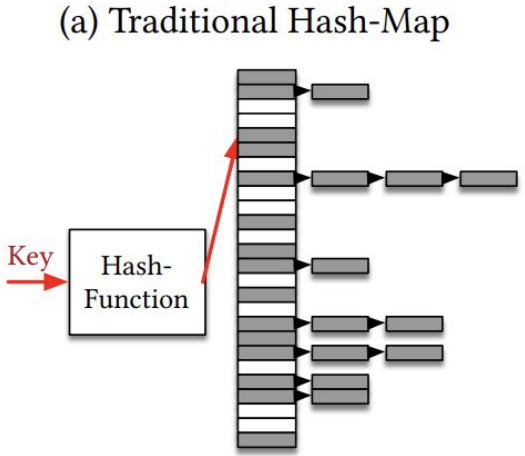
Can Learned Models Replace Hash Functions?

Presenter: Olga Solodova,Yulai Zhao

April 26th, 2024

Motivation + Summary

- Hash functions are used to store and retrieve data from a hash table; data ‘key’ is hashed and the value of the hash determines the position of the data in the table
  - Numerous applications and usages across computer science, particularly important for database management
- This work studies if using learned models instead of traditional hash functions can reduce collisions and whether this translates to improved performance, particularly for indexing and joins. They show that learned models reduce collisions in some cases, which depend on how the data is distributed.



Traditional Hash Functions

Traditional hash functions $h(x) : X \mapsto U$ attempt to map arbitrary inputs to independent and identically distributed (i.i.d.) uniform random outputs.

- True randomness is not feasible in practice
- Quality typically measured by number of collisions that occur (this affects the efficiency of insertion/queries/etc)
- Important that hashing operation is efficient

Examples of traditional hash functions:

- Multiplicative Hashing (MultiplyPrime)
- Fibonacci Hashing (FibonacciPrime)
- Murmur Hashing (Murmur)
- XXHash
- AquaHash

$$h(x) = \left\lfloor \frac{M}{2^w} \cdot (Ax \mod 2^w) \right\rfloor$$

$$h(x) = \left\lfloor M \cdot \left(\left(\frac{A}{2^w} x \right) \mod 1 \right) \right\rfloor$$

Learned models as hash functions

Idea: use a model F to approximate the CDF of the data (i.e. the keys) and use h(Key) = F(Key) * M as the hash function, where M is the size of the hash map

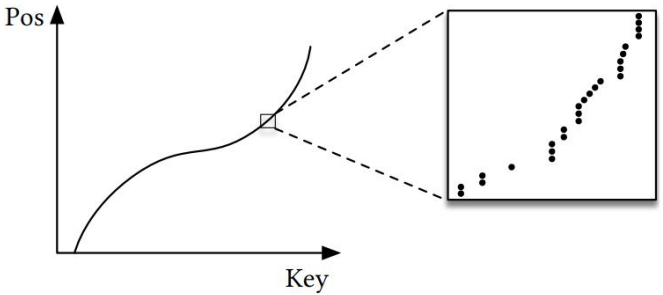


Figure 2: Indexes as CDFs

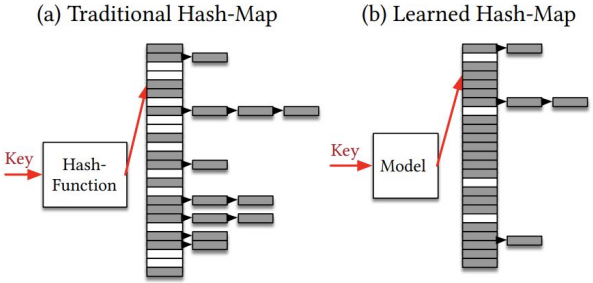


Figure 7: Traditional Hash-map vs Learned Hash-map

- If the model F perfectly learned the empirical CDF of the keys, no conflicts would exist.
- Model should be small and efficient to execute

Learned models as hash functions

Recursive model index (RMI)
(Kraska et. al. 2018)

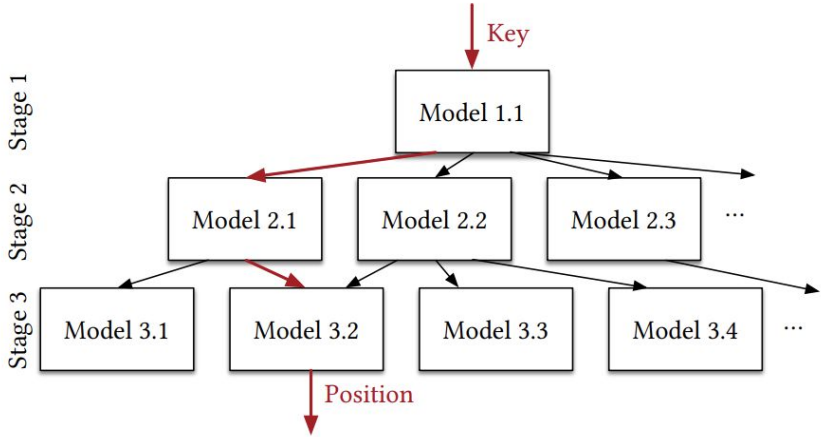


Figure 3: Staged models

Objective:

$$L_\ell = \sum_{(x,y)} (f_\ell^{(\lfloor M_\ell f_{\ell-1}(x)/N \rfloor)}(x) - y)^2$$

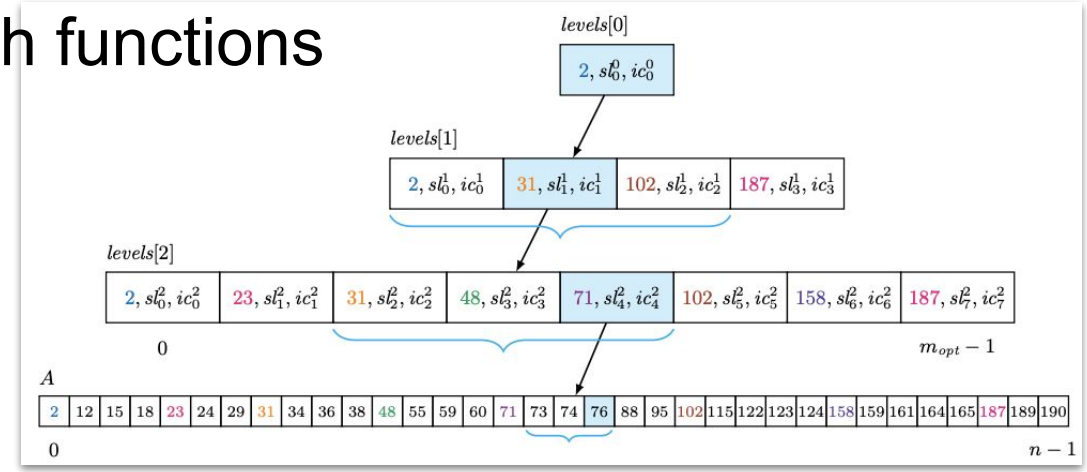
$$L_0 = \sum_{(x,y)} (f_0(x) - y)^2$$

- 2 layers of recursion used
- Linear models perform best

Learned models as hash functions

Piecewise Geometric Model index (PGM-index) (Vinciguerra et. al. 2019) and **RadixSpline** (Kipf et. al. 2020):

- Provide error-bounded approximations to the CDF
  - PGM-index does this with recursive piecewise linear regression
  - RadixSpline does this via its spline-building algorithm
- Initialized with one pass through the data
  - PGM index work per element is logarithmic in number of layers
  - RadixSpline work per element is constant



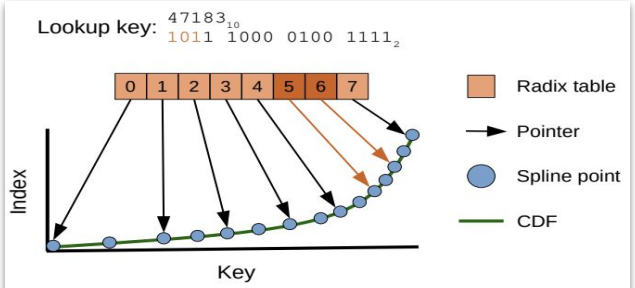


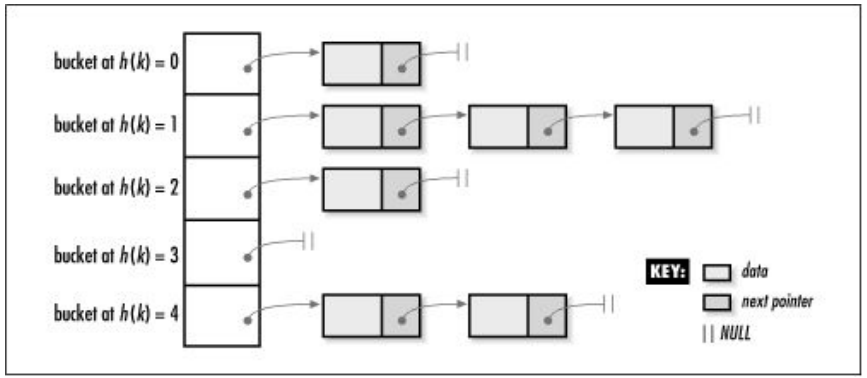
Figure 1: A radix spline index and example lookup process. The r (here, 3) most significant bits b of the lookup key are used as an index into the radix table. Then, a binary search is performed on the spline points between the bth pointer and the $b+1$th pointer.

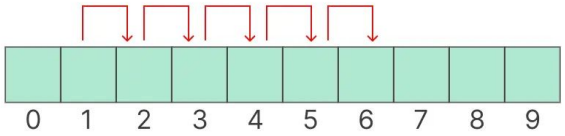
Perfect Hashing

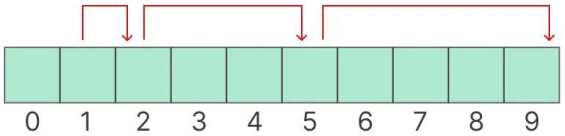
- Perfect hash functions: function is injective, mapping set of inputs into a range [0,N] (equivalently, the hash function achieves no collisions)
  - Minimal perfect: perfect + bijective (i.e. hash table has minimal size)
  - Order-preserving: order in the keys is preserved in the hashed values
- Building a MPHF requires knowing the dataset a-priori
- MPHF require O(n) time to build and are not easily updated, often requiring a full rebuild upon insertion
- Examples:
  - RecSplit (MPHF)
  - AWHC (OMPHF)

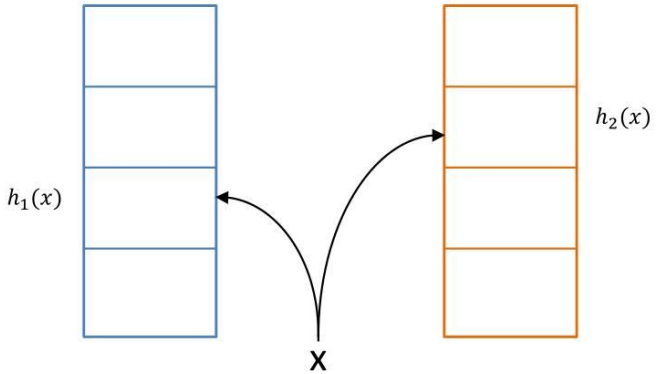
Hashing Schemes: Resolving Collisions

- Bucket chaining
- Open addressing
  - Linear probing
  - Cuckoo hashing









Collisions analysis for Hashing

- Notation: for the task of mapping N keys to N locations
- Assume that we apply a hash function f on the keys, where f could be a traditional hash or a LMH function. Let $x\_0, x\_1, \ldots, x\_N-1$ be the sorted array
  - Let $y\_0, y\_1, \ldots, y\_N-1$ be the sorted array of the hashing outputs $f(x\_0), f(x\_1), \ldots, f(x\_N-1)$.
  - The gaps of the sorted output: g_0, g_1, g_2,... are defined as the difference between consecutive {y_i}

Collisions analysis for Hashing: Main lemma

- Main lemma states that: the expected number of colliding keys is expressed with PDF and CDF of the gap distribution.
  - LMH: more uniform gaps lead to fewer collisions.
  - Traditional Hash Functions: when distributing keys uniformly, are expected to create gaps following an exponential pattern, influencing collision rates.
- LMH Efficiency: Number of submodels in an LMH like RMI impacts accuracy but not necessarily collision rates.

Experiment evaluation

- Main goal: to validate what are the main workload characteristics, scenarios, and operations where employing LMH functions would improve performance?
- Comparisons conducted across 4 aspects
  - Collisions and computation time tradeoffs
  - Usability of basic operations (e.g. lookup and insertion)
  - Other systematic metrics, e.g. construction time
  - High-level operations (range queries and non-partitioned hash join)

Experiment evaluation: Setup

- **Datasets**
  - 4 Real dataset: Facebook user IDs (fb), Wikipedia edit timestamps (wiki), Open Street Map cell IDs (osm), and Amazon book popularity keys (book) from the SOSD benchmark, each with 200 million keys.
  - 4 Synthetic dataset: Sequential keys with regular intervals and deletions (gap_10), uniformly random keys (uniform), and keys from normal and lognormal distributions.
- **Metrics**: computation throughput (operations per second) and collisions (proportion of colliding keys).
- **Methods**
  - Traditional Hash: Multiplicative Hashing (MultiplyPrime), Fibonacci Hashing (FibonacciPrime), Murmur Hashing (Murmur), XXHash, AquaHash
  - Learned Models: RMI, RadixSpline, PGM, perfect hashing: MWHC, RecSplit

Experiment evaluation: Collisions and computation time tradeoffs

- Focus: the balance between hash function efficiency and quality!

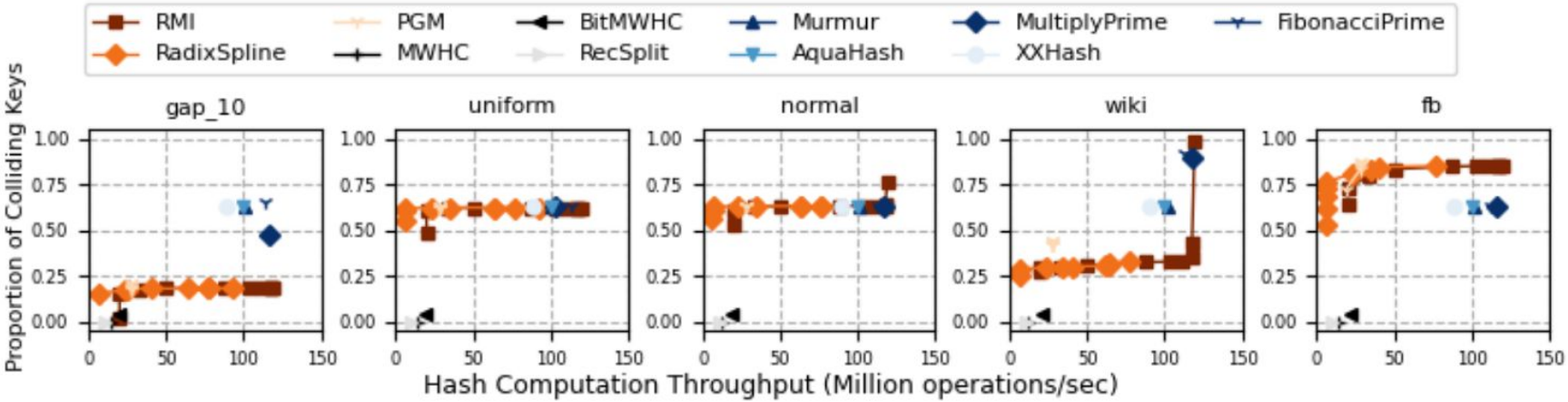


Figure 3: Computation throughput and collisions tradeoffs for various hash functions and using different datasets.

Experiment evaluation: Hash Table Performance - Probing

- A main hash table operations: probing.

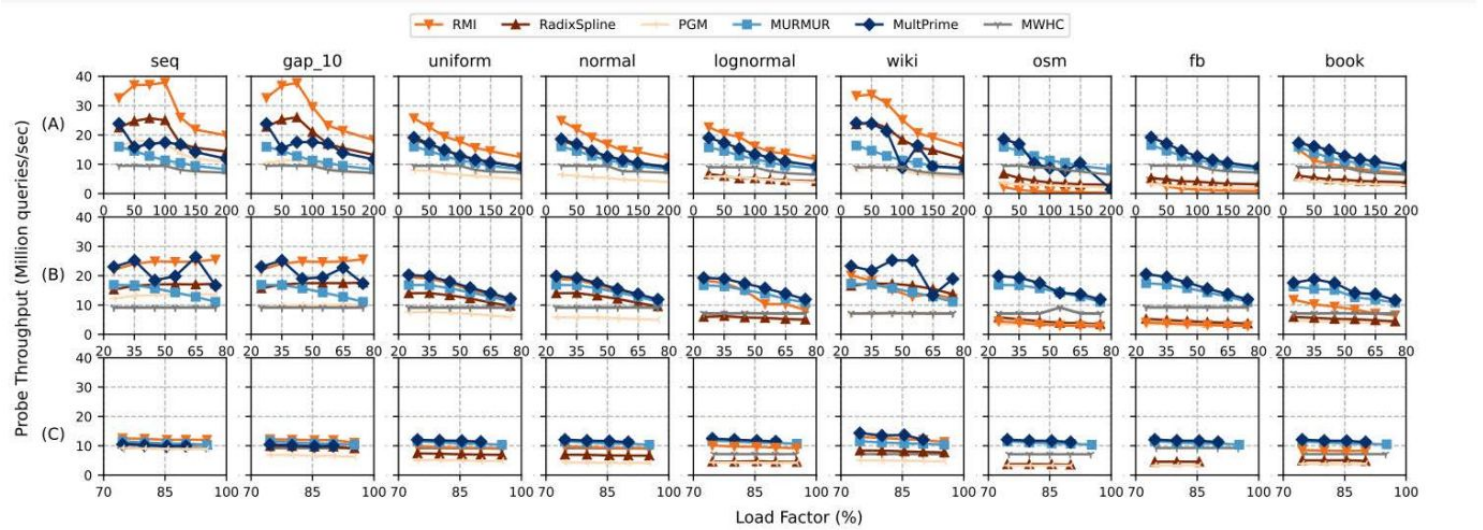


Figure 4: Probe throughput for combinations of 6 hash functions and 3 hashing schemes: (A) bucket chaining, (B) linear probing, and (C) cuckoo hashing. Results are shown for 7 different datasets, and various load factors for each hashing scheme.

Experiment evaluation: Hash Table Performance - Insertion

- Another main hash table operation: insertion.

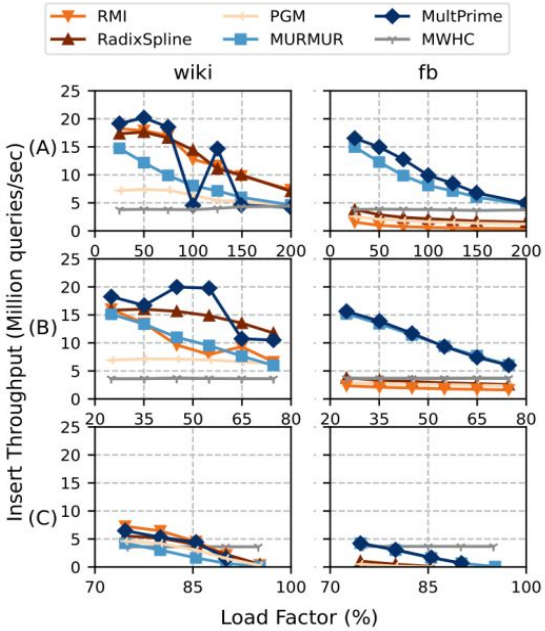


Figure 5: Insert throughput for the same hash functions and schemes used in Figure 4, yet for *wiki* and *fb* datasets only.

Experiment evaluation: High-level operation - Range queries

- How to implement ”range queries" using Hash tables?
  - “monotonic” LMH functions like RMI and RadixSpline can be combined with bucket chaining.

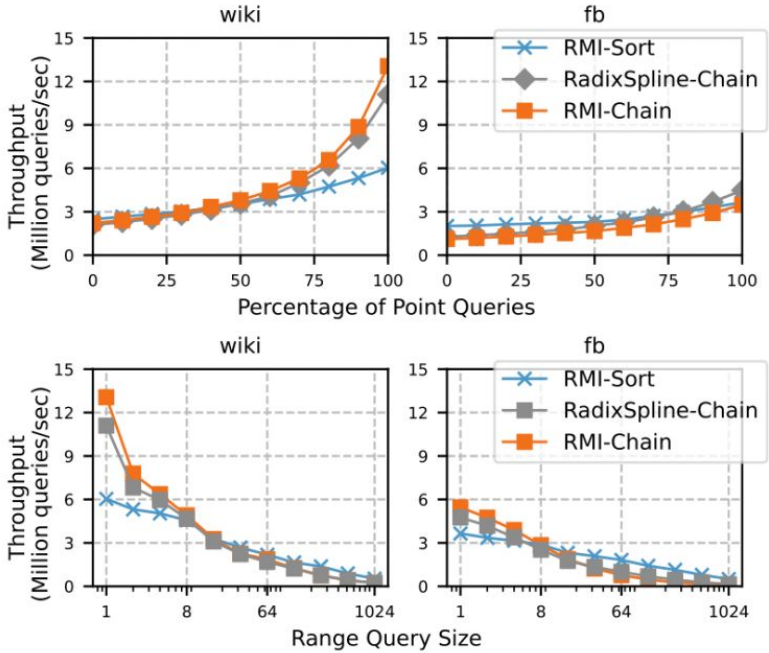


Figure 10: Effect of both point queries percentage (first row), and range query size (second row) on the queries throughput.

Experiment evaluation: High-level operation - Hash-based Join

- Non-Partitioned Hash Join (NPJ) is a method used in database management systems to join two tables.

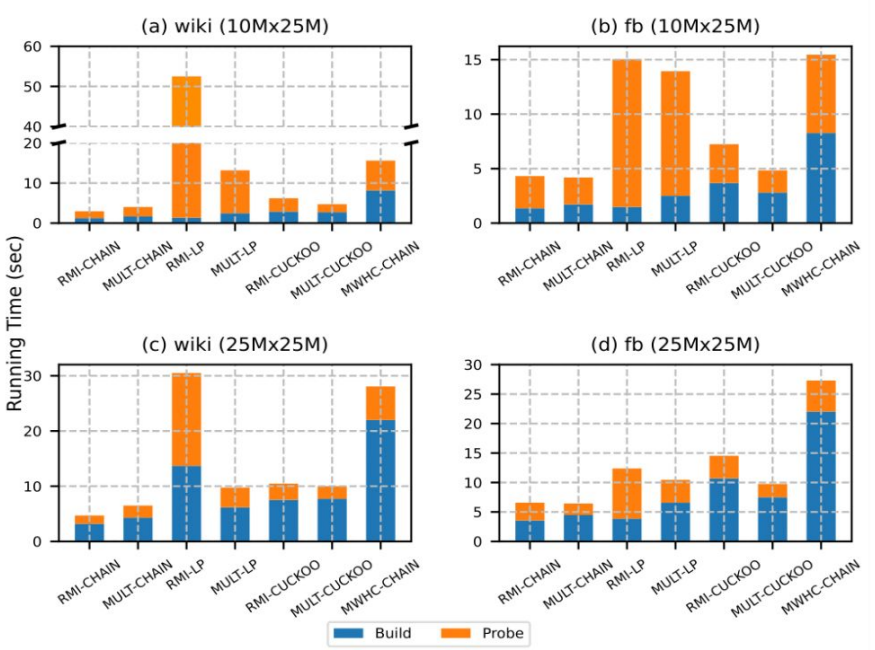


Figure 11: Runtime breakdown for the different implementations of non-partitioned hash join using various hash tables.

Conclusion

- Gap Distribution: Evenly spaced gaps in sorted input keys result in fewer collisions for LMH functions.
- LMH Performance: RMI generally offers the best tradeoff between throughput and collision rates, while RadixSpline performance drops with skewed datasets.
- Hash Table Throughputs: LMH functions show improved probe and insert throughputs with bucket chaining; this advantage diminishes with cuckoo hashing.
- Mixed Workloads & NPJ: Monotonic LMH functions are effective for mixed workloads and NPJ when combined with bucket chaining, with RMI-CHAIN being the most efficient.

Future Work

- Multi-threaded implementations of LMH, perfect, and traditional hash tables have yet to be explored.

- Investigating complex models like decision trees and neural networks could reveal new efficiency tradeoffs.

References

[1] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In SIGMOD, page 489–504, 2018.

[2] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. RadixSpline: A Single-Pass Learned Index. In Proc. of aiDM@SIGMOD, 2020

[3] Paolo Ferragina and Giorgio Vinciguerra. The PGM-Index: A Fully-Dynamic Compressed Learned Index with Provable Worst-Case Bounds. Proc. VLDB Endow., 13(8):1162–1175, 2020.

[4] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. Recsplit: Minimal perfect hashing via recursive splitting. In 2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX), pages 175–185. SIAM, 2020.

[5] Bohdan S Majewski, Nicholas C Wormald, George Havas, and Zbigniew J Czech. A family of perfect hashing methods. The Computer Journal, 39(6):547–554, 1996.

Thank you!

Discussion Questions

- Given that learned models can reduce collisions in hash functions for certain datasets, what are the potential benefits and challenges of integrating these models into traditional database systems?
- Besides training to approximate the CDF of the data, are there other objectives that could be used to train a learned model-based hash function (LMH)?