applied sciences

MDPI

Article

SAHA: A String Adaptive Hash Table for Analytical Databases

Tianqi Zheng [1,2,*], Zhibin Zhang [1] and Xueqi Cheng [1,2]

[1] CAS Key Laboratory of Network Data Science and Technology, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China; zhangzhibin@ict.ac.cn (Z.Z.); cxq@ict.ac.cn (X.C.)
[2] University of Chinese Academy of Sciences, Beijing 100049, China
* Correspondence: zhengtianqi@ict.ac.cn

Received: 03 February 2020; Accepted: 09 March 2020; Published: 11 March 2020

Abstract: Hash tables are the fundamental data structure for analytical database workloads, such as aggregation, joining, set filtering and records deduplication. The performance aspects of hash tables differ drastically with respect to what kind of data are being processed or how many inserts, lookups and deletes are constructed. In this paper, we address some common use cases of hash tables: aggregating and joining over arbitrary string data. We designed a new hash table, `SAHA`, which is tightly integrated with modern analytical databases and optimized for string data with the following advantages: (1) it inlines short strings and saves hash values for long strings only; (2) it uses special memory loading techniques to do quick dispatching and hashing computations; and (3) it utilizes vectorized processing to batch hashing operations. Our evaluation results reveal that `SAHA` outperforms state-of-the-art hash tables by one to five times in analytical workloads, including Google's SwissTable and Facebook's F14Table. It has been merged into the `ClickHouse` database and shows promising results in production.

Keywords: hash table; analytical database; string data

1. Introduction

1.1. Background

We are now in the big data era. The last decade was characterized by an explosion in data-analyzing applications, managing data of various types and structures. The exponential growth of datasets imposes a big challenge on data analytic platforms, among which, one typical platform is the relational database, which is commonly used to store and extract information that can be monetized or contains other business value. An important purpose of a database system is to answer decision support queries via the SQL [1] language. For example, a typical query might be to retrieve all data values corresponding to all customers having their total account balances above required limits. It mainly utilizes two SQL operators: the `join` operator to associate the corresponding values and the `group` operator to summarize the account balances. These two operators appear in almost every decision support query. For instance, among the 99 queries of the most well-known decision support benchmark: `TPC-DS` [2], 80 queries contain `group` operators and 97 queries have `join` operators. Efficient implementations of these two operators will greatly reduce the decision support query's runtime and benefit interactive data exploration. There are many different implementations of `group` and `join` operators. In order to understand how analytical databases handle these operators and what optimizations can be done, we will first describe the well-known implementation of famous analytical databases, such as ClickHouse [3], Impala [4] and Hive [5], and then discuss the hot spots of these operators.

1.1.1. Group Operator

Figure 1 shows a typical implementation of an analytical database's `group` operator. (One possible query statement would be to count the number of each `col` and output `<col, count>` pairs: `SELECT col, count(*) FROM table GROUP BY col`.) It contains two processing phases: phase (1) is to build a hash table using the data from the data source. Every record in the hash table is associated with a counter. If the record is newly inserted, the related counter is set to 1; otherwise, the counter gets incremented. Phase (2) is to assemble the records in the hash table into a format that can be used for further query processing. One important thing note is that the aggregator reads a batch of rows from the scanner and inserts them into the hash table one by one. This is called vectorized processing, and it helps improve cache utilization, devirtualize functions and potentially get compiled into faster SIMD (Single Instruction, Multiple Data) [6] implementations because of smaller loop bodies. (Vectorized processing is to operate on a set of values at one time instead of repeatedly operating on a single value at a time.) It is commonly used in analytical databases.

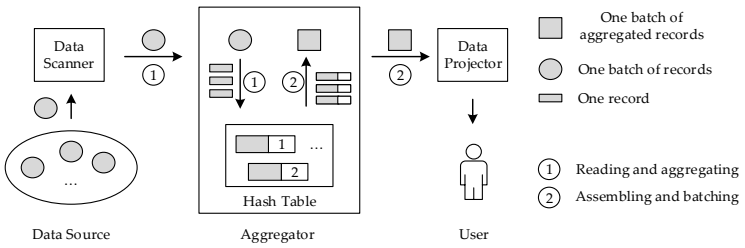


Figure 1. An typical implementation of the `group` operator of vectorized databases.

1.1.2. Join Operator

A typical `join` implemention of an analytical database is shown in Figure 2. (The query statement might be to join two tables, A and B, using the `key_col` and output both columns from each table as `<A.left_col, B.right_col>`: `SELECT A.left_col, B.right_col FROM A JOIN B USING (key_col)`.) It also has two phases: phase (1) is to build a hash table using the data from the right hand side table of the join statement, and phase (2) is to read from the left hand side table and probe the hash table just built in a streamlined way. The building phase is similar to the previous `group` implementation, but with each table slot storing a reference to the right columns. Both phases are vectorized too.

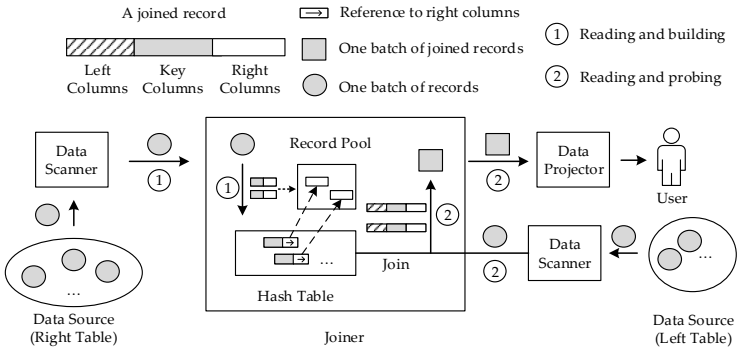


Figure 2. A typical implementation of the `join` operator of vectorized databases.

1.1.3. Hash Table

The core data structure for the `group` and `join` operators is the **Hash Table**. A hash table is a data structure that associates keys with values. It supports querying given keys in constant time by choosing an appropriate hash function. The basic idea is to pick a hash function that maps arbitrary keys to numeric hashes, and these hashes serve as indices to slots used to store values in the hash table.

However, two different keys might end up having the same hash, and thus the same index. If any key collides in the hash table with an already existing key of the same index, a probing function is used to resolve the collision.

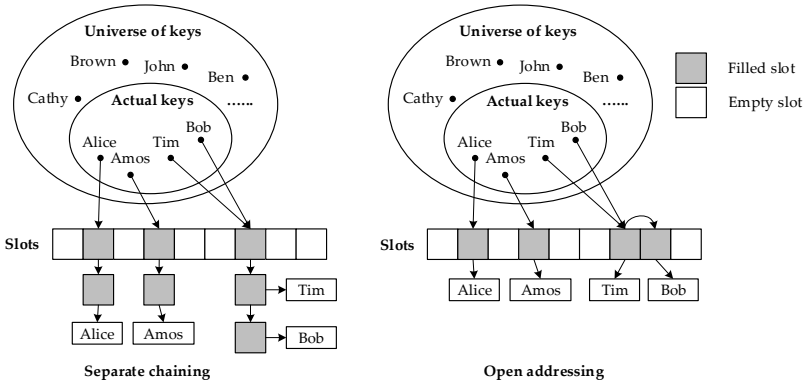


Figure 3. Two main implementations of hash tables.

There are two main ways of resolving collisions, namely, open addressing and separate chaining, as shown in Figure 3. In separate chaining, elements are stored outside of the hash table with linked lists. Items that collide are chained together in separate linked lists. To find a given record, one needs to compute the key's hash value, get back the corresponding linked list and search through it. It is usually inefficient compared to open addressing because of additional cache misses and branch conditions. However, it is easy to implement, and has other properties, such as reference stability (references and pointers to the keys and values in the hash table must remain valid until the corresponding key is removed). Thus, it is still widely used and is the default implementation of C++'s unordered hash tables.

In open addressing, all elements are stored in the hash table. In order to find a given record, table slots are examined systematically during search until the desired element is found. This process is called probing through the conflicting chain, and there are many different probing schemes, such as linear probing, quadratic probing and double hashing. These schemes are intuitive and are named according to the probing process. There are also more sophisticated strategies, such as RobinHood [7] and Hopscotch [8], which contains complex element adjustments. When hash tables become full, many conflicting chains are quite long and probing can become expensive. To remain efficient, rehashing is used by scattering the elements into a larger array. The indicator of doing rehashing is called the `load factor`, which is the number of records stored in the hash table divided by the capacity. Open addressing hash tables have better cache locality, less pointer indirections and are well suited for vectorized processing. Thus, for analytical workloads, open addressing schemes should be used to achieve superior performance. To save words, unless otherwise specified, all mentioned hash tables use the open addressing scheme in this paper.

1.2. Problems

Implementing efficient hash tables for analytical databases is nontrivial, due to the huge design space they belong to and their critical performance impacts on a query's runtime. Richter et al. [9] gives an in-depth analysis of how to implement hash tables for integer keys. They proposed more than 10 conditions to check in order to find the best hash table implementation. However, there is little information in an analytical database to fulfill those checks, and processing string keys is more difficult than integers because strings have variable lengths. Meanwhile, string data in the real world is commonly used as a key column in many queries, such as URL domains, nicknames, keywords, RDF (Resource Description Framework) attributes and IDs, to name a few. These datasets share a common property: the average length is short. There are also some string datasets, such as package delivery addresses—web search phrases that have long average string length. Nevertheless, neither

distribution gets optimized in current analytical databases, and there is no easy way to tell beforehand how the actual length distribution of a string dataset looks in every subset to choose a suitable hash table implementation. In summary, we discuss the following three problems to process analytical workloads on string datasets:

1. When processing string datasets, different optimizations should be applied based on the distribution of string length. There is no single hash table that can fit all cases of string data.
2. Short strings are cost-ineffective when stored with pointers, and it is a waste of space to store saved hash values for short strings.
3. Hashing long strings while doing insertion or lookup in hash tables is cache unfriendly.

1.3. Our Solutions and Contributions

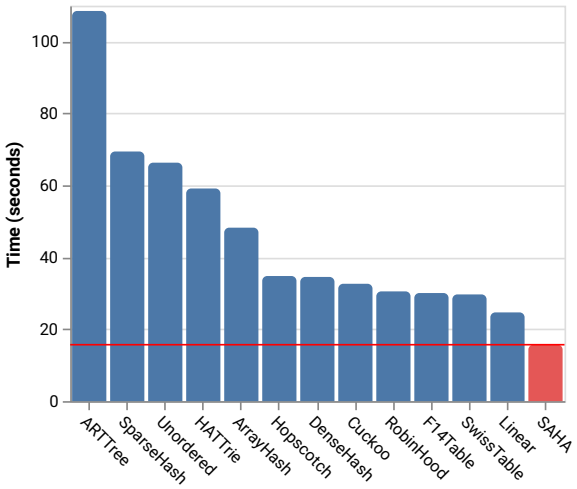


Figure 4. A bar graph of time consumption of various hash tables to process `group` and `join` benchmarks compared to `SAHA`.

To address these problems, we present `SAHA`, a string adaptive hash table for analytical databases. It reinterprets short string keys as an array of integers and stores them directly inside slots without any indirection. For long string keys, it saves the computed hash value along with the string data pointer in each slot to lower the frequency of long string comparisons. `SAHA` precomputes the hash values in each vectorized processing step of `group` and `join` operators, and uses these hash values directly in the following operations. This reduces the cache pollution when probing empty slots during hash table insertion and lookup, especially for long strings. `SAHA` also provides a highly optimized key dispatcher based on string length. It can adapt to different string datasets and choose the best suited hash table implementation dynamically. The main contributions of our work are summarized as follows:

1. **Optimized String Dispatcher.** We implement a string key dispatcher based on the length of a string. It is used to categorize string datasets into subsets with predefined string length distributions. The dispatcher is optimized with special memory loading techniques so that it does not introduce performance regressions to the underlying hash table.
2. **Hybrid String Hash Tables.** We design five implementations of hash tables for strings with different lengths. Each implementation is optimized for one distribution of string length. All implementations are united as one hash table with the standard hash table interface but can adapt to string data. It has the benefit of low peak memory usage because the sub hash tables grow independently, which is smoother than growing as a whole.
3. **Pre-hashing and Vectorized Programming Interface.** We identify the chance of precomputing the hash values of records before doing vectorized processing in the hash table. This optimizes the cache utilization when the current batch is filled with long strings. We propose a new

programming interface accepting callbacks to insertion and lookup functions so that additional information such as the precomputed hash values can be calculated inside the hash table.

4. **Extensive Experiments.** We carry out extensive experiments using both real-world and synthetic datasets to validate the performance of SAHA. Our experiments look into the key performance factors of a hash table, including the hashing functions, the computational time and the effectiveness of main memory utilization. The results reveal that SAHA outperforms the state-of-the-art hash tables in all cases by one to five times, as shown in Figure 4.

The rest of this paper is organized as follows. Section 2 discusses the related works. Section 3 describes the main constructs of SAHA with its optimizations. Section 4 shows the vectorized specific optimization of SAHA and its supporting programming interface. Section 5 contains experimental results. Finally, Section 6 gives the concluding remarks.

2. Related Works

2.1. Hash Tables

Hash tables are a well established and extensively researched data structure that still continues to be a hot topic of research. We have briefly mentioned implementations that are well known in textbooks in Section 1. Here, we discuss three other implementations actively used in the real world: (1) the RobinHood table [7] is a derived linear probing hash table that constrains the length of each slot's conflicting chain by shifting the current elements out of the slot if the probing key is too far away from its slot; (2) Hopscotch [8] is similar, but instead of shifting elements out, it swaps the probing key towards its slot based on the metadata in each slot; (3) Cuckoo hash [10] uses two hash functions, providing two possible locations in the hash table for each key. As a result, it guarantees a low number of memory accesses per lookup. The conflicting chain becomes a graph-like structure with interesting mathematical characteristics, which attracts many researchers [11–13]. All these works are focused on general datasets and are orthogonal to ours. SAHA can use any of these hash tables as the backend with small adaptations.

Some of the hash tables focus on memory saving by supporting high load factors [14,15]. These compact hash tables use a bitmap for occupancy check and a dense array to store the elements. However, the actual amount of memory saved is insignificant when processing string data compared to string encoding techniques. With additional bitmaps to maintain, both insertion and lookup operations become slower. SAHA optimizes memory usage by inlining short strings. It achieves similar memory consumption but is more efficient than these compact hash tables.

In industry, the state-of-the-art hash tables used widely are Google's SwissTable [16] and Facebook's F14Table [17]. Both are designed for general usage and optimized for modern hardware. The technique is to group elements into small chunks and pack each chunk as a small hash table with additional metadata for probing. Each of these chunks can be operated more efficiently compared to operating over the entire hash table. The main difference between SwissTable and F14Table is that in SwissTable, the chunk's metadata is stored separately, while in F14, the metadata and the elements of a chunk are stored together. Though they achieve good performance in common lookup and insert operations, they are slightly slower than carefully crafted linear probing hash tables, such as the one used in ClickHouse [3], which is the fastest aggregation implementation we can find. It uses specialized linear hash tables which are trimmed for `group` and `join` operations and has optimized implementation of rehashing. It is 20% faster than both SwissTable and F14Table in `group` workloads. However, it is not optimized when dealing with string data. SAHA combines the string optimizations with this heavily tuned hash table, which makes it competent for analysis workloads on string datasets.

2.2. Data Structures and Optimizations for Strings

One of the famous string associative containers is a tree-like data structure called `trie` [18]. It provides similar functionalities to hash tables, with additional operations such as string prefix searching.

`Tries` have been well studied and many efficient implementations have been proposed [19–22]. However, it is not appropriate for analytical workloads, such as string aggregations. As pointed out in [23], they observed that ART [22] is almost five times slower when running some simple aggregation query compared to an open addressing hash table. We also noticed similar results in Section 5. This is mainly because any trie structure contains pointers in each node and traversing through the tree takes too many indirections compared to a flat array implementation of an open addressing hash table.

Some hash table implementations [21,24] try to compact the string keys of a conflicting chain into a continuous chunk of memory. This reduces memory consumption and lowers the frequency of pointer indirections, as strings get inlined. As a result, doing lookups in these hash tables is more efficient. However, insertion is much slower due to the additional memory movements required, especially for long strings. `SAHA` only inlines short strings and directly stores them in the slot instead of a separate conflicting chain. Thus, it has the same advantage but without any impact on insertions.

Many analytical databases employ string encoding as a compression scheme [3,25–27]. It maintains a string dictionary for each string column and stores the output codes as integer columns instead. These integer columns then can be used directly for `group` and `join` operations. However, it only works if the string column has low cardinality; otherwise maintaining the dictionary becomes too expensive. `SAHA` works well for any cardinality and can also be used together with string encoding.

2.3. Summary

We select twelve hash tables and `tries` in Table 1 to give an overview of the main data structures that can be used for `group` and `join` on string datasets. Short descriptions and string related features are listed when compared to `SAHA`. We also evaluate all these data structures in detailed benchmarks in Section 5.

1. **Inline string**. Whether the hash table has stored the string data directly without holding string pointers.
2. **Saved hash**. Whether the computed hash value of a string is stored together with the string in the hash table.
3. **Compact**. Whether the hash table has high load factors (we consider `tries` to have 100% load factor).

Table 1. Hash tables.

Name	Inline String	Saved Hash	Compact	Description
SAHA	Hybrid	Hybrid	No	Linear probing after dispatching by length
Linear [3]	No	Yes	No	Linear probing
RobinHood [7]	No	Yes	No	Linear probing within limited distance
Hopscotch [8]	No	Yes	No	Neighborhood probing
Cuckoo [10]	No	Yes	No	Alternatively probing between two tables
Unordered	No	No	No	Chaining (GNU stdlibc++'s implementation)
SwissTable [16]	No	Partial	No	Quadratic probing with separated metadata
F14Table [17]	No	Partial	No	Quadratic probing with inlined metadata
DenseHash [14]	No	No	No	Quadratic probing
SparseHash [14]	No	No	Yes	Quadratic probing over sparse table
HATTrie [21]	Yes	No	Yes	Trie tree with ArrayHash as leaf nodes
ArrayHash [24]	Yes	No	Yes	Chaining with compacted array
ARTTree [22]	Yes	No	Yes	Trie tree with hybrid nodes

3. Constructs

The main constructs of `SAHA` are the two sets of sub hash tables: `Hash Table Sn` and `Hash Table L`, along with a string key dispatcher. Figure 5 shows the architecture of `SAHA`. All the sub tables are implemented as linear probing hash tables, except `Hash Table S0`, which is an array lookup table. An array lookup table interprets keys as array indices directly; as a result, it can do element lookup within one or two CPU instructions while not storing the key data at all. Array lookup tables are commonly

used when the universal data's cardinality is small—small integers, for instance—whereas string data are not feasible because they have infinite cardinality. However, since we limit the string length to two bytes, there can only be 65,536 different keys for `Hash Table S0`; as a result, we can apply the array lookup technique even for string data.

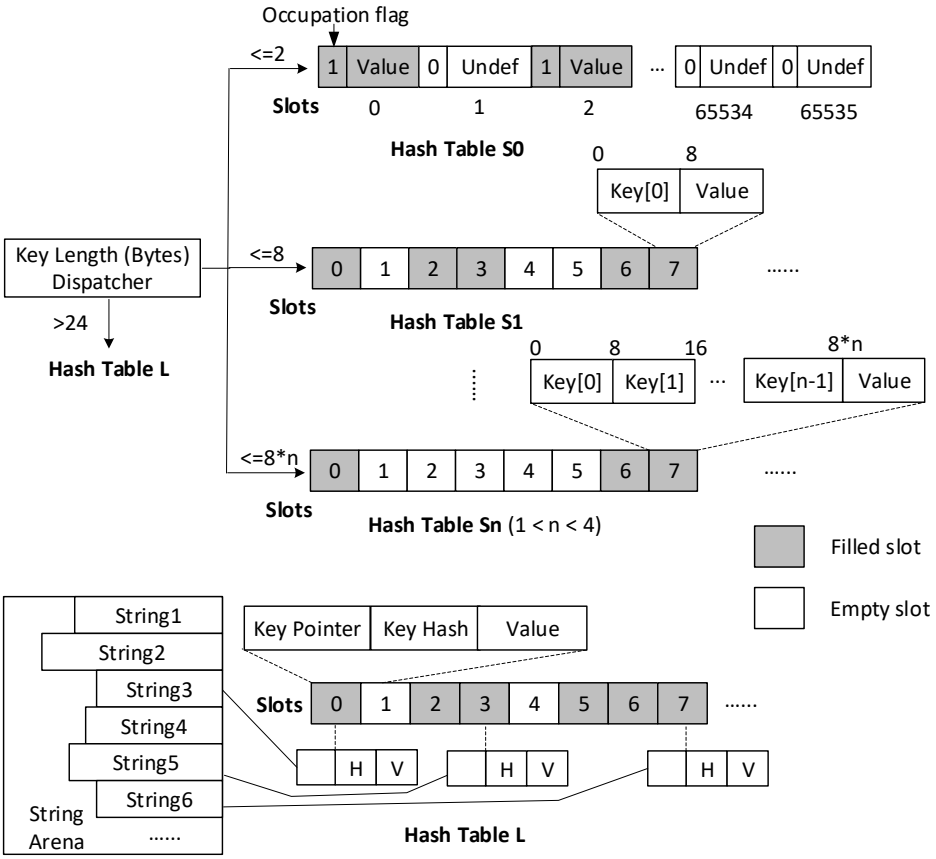


Figure 5. Architecture of `SAHA` with multiple hash tables to store strings with different length ranges.

For string keys that have length longer than two bytes but shorter than 25 bytes, we store them in multiple implementations of `Hash Table S`. The main difference between these tables is the size of their slots. We reinterpret string keys as an array of integers. For string keys with lengths in $(2, 8]$ bytes, we use one integer with eight bytes; for lengths in $(8, 16]$, we use two; and for lengths $(16, 24]$, we use three. These integers are stored in the slots directly without string pointers. The reason for using the linear probing hash table is because it achieves promising performance results in analytical workloads, and its implementation has smaller code size, which is important for `SAHA`. Since we are dispatching over multiple different hash table implementations, if a complex implementation is used, the code size would be bloated and will have a noticeable impact on performance. To verify this choice, we compare the linear probing hash table against an extensive number of other implementations in Section 5.

For string keys longer than 24 bytes, we store the string pointers directly along with their computed hash values in `Hash Table L`. Since long strings are expensive to move around, holding pointers instead of inlining string keys makes insertion and rehashing faster. The reason for choosing 24 bytes as the threshold is because (1) we observe a good amount of string datasets which have 80% of strings shorter than 24 bytes and (2) using more integers to represent strings would offset the performance benefit due to excessive data copy introduced. It is also important to save the computed hash values for long strings, since hash recomputation is expensive and storing additional hash values has a negligible memory overhead compared to storing long strings. The saved hash can also be used to check whether a given slot contains the current key while doing linear probing. Otherwise, we

have to always compare two long strings to check if they are equal, even when their hash values are different. `Hash Table L` also contains a memory pool called `String Arena` to keep the strings inserted. It is necessary to have complete ownership of the inserted data inside the hash table, because when processing each row in `group` or `join` workloads, the current record will be destroyed after insertion.

The string key dispatcher accepts a string key and dispatches it to the corresponding hash table's routine. Because it sits in the critical part of code, the dispatcher needs careful implementation and optimization to avoid introducing performance regressions. In order to dispatch short strings, it requires loading the string data from the given pointer into current stack, and converting it into a number array. To achieve efficient memory copying, a constant number of bytes should be used instead of doing `memcpy(dst, src, len)` directly. We use eight bytes as a minimal memory copy unit to read one integer at a time from the string data into the array. If the string length is not divisible by eight, we load the last bytes starting from the end position minus eight. Thus, we still copy eight bytes, but shifting out the `H` part, as shown in Figure 6. However, if the string is shorter than eight bytes, copying eight bytes might lead to segmentation fault, which is demonstrated as `underflow` and `overflow` in Figure 6. (A segmentation fault is raised by hardware with memory protection, notifying an operating system the software has attempted to access a restricted area of memory.) To resolve this issue, we do a memory address check for the string pointer when dispatching to `Hash Table S1`. Since memory page size is of 4096 bytes, if `pointer & 2048 = 0`, it means the string data is at the top half of a memory page and it can only be underflow. We can then safely copy eight bytes from 0 and get rid of the `T` part. Otherwise, it can only be overflow and our previous algorithm will work correctly.

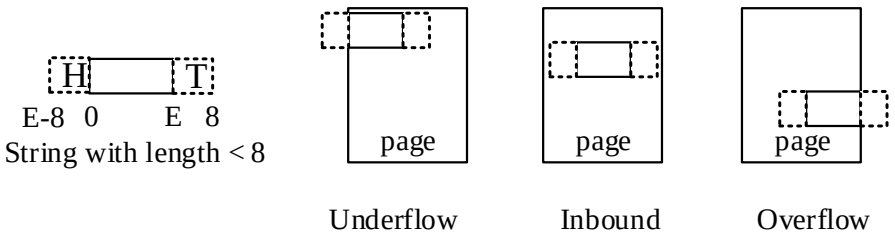


Figure 6. Three different cases of the memory layout of a string key with length shorter than eight bytes.

Listing 1 shows the implementation of the string key dispatcher in pseudo C++. It accepts a string key with pointer and size, and dispatches the key to sub hash tables. Different hash tables require different key types but the same string data; as a result, a union structure is used. The length of the `T` part in each string key can be calculated using `(-key.size & 7) * 8` and is used for fast memory loading. There are two main routines for a hash table when used for `group` and `join` workloads: `emplace` and `find`. The `emplace` routine is for inserting data into the hash table if the given key is not yet found inside, or else returning the data in the hash table with the key equal to the one being inserted. The `find` routine is slightly different from `emplace` in that it does not insert data when the given key is missing in the hash table, but returns a key-not-found indicator to the routine's caller. Here we demonstrate dispatching over `emplace`, and other routines are dispatched similarly. The implementation needs to make sure that all the method bodies of the sub hash tables get inlined into the dispatching code. Otherwise the dispatching process would impact the performance due to introducing additional function calls. In order to achieve this effect, the actual implementation requires careful tuning. One possible implementation can be referred to in [28].

In addition to these constructs, `SAHA` also provides the following optimizations to further reduce the runtime:

- **Slot empty check for Hash Table Sn**. As the string length is not stored in `Hash Table Sn`, one way to check if a given slot is empty is to verify that all the integers in the slot's key array are equal to zero. However, since the zero byte in UTF8 is code point 0, `NUL` character. There is no

other Unicode code point that will be encoded in UTF8 with a zero byte anywhere within it. We only need to check the first integer, that is `key[0] == 0` in Figure 5. This gives considerable improvements, especially for `Hash Table S3`, which contains three integers per key.

- **Automatic ownership handling for Hash Table L**. For long strings, we copy the string data into `String Arena` to hold the ownership of string key when successfully inserted. This requires the insertion method to return the reference of the inserted string key in order to modify the string pointer to the new location in `String Arena`. However, it is expensive to return the key reference, since strings are also stored as integers in `SAHA`; returning references universally would require constructing temporary strings. To address this, we augment the string key with an optional memory pool reference and allow the hash table to decide when the string key needs to be parsed into the memory pool. It is used in `Hash Table L` that automatically parses strings when newly inserted.
- **CRC32 hashing with memory loading optimization**. Cyclic redundancy check (CRC) is an error-detecting code. It hashes a stream of bytes with as few collisions as possible, which makes it a good candidate as a hash function. Modern CPUs provide CRC32 native instructions that compute the CRC value of eight bytes in a few cycles. As a result, we can use the memory loading technique in the string dispatcher to do CRC32 hashing efficiently.

Listing 1. Implementatation of string dispatcher for the emplace routine in pseudo C++.

```
void dispatchEmplace(StringRef key, Slot*& slot) {
    int tail = (-key.size & 7) * 8;
    union { StringKey8 k8; StringKey16 k16; StringKey24 k24; uint64_t n[3]; };
    switch (key.size) {
    case 0:
        Hash_Table_S0.emplace(0, slot);
    case 1..8:
        if (key.pointer & 2048) == 0)
        { memcpy(&n[0], key.pointer, 8); n[0] &= -1 >> tail;}
        else { memcpy(&n[0], key.pointer + key.size - 8, 8); n[0] >>= tail; }
        if (n[0] < 65536) Hash_Table_S0.emplace(k8, slot);
        else Hash_Table_S1.emplace(k8, slot);
    case 9..16:
        memcpy(&n[0], key.pointer, 8);
        memcpy(&n[1], key.pointer + key.size - 8, 8);
        n[1] >>= s;
        Hash_Table_S2.emplace(k16, slot);
    case 17..24:
        memcpy(&n[0], key.pointer, 16);
        memcpy(&n[2], key.pointer + key.size - 8, 8);
        n[2] >>= s;
        Hash_Table_S3.emplace(k24, slot);
    default:
        Hash_Table_L.emplace(key, slot);
    }
}
```

To better understand the effectiveness of our optimizations, we discuss the cost models of the hash table's `find` and `emplace` routines respectively. For the `find` routine, it mainly consists of two steps: (a) computing the current key's hash value to lookup its conflicting chain, denoted as H; and (b) probing the chain to find whether there is a matching key, denoted as P. The complexity of the probing process is related to the average length of the conflicting chain, denoted as $alpha$. We can further divide the probing process into loading each slot's memory content P_L and doing key comparisons P_C.

After adding the complexity O of other maintenance operations such as rehashing, we can formulate the runtime complexity T_{find} of a unsuccessful `find` operation as Equation (1).

$$T_{find} = O + H + alpha * (P_L + P_C) \quad (1)$$

A successful find has almost the same complexity with a smaller $alpha$; that is, it can finish earlier when a match is found. An unsuccessful (we call it unsuccessful because the emplace fails to store the key when a same key is found in the hash table) emplace operation is identical to a successful find, while a successful emplace has one additional step: after finding an empty slot, it needs to store the element being emplaced, denoted as S. For string values, it means parsing the string data into the hash table's memory pool, S_P, and writing the updated string pointer into the slot, S_W. Thus, the total runtime complexity $T_{emplace}$ of a successful `emplace` operation is higher than a `find` operation, as shown in Equation (2).

$$T_{emplace} = O + H + alpha * (P_L + P_C) + S_P + S_W \quad (2)$$

Optimizing all these parameters is essential to improving the hash table's performance, excluding parameter O, which is related to the mechanism of open addressing scheme, and there is little room for its optimization. The $alpha$ parameter is related to the probing scheme, the hash function and the dataset itself. Optimizing $alpha$ is a broad topic and is beyond the scope of this work. Instead, we adopt the strategy of an industry-proved hash table—ClickHouse—with a good quality $alpha$ value as our building block, and it also comes with a low maintenance cost. In order to verify the effectiveness of our optimizations on the remaining parameters, we design a string emplacing benchmark using a real world string dataset gathered from website—twitter.com—to evaluate each parameter's runtime complexity. To cover all the parameters, we only test the successful `emplace` operation and preprocess the dataset with only distinct strings remaining so that all the `emplace` operations are successful. The comparison is against the hash table in ClickHouse, which is served as the baseline, and the test results are shown in Table 2. The micro benchmark shows that our optimizations improve all parameters significantly.

Table 2. Comparison of the complexities of cost model parameters with and without optimization.

Parameter	Baseline	SAHA	Optimizations
H	33	22	Memory loading optimization
$P_L + P_C$	56	37	String inlining and slot empty check optimization
S_P	34	6	String dispatching and automatic ownership handling
S_W	16	14	Selective hash saving
Unit	Cycles per key	Cycles per key	

4. Pre-Hashing and Vectorized Programming Interface

`SAHA` is designed for analytical databases, and the vectorized processing query pipeline gives a unique opportunity for optimizations. One of the main optimizations is `pre-hashing`. When doing element insertion or lookup in hash tables, a required step is to compute the element's hash value. This requires examining the entire content of the element, which means the CPU cache will be populated with this element. When the string is long, it can take up a lot of space in the cache and potentially evict the ones used for hash table slots, leading to cache contention. In order to better utilize CPU cache when probing, we need to avoid reading the string content. This is possible for inserting new long strings or finding non-existing long strings with precomputed hashes. For instance, when a fresh long string is inserted into `SAHA`, it gets dispatched to `Hash Table L`. Then we directly use the precomputed hash for probing. Since the `Hash Table L` also stores saved hashes in slots, it is likely for a fresh string to find an empty slot without any string comparison involved. Even if two different long

strings have the same hash value, comparing these two strings will likely finish early unless these two strings share a long prefix. For long strings, the dataset is usually of high cardinality, and processing fresh keys is the majority case for the hash table. As a result, there is a high chance we can get rid of reading the string content with precomputed hash values.

We can calculate the hash values after receiving a batch of records but before processing them in the hash table. Figure 7 illustrates the comparison with and without pre-hashing. It is trivial to implement `pre-hashing` outside the hash table. However, this is a bad design for analytical databases, since it also aggregates integer keys and compound keys where `pre-hashing` does not make sense. Thus, the optimization should be transparent to the aggregator and live inside the hash table. In order to achieve `pre-hashing`, the hash table should receive a batch of records instead of one row per time. Therefore, we proposed a vectorized programming interface for `SAHA`. It can also be applied to other hash tables.

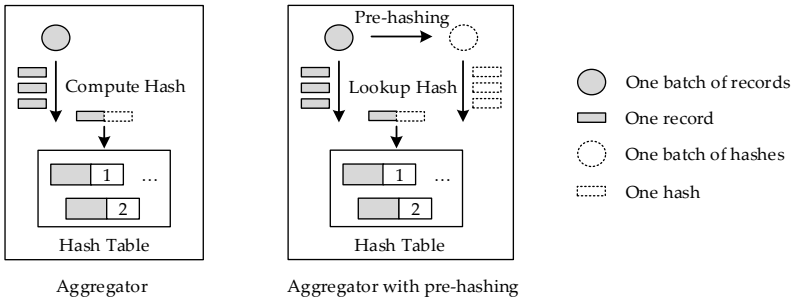


Figure 7. Comparison of vectorized aggregation implementations with and without pre-hashing.

Figure 8 shows a diagram of a hash table which accepts a batch of records. It precomputes the hash values before doing other operations. After processing each record, the registered callback is invoked to finish the operation. In the aggregator example, the callback is an aggregate function applied to the record. Listing 2 shows the comparison of the `emplace` routine with and without vectorization. With the help of the vectorized programming interface, we can achieve other optimizations such as reordering the batch of records to break data dependency.

Listing 2. Comparison of the emplace routine with and without vectorization in pseudo C++.

```
// vanilla emplace
void emplace(
    StringRef key,
    Slot*& slot) {
  uint64_t hash = hashing(key);
  slot = HashTable.probing(hash);
}

// Aggregating
for (StringRef& key : keys) {
  Slot* slot;
  emplace(key, slot);
  if (slot == nullptr)
    slot->value = 1;
  else
    ++ slot->value;
}
```

```
// vectorized emplace with pre-hashing
void emplaceBatch(
    vector<StringRef>& keys,
    Callback func) {
  vector<uint64_t> hashes(keys.size());
  for (int i=0; i<keys.size(); ++i)
    hashes[i] = hashing(keys[i]);
  for (int i=0; i<keys.size(); ++i)
    func(HashTable.probing(hash[i]));
}

void callback(Slot* slot) {
  if (slot == nullptr)
    slot->value = 1;
  else
    ++ slot->value;
}

// Aggregating
emplaceBatch(keys, callback);
```

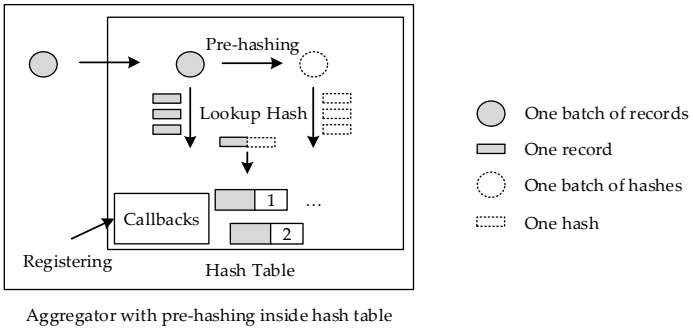


Figure 8. Aggregator using a hash table with vectorized programming interface and pre-hashing.

5. Evaluation

In this section, we evaluate `SAHA`'s performance using a physical server with four benchmarks (SetBuild, SetLookup, group and join). Four real world datasets are used with different string length distributions properties described in Table 3. We also generate additional synthetic datasets to further investigate the performance with regard to different string length distributions. The physical server contains two Intel Xeon E5-2640v4 CPUs with 128 GB memory. We synthesized the `Term` datasets with binomial string length distribution using different mean values.

Table 3. Datasets.

Name	Description	# Strings	# Distinct	String Length Quantiles				
				0.2	0.4	0.6	0.8	1.0
Real World Datasets								
Hotspot	Weibo's hot topics	8,974,818	1,590,098	12	15	18	24	837
Weibo	Weibo's user nicknames	8,993,849	4,381,094	10	12	16	21	45
Twitter	Twitter's user nicknames	8,982,748	7,998,096	8	9	11	13	20
SearchPhrase	Yandex's web search phrases	8,928,475	4,440,301	29	44	57	77	2025
Synthetic Datasets (with Binomial Distributions)								
Term2	Mean length 2	8,758,194	835,147	1	1	2	3	14
Term4	Mean length 4	8,758,194	3,634,680	2	3	4	6	17
Term8	Mean length 8	8,758,194	8,247,994	6	7	9	10	25
Term16	Mean length 16	8,758,194	8,758,189	13	15	17	19	40
Term24	Mean length 24	8,758,194	8,758,194	20	23	25	28	47
Term48	Mean length 48	8,758,194	8,758,194	44	47	49	52	73

We compare `SAHA` to a number of different hash tables, as shown in Table 1. We first evaluate these hash tables with four benchmarks on all the real world datasets, and accumulate the runtime to gain an overview of performance comparison. For each dataset and benchmark, we show a heat map of slowdowns compared to the fastest of all hash tables in Figure 9. Then we evaluate the fastest ones with additional detailed comparison tests. This helps reduce the number of further detailed benchmarks. Additionally, as we are trying to select the leading hash tables, the `pre-hashing` optimization is excluded, since not all hash tables are feasible with `pre-hashing`.

From Figure 9 we can see that `SAHA` achieves the best performance in almost all benchmarks. It is only slightly slower than the ClickHouse's linear probing hash table when dealing with long strings. Trie based implementations are extremely slow and should not be used for `group` and `join` workloads. Of all the open addressing hash tables, linear probing is good enough without complex probing schemes and achieves the second best performance among all implementations. Vectorized hash tables do not show performance advantages in all four benchmarks.

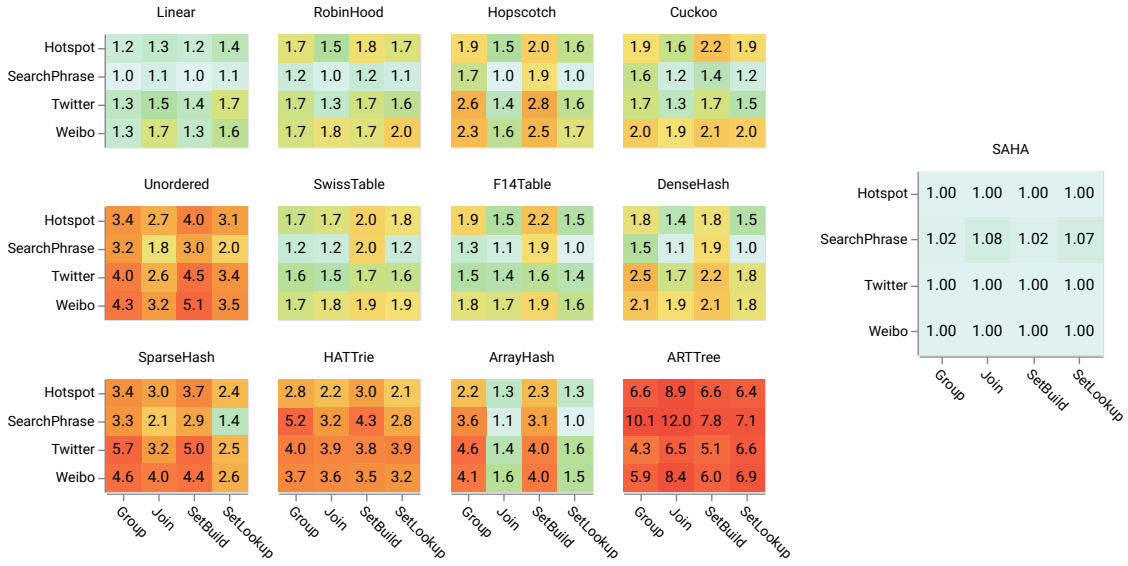


Figure 9. A heat map of slowdowns of various hash tables compared to the fastest of all hash tables for SetBuild, SetLookup, group and join benchmarks, on four datasets with varying string length distributions (HotSpot, Weibo, Twitter, SearchPhrase). Lower numbers (green) are better, with one being the fastest for the specific benchmark running on the specific dataset.

To complete the evaluation, we compare the peak memory allocated when benchmarking each hash table. In order to get accurate memory accounting, we use `malloc hooks` to record every allocation and deallocation, while keeping track of the maximum reached value. We only evaluate the `group` benchmark on two datasets, as it is enough to measure the memory consumption characteristics. As shown in Figure 10, `SAHA` is memory efficient when dealing with short strings, close to the `compact` implementations, and it achieves the best runtime performance. `HATTrie` has the lowest memory consumption, 30 percent lower than `SAHA`, but its runtime is four times slower. For long string datasets, `SAHA` does not provide memory optimizations and uses the same amount of memory as all the implementations with saved hash. As we have not yet included `pre-hasing` optimizations, it also has the same performance characteristics as `Linear`. `Compact` hash tables and trie tree based implementations are very memory efficient but take much more time and should be used with care. Vectorized hash tables consume slightly less memory than other open addressing implementations, because they allocate memory in chunks and have less wasted space introduced by memory aligning.

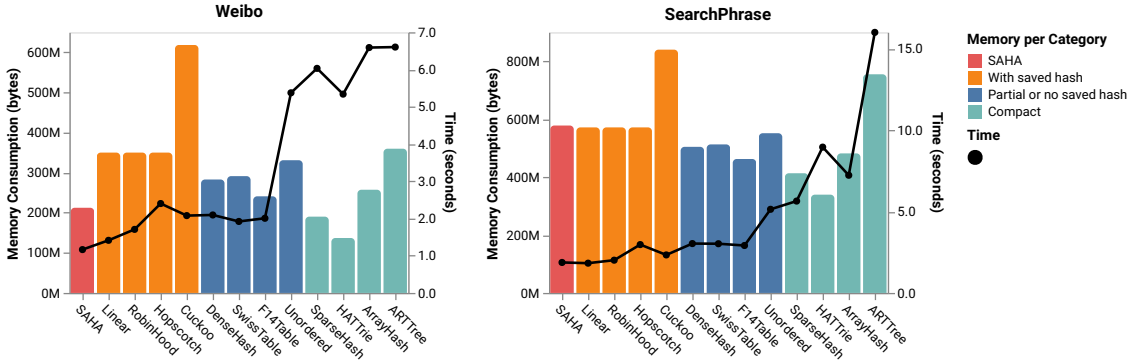


Figure 10. A combined bar and line graph of the memory and time consumption of various hash tables for group benchmarks on two datasets with short and long string length distributions (Weibo and SearchPhrase). The bar graph is grouped by memory consumption characteristics.

As mentioned in previous sections, in order to get better performance when dealing with long strings, we introduced `pre-hashing` optimization. From the previous benchmarks we saw that the

`Linear` hash table can serve as a good baseline. Thus, we do detailed comparison between `Linear` and `SAHA` and demonstrate the benefits of `pre-hashing` for long strings in Figure 11. The evaluation shows `pre-hashing` improves the performance on a large margin for long string datasets, and it also benefits short string datasets too. There are two datasets, `SearchPhrase` and `Term48`, in which `SAHA` without `pre-hashing` introduces a little performance regression due to code bloating; however, it is completely offset by the `pre-hashing` optimization. The only dataset which is slower than without `pre-hashing` is `Term2`, because it contains tiny strings and hash computation is trivial in such cases. But the total runtime is also small and the regression is negligible. As a result, `pre-hashing` should always be used as it improves the hash table's performance in general.

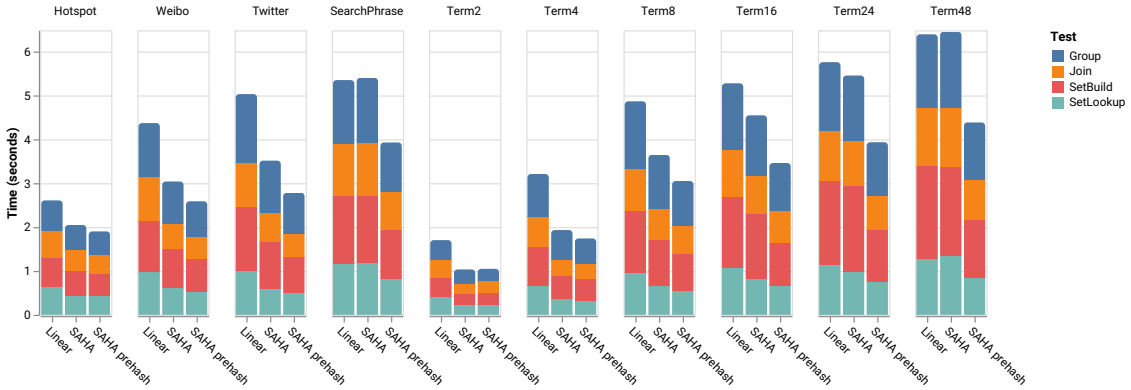


Figure 11. A bar graph of accumulated runtime of `Linear`, `SAHA` and `SAHA` with pre-hashing enabled for SetBuild, SetLookup, group and join benchmarks on all datasets.

We also evaluate the optimization of `SAHA`'s string dispatcher. The benchmark is similar to what we did for evaluating the `pre-hashing` optimization. We use a naive implementation of string dispatcher to serve as a baseline. It is implemented without the memory loading optimization, and the code size is larger. From Figure 12 we can see that compared to naive dispatching, our optimization doubles the performance gain in most test cases. However, the optimization does not affect the performance when dealing with datasets containing tiny or long strings, because these two cases are barely affected by memory loading.

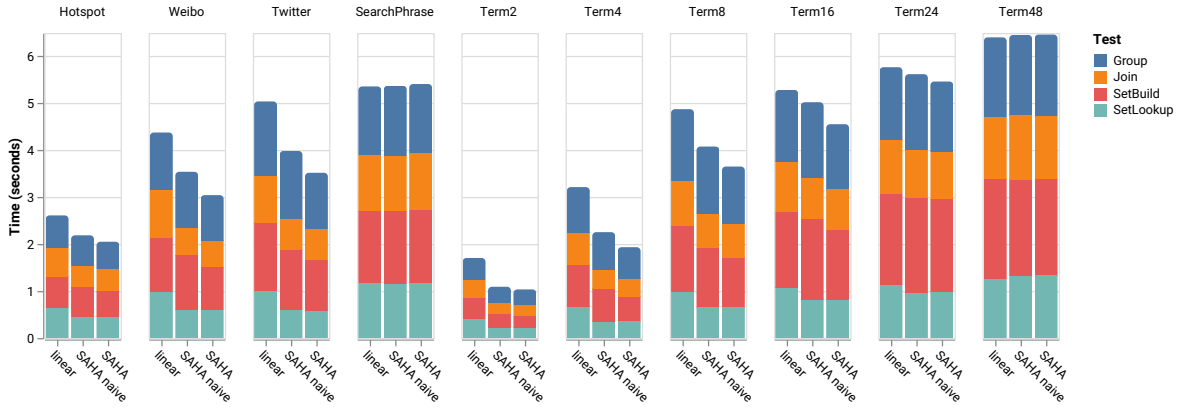


Figure 12. A bar graph of the accumulated runtime of `Linear`, `SAHA` with naive dispatching and `SAHA` for SetBuild, SetLookup, group and join benchmarks on all datasets.

To understand the scalability of different hash tables and their performances on large datasets, we use a real world string dataset gathered from website weibo.com—with one billion rows. Figure 13 compares multiple hash tables using a line graph to reveal the scalability of each hash table. It can be seen that `SAHA` scales linearly, and thanks to our string inlining optimization, `SAHA` is memory friendlier

and thus improves more when processing large datasets. Moreover, since the `join` workload mainly contains key lookup operations, the performance advantage of `SAHA` becomes more prominent.

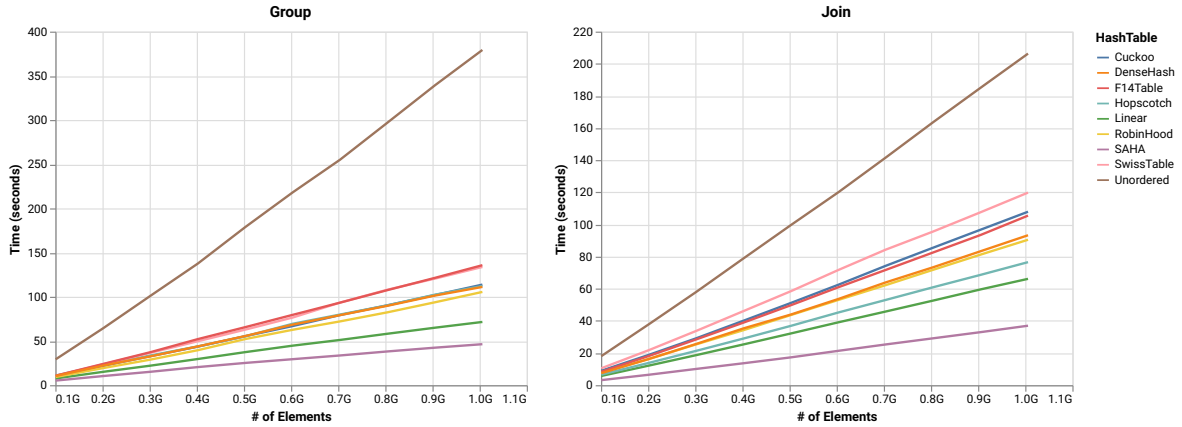


Figure 13. A line figure of time consumption of various hash tables for group and join benchmarks on the real world Weibo dataset.

To validate the advantages of using the `CRC32` instruction as the hash function, we evaluate it against other seven hash functions. Some of them are widely used, such as `MurmurHash`, while others are more modern and claim to achieve promising results, such as `t1ha` and `xxHash`. Due to the similarity of the hashing operations in each benchmark, we only use the `group` benchmark for comparison. Since a hash function might affect the performance of the underlying hash table, we select six hash tables from previous benchmarks and substitute their hash functions to rerun the test. We also use two significantly different datasets, `Weibo` and `SearchPhrase`, to evaluate the influence of string length distributions over the hash functions. As shown in Figure 14, `CRC32Hash` is the fastest among all cases. The result indicates that hash functions perform consistently over different hash tables. The difference among hash functions is higher on the long string dataset, `SearchPhrase`, compared to the `Weibo` dataset, which is dominated by short strings. This is because the hashing operation occupies a smaller proportion of total runtime when processing short strings; thus, the benchmark on the `Weibo` dataset is less sensitive to different hash functions. The performance advantage of `CRC32` is mainly because of the native instructions such as `_mm_crc32_u64` in modern CPUs. As a result it is the default hash function in `SAHA`.

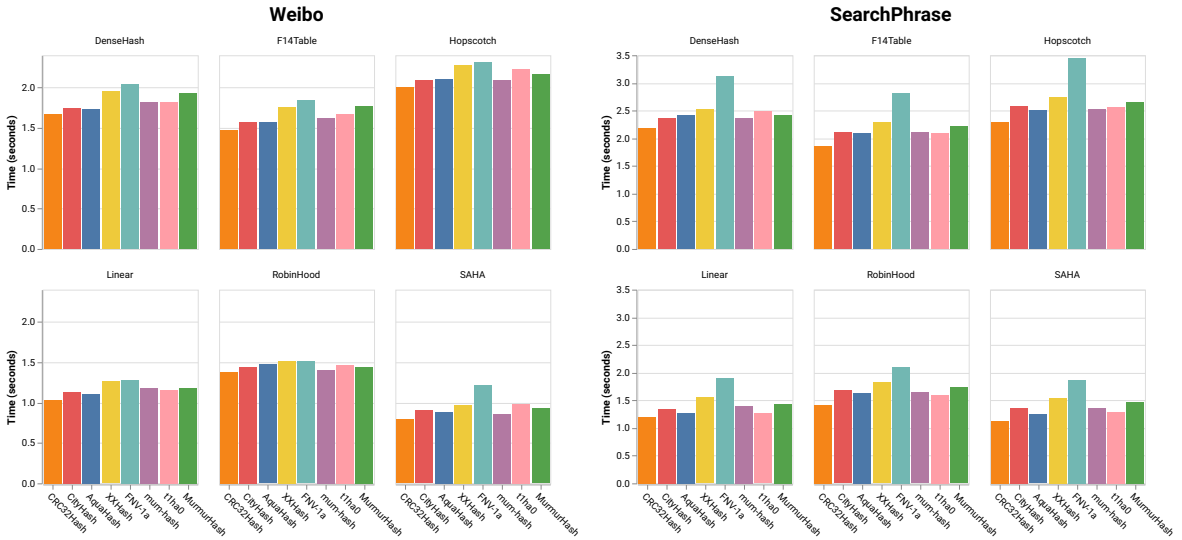
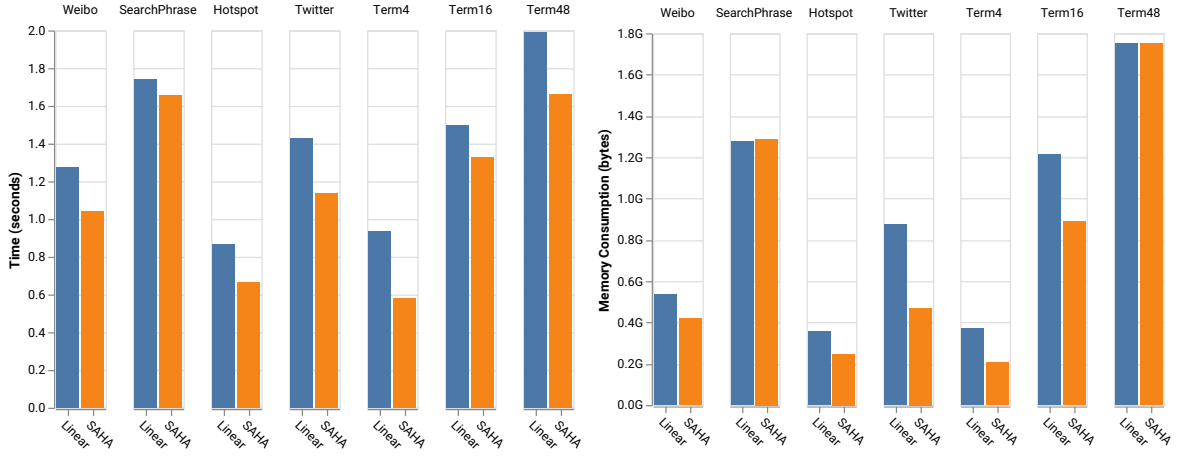


Figure 14. A bar graph of the time consumption of various hash functions for group benchmarks on two datasets with short and long string length distributions (Weibo and SearchPhrase).

As our goal is to optimize the real world workloads in analytical databases. We implemented SAHA in the ClickHouse database and compared its end-to-end time consumption against the builtin hash table implementation. Due to the current limitation of ClickHouse's join mechanism, we only evaluated the group benchmark, and it can serve as a good indicator for the query's runtime improvements. We used "SELECT str, count(*) FROM <table> GROUP BY str" as the benchmark query, where <table> was any of the datasets we used in previous evaluations. It was populated with one column named str and of type string. As can be seen from Figure 15, SAHA helps improve the total query runtime up to 40%, and it is consistently faster on all string data distributions. Because there are other operations participating when doing the end-to-end group benchmarking, such as data scanning, memory copying and various string manipulations, we cannot get the same improvements as shown in the aforementioned hash table evaluations; however, for a highly optimized application such as an analytical database, 40% is remarkable and also shows that hash tables play an important role in analytical queries like group. Moreover, the memory consumption reduced drastically when evaluated on short string datasets, because we had short strings represented as numeric values, and no saved hash values were used.



(a) End-to-end time consumption

(b) End-to-end memory consumption

Figure 15. Two bar graphs of the time and memory consumption of ClickHouse's builtin hash table and SAHA with pre-hashing enabled for an end-to-end group query benchmark on various datasets.

6. Conclusions

In this paper, we proposed SAHA, a hybrid hash table implementation that stores short strings as integers in a set of underlying hash tables. The main idea of SAHA is to use string length dispatching so that small string keys will end up in hash tables that are built for integers, while long string keys will have their hash values saved to avoid expensive hash recomputations. It is both more CPU and memory efficient than using pointers to store short strings. Along with other optimizations, such as pre-hashing and memory-loading, the outcome is a hash table that is nearly 100% faster than the second best we could find. We carried out extensive evaluations and gave a detailed analysis about hash tables, hash functions and string data distributions. Part of the SAHA's code has already been merged into ClickHouse and has been widely used and tested in production.

In our future work, we plan to investigate the vectorization possibilities for the long string sub hash table, and we also plan to combine sorting and other data structures to further optimize analytical queries.

Author Contributions: conceptualization, T.Z.; methodology, T.Z.; software, T.Z.; validation, T.Z.; writing—original draft preparation, T.Z.; visualization, T.Z.; supervision, Z.Z.; project administration, Z.Z.; funding acquisition, X.C.

Funding: This research was funded by The Strategic Priority Research Program of the Chinese Academy of Sciences (XDA19020400).

Acknowledgments: We thank the Yandex ClickHouse team for reviewing the SAHA code and helping merge it to the ClickHouse code base. We thank anonymous reviewers whose comments helped improve and clarify this manuscript.

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

References

1. Wikipedia Contributors. SIMD—Wikipedia, The Free Encyclopedia, 2020. Available online: https://en.wikipedia.org/w/index.php?title=SQL&oldid=938477808 (accessed on 1 Feburary 2020).
2. Nambiar, R.O.; Poess, M. The Making of TPC-DS. *VLDB* **2006**, *6*, 1049–1058.
3. ClickHouse Contributors. ClickHouse: Open Source Distributed Column-Oriented DBMS, 2020. Available online: https://clickhouse.tech (accessed on 1 Feburary 2020).
4. Bittorf, M.K.A.B.V.; Bobrovytsky, T.; Erickson, C.C.A.C.J.; Hecht, M.G.D.; Kuff, M.J.I.J.L.; Leblang, D.K.A.; Robinson, N.L.I.P.H.; Rus, D.R.S.; Wanderman, J.R.D.T.S.; Yoder, M.M. *Impala: A Modern, Open-Source SQL Engine for Hadoop*; CIDR: Asilomar, CA, USA, 2015.
5. Thusoo, A.; Sarma, J.S.; Jain, N.; Shao, Z.; Chakka, P.; Anthony, S.; Liu, H.; Wyckoff, P.; Murthy, R. Hive: A Warehousing Solution over a Map-reduce Framework. *Proc. VLDB Endow.* **2009**, *2*, 1626–1629. doi:10.14778/1687553.1687609.
6. Wikipedia contributors. SQL — Wikipedia, The Free Encyclopedia, 2020. Available online: https://en.wikipedia.org/w/index.php?title=SIMD&oldid=936265376 (accessed on 1 Feburary 2020).
7. Celis, P.; Larson, P.A.; Munro, J.I. Robin hood hashing. In Proceedings of the IEEE 26th Annual Symposium on Foundations of Computer Science (sfcs 1985), Washington, DC, USA, 21–23 Oct 1985; pp. 281–288.
8. Herlihy, M.; Shavit, N.; Tzafrir, M. Hopscotch Hashing. In *Distributed Computing*; Taubenfeld, G., Ed.; Springer: Berlin/Heidelberg, Germany, 2008; pp. 350–364.
9. Richter, S.; Alvarez, V.; Dittrich, J. A seven-dimensional analysis of hashing methods and its implications on query processing. *Proc. VLDB Endow.* **2015**, *9*, 96–107.
10. Pagh, R.; Rodler, F.F. Cuckoo hashing. In *European Symposium on Algorithms*; Springer: Berlin/Heidelberg, Germany, 2001; pp. 121–133.
11. Kirsch, A.; Mitzenmacher, M.; Wieder, U. More robust hashing: Cuckoo hashing with a stash. *SIAM J. Comput.* **2010**, *39*, 1543–1561.
12. Breslow, A.D.; Zhang, D.P.; Greathouse, J.L.; Jayasena, N.; Tullsen, D.M. Horton tables: Fast hash tables for in-memory data-intensive computing. In Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC 16), Denver, CO, USA, 22–24 June 2016; pp. 281–294.
13. Scouarnec, N.L. Cuckoo++ hash tables: High-performance hash tables for networking applications. In Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems, Ithaca, NY, USA, 23–24 July 2018; pp. 41–54.
14. google sparsehash@googlegroups.com. Google Sparse Hash, 2020. Available online: https://github.com/sparsehash/sparsehash (accessed on 1 Feburary 2020).
15. Barber, R.; Lohman, G.; Pandis, I.; Raman, V.; Sidle, R.; Attaluri, G.; Chainani, N.; Lightstone, S.; Sharpe, D. Memory-efficient hash joins. *Proc. VLDB Endow.* **2014**, *8*, 353–364.
16. Benzaquen, S.; Evlogimenos, A.; Kulukundis, M.; Perepelitsa, R. Swiss Tables and absl::Hash, 2020. Available online: https://abseil.io/blog/20180927-swisstables (accessed on 1 Feburary 2020).
17. Bronson, N.; Shi, X. Open-Sourcing F14 for Faster, More Memory-Efficient Hash Tables, 2020. Available online: https://engineering.fb.com/developer-tools/f14/ (accessed on 1 Feburary 2020).
18. Wikipedia contributors. Trie—Wikipedia, The Free Encyclopedia, 2020. Available online: https://en.wikipedia.org/w/index.php?title=Trie&oldid=934327931 (accessed on 1 Feburary 2020).
19. Aoe, J.I.; Morimoto, K.; Sato, T. An efficient implementation of trie structures. *Softw. Pract. Exp.* **1992**, *22*, 695–721.
20. Heinz, S.; Zobel, J.; Williams, H.E. Burst tries: A fast, efficient data structure for string keys. *ACM Trans. Inf. Syst. (Tois)* **2002**, *20*, 192–223.

21. Askitis, N.; Sinha, R. HAT-trie: A cache-conscious trie-based data structure for strings. In Proceedings of the Thirtieth Australasian Conference on Computer Science, Ballarat, VIC, Australia, 30 January–2 February2007; Australian Computer Society, Inc.: Mountain View, CA, USA, 2007; Volume 62, pp. 97–105.
22. Leis, V.; Kemper, A.; Neumann, T. The adaptive radix tree: ARTful indexing for main-memory databases. In Proceedings of the 2013 IEEE 29th International Conference on Data Engineering (ICDE), Brisbane, QLD, Australia, 8–12 April 2013; pp. 38–49. doi:10.1109/ICDE.2013.6544812.
23. Alvarez, V.; Richter, S.; Chen, X.; Dittrich, J. A comparison of adaptive radix trees and hash tables. In Proceedings of the 2015 IEEE 31st International Conference on Data Engineering, Seoul, Korea, 13–17 April 2015; pp. 1227–1238.
24. Askitis, N.; Zobel, J. Cache-conscious collision resolution in string hash tables. In *International Symposium on String Processing and Information Retrieval*; Springer: Berlin/Heidelberg, Germany, 2005; pp. 91–102.
25. Lamb, A.; Fuller, M.; Varadarajan, R.; Tran, N.; Vandiver, B.; Doshi, L.; Bear, C. The Vertica Analytic Database: C-store 7 Years Later. *Proc. VLDB Endow.* **2012**, *5*, 1790–1801. doi:10.14778/2367502.2367518.
26. Färber, F.; Cha, S.K.; Primsch, J.; Bornhövd, C.; Sigg, S.; Lehner, W. SAP HANA Database: Data Management for Modern Business Applications. *SIGMOD Rec.* **2012**, *40*, 45–51. doi:10.1145/2094114.2094126.
27. Stonebraker, M.; Abadi, D.J.; Batkin, A.; Chen, X.; Cherniack, M.; Ferreira, M.; Lau, E.; Lin, A.; Madden, S.; O'Neil, E.; et al. C-store: A column-oriented DBMS. In *Making Databases Work: The Pragmatic Wisdom of Michael Stonebraker*; ACM: New York, NY, USA, 2018; pp. 491–518.
28. Amos Bird (Tianqi Zheng). Add StringHashMap to Optimize String Aggregation, 2019. Available online: https://github.com/ClickHouse/ClickHouse/pull/5417 (accessed on 1 Feburary 2020).