Composite Group-Keys

Space-efficient Indexing of Multiple Columns for Compressed In-Memory Column Stores

Martin Faust, David Schwalb, and Hasso Plattner

Hasso Plattner Institute for IT Systems Engineering
University of Potsdam, Germany
firstname.lastname@hpi.de

Abstract. Real world applications make heavy use of composite keys to reference entities. Indices over multiple columns are therefore mandatory to achieve response time goals of applications. We describe and evaluate the Composite Group-Key Index for fast tuple retrieval via composite keys from the compressed partition of in-memory column-stores with a main/delta architecture. Composite Group-Keys work directly on the dictionary-encoded columns. Multiple values are encoded in a native integer and extended by an inverted index. The proposed index offers similar lookup performance as alternative approaches, but reduces the storage requirements significantly. For our analyzed dataset of an enterprise application the index can reduce the storage footprint compared to B+Trees by 70 percent. We give a detailed study of the lookup performance for a variable number of attributes and show that the index can be created efficiently by working directly on the dictionary-compressed data.

1 Introduction

Today's hardware is available in configurations and at price points that make in-memory database systems a viable choice for many applications in enterprise computing. We focus on columnar in-memory storage with a write-optimized delta partition and a larger read-optimized main partition. This architecture supports high performance analytical queries [12, 2], while still allowing for sufficient transactional performance [5]. The results from an analysis of all primary keys of a large enterprise resource planning (ERP) system installation provide the input for the evaluation of different indexing techniques. The Composite Group-Key index is built on top of multiple dictionary-encoded columns by storing compact key-identifiers derived from the encoded representation of the key's fields. The key-identifiers maintain the sort order of the tuples and therefore, the index supports range lookups, which have a significant share in enterprise workloads [5].

Applications use composite keys to model entities according to their real world counterpart and the relationships between them. Redesigning database schemata to avoid the usage of composite keys is cumbersome and often contrary

to the goal of achieving a good abstraction of entities. To avoid the high costs of composite keys, database designs might use surrogate keys. However, the introduction of surrogate keys brings new problems, such as a disassociation of the key and the actual data and problems of uniquely referencing entities, among others. This is also visible in industry benchmarks like the TPC-C: of the nine tables in the TPC-C schema, seven have a composite key, two thereof have additional, secondary composite indices. TPC-H's largest table *lineitems* has a composite key as well. Consequently, nearly all row-based relational database systems support composite indices. Looking at the internal record based storage scheme of row stores, the support for composite indices is a straightforward extension of the single attribute index. The primary key is often automatically set as the cluster key of of the table, e.g. it establishes the sort order of a table on disk.

In-memory column stores with a main/delta architecture like Hyrise [4] and SAP HANA [12] keep the majority of the data in highly compressed, read-only partitions. Therefore, an additional index on record-level on such partitions can impose a significant part of the overall storage consumption of a table. To maintain a high query performance, the main and delta partition is combined into a new compressed main partition whenever the delta partition grows too large. To keep this merge process simple and fast and the compression scheme flexible, we do not consider the tables to be kept in the sort order of the primary key [5]. Consequently, a separate index structure is needed to enforce uniqueness constraints and fast single tuple access.

In the following sections, we describe the Composite Group-Key Index and benchmark it against alternative indexing schemes for the dictionary-compressed main partition of in-memory column stores with regard to their storage consumption and applicability in a real world enterprise application. We show that the lookup performance of Composite Group-Keys can keep up with alternative implementations while imposing a significantly smaller space overhead. A detailed analysis of a large enterprise application with several thousand tables and billions of records shows its applicability and limitations.

2 Real World Enterprise Application: SAP ERP

An Enterprise Resource Planning (ERP) application is the central planning software for large companies. It typically stores all invoices, sales orders, deliveries, and general ledger documents, and the connections between them, among other relevant data. We had the opportunity to obtain a complete system copy from a large, productive installation of the SAP ERP application from a Fortune 500 company. Although the analysis of a single instance of the product does not cover the entire ERP market, we believe that the findings are valuable and applicable to a larger scope of enterprise applications. The SAP ERP software has about 25 percent market share in the global ERP market and is used by more than half of the Fortune 500 companies. We verified the results from selected tables in a second instance of the application that is used in a different industry. The

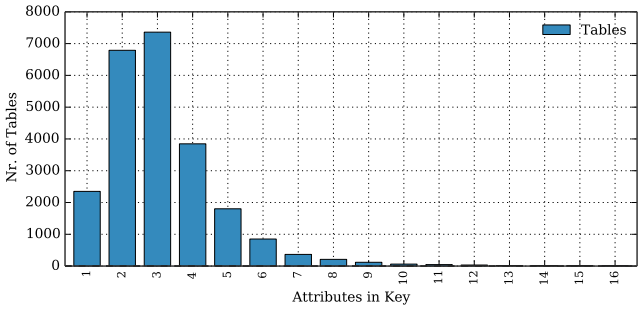

Fig. 1: Analysis of the data in a ERP system from a Fortune 500 company.

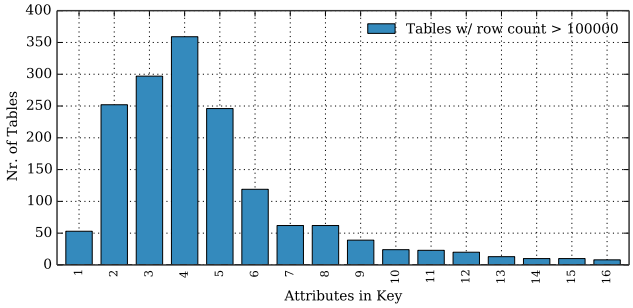

Fig. 2: Number of attributes in primary keys of large tables.

analyzed ERP system's size is about 5 TB in uncompressed format, it stores 10 billion records in 23886 tables. Each table has a primary key, which is usually a composite key. As an example, the general ledger accounting header's key is composed of the tenant-id, a company code, the document number and the fiscal year. Figure 1 shows number of tables in our database system grouped by the number of attributes in their key, in Figure 2 a detailed view of the tables with more than 100,000 records is presented. Only 2350 of the 23886 non-empty tables have a primary key of only one attribute, 6789 with two attributes and 14747 have composite keys of three or more attributes. If only the tables with more than 100,000 records are taken into account, 96 percent have a composite key, 81 percent with three or more fields.

Since the application is designed as a multi-tenant system with the tenant-id as the first key in all transactional tables, two keys are the norm. However, even if multi tenancy is implemented on a different layer, there are many more composite keys of higher order. The important finding of the analysis is, that more than 90 percent of the tables have a composite primary key.

	Symbol		Symbol
Table Length	n	Position List	P
Key-Identifier List	K	Concatenated Key	c
Attribute Vector of Column x	AV_x	Dictionary of Column x	D_x
Column x	C_x	Key-identifier	k_{id}

Table 1: Symbols

3 Composite Group-Key

The Composite Group-Key is our proposal for indexing the main partition of in-memory column stores with dictionary compression. Table 1 summarizes the used symbols.

The dictionary compression on the main partition uses sorted dictionaries (D) and bit-packed attribute vectors (AV). We refer to the compressed representation of a value, its bit-packed index into the dictionary, as value-id. Because all dictionaries of the main partition are sorted, the value-ids follow the same sort order as the original values and the value-ids of one column can be directly compared to evaluate the order between values. Therefore, range queries can also be evaluated directly on the compressed format.

The composite Group-Key contains two data structures: a key-identifier list K and a position list P. The key-identifier list contains integer keys k_{id} which are composed of the concatenated value-ids of the respective composite key's values. The bit-wise representation of k_{id} equals the concatenation of the value-ids of the keys fields, as illustrated in Figure 3(b). The creation of key-identifiers can be implemented efficiently through bit shifts.

The key-identifiers are similar to BLINK's data banks, but as they are composed of fixed-length values, they are binary-comparable across the complete main partition [9]. In the successor, DB2 BLU [8] indices are only used to enforce uniqueness constraints. Best practice guides advice to disable constraint checking, as the B+Tree organized indices consume space and introduce processing overhead [1].

Storage Requirements The Composite Group-Key maintains two data structures, the key-identifier list K with either 8,16,32 or 64 bits per indexed key and a bit-packed position list P. K is always composed of native integer datatypes, to avoid costly bit un-packing during the binary search. P is only accessed to retrieve the respective row-id, hence it is stored with $\lceil \log_2 n \rceil$ bits to save memory space.

[1] Rockwood et al.:*Best practices: Optimizing analytic workloads using DB2 10.5 with BLU Acceleration May 2014* on IBM.com

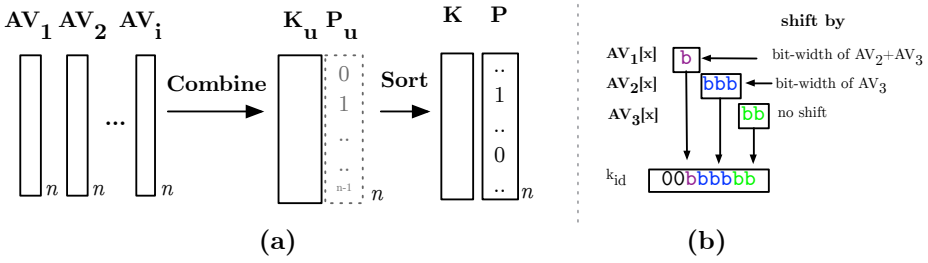

Fig. 3: Composite Group-Key creation: (a) schematic overview, (b) k_{id} creation.

$$K_x = n * \frac{x}{8}\ bytes \mid \text{x in } \{8,16,32,64\} \tag{1}$$

$$P = \frac{\lceil log_2(n) \rceil * n}{8}\ bytes \tag{2}$$

$$Memory_{Comp.GK} = K_x * n + P \tag{3}$$

Key Lookups The first step of the lookup with the Composite Group-Key Index consists of the translation of all key attributes of the predicate to their respective value-id, using binary search on each key attribute's dictionary. The complexity of each dictionary lookup is $\mathcal{O}((\log |Dict|) * k_i)$, with k_i being the length of the respective key attribute. Afterwards the key-identifier is created by concatenating the value-ids through bit shifts. The search key is used for a binary search on the key-identifier list, which is within $\mathcal{O}(\log n)$. The results, the matching row-id, can be read directly from the offset in P.

Index Creation The process of creating the index is shown schematically in Figure 3 and by example in Figure 4.

In the first step, value-ids from all columns of the composite key are combined to a vector of key-identifiers (K_u). This intermediate data structure is extended by an ascending list of row-ids (P_u). Afterwards both structures are sorted according to the key-identifiers to obtain K and P.

The appropriate native integer type for the key-identifier list is calculated by adding up the length of the value-ids of all indexed attributes and rounding up to the next power of two.

4 Alternative Index Implementations

This section briefly introduces two alternatives for secondary indexing of multiple columns. Both allow the efficient execution of single key and range queries.

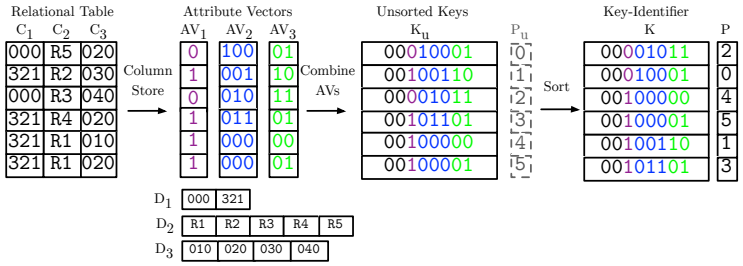

Fig. 4: Composite Group-Key creation: Example with 8 bit integer key-identifier.

However, they index the full composite key, instead of a shorter integer representation. Our goal is to show, that it is viable to transform the key into its compressed representation, although a binary search on each dictionary is necessary before searching for the actual key-identifier.

4.1 Tuple-Based B+tree

A classic implementation of an index stores pairs of the actual composite key and a row-id in a tree structure. Since the tree stores the uncompressed keys, no additional dictionary lookups have to be performed upfront, and the search takes place directly on the tree. The drawback of this approach is the need for expensive comparisons of the actual composite key while traversing the tree and its higher storage requirements (roughly 2x the data [11]) for internal pointers. Newer trie-based structures, such as the Generalized Prefix Trees proposed by Boehm et al. [1] and further developed in the Adaptive Radix Tree (ART) by Leis et al. [6] address some of the problems that classic B+Trees have. However, also tries require the replication of keys in the index and additional space for auxiliary structures.

For a basic performance comparison, we use the STX B+Tree library[2], a drop in STL map replacement, which is optimized for modern CPUs and more storage efficient than the GNU STL red-black trees. C++ tuples of char-arrays are used to store the key. The number of attributes in the key is a template parameter, i.e. there is no additional runtime overhead to determine the number of keys.

Storage Requirements For our comparison we ignore the internal overhead of the B+Tree's structures, and only assume that the indexed keys are replicated once into the tree structure, and an additional 8 bytes for the row-id pointer are stored. The resulting value is a lower-bound for any indexing scheme that replicates the keys into the index structure without further compression of the

[2] http://panthema.net/2007/stx-btree/

keys or row-id pointers.

$$Memory_{B+Tree} = (c + 8) * n \text{ bytes} \tag{4}$$

Key Lookups To find the corresponding row-ids for a predicate on the composite key, the key's attributes are concatenated to a single search key. In our implementation a fixed-length byte-array is indexed. To search the index for matches, the byte-array has to be created from the query predicates. Then, a search on the tree is performed and the row-id is read from the leaf. Let k be the length of the composite key, e.g. the sum of the length of all attributes that form the key. The complexity of building the key is within $\mathcal{O}(k)$ and the actual search on the index within $\mathcal{O}(log(n) * k)$, since the key comparison is in itself a $\mathcal{O}(k)$ operation.

4.2 Concatenated Attribute with Inverted Index

An alternative implementation to index composite primary keys adds an additional column to the table. The additional column holds concatenated values of all key attributes. It is extended by an inverted index to allow for fast tuple retrieval through the concatenated key. This essentially creates a clustered in-memory row store for the vertical partition of the composite key, and allows other database operations, like joins and aggregations, to work on the single concatenated column instead of handling multiple columns. Its integration into existing analytical column store engines without indices promises to be feasible with less effort than the introduction of new data structures and operators. If the key is composed of fixed length fields, the concatenated values follow the same sort order as the original values, otherwise a specialized encoding scheme has to be employed to support range queries. If a primary key is indexed, the resulting column has 100 percent distinct values and the the dictionary is essentially an uncompressed representation of the Composite Group-Keys key-identifier list.

Storage Requirements The concatenated key column consists of a sorted dictionary of string-keys (D), the attribute vector (AV) and a bit-packed position list (P). For primary keys the resulting key column has 100 percent distinct values, therefore we avoid adding a level of indirection [3] to cope with differently sized position lists, and instead store the positions directly in P. The differences to the B+Tree lower-bound stem from the bit-packed row-ids, an optimization that is only possible, if row-ids are stored consecutively. The resulting size is dominated by the dictionary, which is further compressed in practice. Mueller et al. [7] inspect the compression of the dictionary, and report compression factors between two and eight [7]. We show the results of the uncompressed column, as well as with a dictionary compression factor four.

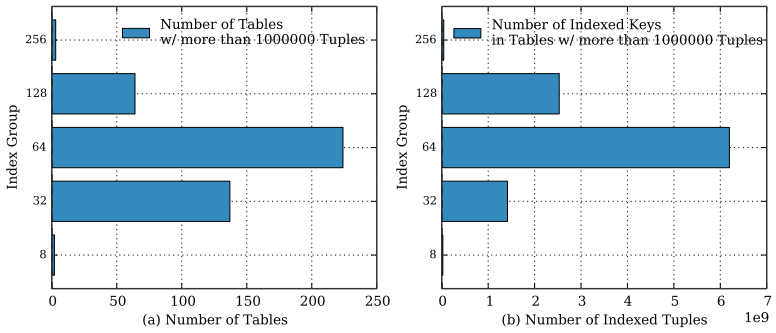

Fig. 5: Large tables from the ERP system by their respective Composite Group-Key class.

$$AV_{Concat} = \frac{\lceil log_2(n) \rceil * n}{8bit} \; bytes \tag{5}$$

$$D_{Concat} = n * sizeof(c) \tag{6}$$

$$P = \frac{\lceil log_2(n) \rceil * n}{8} \; bytes \tag{7}$$

$$Memory_{Concat} = AV_{Concat} + D_{Concat} + P \tag{8}$$

$$Memory_{CompressedConcat} = AV_{Concat} + 0.25 * D_{Concat} + P \tag{9}$$

Key Lookups A predicate on the key columns is translated to the concatenated version of the composite key by the query processor, similar to query processing with B+Trees. Next, a binary search for the concatenated key is performed on the concatenated column's dictionary. The respective row-id is obtained from the inverted index through a direct offset lookup in constant time. The lookup complexity is equal to the B+Tree lookup.

5 Evaluation

We compare the different indices with regard to the storage requirements, lookup performance, and index rebuild costs.

5.1 Storage Requirements of ERP Primary Keys

We use the insights from the ERP dataset analysis to compare the expected storage footprints of the Composite Group-Key Index and the presented alternatives.

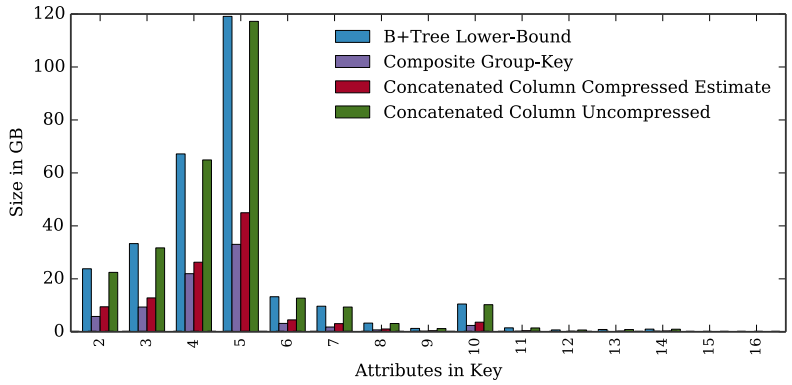

Fig. 6: Calculated index sizes grouped by attributes in key for the 23800 tables of the analyzed ERP system where the Composite Group-Key is applicable.

To evaluate the applicability of our proposed Composite Group-Key index we calculate the size of the key-identifier for all tables: Figure 5(a) shows the aggregated counts of the tables that are found in the system and have more than one million rows, and Figure 5(b) the indexed tuples within these tables. It highlights the importance of the 32 bit and 64 bit index cases, however, 86 tables of the analyzed dataset would need a 128 or 256 bit key-identifier, if the Composite Group-Key is applied. We focus on the configurations in which an native integer type is sufficient and leave the other cases for future work. Nevertheless, tables that use the Composite Group-Key can still grow at runtime, without leading to problems: as the size of the key-identifier is known at merge-time, the decision to use the Composite Group-Key can be safely made for each table. The limitations cannot be hit during normal query processing, i.e. during insertions or updates, but only when a re-encode of the main partition occurs during the merge process.

The total memory footprint of all primary keys in the ERP dataset is 287 GB for the calculated B+Tree lower-bound, 278 GB for the concatenated attribute, 108 GB for the estimate of compressed concatenated column, and 79 GB for the Composite Group-Key Index. The Composite Group-Key has a memory footprint advantage of about 70 percent less than the lower-bound of B+Trees and the uncompressed concatenated attribute. Even with an assumed compression factor of 4 for the dictionary of the concatenated attribute, the Composite Group-Key still leads to a 30 percent reduction. The storage footprint of the concatenated column and the Composite Group-Key are equal at an assumed compression factor of eight for all concatenated dictionaries.

In Figure 6, we compare the resulting index sizes of the Composite Group-Key Index and the other indexing schemes grouped by the number of fields in the composite key. It shows the storage savings of the Composite Group-Key

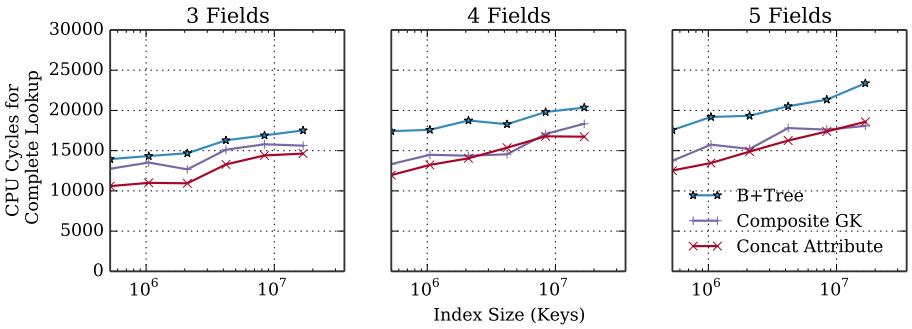

Fig. 7: Uncached Performance of Lookups. The CPU cache has been cleared between each access.

compared to the presented alternatives. It highlights that most savings in the ERP system can be made in keys with 4 and 5 attributes.

5.2 Lookup Performance

We benchmark the performance of key selects via the index. For each of the introduced indices we randomly pick 100 keys and report the average access time in CPU cycles. The benchmarks include the complete predicate-to-result translation, e.g. in case of the concatenated attribute the predicates are copied to create the char-array search key. For the Group-Key Index a binary search on each dictionary is performed. All measurements were performed on an Intel Core i5-3470 3.2GHz CPU with 8 GB RAM running Ubuntu 13.10 and using the GCC 4.8.1 compiler. The results are plotted for three to five attributes in the key in Figures 7 and 8. In Figure 7, the lookup performance of a single, uncached access to the index is reported. The three index types show a similar performance, with a minimal penalty for the B+Tree. Figure 8 reports the results for 100 consecutive index accesses to different values without any forced CPU cache invalidation. Here, the smaller size of the Composite Group-Key is beneficial for cache locality, and it outperforms the alternatives consistently. We conclude that the Composite Group-Key's performance is on-par with other established indexing schemes.

5.3 Index Creation and Maintenance

To keep the delta partition small and fast, its contents are merged from time to time into the main partition [5]. Only at *merge time*, the main partition index has to be maintained, as all other write operations during runtime are handled by the delta partition and a special invalidation vector of the main partition.

During the merge process, the delta and main partition are combined to a new main partition, thereby potentially changing the value-ids of every value in

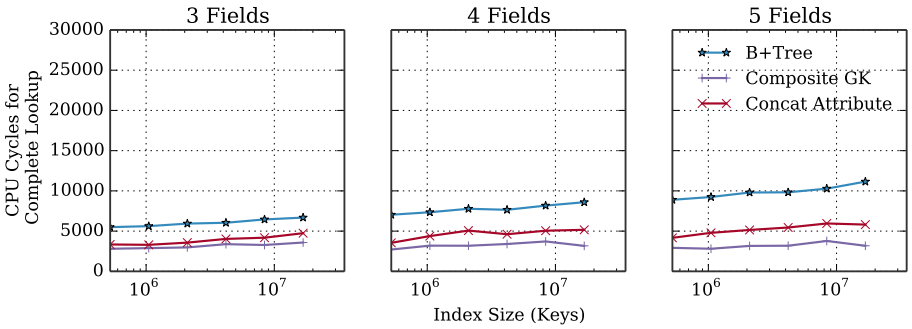

Fig. 8: Cached Performance of Lookups. Same experiment as in Figure 7 but without invalidation of the CPU cache between runs.

the former main store [5]. Additionally, the merge process handles a column at a time, making in difficult to handle composite keys, as multiple columns have to be considered. The merge process runs concurrently to transactions, hence, the current index cannot be modified in-place. Therefore, after the merge process created a new main partition, a new index is built from scratch.

This works for all index types, but as Figure 9 shows, the costs vary. The high costs for the concatenated attribute are due to the expensive byte-wise operations on all values, especially the sorting to create the inverted index. The B+Tree shows better performance due to the better cache locality during the sort. The Composite Group-Key outperforms the two alternative, since it does not work on byte-arrays, but native integers. Since K is a vector of integers, the sorting operation is much faster than the respective sorting of char-arrays.

6 Conclusion and Future Work

We showed the importance of composite keys and proposed a novel index structure tailored towards dictionary encoded column-stores with a main/delta architecture. The Composite Group-Key's lookup performance is on par with other established indexing schemes while significantly reducing the storage footprint for a variety of real world tables. Its implementation leverages the encoding of the primary data by encoding value-ids instead of values. It can therefore avoid costly byte-wise comparisons and perform the comparison of multiple parts of the key in a single integer comparison. Although the Composite Group-Key's lookup complexity suggests that a lookup operation is more costly than in the other cases, its actual performance on modern CPUs keeps up with the alternatives. It is a viable choice to use the compressed representation of a key to perform fast single-tuple lookups in in-memory column-stores with a main/delta architecture.

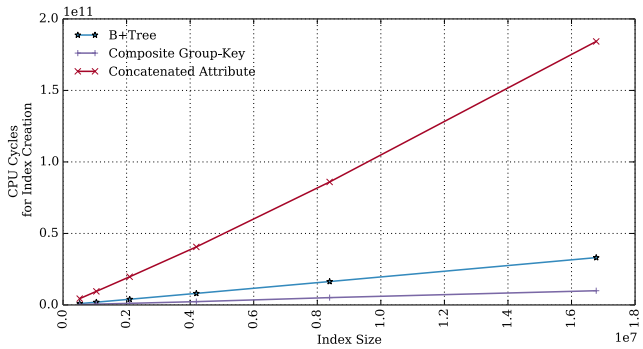

Fig. 9: Index creation performance for different main partition sizes

In future work we plan to evaluate how additional optimizations, such as storing the key-identifier list as a CSS tree [10] or trie compare in this setting. Bit-packing row-ids in tree leaf nodes is another option to reduce the memory footprint of tree structures. Additionally, clustered indices can be applied to our columnar in-memory storage engine. The binary search on the sorted compressed columns is similar to the Composite Group-Key lookup, since the predicate needs to be translated to the compressed representation as well, before the search on the partition can be performed. Nevertheless, additional index structures could improve search performance on the sorted table.

References

1. M. Böhm, B. Schlegel, P. B. Volk, U. Fischer, D. Habich, and W. Lehner. Efficient in-memory indexing with generalized prefix trees. In T. Härder, W. Lehner, B. Mitschang, H. Schöning, and H. Schwarz, editors, *BTW*, volume 180 of *LNI*, pages 227–246. GI, 2011.
2. F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Record*, 40(4):45–51, 2011.
3. M. Faust, D. Schwalb, J. Krueger, and H. Plattner. Fast Lookups for In-Memory Column Stores: Group-Key Indices, Lookup and Maintenance.
4. M. Grund, J. Krueger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE—A Main Memory Hybrid Storage Engine. *VLDB '10*, 2010.
5. J. Krüger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast updates on read-optimized databases using multi-core CPUs. *PVLDB*, 5(1):61–72, 2011.
6. V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In C. S. Jensen, C. M. Jermaine, and X. Zhou, editors, *ICDE*, pages 38–49. IEEE Computer Society, 2013.
7. I. Müller, C. Ratsch, and F. Faerber. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. *EDBT*, 2014.

8. V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and L. Zhang. DB2 with BLU acceleration: so much more than just a column store. In *Proceedings of the VLDB Endowment*, pages 1080–1091. VLDB Endowment, Aug. 2013.
9. V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-Time Query Processing. In *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*. IEEE Computer Society, Apr. 2008.
10. J. Rao and K. Ross. Cache conscious indexing for decision-support in main memory. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.
11. J. Rao and K. A. Ross. *Making B+- trees cache conscious in main memory*, volume 29. ACM, June 2000.
12. V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in SAP HANA database: the end of a column store myth. In K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, editors, *SIGMOD Conference*, pages 731–742. ACM, 2012.