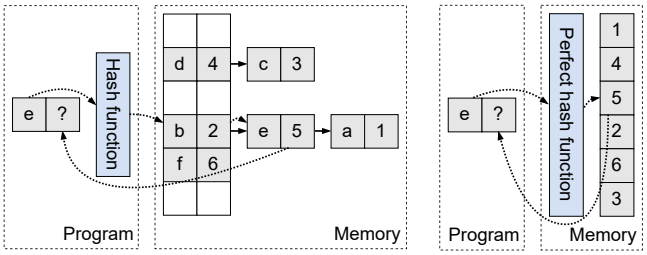
Is Perfect Hashing Practical for OLAP Systems?

Kevin P. Gaffney
University of Wisconsin-Madison
kpgaffney@wisc.edu

Jignesh M. Patel
Carnegie Mellon University
jignesh@cmu.edu

ABSTRACT

A perfect hash function (PHF) maps a set of keys to a range of integers with no collisions. Compared to conventional hash methods, PHFs are attractive for their low space overhead and reduced control flow. Despite their advantages, there has been little investigation into the use of PHFs for online analytical processing (OLAP). This paper is an initial guide to practical perfect hashing for OLAP. We identify several promising applications for PHFs in OLAP and survey their current use in systems and research prototypes. We then evaluate existing PHF approaches and quantify their impact on query performance. Our results are encouraging: in a real OLAP system, PHFs achieve end-to-end speedups of 1.7X and 3.1X for join and aggregate queries, respectively. Nevertheless, there is room for improvement. Future approaches that simultaneously achieve low build time and high probe throughput could offer additional performance increases.



(a) Conventional hashing. (b) Perfect hashing.

Figure 1: Comparing conventional and perfect hashing. A conventional hash function is typically embedded in the program and produces collisions. A perfect hash function has data that resides in memory and avoids collisions.

1 PERFECT HASHING

Hashing is a fundamental operation in online analytical processing (OLAP), providing the foundation for several key data structures and algorithms such as joins, aggregates, partitions, and filters [38]. Typical OLAP queries spend a substantial portion of their time on hashing-related tasks [27]. Given its prevalence and impact, hashing remains a focus of ongoing research efforts aimed at improving OLAP efficiency (*e.g.*, [1–3, 7, 18, 23, 36, 40–42]).

A hash function is a function that maps a set S of n keys to positions in the range $[0, m)$. A collision occurs when multiple keys are mapped to the same position. While good hash functions return each position with roughly equal probability, collisions are nearly inevitable, even when $m \gg n$ as illustrated by the birthday problem [10]. Conventional hash functions use strategies such as probing, chaining, or cuckoo hashing to resolve collisions. Probing searches for neighboring unoccupied positions. Chaining stores multiple keys in the same position. Cuckoo hashing uses multiple hash functions to find alternative positions for colliding keys.

A perfect hash function (PHF) is a hash function that maps each key in a given set to a unique position; *i.e.*, there are no collisions. In specific cases, PHFs may be constructed through simple means. If the keys are (representable as) integers and m is larger than the maximum key, then a trivial PHF is the identity function. However, this approach may waste considerable space if the keys are strings or sparse integers. Alternatively, if n is small, then it may be possible to find a PHF with brute force, iteratively trying candidate hash functions until one is found that produces no collisions. However, more sophisticated algorithms are necessary to construct PHFs over arbitrary sets of keys.

In exchange for avoiding collisions, a PHF requires linear space with respect to the number of keys in the set. As illustrated in Figure 1, a conventional hash function can be represented in constant space, and its evaluation is often inlined into the calling code. In contrast, a PHF has data that resides in memory and requires one or more memory accesses to evaluate. However, PHFs are surprisingly space-efficient. The lower bound for a minimal PHF (MPHF), for which $m = n$, is $log_2 e \approx 1.44$ bits per key [17, 26]. Existing approaches achieve MPHFs within a small factor of the lower bound, often just 2-4 bits per key. The space lower bound for non-minimal PHFs decreases as $\frac{m}{n}$ increases.

1.1 Approaches

In general, existing PHF algorithms can be categorized into the following four approaches [15, 33].

1.1.1 Hash and displace. In the hash and displace approach, keys are first mapped to buckets B_i, in which collisions are expected. Then, buckets are processed in order of decreasing occupancy. For each bucket B_i, a displacement value d_i is found such that keys in the bucket are mapped to positions without collisions. For example, given some conventional hash function h, the algorithm may find some d_i such that $(h(x) + d_i) \mod m$ is a collision-free mapping for all $x \in B_i$. Finally, the displacement values are stored in a compact form that preserves constant-time lookup. Intuitively, this approach processes the heaviest buckets first, taking advantage of the higher number of unoccupied positions early on. Towards the end of the construction, when most positions are occupied, the remaining buckets contain very few keys (perhaps just one). There are several variants of this approach that differ mainly in whether keys are mapped uniformly or non-uniformly into buckets, how displacement values are used to map keys to positions, and what

method is used to encode the displacement values [6, 16, 30, 32, 33, 43].

1.1.2 Random hypergraphs. Algorithms based on random hypergraphs evaluate r independent hash functions for each key, each of which maps keys to positions in the range $[0, m)$. This process generates an r-uniform random hypergraph with m vertices and n edges. The algorithm then assigns values to vertices such that, for each edge, some function of the values at the edge's vertices gives a collision-free mapping to positions the range $[0, m)$. For example, the algorithm may assign values w_i such that the PHF is given by

$$f(x) = \left(w_{h_1(x)} + w_{h_2(x)} + ... + w_{h_r(x)}\right) \mod m,\ x \in S.$$

Due to the bounds on acyclicity of random hypergraphs, if the ratio m/n is above a certain threshold for a given r (minimum of 1.23 for $r = 3$), with high probability the values can be assigned using linear-time hypergraph peeling. The algorithm may include an additional ranking step to produce an MPHF. The random hypergraph approach was originally proposed in [25] and subsequently improved in later work [5, 11, 12, 19, 20].

1.1.3 Fingerprinting. In the fingerprinting approach, keys are first mapped to a bitmap A_0 of length γn_0, where $\gamma \geq 1$ is a parameter of the algorithm and $n_0 = n$. The positions in the bitmap to which exactly one key was mapped are set to 1; all other positions are set to 0. If $n_1 > 0$ collisions are produced, then the same process is repeated with the n_1 colliding keys using another bitmap A_1 of size γn_1. The process repeats until there are no colliding keys. To evaluate the MPHF for a given key, the key is mapped to a position in each successive bitmap until a 1 bit is found. The MPHF then returns the number of 1 bits that precede the current position in the bitmaps using a constant-time ranking structure [22]. The fingerprinting approach was introduced in [28] and further optimized in [24].

1.1.4 Recursive splitting. The recursive splitting approach exploits the observation that for very small sets of keys, it is possible to find an MPHF through brute force. Given parameters b and $\ell \geq 1$, the algorithm divides keys into buckets of average size b. Then, each bucket is recursively split until a bucket of size at most ℓ is obtained, for which it is feasible to directly search for an MPHF. The parameters b and ℓ provide different space and time tradeoffs. The recursive splitting approach was introduced in [15] under the name RecSplit.

1.2 Advantages over conventional methods

PHFs are appealing for their ability to produce extremely space-efficient data structures. As noted earlier, the space lower bound for an MPHF is about 1.44 bits per key, and existing approaches can achieve 2-4 bits per key in practice. Furthermore, if the PHF is allowed to return arbitrary positions for keys that are not in the set, PHF-based hash tables need not store their keys at all. We refer to such hash tables as *keyless*. Observe that conventional hash tables store their keys to handle two cases: to resolve collisions and to return *not found* for keys not in S. PHFs eliminate the first case, and if the application can avoid the second, it has no need for the keys. As a result, PHFs may drastically reduce the size of hash tables whose keys are large relative to their values. For example, consider a conventional hash table that maps 1 billion 8-byte keys to 1-byte values. At a load factor of 0.9, this hash table would be 10 GB in size, excluding any space used to mark occupied positions. In contrast, a keyless hash table using an MPHF with 4 bits per key would be 1.5 GB in size (including the 1-byte values). For memory-based databases, smaller hash tables are more likely to reside in upper levels of the processor cache hierarchy. For disk-based databases, smaller hash tables may avoid expensive disk-based join and aggregate algorithms.

Ostensibly, PHFs are also attractive for their computational efficiency. After a PHF returns a position for a given key, we can immediately access the value at that position. Compared to conventional hash tables, this eliminates the control flow responsible for (1) checking if the position is occupied, and if so, (2) checking if the key stored at the position is equal to the requested key, and if not, (3) looping over alternative positions or entries in the chain as determined by the collision resolution strategy. The elimination of control flow is particularly important for today's OLAP systems that use vectorized execution or runtime code generation. Indeed, there are several recent efforts focused on making the structure and logic of hash tables more amenable to SIMD vectorization [4, 34].

Despite their apparent advantages, PHFs must be carefully examined. Many existing PHF approaches involve multiple memory accesses and incur significant computational overhead, so these perceived benefits may not be realized. However, recent efforts have produced increasingly performance-focused PHF algorithms [24, 32, 33]. These advances motivated us to ask the following questions, which guide the remainder of this paper.

(§2) What roles could PHFs play in accelerating OLAP queries?
(§3) How are PHFs currently being used in OLAP systems?
(§4) How well do PHFs perform in a real system?
(§5) What future work is needed to make PHFs more practical?

2 APPLICATIONS TO OLAP

While PHFs have notable advantages over conventional methods, they come with additional constraints that appear to restrict their applicability. However, we identify several operations in standard OLAP benchmarks that satisfy these constraints and thus stand to benefit from the use of PHFs.

2.1 Criteria

There are two main criteria that should be considered when deciding whether to use a PHF.

(1) Is the set of keys static? Most existing PHF approaches are designed to be built over a static set of keys, assuming a one-time construction cost followed by repeated evaluation of the PHF. However, it may be possible to extend existing approaches to support limited insertion and deletion. Insertion could be achieved by storing the inserted keys in a small buffer and rebuilding the PHF when the buffer grows too large. Deletion could be achieved by simply marking positions of deleted keys. There is some work on dynamic perfect hashing [13], which partitions the set of keys into smaller subsets and then maintains a PHF for each subset. When a key is inserted or deleted, only the PHF for the appropriate subset is rebuilt. However, these approaches come with the additional space and computational overhead of maintaining a buffer or rebuilding

PHFs. Therefore, we focus on opportunities for static, short-lived PHFs (*e.g.*, for hash joins and hash aggregates).

(2) Are lookups limited to the original set of keys? Typically, a PHF returns an arbitrary position for any key not in the set over which it was built. If the application wishes to take advantage of the space reduction offered by keyless hash tables described in Section 1.2, it must only request keys in the original set. In the context of OLAP, one way to achieve this is to exploit integrity constraints to limit lookups to a certain set of keys. We discuss concrete examples of this approach in the following paragraphs.

2.2 Operations

2.2.1 Hash joins with high selectivity. Join queries commonly include selections that can be pushed down to improve efficiency. On the build side, selection pushdown reduces hash table size and enables lightweight approximate semijoin techniques such as LIP [45]. On the probe side, selection pushdown avoids costly hash table probes for eliminated tuples.

However, joins with no selections do appear in OLAP workloads and are often quite expensive. For example, TPC-H Q9 (Listing 1) involves a join between the two largest tables, `lineitem` and `orders`, with no selections on either [44]. To execute this query, a typical OLAP database system builds and probes a hash table with an entry for every row of the `orders` table mapping `o_orderkey` to the year of `o_orderdate`. This join has been identified as a choke point in prior work [9].

We observe that if referential integrity is enforced (*i.e.*, each `l_orderkey` references a valid row in the `orders` table), then each `l_orderkey` is guaranteed to be contained in the hash table—the hash table will never return *not found*. Therefore, if we build the hash table using a PHF, we need not store the `o_orderkey` values. As the majority of space in the hash table is occupied by the `o_orderkey` values, using a PHF can drastically reduce its size.

A small modification to the above technique enables PHFs to be used in hash joins *with* selections. In the hash table, a *valid* bit is reserved for each position to indicate whether the key at that position is retained by the selection. If a probe encounters an unset valid bit, it returns *not found*. Alternatively, a special value may be reserved to indicate an invalid position. For example, if the hash table contains only values in the range $[1, 9]$, then the value 10 could represent *not found*. However, to ensure correctness, keyless hash tables must contain an entry for every row in the original table, regardless of the fraction retained by the selection. Consequently, their space-reducing potential is greatest for high-selectivity cases.

2.2.2 Hash aggregates with known groups. Hash aggregate is an essential yet expensive operator in OLAP systems that computes aggregate statistics over groups of data. Central to this operator is the aggregate hash table, which often accounts for the majority of execution time in aggregate-focused OLAP pipelines [27].

At times, the operator builds the aggregate hash table on the fly. When a new group is encountered, the operator inserts an entry into the hash table mapping the group key value to the aggregate accumulators. Otherwise, the operator updates the existing aggregate accumulators. This approach is most often used when the groups are not known in advance.

Listing 1: Snippet of TPC-H Q9

```
SELECT EXTRACT(YEAR FROM o_orderdate), ...
FROM lineitem, orders, ...
WHERE l_orderkey = o_orderkey AND ...
```

Listing 2: Snippet of TPC-H Q18

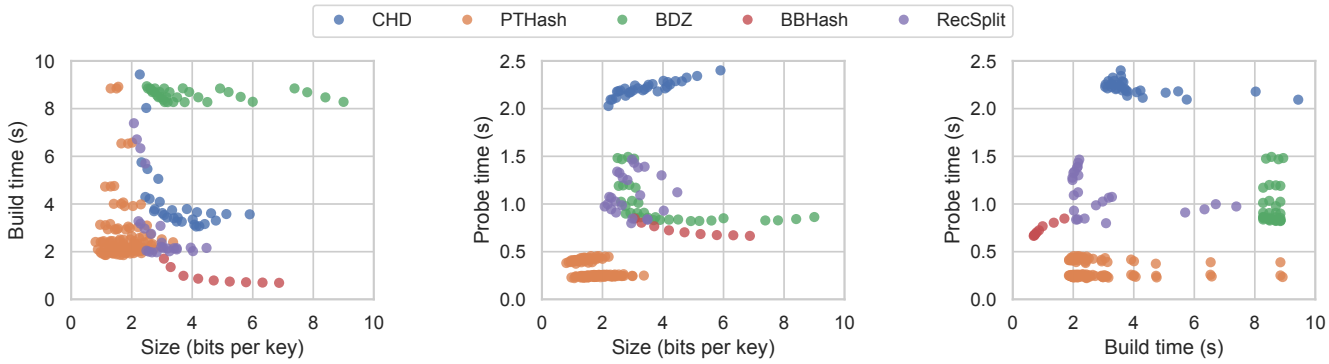
```
SELECT l_orderkey
FROM lineitem
GROUP BY l_orderkey
HAVING SUM(l_quantity) > ?
```

Frequently, however, the groups are known in advance, enabling the operator to pre-allocate the aggregate hash table. Known groups may result from earlier operators in the query plan. For example, SSB Q2.1 requests the sum of `lo_revenue` for each `d_year`, `p_brand1` pair obtained from prior joins [29]. Sets of `d_year` and `p_brand1` values can be recorded during the build phase of each join and then used to compute all possible groups (less than 300, in this case) before the hash aggregate begins. It is also possible to infer the groups by exploiting integrity constraints. For example, TPC-H Q18 (Listing 2) involves a group-by aggregate over the `lineitem` table, where the grouping key is `l_orderkey` [44]. If referential integrity is enforced, then the set of `l_orderkey` values is a subset of `o_orderkey` values. Therefore, one possible strategy is to pre-allocate the hash table with `o_orderkey` values prior to the hash aggregate. In each of these cases, it is possible to build a keyless hash table using a PHF.

3 ADOPTION IN CURRENT SYSTEMS

To our knowledge, there has been little adoption of PHFs in OLAP systems. This is likely due to several factors. The first is their perceived complexity. PHFs are not part of the core algorithm toolbox found in many programming languages or undergraduate textbooks, and their construction and evaluation can be difficult to comprehend. However, in our observations, much of the complexity arises from efforts to reach the space lower bound. Simpler, though slightly less space efficient PHFs can be sketched in a paragraph and implemented in a few hundred lines of code. The second reason is their apparent restrictiveness. Typically, a PHF is built over a static set of keys and returns arbitrary positions for lookups of keys not in the set. Despite these constraints, there are promising applications of PHFs that appear in standard OLAP benchmarks as discussed in Section 2. The third reason is related to the previous two: it is unclear if their performance benefits justify their complexity and applicability. Historically, PHF approaches have prioritized space-efficiency over speed. However, several recent approaches are increasingly focused on performance, trading off a small amount of space for faster construction and lookup time [24, 32, 33]. As we will demonstrate in our results, these approaches can provide substantial speedups for OLAP queries.

In modern OLAP systems [31, 37] and research prototypes [42], a common optimization for hash joins and hash aggregates is to use a specialized hash table when the keys are dense. Typically, the hash table consists of a bitmap indicating occupied positions and an array of payloads; the key values are omitted. Lookups



(a) Build time vs. size. **(b) Probe time vs. size.** **(c) Probe time vs. build time.**

Figure 2: Microbenchmark performance of various PHF algorithms and parameter configurations for 10 million keys.

use the key as an index into the bitmap and payload array. This approach is often more efficient than a conventional hash table because it eliminates the space overhead of storing the keys and the computational overhead of resolving collisions. However, it wastes considerable space for sparse sets of keys. Following prior work [42], we refer to this approach as *array join* and *array aggregate* when applied to join and aggregate, respectively.

Despite the prevalence of array join and array aggregate, we are aware of few examples of general PHFs being used for OLAP. In the prototype query engine Blink [39], MPHFs are used for hash aggregates. However, they must be pre-built during the loading phase rather than built on the fly during query processing. Consequently, they can only be used for a fixed set of group keys and must be rebuilt when the data changes. PHFs were also applied to enable SIMD acceleration of hash aggregates in [35]. However, PHFs were constructed with brute force and hence limited to small sets of keys. Beyond about 250 keys, the implementation switches to conventional hashing.

4 EXPERIMENTAL RESULTS

We now present an experimental evaluation of several state-of-the-art PHF algorithms and their impact on OLAP performance. Our evaluation focuses on the five algorithms shown in Table 1. We selected these algorithms because they have been shown to outperform many others in one or more of the following categories: build time, probe time, and space overhead. We use the source code provided by the original authors.

The details of our experimental setup are as follows. Experiments were run on a dedicated machine with a 2.2 GHz Intel Xeon Silver 4114 processor. The processor has 10 physical cores, each with its own 32 KiB L1 cache and 1 MiB L2 cache, with a 13.75 MiB L3 cache and 100 GB of DDR4 SDRAM shared among cores. Each physical core has two logical cores for a total of 20 logical cores. The machine was provisioned with CloudLab [14]. Input data and results were stored in memory. In Section 4.1, all experiments are single-threaded. In Section 4.2, the build phase is single-threaded and the probe phase is multi-threaded with all 20 cores. Hardware performance counter values were obtained with Intel VTune Profiler [21].

Name	Category	References
CHD	Hash and displace	[6]
PTHash	Hash and displace	[32, 33]
BDZ	Random hypergraphs	[11]
BBHash	Fingerprinting	[24]
RecSplit	Recursive splitting	[15]

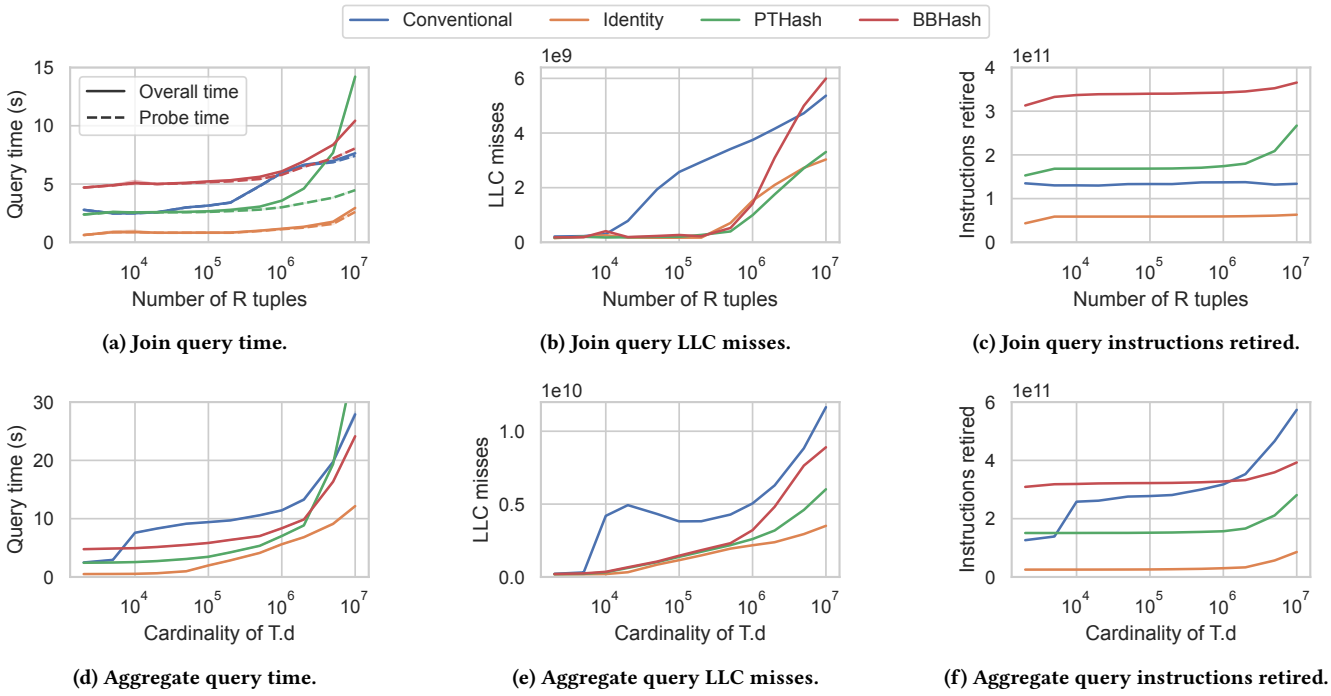
Table 1: Evaluated PHF implementations.

4.1 Microbenchmarks

To explore the tradeoffs of various PHF algorithms and their parameters, we first conducted microbenchmarks. We generated a set of 10 million 64-bit uniform random integers. For each algorithm, we measured the time taken to build a PHF over the integers (build time) and the time taken to return the position of all integers (probe time). We swept the parameters of each algorithm across the range of values recommended in their publications and software documentation.

Results are shown in Figure 2. We first observe that the choice of parameter values has a substantial effect on PHF size and performance. Our chosen parameter combinations produced several PHFs with size between 0.8 and 9 bits per key, build time between 0.7 and 10 seconds, and probe time between 0.2 and 2.4 seconds. For some algorithms, build time appears to increase as size decreases. PHF algorithms must often perform more work to achieve lower space overhead. Probe time also increases as size decreases for algorithms such as BBHash, which incurs additional memory accesses and computation when fewer bits are allocated. However, for others, probe time appears to decrease as size decreases, likely due to greater cache efficiency.

For certain parameter values, PTHash and BBHash outperform the other three algorithms in both build time and probe time. However, of the two algorithms, BBHash achieves a faster build time while PTHash achieves a faster probe time. PTHash also achieves the best space efficiency, requiring 1 bit per key in its fastest configuration.



(a) Join query time.

(b) Join query LLC misses.

(c) Join query instructions retired.

(d) Aggregate query time.

(e) Aggregate query LLC misses.

(f) Aggregate query instructions retired.

Figure 3: End-to-end query performance of PHF algorithms in DuckDB.

4.2 End-to-end queries

We now examine the impact of PHFs on end-to-end query performance. For this part of our evaluation, we selected PTHash and BBHash out of the five algorithms above for their outstanding build and probe performance. We use the default parameters recommended by the authors (for PTHash, $c = 4.5$ and $\alpha = 0.98$; for BBHash, $\gamma = 2$). We integrated these algorithms into the execution engine of DuckDB [37], a state-of-the-art open-source OLAP database system. We then created a database with the schema described in Listing 3 and executed the queries that follow. Throughout this section, the term "Conventional" refers to DuckDB's general hash join/aggregate algorithm. "Identity" refers to DuckDB's unmodified array join/aggregate algorithm discussed in Section 3. "BBHash" and "PTHash" refer to a modified array join/aggregate algorithm in which the respective PHF is used to index into the array rather than the identity function.

4.2.1 Join query. To evaluate the impact of PHFs on high-selectivity joins, we populated column R.a with unique integers in the range $[0, |R|)$ and column R.b with uniform random integers in the range $[0, 10)$, where $|R|$ is the number of tuples in R. We populated S.c with uniform random integers in the range $[0, |R|)$. We varied $|R|$ from 2 thousand to 10 million tuples and fixed $|S|$ at 1 billion tuples. We then executed the query shown in Listing 4 five times for each $|R|$. For this query, DuckDB builds a hash table on R using one of the four algorithms described above, then probes the hash table with the tuples in S, computing the sum.

Results are shown in Figure 3 (a-c). We first observe that for all join algorithms, execution time increases as the number of build tuples increases. We note that there are substantial performance differences among algorithms across build sizes. The Identity algorithm outperforms the others in all cases. However, unlike the other three algorithms, Identity is infeasible when the range of join key values is large. PTHash outperforms or matches Conventional when the build input has fewer than 5 million tuples. The most significant difference between PTHash and Conventional is at 1 million tuples, where PTHash is about 1.7X faster than Conventional. Beyond 5 million tuples, the cost of constructing the PHF dominates the overall query time. However, PTHash considerably outperforms Conventional for probing alone (the dotted lines in Figure 3a), especially when the build input is large. Between build input sizes of 200 thousand to 2 million, Conventional exhibits a

Listing 3: Database schema

```
CREATE TABLE R (a UBIGINT, b UTINYINT);
CREATE TABLE S (c UBIGINT);
CREATE TABLE T (d UBIGINT, e UTINYINT);
```

Listing 4: Join query

```
SELECT SUM(R.b)
FROM R, S
WHERE R.a = S.c;
```

Listing 5: Aggregate query

```
SELECT d, SUM(e)
FROM T
GROUP BY d;
```

sharp increase in probe time, corresponding to the spilling of the hash table to DRAM from the last level cache (LLC) as shown in Figure 3b. The other three algorithms do not exhibit this increase until 1 million tuples due to their better space efficiency. Due to its costly probe time and computational overhead (Figure 3c), BBHash does not significantly outperform Conventional in overall query time.

4.2.2 Aggregate query. To evaluate the impact of PHFs on hash aggregates, we populated `T.d` with uniform random integers in the range $[0, C)$ and column `T.e` with uniform random integers in the range $[0, 10)$. We fixed the size of `T` at 1 billion tuples. We varied the cardinality C of `T.d` from 2 thousand to 10 million. We then executed the query in Listing 5 five times for each C.

The result for this experiment are shown in Figure 3 (d-f). Similar to the join query, while Identity outperforms all other algorithms across cardinalities, it is limited by the constraints discussed earlier. For C up to 5 million, PTHash outperforms Conventional. At greater cardinalities, PHF construction dominates overall query time. Between C of 5 and 10 thousand, Conventional exhibits a sharp spike due to an increase in LLC misses (Figure 3e), after which it is outperformed by BBHash as well. The maximum speedup of PTHash over Conventional is 3.1X at C = 20 thousand. The maximum speedup of BBHash over Conventional is 1.7X at C = 50 thousand.

5 CONCLUSION

PHFs have notable advantages over conventional methods and several promising applications in OLAP. In a real OLAP system, recent PHF algorithms can speed up joins and aggregates by up to 1.7X and 3.1X, respectively. However, there is substantial room for improvement. Further gains in performance may result from new approaches that achieve outstanding probe throughput while remaining lightweight enough to be built on the fly. In particular, the algorithms we examined tended to favor either build or probe performance at the expense of the other. For example, the lightweight fingerprinting approach of BBHash builds PHFs efficiently, but probing requires several memory accesses. In contrast, PTHash achieves better probe performance but requires more work to build PHFs.

Although this paper focuses on joins and aggregates, there are likely several more applications for perfect hashing in data management systems that we have not discussed. For example, PHFs could be used as the building blocks for approximate membership query data structures such as Bloom filters [8]. In addition, k-perfect hash functions, generalizations of PHFs that map at most k keys to the same position, may be useful for partitioning. Exploring these avenues is part of future work.

REFERENCES

[1] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 362–373. https://doi.org/10.1109/ICDE.2013.6544839

[2] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That Is the Join Question in a Real System. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 168–180. https://doi.org/10.1145/3448016.3452831

[3] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. 2014. Memory-Efficient Hash Joins. In *Proceedings of the VLDB Endowment*, Vol. 8. 353–364. https://doi.org/10.14778/2735496.2735499

[4] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2326–2339. https://doi.org/10.1145/3514221.3526054

[5] Djamal Belazzougui, Paolo Boldi, Giuseppe Ottaviano, Rossano Venturini, and Sebastiano Vigna. 2014. Cache-Oblivious Peeling of Random Hypergraphs. In *2014 Data Compression Conference*. 352–361. https://doi.org/10.1109/DCC.2014.48

[6] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. 2009. Hash, Displace, and Compress. In *Algorithms - ESA 2009 (Lecture Notes in Computer Science)*, Amos Fiat and Peter Sanders (Eds.). Springer, Berlin, Heidelberg, 682–693. https://doi.org/10.1007/978-3-642-04128-0_61

[7] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/1989323.1989328

[8] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (July 1970), 422–426. https://doi.org/10.1145/362686.362692

[9] Peter Boncz, Thomas Neumann, and Orri Erling. 2014. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Performance Characterization and Benchmarking (Lecture Notes in Computer Science)*, Raghunath Nambiar and Meikel Poess (Eds.). Springer International Publishing, Cham, 61–76. https://doi.org/10.1007/978-3-319-04936-6_5

[10] Mario Cortina Borja and John Haigh. 2007. The Birthday Problem. *Significance* 4, 3 (Sept. 2007), 124–127. https://doi.org/10.1111/j.1740-9713.2007.00246.x

[11] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. 2007. Simple and Space-Efficient Minimal Perfect Hash Functions. In *Algorithms and Data Structures (Lecture Notes in Computer Science)*, Frank Dehne, Jörg-Rüdiger Sack, and Norbert Zeh (Eds.). Springer, Berlin, Heidelberg, 139–150. https://doi.org/10.1007/978-3-540-73951-7_13

[12] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. 2013. Practical Perfect Hashing in Nearly Optimal Space. *Information Systems* 38, 1 (March 2013), 108–131. https://doi.org/10.1016/j.is.2012.06.002

[13] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. 1994. Dynamic Perfect Hashing: Upper and Lower Bounds. *SIAM J. Comput.* 23, 4 (Aug. 1994), 738–761. https://doi.org/10.1137/S0097539791194094

[14] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of Cloudlab. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19)*. USENIX Association, USA, 1–14.

[15] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. 2019. RecSplit: Minimal Perfect Hashing via Recursive Splitting. In *2020 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX) (Proceedings)*. Society for Industrial and Applied Mathematics, 175–185. https://doi.org/10.1137/1.9781611976007.14

[16] Edward A. Fox, Qi Fan Chen, and Lenwood S. Heath. 1992. A Faster Algorithm for Constructing Minimal Perfect Hash Functions. In *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '92)*. Association for Computing Machinery, New York, NY, USA, 266–273. https://doi.org/10.1145/133160.133209

[17] Michael L. Fredman, János Komlós, and Endre Szemerédi. 1984. Storing a Sparse Table with 0(1) Worst Case Access Time. *J. ACM* 31, 3 (June 1984), 538–544. https://doi.org/10.1145/828.1884

[18] Kevin P. Gaffney, Martin Prammer, Larry Brasfield, D. Richard Hipp, Dan Kennedy, and Jignesh M. Patel. 2022. SQLite: Past, Present, and Future. In *Proceedings of the VLDB Endowment*, Vol. 15. 3535–3547. https://doi.org/10.14778/3554821.3554842

[19] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. 2016. Fast Scalable Construction of (Minimal Perfect Hash) Functions. In *Experimental Algorithms (Lecture Notes in Computer Science)*, Andrew V. Goldberg and Alexander S. Kulikov (Eds.). Springer International Publishing, Cham, 339–352. https://doi.org/10.1007/978-3-319-38851-9_23

[20] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. 2020. Fast Scalable Construction of ([Compressed] Static | Minimal Perfect Hash) Functions. *Information and Computation* 273 (Aug. 2020), 104517. https://doi.org/10.1016/j.ic.2020.104517

[21] Intel. 2023. Intel VTune Profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

[22] G. Jacobson. 1989. Space-Efficient Static Trees and Graphs. In *30th Annual Symposium on Foundations of Computer Science*. 549–554. https://doi.org/10.1109/SFCS.1989.63533

[23] Aarati Kakaraparthy, Jignesh M. Patel, Brian P. Kroth, and Kwanghyun Park. 2022. VIP Hashing: Adapting to Skew in Popularity of Data on the Fly. In *Proceedings of the VLDB Endowment*, Vol. 15. 1978–1990. https://doi.org/10.14778/3547305.3547306

[24] Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. 2017. Fast and Scalable Minimal Perfect Hashing for Massive Key Sets. https://doi.org/10.48550/arXiv.1702.03154 arXiv:1702.03154 [cs]

[25] B. S. Majewski, N. C. Wormald, G. Havas, and Z. J. Czech. 1996. A Family of Perfect Hashing Methods. *Comput. J.* 39, 6 (Jan. 1996), 547–554. https://doi.org/10.1093/comjnl/39.6.547

[26] Kurt Mehlhorn. 1982. On the Program Size of Perfect and Universal Hash Functions. In *23rd Annual Symposium on Foundations of Computer Science (Sfcs 1982)*. 170–175. https://doi.org/10.1109/SFCS.1982.80

[27] Hannes Mühleisen and Mark Raasveldt. 2022. Parallel Grouped Aggregation in DuckDB. https://duckdb.org/2022/03/07/aggregate-hashtable.html

[28] Ingo Müller, Peter Sanders, Robert Schulze, and Wei Zhou. 2014. Retrieval and Perfect Hashing Using Fingerprinting. In *Experimental Algorithms (Lecture Notes in Computer Science)*, Joachim Gudmundsson and Jyrki Katajainen (Eds.). Springer International Publishing, Cham, 138–149. https://doi.org/10.1007/978-3-319-07959-2_12

[29] Pat O'Neil, Betty O'Neil, and Xuedong Chen. 2009. Star Schema Benchmark. https://www.cs.umb.edu/~poneil/StarSchemaB.PDF

[30] Rasmus Pagh. 1999. Hash and Displace: Efficient Evaluation of Minimal Perfect Hash Functions. In *Algorithms and Data Structures (Lecture Notes in Computer Science)*, Frank Dehne, Jörg-Rüdiger Sack, Arvind Gupta, and Roberto Tamassia (Eds.). Springer, Berlin, Heidelberg, 49–54. https://doi.org/10.1007/3-540-48447-7_5

[31] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: A Data Platform Based on the Scaling-up Approach. In *Proceedings of the VLDB Endowment*, Vol. 11. 663–676. https://doi.org/10.14778/3184470.3184471

[32] Giulio Ermanno Pibiri and Roberto Trani. 2021. Parallel and External-Memory Construction of Minimal Perfect Hash Functions with PTHash. https://doi.org/10.48550/arXiv.2106.02350 arXiv:2106.02350 [cs]

[33] Giulio Ermanno Pibiri and Roberto Trani. 2021. PTHash: Revisiting FCH Minimal Perfect Hashing. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '21)*. Association for Computing Machinery, New York, NY, USA, 1339–1348. https://doi.org/10.1145/3404835.3462849

[34] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1493–1508. https://doi.org/10.1145/2723372.2747645

[35] Orestis Polychroniou and Kenneth A. Ross. 2013. High Throughput Heavy Hitter Aggregation for Modern SIMD Processors. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware (DaMoN '13)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/2485278.2485284

[36] Orestis Polychroniou and Kenneth A. Ross. 2014. A Comprehensive Study of Main-Memory Partitioning and Its Application to Large-Scale Comparison- and Radix-Sort. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 755–766. https://doi.org/10.1145/2588555.2610522

[37] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212

[38] Raghu Ramakrishnan and Johannes Gehrke. 2003. *Database Management Systems* (3rd ed ed.). McGraw-Hill, Boston.

[39] Vijayshankar Raman, Garret Swart, Lin Qiao, Frederick Reiss, Vijay Dialani, Donald Kossmann, Inderpal Narang, and Richard Sidle. 2008. Constant-Time Query Processing. In *2008 IEEE 24th International Conference on Data Engineering*. 60–69. https://doi.org/10.1109/ICDE.2008.4497414

[40] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A Seven-Dimensional Analysis of Hashing Methods and Its Implications on Query Processing. In *Proceedings of the VLDB Endowment*, Vol. 9. 96–107. https://doi.org/10.14778/2850583.2850585

[41] Ibrahim Sabek, Kapil Vaidya, Dominik Horn, Andreas Kipf, Michael Mitzenmacher, and Tim Kraska. 2022. Can Learned Models Replace Hash Functions?. In *Proceedings of the VLDB Endowment*, Vol. 16. 532–545. https://doi.org/10.14778/3570690.3570702

[42] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1961–1976. https://doi.org/10.1145/2882903.2882917

[43] Robert Endre Tarjan and Andrew Chi-Chih Yao. 1979. Storing a Sparse Table. *Commun. ACM* 22, 11 (Nov. 1979), 606–611. https://doi.org/10.1145/359168.359175

[44] Transaction Processing Performance Council (TPC). 2022. TPC Benchmark H. https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v3.0.1.pdf

[45] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. 2017. Looking Ahead Makes Query Plans Robust: Making the Initial Case with in-Memory Star Schema Data Warehouse Workloads. In *Proceedings of the VLDB Endowment*, Vol. 10. 889–900. https://doi.org/10.14778/3090163.3090167