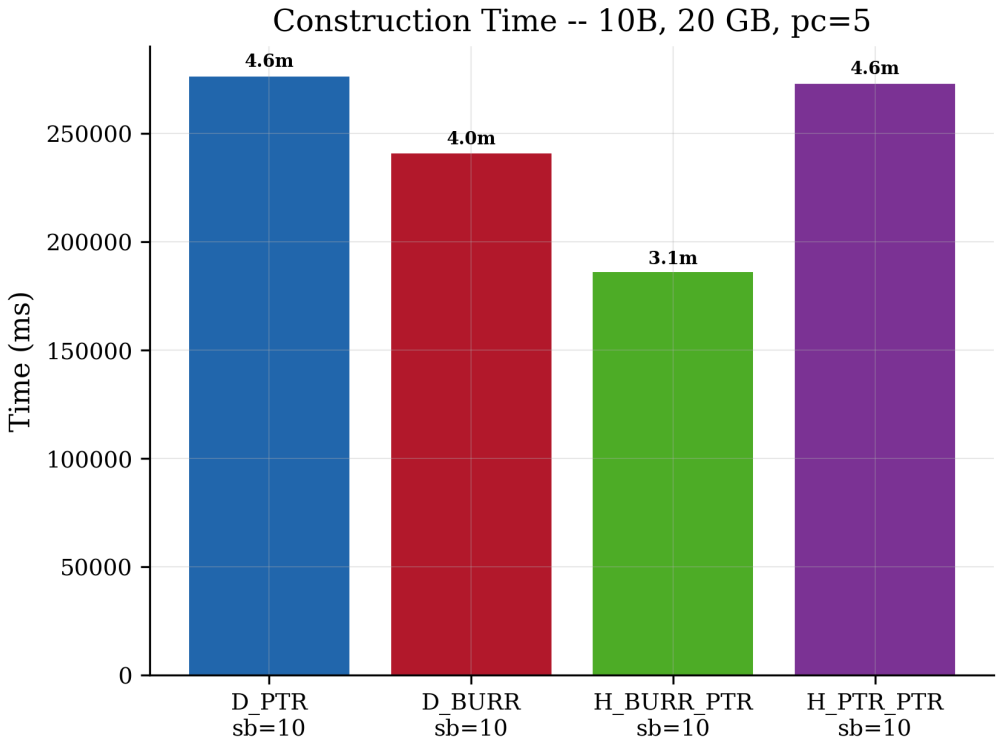

Figure 4.30: 10B, 20 GB heap: construction time per configuration (focused).

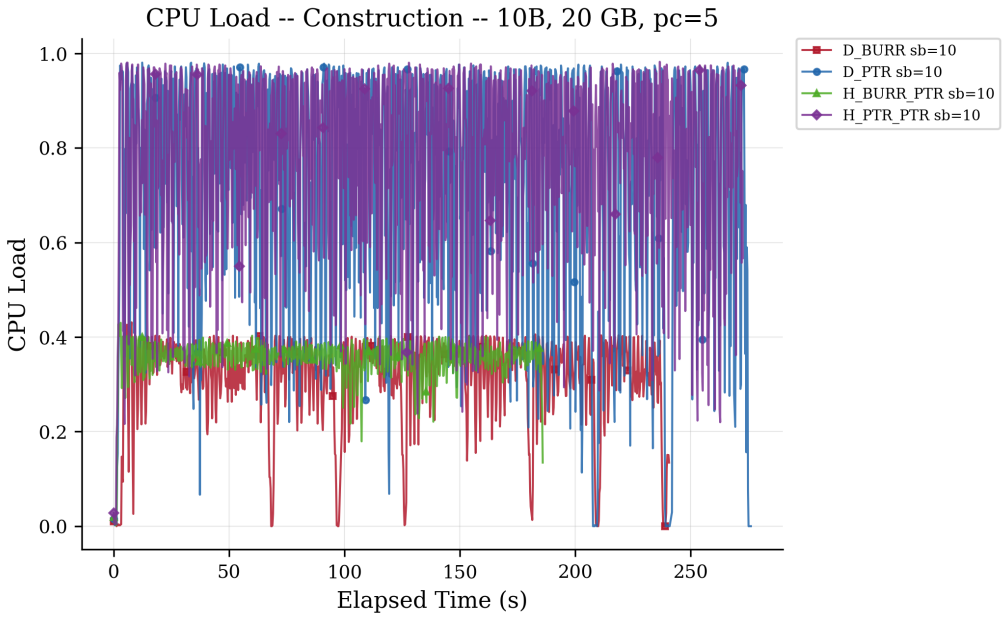

Figure 4.31: 10B, 20 GB heap: CPU load during construction (focused).

Figure 4.31 shows a high-frequency sawtooth from the multi-shard build: each dip is the gap between finishing one shard and starting the next. The same backend split as at 1 B is still visible — PtrHash configurations average ~ 0.7–0.95 between dips, BuRR configurations average ~ 0.3–0.4 — confirming that the per-backend CPU profile is independent of dataset size.

Query and Well Partitioning

In Table 4.19, Hybrid_PTR_PTR (sb = 10) is the fastest at 1 983 s (39.67 ns/key), with Hybrid_BuRR_PTR (sb = 10) close behind at 2 002 s (40.04 ns/key). Direct_BuRR (sb = 10) is the slowest at 3 374 s (67.48 ns/key) — about 1.7× slower than the fastest. The query phase accounts for the bulk of total time at this scale, so the same hybrid-favouring ranking carries through to the end-to-end result. See Section 3.4.1 for a description of this phase.

Table 4.19: Query and well partitioning time for 10B, 20 GB heap, passCount = 5 (single run).

Architecture	SB	shardGroups	Time (ms)	Avg time / key (ns)
Direct_PTR	**10**	**4**	**2 432 378**	**48.65**
Direct_PTR	14	4	2 944 692	58.89
Direct_BuRR	**10**	**4**	**3 374 056**	**67.48**
Direct_BuRR	14	4	3 847 016	76.94
Hybrid_BuRR_PTR	**10**	**1**	**2 001 992**	**40.04**
Hybrid_BuRR_PTR	14	1	3 003 034	60.06
Hybrid_PTR_PTR	**10**	**2**	**1 983 490**	**39.67**
Hybrid_PTR_PTR	14	2	2 008 619	40.17

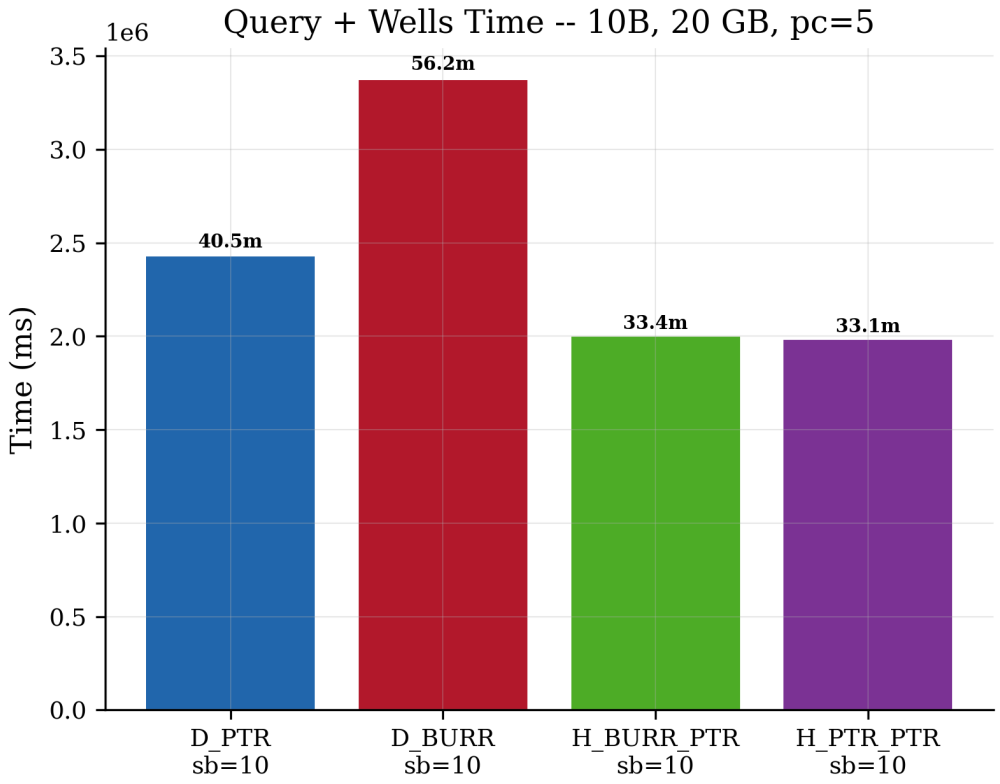

Figure 4.32: 10B, 20 GB heap: query + wells time per configuration (focused).

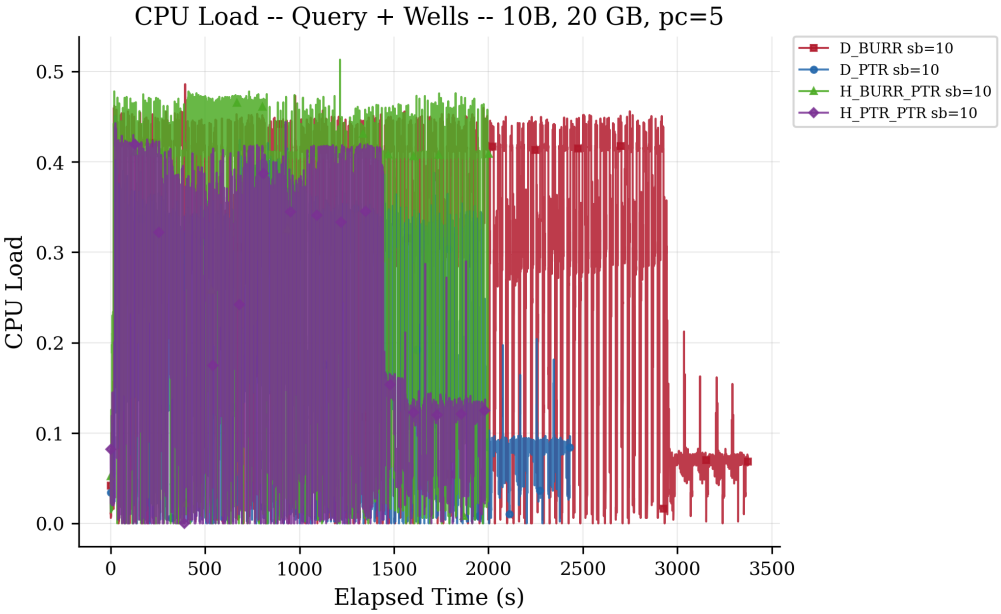

Figure 4.33: 10B, 20 GB heap: CPU load during query + wells (focused).

In Figure 4.33, all four configurations oscillate between ~ 0.05 and ~ 0.5 CPU load throughout the multi-thousand-second phase, never approaching saturation. The two hybrids finish first (~ 1500–2000 s), Direct_PTR follows (~ 2400 s), and Direct_BuRR runs the longest (~ 3300 s); the load profile is similar across all four, again indicating an I/O-bound phase.

Well Processing

Table 4.20 continues the direct/hybrid split seen at 1 B keys: Direct_BuRR (sb=14) is fastest at 105 s (2.10 ns/key), Direct_PTR (sb=14) is tied at 107 s, while Hybrid_BuRR_PTR (sb=10) is slowest at 318 s — roughly 3× slower. See Section 3.4.1 for a description of this phase.

Table 4.20: Well processing time for 10B, 20 GB heap, passCount = 5 (single run).

Architecture	SB	shardGroups	Time (ms)	Avg time / key (ns)
Direct_PTR	10	4	107 298	2.15
Direct_PTR	**14**	**4**	**107 153**	**2.14**
Direct_BuRR	10	4	109 793	2.20
Direct_BuRR	**14**	**4**	**105 118**	**2.10**
Hybrid_BuRR_PTR	**10**	**1**	**317 970**	**6.36**
Hybrid_BuRR_PTR	14	1	332 521	6.65
Hybrid_PTR_PTR	**10**	**2**	**288 606**	**5.77**
Hybrid_PTR_PTR	14	2	328 926	6.58

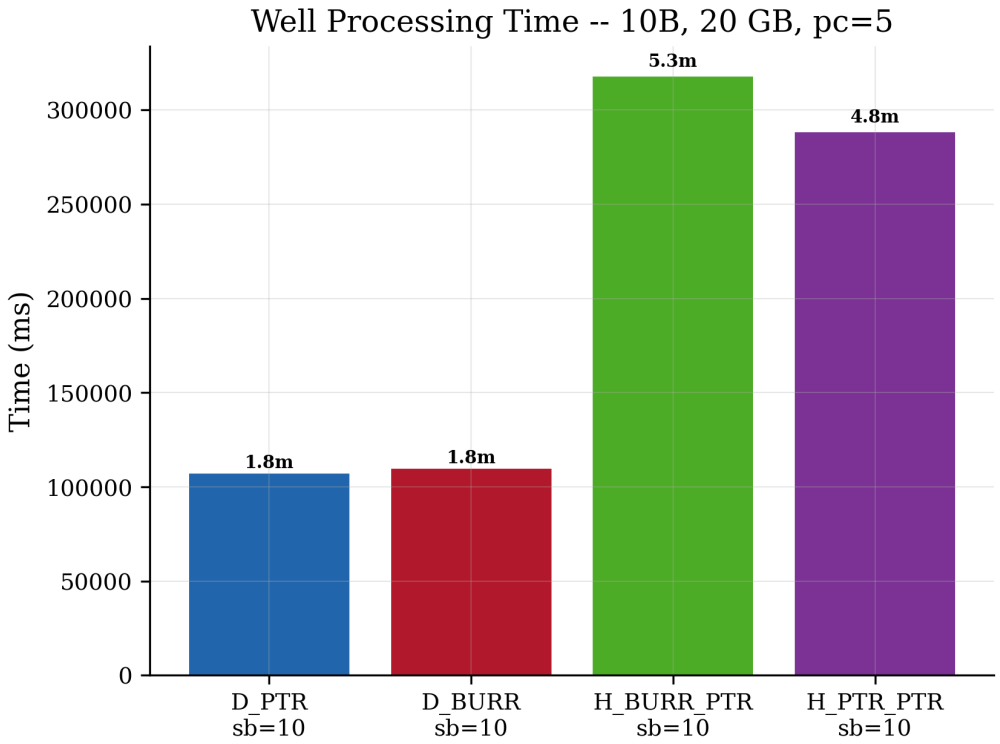

Figure 4.34: 10B, 20 GB heap: well processing time per configuration (focused).

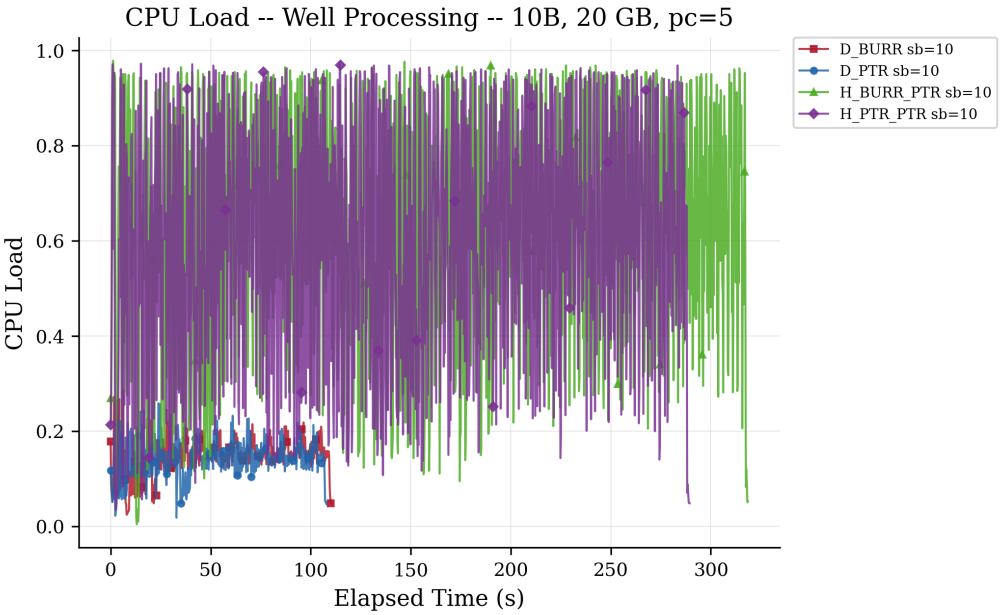

Figure 4.35: 10B, 20 GB heap: CPU load during well processing (focused).

Figure 4.35 shows that the hybrids sustain a **0.5–0.95** CPU load for their full **~ 290–320** s well phase, while the direct backends finish in **~ 110** s with much lower utilisation around **0.1–0.2**. The same direct/hybrid split observed at 1 B carries over unchanged to 10 B.

Chapter 5
Discussion

5.1 Analysis of Benchmark Results

5.1.1 1B keys experiment

This section interprets the benchmark results presented in Section 4.1.

Total Time The 10^9-key benchmark represents a scale where the lookup structures are still small enough to be handled efficiently on the benchmark machine, but the results show that the best architecture depends on both the available heap and the number of query passes. With `passCount = 3`, the direct PtrHash-based architecture performs best overall. Under the constrained 3 GB heap, `Direct_PTR` has the lowest average total time, although `Hybrid_PTR_PTR` is close. Under the 40 GB heap, `Direct_PTR` with 8 shard bits is the fastest configuration.

When `passCount` is increased to 20, the trend changes. In both the 3 GB and 40 GB heap experiments, `Hybrid_PTR_PTR` becomes the fastest architecture. This indicates that the hybrid design is not universally faster at the 1B scale, but that it becomes advantageous when the query and well-partitioning phase is repeated many times, corresponding to larger relationship files in real world scenarios. In this setting, the reduced shard traffic and smaller per-well structures compensate for the extra lookup stage introduced by the hybrid architecture.

Construction No single architecture dominates construction time across all 1B configurations. Under the 3 GB heap, `Direct_BuRR` constructs fastest, while the hybrid BuRR-based design is close. Under the 40 GB heap, the fastest construction times are generally obtained by the hybrid architectures, especially at higher shard-bit values. This suggests that construction time is affected not only by the choice of underlying structure, but also by the shard size and the value width used inside each structure.

When `passCount` is high, the construction phase is a small part of the total runtime. Differences in construction time therefore have less influence on the overall result than differences in the query and well-partitioning phase.

Query and Well Partitioning The query and well-partitioning phase is the dominant phase when `passCount` is high. The updated results show that the fastest architecture in this phase is usually `Hybrid_PTR_PTR`, not `Direct_PTR`. This is especially clear for `passCount = 20`, where `Hybrid_PTR_PTR` achieves the lowest average query and well-partitioning time in both heap configurations.

This result suggests that the benefit of the hybrid design comes primarily from the query phase. Although each hybrid lookup requires an additional step, the smaller structures and reduced shard-group pressure can outweigh this overhead. Note that in the 3GB heap case `Direct_PTR` had to be split into 4 shard groups while `Hybrid_PTR_PTR` only had to be split into 2, thus halfing the number of passes during phase 2. This significant decrease in IO operations is likely the reason behind the improved performance compared to the 40GB case. In contrast, `Direct_BuRR` remains slower in this phase, which is consistent with the higher query cost of retrieving larger values directly from a BuRR structure.

Well Processing The well-processing phase is generally faster for the direct architectures, particularly when `passCount = 3`. This is expected, since the direct architectures already map directly to the final value, whereas the hybrid architectures require an additional resolution step during processing.

When `passCount = 20`, the well-processing phase is small relative to the query and well-partitioning phase. Therefore, even though the hybrid architectures often pay an extra cost during well processing, this cost is outweighed by their faster query phase in the high-pass-count experiments.

CPU Utilization The CPU-utilization traces support the interpretation that the bottlenecks differ between phases. In construction, lower CPU utilization for some BuRR-based configurations indicates that the implementation may not fully saturate the available compute resources. In the query and well-partitioning phase, the runtime differences are more closely tied to memory access patterns, shard-group handling, and repeated input scans. During well processing, the low CPU utilization across several configurations suggests that memory access and page/cache behaviour are more important than raw computation.

Overall, the 1B results show that `Direct_PTR` is the strongest option when the entire data structure can fit in memory. `Hybrid_BuRR_PTR` and especially `Hybrid_PTR_PTR` become more likely to be better perforers in cases where the ratio of node data to memory is higher and even more so when the query phase contains more data. Thus, at this scale, the hybrid architecture should be interpreted as a workload-dependent optimization rather than a universally better replacement for the direct lookup table. For most practical cases, `Direct_PTR` is likely the best contender at this scale as long as memory is not an issue.

5.1.2 10B keys experiment

This section interprets the benchmark results presented in Section 4.2 and compares them with the smaller 10^9-key experiments from Section 4.1. The main difference at this scale

is that memory pressure and shard management become more important than at 10^9 keys. At 10^9 keys, the direct PtrHash-based architecture is highly competitive because the lookup structures are still small enough that the extra indirection of the hybrid architectures is difficult to justify. At 10^{10} keys, this changes: the reduced memory footprint of the hybrid architectures becomes large enough to compensate for the additional lookup stage.

With a 20 GB heap and `passCount = 5`, `Hybrid_BuRR_PTR` achieves the strongest result, completing in 2 506 321 ms versus 2 816 066 ms for the fastest direct architecture, `Direct_PTR`. This corresponds to an improvement of approximately 11%. This behavior contrasts with the 10^9-key results. In the smaller benchmark, the hybrid architectures do not consistently outperform the direct architectures. For example, with a 40 GB heap and `passCount = 3`, `Direct_PTR` is the fastest overall. Only when the query phase is repeated more heavily, as in the `passCount = 20` experiment, do the hybrid architectures become competitive. This suggests that the hybrid approach is most useful when either the data set is large enough to create memory pressure, or the lookup phase is repeated enough times for reduced shard traffic to dominate the extra cost of the second lookup stage.

Overall, the 10^{10}-key experiments support the motivation behind the hybrid design. At smaller scales, the additional complexity of the hybrid lookup is not always worthwhile. At larger scales, where memory footprint and shard loading become more significant, the hybrid architecture becomes the better option. This indicates that the hybrid approach is primarily a scalability optimization rather than a universally faster replacement for the direct lookup table.

5.2 Methodology

This section reflects on the strengths, weaknesses, and key learnings from each methodological step taken in this thesis. Section 3 described the actions performed to answer the research questions, which produced the results presented in Section 4. The discussion below evaluates how each step was executed and what could have been done differently.

5.2.1 Literature Review

Strengths. Framing the search around the problem itself, a static key set known in full before queries begin, kept the scope narrow and directed the review toward perfect hashing and retrieval structures from the outset. Discovering the survey by Lehmann et al. [28] early in the process provided a good overview of the field and acted as a roadmap for subsequent reading, reducing the risk of overlooking recent constructions.

Weaknesses. The review was exploratory rather than systematic. No formal inclusion or exclusion criteria were defined, search queries were not recorded, and the snowball sampling strategy [32] introduces a bias toward the citation neighborhood of the initial seed papers. This made us overlook other potential data structures, such as interpolation search [10].

Learnings. A brief pre-registration of search terms, queried databases, and date ranges would have made the review reproducible at minimal cost. In future work of this kind, alter-

nating between problem-driven and structure-driven searches would help guard against the tunnel vision induced by following a single citation chain.

5.2.2 Implementation

Strengths. Sharing the sharding, streaming, and pipelining layers between PtrHash and BuRR means that differences observed in the benchmarks reflect the structures themselves rather than incidental differences in surrounding code, which is important for internal validity. The MultiPass query design addressed the I/O thrashing problem pragmatically and was diagnosed and corrected before it could undermine the benchmark results.

Weaknesses. The ShardWindow strategy was designed, but due to time constraints, was not implemented or evaluated, leaving an open question rather than a closed answer. Several engineering heuristics, such as the 128 MB I/O buffer, the cache fraction $f = 0.8$, and the group size formula. Finally, the fixture omits the 8-byte pointer used in production to resolve hash collisions; this was a deliberate trade-off to make larger benchmarks possible, but it is a deviation from production that should not be understated.

Learnings. Allocating time at the end of the implementation phase for a small sensitivity sweep over the chosen heuristics would have improved confidence in the configuration choices.

5.2.3 Benchmarks

Strengths. Running a single comprehensive benchmark per architecture and instrumenting each phase, construction, partitioning, and lookup, individually, allows the reader to see where time is spent rather than only the totals. Including `BinarySearchLookup` as a baseline grounds the absolute numbers in a familiar structure and provides a reference point for judging the new architectures.

Weaknesses. The benchmarks were executed on a single hardware configuration, a consumer-class laptop. No server-class node, no alternative storage tier, and no varying memory-pressure regime were evaluated, which directly limits external validity. The fixture is drawn from a uniform distribution of synthetic 64-bit keys, whereas real graph imports exhibit clustered identifiers and structured key formats before hashing; this restricts construct validity. The number of measured runs, three at 10^9 keys and one at 10^{10}, was a deliberate compromise driven by computational cost. Still, it limits the statistical strength of the conclusions. Finally, the `BinarySearchLookup` baseline is a simplification of the production import path rather than the production code itself, so comparisons to "Neo4j today" are indicative rather than precise.

Learnings. Even a single additional hardware target, such as a server-class machine with NVMe storage, would have substantially improved external validity for relatively modest extra effort. Extending the fixture generator to support optional skew and duplicate injection

would have strengthened the construct validity story without changing the experimental harness.

Chapter 6
Conclusion

This thesis implemented four architectures based on the two state-of-the-art data structures BuRR and PtrHash. These were evaluated against the current `BinarySearchLookup` baseline. The remainder of this chapter restates the research questions and summarizes the answers obtained.

RQ1: How do different state-of-the-art lookup table data structures perform in terms of optimizing the import process for graph databases? Compared to the current `BinarySearchLookup` baseline, the fastest configuration reduced end-to-end import time by 83.6 % (from 479 816 ms to 78 644 ms, a 6.1× speedup). This was achieved by the `Direct_PTR` architecture with shard bits = 8 and the `fast` PtrHash configuration. State-of-the-art MPHF-based lookup structures therefore offer a substantial improvement over the existing baseline at the $1{\times}10^9$ key scale. The lookup phase shows an 8.8× improvement, indicating that it will scale even better with higher relationship-to-node ratios than the 3:1 we ran this benchmark on. Real-world workloads can see ratios up to 20:1 in typical cases.

RQ2: Can modification/combinations of different state-of-the-art lookup table data structures be more efficient at performing graph database imports? The hybrid architectures combine a coarse-grained well function with sharded per-well PtrHash instances (see Section 3.2.1). Whether this combination is faster than a single lookup table depends on scale. At $1{\times}10^9$ keys with the default passCount, `Direct_PTR` remains the fastest configuration at 78 644 ms, against 91 306 ms for the best hybrid (Section 4.1); the hybrids only overtake `Direct_PTR` once passCount is raised to 20. At 1×10^{10} keys, the hybrids overtake the direct approach: `Hybrid_BuRR_PTR` achieves 2 506 321 ms compared to 2 816 066 ms for `Direct_PTR`, an 11.0 % reduction (Section 4.2). The hybrid advantage grows with scale because the eliminated value array would otherwise force additional sharding and disk I/O. RQ2 is therefore answered in the affirmative, with

the qualification that the combination only pays off once the working set is large enough for the value array to become the dominant cost.

6.1 Threats to Validity

Runtime variability The benchmarking results may be affected by runtime variability introduced by the Java Virtual Machine (JVM), including just-in-time compilation and garbage collection pauses. These effects can lead to inconsistent execution times, particularly during initial runs.

For the 10^9-key experiments, each configuration was measured 3 times and the results averaged. For the 10^{10}-key experiments, due to computational cost, each architecture was measured with a single run.

Hardware dependency Performance results are influenced by the underlying hardware, including CPU cache sizes, memory bandwidth, and storage performance. Since the evaluated data structures are sensitive to memory access patterns, results may vary across systems.

Implementation bias Differences in implementation quality, optimization, and integration with the Neo4j codebase may affect performance. Observed differences may therefore reflect implementation details rather than inherent algorithmic properties.

Hash function choice The choice of hash functions affects key distribution and collision behavior. Since this study reuses hash functions from the Neo4j codebase, alternative choices may lead to different results.

JVM optimizations JVM-specific optimizations, such as escape analysis and automatic vectorization, may influence execution behavior in ways that are difficult to control or reproduce.

I/O and sharding effects The use of sharding and disk-based storage introduces additional variability due to I/O performance. This may obscure the pure in-memory performance characteristics of the evaluated data structures.

Isolated evaluation The data structures are evaluated outside the full Neo4j import pipeline. Interactions with components such as CSV parsing, page caching, and disk I/O are therefore not captured.

Synthetic datasets The use of synthetic datasets enables controlled experimentation but may not reflect real-world data characteristics, such as skewed distributions, varying key sizes, or duplicate frequencies.

Scale limitations Although large datasets are considered, practical constraints prevent evaluation at the extreme scales discussed theoretically. Generalization to such scales therefore remains uncertain.

6.2 Future Work

Several directions for future research emerge from this work.

ShardWindow lookup strategy As described in Section 3.4.1, the MultiPass approach re-reads the entire input file once per shard group, which becomes expensive when the number of passes is large. The ShardWindow strategy, which sorts the intermediate representation and enforces a sliding window over loaded shards, was designed but not implemented due to time constraints. An implementation and empirical comparison of ShardWindow against MultiPass would clarify whether eliminating redundant I/O passes yields a net improvement, or whether the upfront sorting cost dominates.

End-to-end integration The benchmarks in this study evaluate the lookup structures in isolation from the full Neo4j import pipeline. Integrating the proposed architectures into the production importer and measuring end-to-end import time, including CSV parsing, page caching, and store writes, would reveal whether the gains observed in isolation translate to real-world speedups or are masked by other bottlenecks.

Scaling to larger datasets While datasets of 10^{10} keys were evaluated, the theoretical analysis targets key counts up to 10^{14}. Benchmarking at the 10^{11} scale and beyond, with particular attention to I/O behavior and shard management overhead, would strengthen the practical relevance of the results.

Alternative hardware All experiments ran on a single Lenovo ThinkPad. Evaluating the architectures on other hardware platforms, including ARM processors (e.g., Apple M-series or AWS Graviton) and cloud environments with network-attached storage, would clarify how sensitive the results are to hardware characteristics such as cache hierarchy, memory bandwidth, and storage latency.

More extensive benchmarks Running the benchmarks with more runs at different times of day, with an isolated benchmark machine, might give more accurate benchmark numbers.

Different well sizes Currently, only 1024 wells are tested and benchmarked. This is an arbitrary number selected to be reasonable for a production use case, but the actual number of wells will vary. Therefore, it would have been interesting to benchmark how well the effects affect execution time.

References

[1] Aura. URL: https://neo4j.com/product/auradb/.

[2] Community Edition. URL: https://neo4j.com/product/community-edition/.

[3] *Computer Architecture: A Quantitative Approach 6th Edition* .

[4] Importing data - Operations Manual. URL: https://neo4j.com/docs/operations-manual/2026.01/tutorial/neo4j-admin-import/.

[5] Neo4j Surpasses $200M in Revenue, Accelerates Leadership in GenAI-Driven Graph Technology. URL: https://neo4j.com/press-releases/neo4j-revenue-milestone-2024/.

[6] SplittableRandom (Java Platform SE 8). URL: https://docs.oracle.com/javase/8/docs/api/java/util/SplittableRandom.html.

[7] (PDF) Hash Tables as Engines of Randomness at the Limits of Computation: A Unified Review of Algorithms. *ResearchGate*, December 2025. URL: https://www.researchgate.net/publication/398835964_Hash_Tables_as_Engines_of_Randomness_at_the_Limits_of_Computation_A_Unified_Review_of_Algorithms, doi:10.3390/a18120804.

[8] Binary search. *Wikipedia*, April 2026. URL: https://en.wikipedia.org/w/index.php?title=Binary_search&oldid=1351347892.

[9] Database. *Wikipedia*, February 2026. URL: https://en.wikipedia.org/w/index.php?title=Database&oldid=1337009515.

[10] Interpolation search. *Wikipedia*, January 2026. URL: https://en.wikipedia.org/w/index.php?title=Interpolation_search&oldid=1332633084.

[11] Neo4j Graph Database, February 2026. URL: https://neo4j.com/product/neo4j-graph-database/.

[12] SQL. *Wikipedia*, February 2026. URL: https://en.wikipedia.org/w/index.php?title=SQL&oldid=1338098875.

[13] Thrashing (computer science). *Wikipedia*, April 2026. URL: https://en.wikipedia.org/w/index.php?title=Thrashing_(computer_science)&oldid=1351744482.

[14] A. Apostolico and A. Fraenkel. Robust transmission of unbounded strings using Fibonacci representations. *IEEE Transactions on Information Theory*, 33(2):238–245, March 1987. URL: https://ieeexplore.ieee.org/document/1057284, doi:10.1109/TIT.1987.1057284.

[15] S. Asraf, Y. Gross, S. T. Klein, R. Revivo, and D. Shapira. New compression schemes for natural number sequences. *Discrete Applied Mathematics*, 327:18–27, March 2023. URL: https://www.sciencedirect.com/science/article/pii/S0166218X22004279, doi:10.1016/j.dam.2022.11.004.

[16] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone minimal perfect hashing: Searching a sorted table with O (1) accesses. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 785–794, USA, January 2009. Society for Industrial and Applied Mathematics. URL: https://dl.acm.org/doi/10.5555/1496770.1496856.

[17] Valerio Bioglio, Marco Grangetto, Rossano Gaeta, and Matteo Sereno. On the fly gaussian elimination for lt codes. *IEEE Communications Letters*, 13(12):953–955, 2009. doi:10.1109/LCOMM.2009.12.091824.

[18] Martin Dietzfelbinger and Rasmus Pagh. Succinct data structures for retrieval and approximate membership. In *Lecture Notes in Computer Science*, pages 385–396. Springer, 2008.

[19] Martin Dietzfelbinger and Rasmus Pagh. Succinct Data Structures for Retrieval and Approximate Membership (Extended Abstract). In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halld órsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming*, pages 385–396, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-70575-8_32.

[20] Peter C. Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer. Fast Succinct Retrieval and Approximate Membership using Ribbon, February 2022. URL: void, arXiv:2109.01892, doi:10.48550/arXiv.2109.01892.

[21] Peter C. Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer. Fast Succinct Retrieval and Approximate Membership Using Ribbon . In Christian Schulz and Bora Uçar, editors, *20th International Symposium on Experimental Algorithms (SEA 2022)*, volume 233 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:20, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SEA.2022.4, doi:10.4230/LIPIcs.SEA.2022.4.

[22] Peter C. Dillinger and Stefan Walzer. Ribbon filter: Practically smaller than Bloom and Xor, March 2021. URL: http://arxiv.org/abs/2103.02515, arXiv:2103.02515, doi:10.48550/arXiv.2103.02515.

[23] Ulrich Drepper. What Every Programmer Should Know About Memory.

[24] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, March 1975. URL: https://ieeexplore.ieee.org/document/1055349, doi:10.1109/TIT.1975.1055349.

[25] Stefan Hermann, Hans-Peter Lehmann, Giulio Ermanno Pibiri, Peter Sanders, and Stefan Walzer. PHOBIC: Perfect Hashing with Optimized Bucket Sizes and Interleaved Coding, April 2024. URL: http://arxiv.org/abs/2404.18497, arXiv:2404.18497, doi:10.48550/arXiv.2404.18497.

[26] Yang Hu, William Kuszmaul, Jingxun Liang, Huacheng Yu, Junkai Zhang, and Renfei Zhou. Static Retrieval Revisited: To Optimality and Beyond, October 2025. URL: http://arxiv.org/abs/2510.18237, arXiv:2510.18237, doi:10.48550/arXiv.2510.18237.

[27] Ragnar Groot Koerkamp. PtrHash: Minimal Perfect Hashing at RAM Throughput. *LIPIcs, Volume 338, SEA 2025*, 338:21:1–21:21, 2025. URL: http://arxiv.org/abs/2502.15539, arXiv:2502.15539, doi:10.4230/LIPIcs.SEA.2025.21.

[28] Hans-Peter Lehmann, Thomas Mueller, Rasmus Pagh, Giulio Ermanno Pibiri, Peter Sanders, Sebastiano Vigna, and Stefan Walzer. Modern Minimal Perfect Hashing: A Survey, January 2026. URL: http://arxiv.org/abs/2506.06536, arXiv:2506.06536, doi:10.48550/arXiv.2506.06536.

[29] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. Operating System Concepts.

[30] Ludvig Svedberg and Pernilla Åström. Optimised Retrieval of Properties in a Graph Database with Machine Learning. *LU-CS-EX*, 2025. URL: http://lup.lub.lu.se/student-papers/record/9202024.

[31] Sebastiano Vigna. Quasi-Succinct Indices, June 2012. URL: http://arxiv.org/abs/1206.4300, arXiv:1206.4300, doi:10.48550/arXiv.1206.4300.

[32] Lauren Vizor. Cranfield Libraries: Conducting your literature review: Snowballing and grey literature. URL: https://library.cranfield.ac.uk/literature-review/grey-literature.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2026-06-04

EXAMENSARBETE High-performance, High-Throughput memory-efficient lookup tables for schem
STUDENT Yazan Al-Aswad, Hugo Persson
HANDLEDARE Jonas Skeppstedt (LTH), Lukas Gustavsson (Neo4j)
EXAMINATOR Per Andersson (LTH)

Hur gör man för att snabbt överföra information mellan olika uppslagskatalog

POPULÄRVETENSKAPLIG SAMMANFATTNING **Yazan Al-Aswad, Hugo Persson**

Tänk dig att du ska göra uppslag i en telefonkatalog för att hitta vem du ska ringa. Tänk dig nu att denna telefonkatalog innehåller 100 miljarder människor. Hur ska du kunna hitta personen du söker på ett effektivt sätt?

Digitala databaser är den moderna världens informationshanterare. De uppfyller den roll som bibliotek och arkiv förr hade. Dvs. att hantera och lagra information. Moderna databassystem är ofta gjorda för att hantera extremt stora mängder information, eller så kallad data".

De olika databassystemen skiljer sig oftast åt när det gäller hur de hanterar eller lagrar information. Tänk på skillnaden mellan en historiabok från grundskolan och en perm eller en mapp med händelser. Informationen i sig kan vara densamma för det mesta, men sättet man lagrar informationen på förändrar hur vi kan använda den och kan därmed möjliggöra nya användningssätt.

Neo4j är en specifik databas som specialiserar sig på att kartlägga relationer. När en användare vill sätta upp en Neo4j-databas kan de överföra (importera) sin befintliga data från t.ex. en annan databas. Det kan ta upp mot 2 dagar att importera all data när kunder har en stor mängd befintlig data. Detta ger användaren en dålig första upplevelse med Neo4j och kan göra vissa användningsfall omöjliga. Till exempel kan kunder vilja skapa en temporär Neo4j-databas för att göra en särskild analys av sin data. För att detta ska ge ett värde för användaren måste det gå snabbt att överföra datan till den nya databasen.

En stor anledning till att importeringsprocessen är långsam är att det för närvarande tar mycket tid att översätta ett externt ID till ett internt ID. Detta kan vara en godtycklig text eller en siffra till en siffra.

I vårt arbete undersöker vi om man med hjälp av toppmoderna datastrukturer kan göra översättningen snabbare. Vi använder två datastrukturer, *PtrHash* och *BuRR*, som vi även kombinerar för att skapa fyra olika potentiella implementeringar.

Vi jämför dessa implementationer med varandra samt med den implementation som Neo4j använder idag för att se vilken idöversättningsalgoritm

som kan ge störst förbättring för Neo4j. Vi jämför med 1 miljard nycklar och 10 miljarder nycklar för att se hur vår implementation presterar vid olika datamängder.

Vårt arbete visar att för den mindre datamängden är vår arkitektur `Direct_PTR` runt $6\times$ snabbare än den nuvarande arkitekturen. För den större datamängden är den nuvarande implementationen för långsam för att jämföra med. Vi ser att arkitekturen `Hybrid_BuRR_PTR` var $11\,\%$ snabbare än Direct_PTR.

Detta ger Neo4j en bra grund för att integrera våra algoritmer i deras överföringsprocess och ge en bättre upplevelse för deras användare. Det ger också forskningen en intressant jämförelse mellan PtrHash och BuRR samt hur dessa algoritmer presterar i en Java-kodbas. Samt att vi introducerar en helt ny arkitektur som visar en $11\,\%$ förbättring, vilket är signifikant.