Footprint Reduction and Uniqueness Enforcement with Hash Indices in SAP HANA

Martin Faust[1], Martin Boissier[1], Marvin Keller[1], David Schwalb[1], Holger Bischoff[2], Katrin Eisenreich[2], Franz Färber[2], and Hasso Plattner[1]

[1] Hasso Plattner Institute, Potsdam, Germany
{firstname.lastname}@hpi.de
[2] SAP SE, Walldorf, Germany
{firstname.lastname}@sap.com

Abstract. Databases commonly use multi-column indices for composite keys that concatenate attribute values for fast entity retrieval. For real-world applications, such concatenated composite keys contribute significantly to the overall space consumption, which is particularly expensive for main memory-resident databases. We present an integer-based hash representation of the actual values for the purpose of reducing the overall memory footprint of a system while maintaining the level of performance. We analyzed the performance impact as well as the memory footprint reduction of hash-based indices in SAP HANA in a real-world enterprise database setting. For a production SAP ERP system, the introduction of hash-based primary key indices alone reduces the entire memory footprint by 10% with comparable performance.

Keywords: in-memory databases, hash indices, footprint reduction, enterprise systems

1 Composite Keys in Enterprise Applications

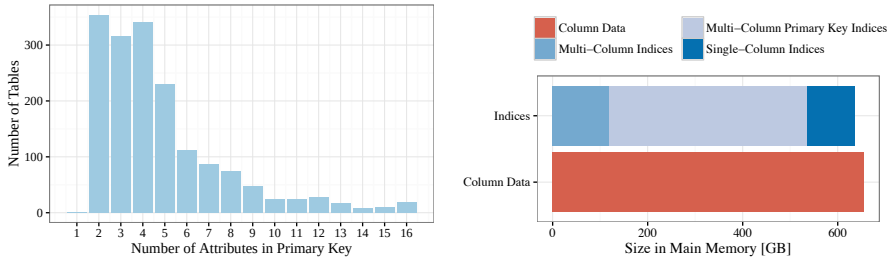
Today's trends in hardware development render in-memory databases as a viable platform for enterprise applications. In-memory databases use compression techniques for the purpose of reducing the required main memory. We analyzed the primary keys of a large enterprise resource planning (ERP) installation of a Global 2000[1] company. We found that most tables' primary keys contain multiple columns as shown in Figure 1(a). To achieve fast data retrieval on these tables, multi-column indices are used. Looking at the memory breakdown shown in Figure 1(b), we see that composite keys account for nearly 30% of the entire memory footprint.

In SAP HANA, these multi-column indices are stored as a simple concatenation of the primary key values (hereafter called *value-based indices*). Although various forms of compression are applied to these indices, they introduce additional data stored in DRAM and therefore further add to the memory footprint.

[1] Global 2000: http://www.forbes.com/global2000/

27th International Conference Database and Expert Systems Applications, DEXA 2016, Porto, Portugal, September 5-8, 2016, LNCS 9828, pp 137-151. The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-44406-2_11.

In this paper, we evaluate whether we can reduce the size of composite keys by storing a hash-based integer representation of the composite values instead of the actual values concatenated while maintaining the same level of performance.



(a) Overview of the number of attributes in primary keys for tables with more than 100,000 rows. In the most recent SAP ERP version, all of the larger tables have a primary key with at least two attributes.

(b) Memory consumption: index structures consume almost as much main memory as the actual data, whereby multi-column primary key indices alone are responsible for over 400 GB.

Fig. 1. Statistics for a live production SAP ERP system of a Global 2000 company: (a) overview of primary key lengths and (b) break down of memory consumption.

2 Production Enterprise System: SAP ERP & Columnar In-Memory Databases

An Enterprise Resource Planning (ERP) application is the central management software for large companies. We had the opportunity to analyze a live production system of an SAP ERP system of a Global 2000 company. This system stores over 10 billion records in 23,886 tables with a total main memory footprint of about 1.3 TB. 90% of these tables have multi-column primary keys, emphasizing the impact a change of the primary key type could have. While analyzing a single instance does not cover the whole ERP market, we consider this system representative since SAP ERP systems have a share of 25% of the global ERP market and are used by more than half of the Fortune 500 companies.

The ERP system runs on a columnar in-memory database optimized for OLxP workloads: SAP HANA. In-memory database systems like SAP HANA [5] and HYRISE [9] use a main/delta architecture to store database relations. Inserts and updates are handled by a comparatively small and write-optimized partition, called *delta partition*. The delta partition is frequently merged with the *main partition* [5, 9]. The main partition is read-only, compressed, and read-optimized towards analytical workloads. This allows for fast analytical queries while still supporting sufficient transactional performance.

Each column is dictionary-encoded consisting of an attribute vector and a dictionary. The dictionary stores all distinct values in a sorted manner while the

attribute vector contains bit-packed valueIDs for each record. These valueIDs reference the actual, uncompressed values stored in the dictionary by their offset.

3 Related Work

Database index structures have been optimized in many ways. The general goal is to increase lookup performance while minimizing the additional storage needed for these indices. But with changing trends in hardware, there is the need to further optimize these index structures for their target systems. In-memory databases require new and optimized in-memory indexing structures since traditional indexing strategies become inefficient on modern hardware [1].

Tree-Based Indices Leis et al. [12] introduced the DRAM-optimized *adaptive radix trees* (ART). By adaptively choosing efficient data structures used in ART, they were able to achieve high space efficiency while surpassing the performance of traditional tree-based index structures.

Athanassoulis and Ailamaki [3] introduced a method of reducing memory requirements of tree-based index structures by employing probabilistic data structures (Bloom filters). By trading accuracy for size, they were able to reduce the footprint of tree-based indices by up to 4× for real-world scenarios while keeping the performance on par with traditional tree indexing. Their motivation was the trend of solid-state disks emerging as a viable alternative to traditional hard disk drives.

Hash-Based Indices An alternative to tree-based structures is a hash-based data structure that is typically employed in two types: (1) hash tables with fixed size and no reorganization of data and (2) hash tables with variable size and dynamic reorganization. An example for the former is *chained bucket hashing* [8]. Dynamic structures include *extensible hashing* [4], *linear hashing* [10, 13] and *modified linear hashing* [11]. Ross presented a method of hash probing for typical database workloads using SIMD instructions [16].

With the usage of in-memory databases with column stores, the problem of efficiently accessing disk blocks is replaced by accessing the main memory and therefore the problem of minimizing cache misses [14]. Sidirourgos and Kersten introduced column imprints as a cache conscious secondary indexing structure for column stores [17]. For each column, a histogram of a few equal-height bins is created. For every cache line of data, a bit vector is created with each bit corresponding to a bin of the histogram. A bit is set if the cache line contains at least one value in the corresponding bin. The authors have shown significantly improved query speed with a storage overhead of only 12%.

Composite Keys Faust et al. [6] introduced the composite group-key as an alternative indexing method for composite keys. They utilize the existing dictionary compression by concatenating the compressed values (i.e., valueIDs) of the primary key column's dictionaries and storing them in an additional data structure

named key-identifier list. This structure contains integer values with 8, 16, 32, or 64 bits per key and is stored alongside a bit-packed position list to retrieve the record's position. Because they are storing an integer representation of the primary key attributes, the size of this index is significantly smaller than the size of the previously introduced index types that store the key attributes in an uncompressed format. Tests have shown that the composite group-key's performance is on par with established indexing methods while decreasing the storage requirements.

The default method of storing composite keys in SAP HANA is adding an additional column to the table that contains the concatenated values of all primary key columns. For each entry, a compressed (using *Golomb* or *Simple9* compression [2]) position list is stored for fast record retrieval. This allows database operations to only use a single column instead of having to scan every column of the composite primary key, but adds significantly to the overall size of the table, because the primary key values are basically stored twice. The additional key column consists of a sorted dictionary containing the key values, the attribute vector and the position list. For key lookups, the primary key values are concatenated into a single search string that is used in a binary search on the composite key column's dictionary. The position list is used to find the records' positions in constant time. Because this is the current default method of indexing composite primary keys in HANA, we compare the performance and storage requirements of this index type with hash-based indices introduced in the next section.

4 Hash-based Unique Index

Hash-based indices hash the attributes of a composite-key to obtain a single, fixed-length representation.

4.1 Index Structure

The index is modeled as a dictionary-compressed column, and therefore contains a main and a delta partition. For the main partition, the sorted dictionary D_M stores hashed keys and is extended with an inverted index I_M to provide a mapping to row identifier. For this work, we assume that the inverted index establishes a one-to-one mapping of dictionary entries to position lists, hence, no additional logic is needed to support variable length position lists. Per definition, storing primary key values means there are 100% unique values in the dictionary what makes traditional dictionary encoding pointless. To reduce the dictionary size, delta encoding is used. The inverted index has the same length like D_M and is bit-packed. The attribute vector is a bit-packed list of offsets in D_{Hash}. The hash index dictionary of the delta partition is unsorted and again each entry is extended by a position list. Figure 2 shows the schematic process to create the inverted hash-based index.

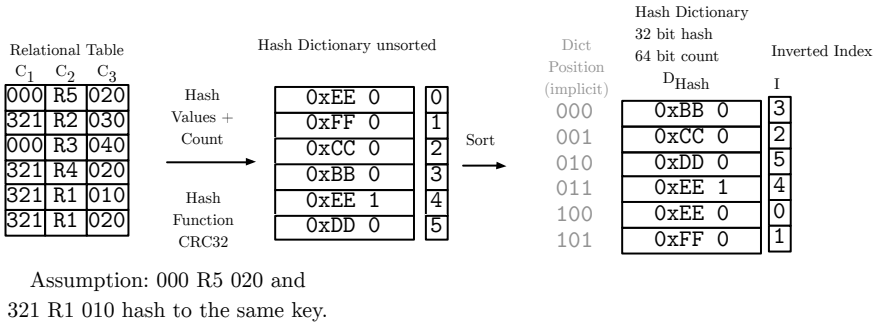


Fig. 2. Schematic overview of the hash-based multi-attribute index on the main partition (delta encoding not shown).

4.2 Lookup Algorithm

The index allows efficient point queries, i.e. the lookup of a key. For a primary key lookup, the predicate has to be translated into its hash representation for comparison with the values in the hash dictionary. This is achieved by concatenating the values of the primary key columns and applying the hash function to it. The resulting hash value is used in a binary search to find matching hashes in the hash dictionaries of the main partition as well as the delta partition. The position of the matching rows is extracted from the inverted index. Because of possible hash collisions, the actual values of all matching tuples have to be compared to find the tuples matching all predicates. The lookup algorithm, including the handling of collisions is shown in Algorithm 1.

4.3 Insert Algorithm

A frequent operation accessing the index is the lookup of a non-existing key for uniqueness constraints, when a new tuple is about to be inserted. The lookup has to be performed first to find rows that would potentially cause uniqueness violations (see Section 4.2). For every matching hash that was found, the actual attribute values are compared to ensure the uniqueness constraint. If the actual values are different, the hash is inserted into the hash dictionary with an 8-byte collision counter. If the values match, the new record will not be inserted, because of a violation of uniqueness for the primary key. The insert algorithm with verification is detailed in Algorithm 2.

4.4 Limitations

Because the used hash is an integer representation of the whole primary key and does not store the actual attribute values, it is not possible to use hash-based indices for range queries or partial key lookups. SAP HANA automatically creates

Algorithm 1 Lookup of key with n attributes

$h \leftarrow crc32(concat(k_0, ..., k_{n-1}))$
$match_M \leftarrow I_M[D_M[(h, min)..(h, max)]]$
$match_D \leftarrow I_D[D_D[(h, min)..(h, max)]]$
$MVCCverify(match_M), MVCCverify(match_D)$
$results \leftarrow []$
for P in (M, D) **do**
  for $rowID$ in $match_P$ **do**
    $equal \leftarrow True$
    for $i \leftarrow 0...(n-1)$ **do**
      $equal \leftarrow equal \ \& \ D_P i[AV_i[rowID]] == k_i$
    end for
    if $equal$ **then**
      $results \leftarrow [results.rowID]$
    end if
  end for
end for
return results

single-column indices on all attributes of the primary key that are hence used to answer non-full primary key selects. For the value-based index in contrast, range selects and partial key lookups can often be executed directly on the index via binary substring searches (depending on the selected attributes). Consequently, depending on the query filters on multiple columns have to be evaluated for the case of a hash-based primary key while a single access to the value-based index is sufficient for many typical OLTP queries. Further, for partitioned tables hash-based primary keys are only beneficial if the complete key is included in the partitioning criteria.

4.5 Hash Function

The hash function for the index ought to be fast and provide well-distributed hashes for continuously ascending keys. We use CRC32(C) as the hashing function for several reasons. First, cryptographic properties are not needed. Also, other hashing alternatives yield fewer collisions, but the number of expected collisions is limited anyway by SAP HANA's partition size limit of 2^{31} rows per partition. Second, recent Intel CPUs implement the CRC32 instruction in hardware (see Section 4.6) with a latency of only three CPU cycles.

Cyclic Redundancy Check (CRC) is a code commonly used for error-detection in digital networks or storage devices to detect unintentional changes in data. A message is encoded by appending a fixed-length check value. The check value is the remainder of the division of a given message by a specified polynomial. The receiver of a message can check its integrity by performing the same division and comparing the check values. The length of the remainder determines the name of the CRC. A CRC with a check value of n bits is called an n-bit CRC or CRCn. We use CRC32, i.e., the remainder has a length of 32 bits. For hash-based

Algorithm 2 Insertion of key with n attributes

$h \leftarrow crc32(concat(k_0, ..., k_{n-1})$
$collisions_D, collisions_M \leftarrow []$
if (h, min) in D_M **then**
  $collisions_M \leftarrow I_M[D_M[(h, min)..(h, max)]]$
end if
if (h, min) in D_D **then**
  $collisions_D \leftarrow I_D[D_D[(h, min)..(h, max)]]$
end if
MVCCverify($collisions_M$)
MVCCverify($collisions_D$)
for P in (M, D) **do**
  for c in $collisions_P$ **do**
    $equal \leftarrow True$
    for $i \leftarrow 0...(n-1)$ **do**
      $equal \leftarrow equal \ \& \ \& \ D_P i[AV_i[c]] == k_i$
    end for
    if *equal* **then**
      abort: unique violation
    end if
  end for
end for
$count \leftarrow |collisions_M| + |collisions_D|$ #all collisions refer to different keys
$InsertDelta(h, count)$

indices, we do not use CRC-32 to check data integrity. We use the check value as a shorter (32 bits) integer representation of the primary key values.

The message used as dividend in the polynomial division is the concatenation of the primary key values. To concatenate the key, we create a prefix-free encoding, by prefixing each key attribute with its length. The concatenated string follows the form `''<len(key1)>,key1;<len(key2)>,key2;''`. Since single partitions do not grow larger than two billion records, a hash length of 32 bits is sufficient.

4.6 CRC32: Hardware-Assisted Hashing

With the SSE4.2 instruction set, Intel added support for hardware-assisted CRC32C to their processors. Traditional CRC32, used for example in ZIP and Ethernet, uses the polynomial *0x04C11DB7* as divisor while CRC32C, which is supported by SSE4.2, uses the Castagnoli polynomial *0x1EDC6F41*. The SSE4.2 instruction uses a precalculated, built-in lookup table for the Castagnoli polynomial and is therefore limited to this specific polynomial while software implementations can choose the polynomial best suited for their use case. Using different polynomials results in different checksums, i.e. different hashes for the same key.

The CRC32 instruction expects two parameters: a destination operand and a source operand. It uses the fixed polynomial (Castagnoli) to accumulate the

CRC32 value for the source operand (i.e., the concatenated key values) and stores the result in the destination operand. The source operand can be a register or a memory location while the destination operand must be a register. This instruction can operate on a maximum data size of 64 bits and is implemented with a latency of three CPU cycles and a throughput of a single CPU cycle. To incrementally accumulate a CRC32 value, the result of the previous CRC32 operation is used to execute the CRC32 instruction again with new input.

The hardware implementation is 2-3× faster than highly optimized software implementations and its performance can be further increased by parallelizing the CRC computations [7]. These capabilities emphasize the viability of CRC32 for the use case of hash-based indices.

4.7 Collision Handling

By definition, any function that maps an unlimited range to a fixed range is prone to collisions. Collisions occur when a hash function creates the same hash for different values. Using CRC32, there are 2^{32} possible hash values. Although this is sufficient for the SAP HANA's maximum of two billion records that can be stored per partition, hash collisions are inevitable and have to be dealt with.

SAP HANA appends an 8-byte counter to the hashes before adding them to the dictionary. The value of this counter is unique and thereby ensures that all values in the dictionary are unique even if hashes for different values match. If a collision occurs while inserting a new record, the insert algorithm compares the actual values to enforce uniqueness, as described in Section 4.3. Collisions also have to be expected during key queries. As a consequence, lookups need to verify the actual key components against the predicate, as outlined in Section 4.2.

4.8 Column Merge

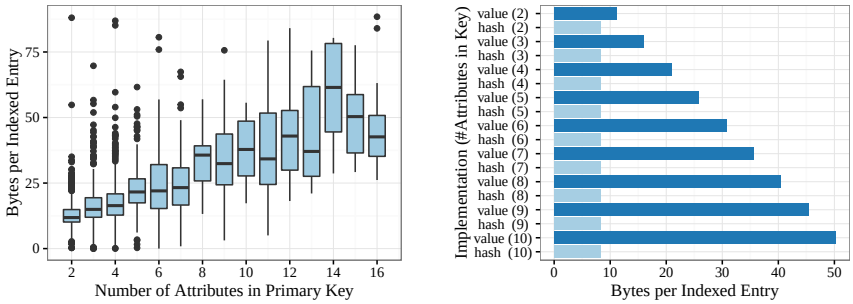
When merging the content of the delta partition into the main partition, a new main dictionary for the primary key is created. This dictionary contains all distinct hashed key values from the delta dictionary as well as from the old main dictionary. Since any insert into the table has to check for uniqueness in the main partition as well as in the delta partition, primary keys are ensured to be unique and thus the dictionaries can be directly merged. When a hash value of the delta partition already exists in the main partition, the collision counters are simply added and the inverted position list is updated.

4.9 Memory Footprint

Per dictionary entry, a 4-byte hash value is stored along with an 8-byte collision counter to resolve hash collisions. As mentioned earlier, the dictionary containing the hash values is compressed using delta encoding. Since there are only unique values stored in the dictionary, traditional dictionary encoding would have a negative effect on compression. Instead of storing the full values or compressing

single values, delta encoding stores only the difference of consecutive values. As a rule of thumb, after compressing the hash-index's dictionary, the average size per entry is 8-10 bytes.

5 Evaluation



(a) ERP system of a Global 2000 company: space consumption for value-based primary keys (tables with over 100,000 entries).

(b) Bytes per indexed item for a table with 1M integer values.

Fig. 3. Space consumption of composite keys.

We evaluate the potential memory footprint reductions of the hash-based index both on tables of the analyzed live production enterprise system and on a synthetic table of the TPC-C benchmark.

We evaluated three tables to cover a broad range of use cases for hash-based indices (an overview of the primary keys is shown in Table 1). The tables `BSEG` and `SKA1` are both table copies of the production SAP ERP system. `BSEG` is a transactional table storing accounting documents and is the central part of the financial module. `SKA1` is a master data table storing the chart of accounts of the general ledger module. Since it is a master data table, it is considerably smaller than the `BSEG` table. The third table is TPC-C's largest transactional table `ORDERLINE`, which we created with a scaling factor of 2,000.

The benchmarks have been executed on the same system with a varying number of benchmark processes. Each benchmark process runs 16 threads (8 for the insert benchmarks) that share the same database connection. SELECT queries solely project the first attribute of the primary key in order to exclude time required for tuple materialization. The benchmark system was a four-socket server equipped with Intel Xeon E7-4880 v2 CPUs and 2 TB of DRAM running SAP HANA SPS 11, revision 111. Error bars denote the standard error.

	Primary Key Attributes				
BSEG 70 M tuples	MANDT varchar(3) Distinct values: 1	BUKRS varchar(4) Distinct values: 476	BELNR varchar(10) Distinct values: 7,777,105	GJAHR varchar(4) Distinct values: 31	BUZEI varchar(3) Distinct values: 999
ORDERLINE 600 M tuples	OL_W_ID integer Distinct values: 2,000	OL_D_ID integer Distinct values: 10	OL_O_ID integer Distinct values: 3,000	OL_NUMBER integer Distinct values: 15	-
SKA1 67,618 tuples	MANDT varchar(3) Distinct values: 1	KTOPL varchar(4) Distinct values: 54	SAKNR varchar(10) Distinct values: 53,598	-	-

Table 1. Overview of the primary keys and their characteristics of benchmarked tables.

5.1 Main Memory Footprint

We measured the space consumption of all multi-column indices of the analyzed production enterprise system. Figure 3(a) shows a box plot of the bytes per indexed entry. For the 1,736 tables with more than 100,000 entries, the average size of an indexed key is about 24 bytes.

The size of value-based indices in large (>100,000 entries) tables in our analyzed system amounts to 386 GB. If we conservatively assume a size of 10 bytes per entry for the hash-based index (see Section 4.9), the memory footprint of all composite primary key indices can be reduced by up to 36% (or 148 GB).

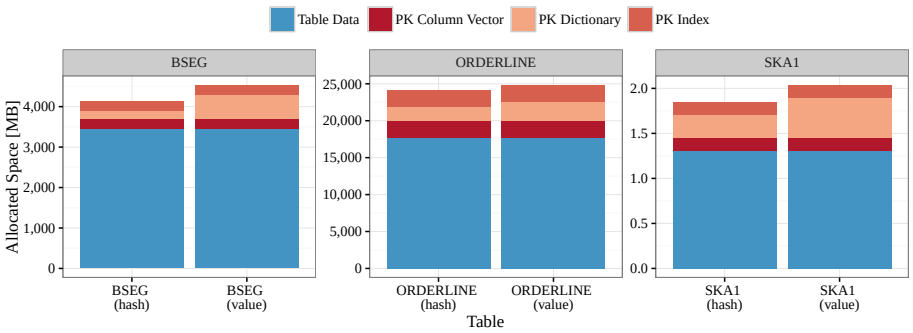


Fig. 4. Break down of memory consumptions for benchmarked tables.

The potential space savings depend on the characteristics of the primary key. Larger keys (i.e., longer concatenations of attribute values) result in larger savings when compressed to 10-byte hashes than smaller keys. Further, with increasing share of primary key columns compared to the total number of columns,

the potential space savings of the whole table increase as well. Figure 3(b) illustrates the high impact, the size of the primary key has on memory savings.

Figure 4 shows a breakdown of the memory used by the three benchmarked tables. As discussed, the total memory savings by using the hash-based index depend on both the number of attributes of the primary key and on the data types of the attributes. For the `BSEG` table with five `varchar` attributes, the footprint reduction for the whole table is around ∼10% due to a 3× smaller dictionary. The reduction of the `ORDERLINE` table with four integer attributes is smaller with ∼1% for the whole table. To assess hash-based indices for SAP ERP systems it is important to know that the majority of primary key attributes are of type `varchar`. After analyzing the number of primary key attributes in all large tables of our ERP system (depicted in Figure 1(a)), we saw that most of these large tables have primary keys with four or more attributes. By using the hash index instead of the value-based index we estimate a footprint reduction of the whole ERP system by 10%.

5.2 Lookup Performance

We analyzed the latency for three kinds of select queries, all of which are typical for OLTP workloads. We discard OLxP and OLAP queries, because they are usually not accessing primary key indices.

Full Primary Key Selects A full primary key select describes a lookup query that filters on all attributes of the composite primary key and therefore returns a single record or an empty result set. Our benchmark script executed 10,000 queries per thread and measured the end-to-end latency from sending the query till receiving the data records. The results are shown in Figure 5. For full primary key selects, we saw a latency increase between 5-15% for hash-based indices.

Partial Key Selects Partial key queries describe SELECT statements that select on a true subset of the primary key attributes. These queries are very common in real-world applications and in particular in ERP systems. We modified the full primary key queries to not select on the last attribute of the primary key (e.g., `ORDERLINE.OL_NUMBER`).

As mentioned in Section 4.4, hash-based indices are not accessed for queries selecting anything but the complete primary key. For those queries, the single-column indices created on each primary key attribute are accessed instead. The query latencies are shown in Figure 5. Depending on the size of the table, the hash-based index in on par with the value-based index (`SKA1` table) or is clearly outperformed by up to two orders of magnitude (`ORDERLINE` table).

Range Queries As a third reading access pattern, we evaluated range queries. Similar to partial key selects, range queries select on a subset of the primary key attributes but additionally execute a range selection (e.g., `ORDERLINE.OL_NUMBER`

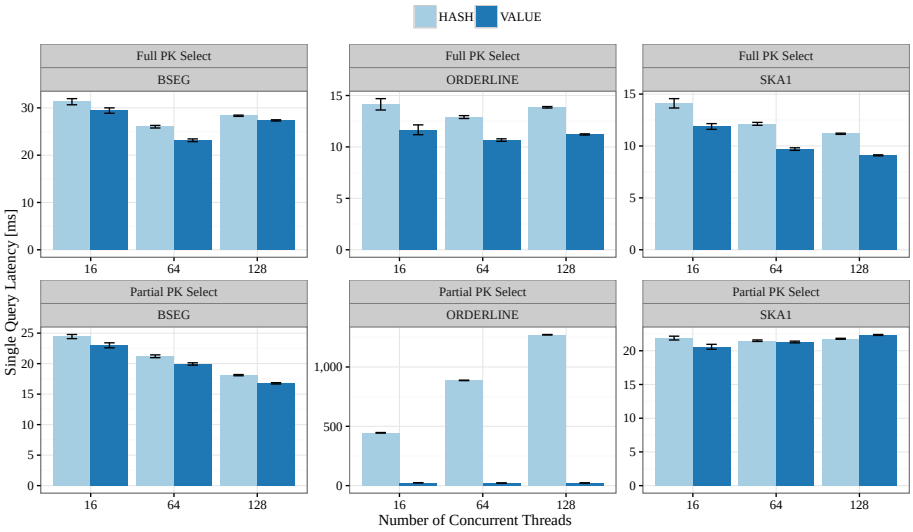


Fig. 5. Latency comparison of full and partial primary key selections.

> 10 AND ORDERLINE.OL_NUMBER < 20). We select all rows with BSEG.BELNR and ORDERLINE.OL_NUMBER in a specified range. The size of the ranges was set to return 100 tuples on average.

As mentioned before, hash-based indices cannot be used for range queries. That means, that we are again testing the performance of the additional single columns indices compared to direct binary searches on the dictionary of the value-based index. Similar to the partial select, the performance is depending on the size of the table with decreasing performance for increasingly large tables (see Figure 6).

5.3 Insert Performance

We measured the insert latency on the following three synthetic tables.

SYNTH3: a table with three attributes (varchar and two integers), all are part of the primary key.
SYNTH8: a table with eight attributes (three varchars and 5 integers), all are part of the primary key.
SYNTH100: a table with 100 attributes (30 integers, remainder varchars) of which eight are primary key columns (similar to SYNTH8).

All three tables contain 100 M tuples at the beginning of each test run. The results are shown in Figure 7. The graphs show that the hash-based index is on par performance-wise with the value-based index for a variety of insert scenarios.

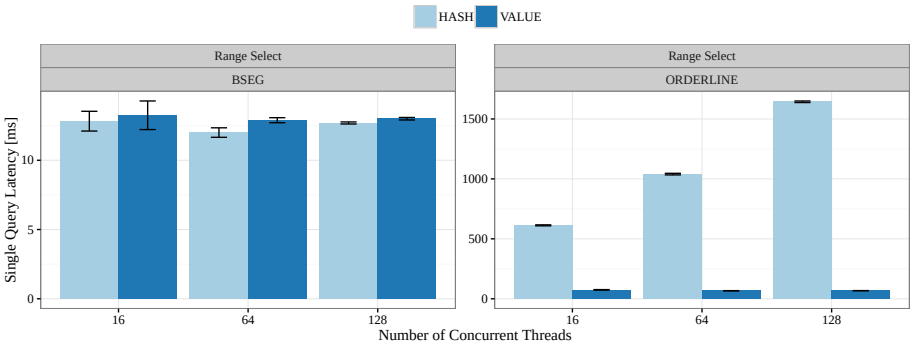


Fig. 6. Latency comparison of range selections.

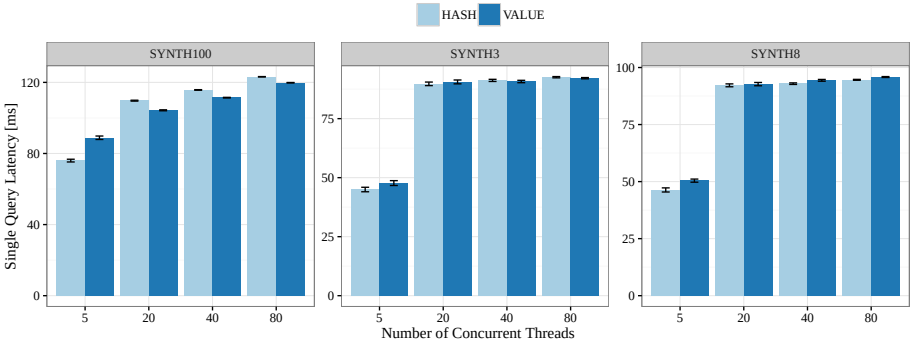


Fig. 7. Comparison for INSERT operations on synthetic tables (100 M tuples) with varying widths.

5.4 Applicability on Enterprise Workloads

The analysis of enterprise system workload by Krueger et al. [9] has shown a trend towards read-dominated workloads. Contrary to benchmarks like TCP-C, OLAP as well as OLTP workloads in modern enterprise applications consist of mostly read queries. Further, applications optimized for a column-based architecture and without materialized aggregates as in SAP's *simplified Financials* (sFIN) applications emphasize that trend [15]. The analysis of the sFIN workload, which is illustrated in Figure 8, has shown that over 98% of the application's total execution time is spent on read queries. 14% of the total time are spent on primary key selects while the remaining 84% are more complex select queries like joins and aggregations. Insert statements only account for 1.3% of the total execution time.

We estimate the overall impact of hash indices based on the analyses in Section 5 to be rather low from a performance perspective. With the exception of range queries, the hash-based indices perform on par with the value-based indices for OLTP-like queries. Since the share of range queries on the (partial) primary key is rather low, the performance drop is neglectable. With an increasing share of complex and computation-intensive OLxP and OLAP queries in future systems, the performance of the primary key will have a decreasing impact. Especially since query run times are often bound to the calculation of aggregates rather than bound to the selection.

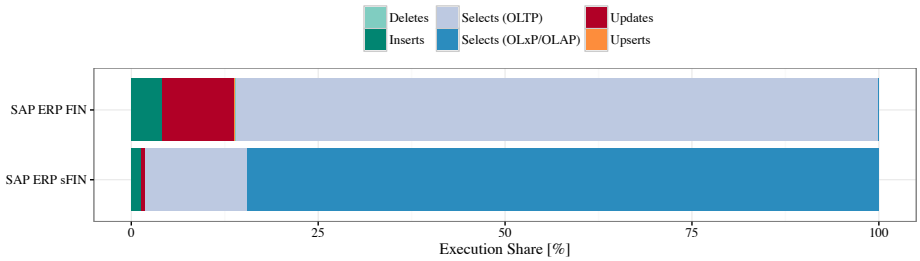


Fig. 8. Workload analysis: accumulated execution time of query types in a live production SAP ERP system.

From a main memory footprint perspective, hash-based indices provide a clear advantage over value-based indices for the current system with the dominance of `varchar` columns. In case many of the current `varchar` columns will be converted to numeric columns (their actual value domain) in the future, which is also advisable for query performance and compression, the potential footprint reduction by introducing hash-based indices will be significantly smaller (compare table `ORDERLINE` with integer attributes in Figure 4). In that case, the composite group-key is a viable alternative (see Section 3).

6 Conclusion

Hash-based primary key indices can be used to reduce the main memory footprint of an enterprise application while maintaining the level of performance for typical OLTP query patterns. We saw that footprint reductions and performance of hash-based indices depend on the characteristics of the tables they are applied to and the workload of the application. For recent enterprise systems optimized for column-based architectures, we expect a comparable performance when using hash-based indices over value-based indices while decreasing the entire main memory footprint by 10%.

References

1. Anastassia Ailamaki et al. DBMSs on a modern processor: Where does time go? In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, 1999*, pages 266–277, 1999.
2. Vo Ngoc Anh and Alistair Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8(1):151–166, 2005.
3. Manos Athanassoulis and Anastasia Ailamaki. BF-Tree: Approximate tree indexing. *Proceedings of the VLDB Endowment*, 7, 2014.
4. Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H Raymond Strong. Extendible hashing—a fast access method for dynamic files. *ACM Transactions on Database Systems (TODS)*, 4(3):315–344, 1979.
5. Franz Färber et al. SAP HANA database: data management for modern business applications. *ACM Sigmod Record*, 40(4):45–51, 2012.
6. Martin Faust, David Schwalb, and Hasso Plattner. Composite group-keys. In *In Memory Data Management and Analysis*, pages 139–150. Springer, 2015.
7. Vinodh Gopal et al. Fast CRC computation for iSCSI Polynomial using CRC32 instruction. Technical report, Intel Corporation, 2011.
8. Donald Ervin Knuth. *The art of computer programming: sorting and searching*, volume 3. Pearson Education, 1998.
9. Jens Krueger et al. Fast updates on read-optimized databases using multi-core cpus. *Proceedings of the VLDB Endowment*, 5(1):61–72, 2011.
10. Per-Ake Larson. Linear hashing with separators—a dynamic hashing scheme achieving one-access. *ACM Transactions on Database Systems (TODS)*, 13(3):366–388, 1988.
11. Tobin J Lehman and Michael J Carey. A study of index structures for main memory database management systems. In *Conference on Very Large Data Bases*, volume 294, 1986.
12. Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 38–49. IEEE, 2013.
13. Witold Litwin. Linear hashing: a new tool for file and table addressing. In *VLDB*, volume 80, pages 1–3, 1980.
14. Stefan Manegold, Martin L Kersten, and Peter Boncz. Database architecture evolution: mammals flourished long before dinosaurs became extinct. *Proceedings of the VLDB Endowment*, 2(2):1648–1653, 2009.
15. Hasso Plattner. The impact of columnar in-memory databases on enterprise systems. *Proceedings of the VLDB Endowment*, 7(13), 2014.
16. Kenneth A Ross. Efficient hash probes on modern processors. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 1297–1301. IEEE, 2007.
17. Lefteris Sidirourgos and Martin Kersten. Column imprints: a secondary index structure. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 893–904. ACM, 2013.