The Case for Hybrid Succinct Data Structures

Christoph Anneser Andreas Kipf Harald Lang Thomas Neumann Alfons Kemper
Technical University of Munich
{anneser,kipf,harald.lang,neumann,kemper}@in.tum.de

ABSTRACT

One of the biggest challenges in data management is to retain the high performance of in-memory processing with the ever increasing data volumes. Recent years have shown that the amount of collected data is increasing at a faster pace than DRAM capacities. Many state-of-the-art index data structures are optimized for performance rather than for low space consumption and quickly exceed the limited main-memory capacities when indexing larger data sets. *Succinct data structures* on the other hand allow for space efficient indexing, but compromise performance while still being orders of magnitude faster than disk-based data structures.

In this work, we propose a novel framework that combines state-of-the-art indexes with succinct data structures to form new *hybrid succinct data structures*. These hybrids enable fine-grained trade-offs between space and performance. Frequently accessed parts of an index, e.g. the upper levels of a tree, are thereby maintained in performance-optimized structures whereas less frequently accessed parts have a space-optimized representation. Our evaluation shows that our approach can significantly reduce the amount of used space by up to 50% (resp. 90% for our compressed version) while retaining 93% (resp. 87%) of the performance.

1 INTRODUCTION

Back in 2006, Jim Gray stated that memory is the new disk and disk is the new tape [7]. This also applies to modern database systems that store the entire data in random access memory (RAM) to allow near real-time analyses for trading companies and finance services. They need to efficiently process large datasets to react within a few milliseconds to new developments or updates.

To achieve the required performance for near real-time data processing, index structures such as B-trees, hash tables, tries, amongst others are used to efficiently find specific elements. In modern database systems, these index structures are also stored in main memory and are most often highly optimized in terms of performance and underlying hardware rather than space efficiency. However, while the main memory capacities tend to double every three years and its costs decrease by a factor of 10 every five years, the data collected by sensors, smartphones, social media platforms, and IoT-devices *increases at an even higher rate* resulting in data overflows [12]. This development requires in-memory data structures to optimize both performance *and* space.

In this paper, we focus on hierarchical indexes comprising nodes and relationships between them, such as trees, tries, and graphs, and distinguish two different types of data structures:

(1) **Pointer Based Data Structures** (PBDS): These state-of-the-art data structures model relationships between elements *explicitly* using `machine addresses`. Traversing to another node directly translates to a pointer resolution.
(2) **Succinct Data Structures** (SDS): These data structures encode relationships in the data *implicitly* using bitmaps and still allow accessing nodes in constant time [13]. While SDS tend to be smaller than PBDS, they are often slower since traversing requires more complex operations [6].

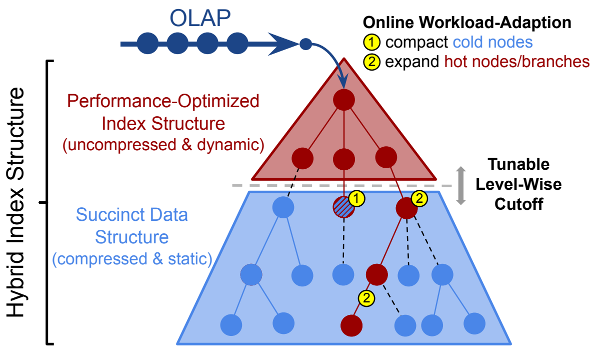
OLAP
Online Workload-Adaption
1 compact cold nodes
2 expand hot nodes/branches
Performance-Optimized Index Structure (uncompressed & dynamic)
Tunable Level-Wise Cutoff
Succinct Data Structure (compressed & static)
Hybrid Index Structure

Figure 1: We propose a novel framework that combines state-of-the-art indexes with succinct data structures. It allows indexing even larger data sets entirely in main memory by taking advantage of space-efficient succinct indexes. Furthermore, our hybrid index allows online adaptation to the actual OLAP workload by storing hot nodes in performance-optimized structures.

So far, only a few SDS-based approaches, such as succinct range filters [13] and tree-encoded bitmaps [9], have been successfully applied to database systems, as modern PBDS tend to offer much better performance. However, the explicit maintenance of relationships in PBDS also consumes more space. Using 64-bit addresses may introduce significant overheads, since pointers theoretically allow for differentiating between 2^{64} (more than 18 trillion) items, which is not required for most applications. When indexing larger datasets, SDS become more interesting for the case when PBDS do not fit into main memory anymore [6] and would require staging parts of the data structure to disk.

We propose a new, lightweight framework that takes advantage of both types of data structures and allows combining any hierachical PBDS and SDS to a new hybrid index that consumes less space than the PBDS and offers higher performance than the SDS. It also allows online workload adaptation for use cases, in which some elements tend to be more important than others and get accessed more frequently.

The rest of this paper is organized as follows: In Section 2, we present the foundations of succinct index structures as well as the Fast Succinct Trie [13] which is used in our evaluation. We also give a short overview of point-polygon joins, since our evaluation is based on this use case. In Section 3, we present a detailed overview of our approach. We evaluate our framework for the real-world use case of geospatial point-polygon joins in Section 4 and conclude with our next steps and future work in Section 5.

2 BACKGROUND

In this section, we present the foundations of succinct data structures and the Fast Succinct Trie, which is a succinct trie data structure that almost achieves the performance of uncompressed PBDSs [13]. We also discuss a state-of-the-art index for point-polygon joins where we applied and evaluated our framework.

Succinct Data Structures. A data structure is called *succinct* if its space is close to the information-theoretic optimum while most basic operations are still executable in constant time. In the literature, *close* is defined in different ways – we refer to a data structure as succinct if it uses $O(\text{opt})$ bits, with opt being the minimal number of required bits to represent the data and its relationships.

As stated in Section 1, instead of explicitly modeling relationships between elements using machine addresses, this information is encoded *implicitly* in bitmaps. Consider the trie data structure in Figure 2 where each level encodes *two bits* and keys *are not a prefix* of other keys. For a succinct encoding, we store two bitmaps *labels* and *hasChild* that encode for each *node label* (branch) 00_2, 01_2, 10_2, and 11_2 whether it exists and if there is a following child node. E.g., when labels$[4n+2]$ is set, it indicates that the n-th node contains the label 10_2. Then, all encoded nodes are concatenated in *breadth-first ordering* resulting in the above mentioned bitmaps (cf. labels and hasChild in Figure 2). We define $\mathtt{rank}(x)$ to count the number of set bits up and including position x, and $\mathtt{select}(x)$ to return the index of the x-th set bit:

$$\mathtt{rank}(x) = \sum_{i=0}^{i \le x} \text{hasChild}[i] \tag{1}$$

$$\mathtt{select}(x) = i, \text{ with } \mathtt{rank}(i) = x \tag{2}$$

Based on rank and select, more complex operations for a node n starting at position p can be defined. E.g., we can calculate the position for the child node at label/branch x (3), and we can find the position of n's parent node (4) (assuming *fixed-sized* nodes comprising w bits) [13]:

$$\mathtt{child}(x,p) = \mathtt{rank}(x+p) \times w, \text{ with } \text{hasChild}[x+p] = 1 \tag{3}$$

$$\mathtt{parent}(p) = \mathtt{select}(\lfloor p/w \rfloor) \tag{4}$$

As one can see in Figure 2, *six bytes* are sufficient to store the trie structure and its relationships (excluding the values). However, while succinct data structures optimize space, they tend to introduce higher latencies than PBDS since resolving a neighboring element involves multiple rank and select statements.

In 2018, Zhang et al. proposed a new data structure called Fast Succinct Trie (FST) [13]. It almost achieves the performance of state-of-the-art trie data structures such as the Adaptive Radix Tree (ART) [10] but needs less space and allows for efficient value compression. It internally uses a hybrid encoding scheme where upper levels are represented similarly to the trie in Figure 2, and lower levels store the data in a more compressed way by only representing branches that actually exist. In Section 3, we apply our framework to this data structure.

Efficient Point-Polygon Joins. We applied and tested our framework for the use case of point-polygon joins according to the approach introduced by Kipf et al. in 2020 [8]. *Dynamic points* get joined with a *static set of polygons* using prefix lookups on *one-dimensional* 64-bit keys. First, a given space (e.g. the minimum bounding rectangle of the polygons, cf. *green cell* in Figure 3) is recursively decomposed into smaller sub-cells. Then, the cells are

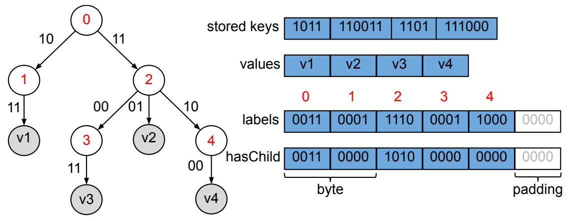

Figure 2: Succinct encoding of a trie with maximum fanout of four. The relationships between nodes are implicitly encoded in the *labels* and *hasChild* bitmaps in breadth-first order.

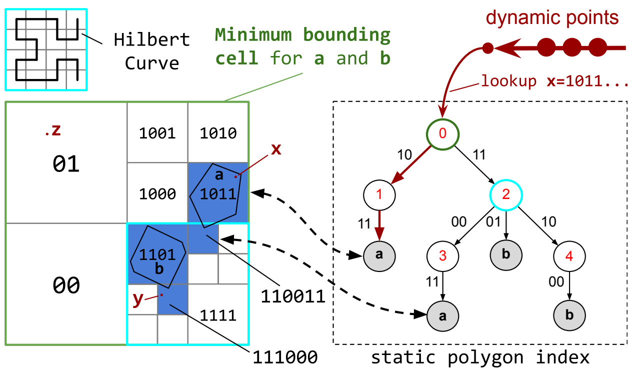

Figure 3: Left: space decomposition into quadtree cells that cover polygons a and b. Right: trie data structure indexing the blue cells level-wise and the exemplary lookup of point x.

enumerated (discretized) by a space-filling curve (e.g. the Hilbert curve) and can be identified by *one-dimensional* keys. Each decomposition divides a given cell into four smaller sub-cells for which reason we store two bits per cell level that uniquely identify one of the four sub-cells. Up to 31 levels can be addressed by a single 64-bit integer where the *least significant set bit* determines the encoded level. Applied to the Earth's surface, we can address every single square centimeter within 64 bits [2].

As a next step, we compute for each polygon p a so-called *covering*, which comprises a set of cells that cover p (cf. blue cells in Figure 3). A specialized pointer-based radix trie with fan-out four, called Adaptive Cell Trie (ACT), indexes the cells of *all combined coverings* (refer to [8] for more details). When joining an incoming point, we first transform the point to the smallest cell level (in this case 31) and then use ACT to find matching polygons using *level-wise prefix checks.* If a cell c_1=`0110` is prefix of another cell c_2= $\underline{\mathtt{0110}}\mathtt{1101}$, then c_2 is fully contained in c_1. E.g., the binary key for point z starts with `01` and we can already detect at ACT's root node *rn* that no polygon encloses the point since the label `01` is not present in *rn*. For point x (`1011...`), we find the enclosing polygon **a** at level two (cf. exemplary lookup in Figure 3). Point y is an example for a *false positive* match since it is not enclosed by any polygon, but querying ACT indicates that y could be contained by polygon **b**. By this means, this point-polygon join guarantees a precision which corresponds to the diagonal of the largest cell that is not completely enclosed by a polygon.

3 HYBRID SUCCINCT DATA STRUCTURES

We propose a new, lightweight framework that allows combining any hierarchical PBDS and SDS *level-wise* to significantly reduce the memory footprint, while retaining the performance at a large extent. *Online workload adaptation* supports *branch-wise* PBDS refinements for use cases with skewed workloads. Our framework comprises four steps to build a new hybrid index structure for a given dataset:

(1) Build the static, read-only SDS.
(2) Build the PBDS for the upper *n* levels.
(3) Connect the PBDS nodes at level *n* to the corresponding child nodes in the SDS (e.g. by using pointer tagging).
(4) Extend the PBDS interface by two operations to allow *branch-wise* online workload adaptation:
    (a) `expand(node)`: Frequently accessed nodes are encoded in the faster PBDS when a specified *threshold* is exceeded.
    (b) `compact(node)`: Colder nodes (under the *threshold*) are compacted, removed from PBDS and indexed in SDS only.

In our approach, we exploit the fact that real-world workloads tend to be *skewed* and therefore, we periodically evaluate the actual queries *at runtime* to determine frequently accessed nodes. For a node *n* whose accesses exceed a predefined threshold *t*, we call `expand(n)` to add *n* to the faster PBDS. In the case that the accesses to *n* precede *t*, we simply remove *n* from the PBDS.

Since *all queries* start at the upper levels of a hierarchical data structure, we encode the frequently queried upper levels *l* using the performance-optimized PBDS and connect it *level-wise* to the SDS. Furthermore, the upper levels represent only a small fraction of the overall data structure and encoding them as PBDS may not have a noticeable impact on the total size. In contrast to branch-wise refinements, level-wise cutoffs do not introduce additional branch misses since we do *not have to differentiate* between PBDS and SDS references *before* reaching level *l*.

We deliberately accept that common parts are stored redundantly as it allows the PBDS to become a *dynamic meta index structure* for the static SDS where *lightweight refinements* do affect only the dynamic PBDS. Despite the introduced redundancy, our framework focuses on *minimizing the memory overhead* while our secondary goal is keeping the *read overhead as small as possible*. In accordance with the RUM conjecture by Thanassoulise et al. [5], "there is always a price to pay for every optimization", as our approach does *not* handle updates efficiently so far.

To the best of our knowledge, this is the first approach that applies *tuneable level-wise* cutoffs combined with *branch-wise* workload adaptations to hybrid succinct index structures. In contrast to the proposed framework by Zhang et al. in 2016 [11], we *completely avoid searching redundant parts* of the key space by *directly* pointing from PBDS into SDS using pointer tagging. While Zhang's approach is optimized for OLTP workloads where recently inserted tuples are assumed to be accessed more frequently and therefore are kept in the dynamic structure, we do *not rely* on this assumption but rather adapt to the *actual workload at runtime* using fine-grained branch-wise refinements. Additionally, we connect PBDS and SDS *level-wise*, whereas Zhang et al. completely separate both indexes *tuple-wise*.

Prefix Lookups Based on ACT and FST. In the following, we explain our framework using the Adaptive Cell Trie [8] as PBDS and the Fast Succinct Trie [13] as SDS and apply them to the use case of point-polygon joins as introduced in Section 2.

Given: We start with a set of polygons and its coverings that were calculated according to the aforementioned approach in Section 2. The coverings contain *unique* cell keys where each cell key is represented by a `uint64_t` and for each *key*, the referenced polygon(s) are stored in an `uint64_t`-payload (*value*).

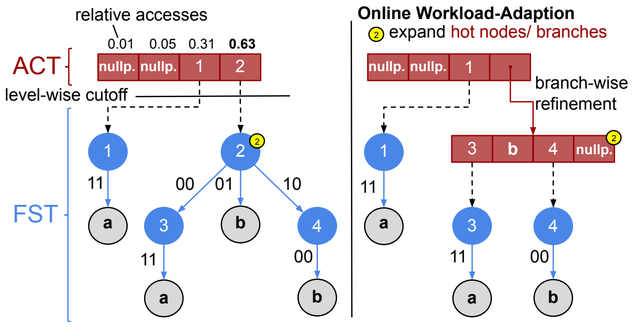

Figure 4: Initial hybrid trie with a threshold of 0.4 and ACT-encoded root node (left). Illustration of hybrid trie after expanding FST node 2 and indexing it in ACT (right).

Step 1: First, we adapted the FST to store *two bits per level* (instead of a *byte*) so that one trie level stores exactly one cell level (cf. Figure 3). The trie indexes the 64-bit cell keys *up to* the cell level while the remaining, unused levels are ignored.

In addition to our framework, we compressed the values using *run-length encoding*. In this use case, neighboring cells are likely to cover the same polygon(s) and thereby share the same polygon reference (value). These values occur directly one after the other in the values vector and offer a high compression potential.

Step 2: As PBDS, we use the Adaptive Cell Trie [8] where one trie level also stores exactly one cell level. Each ACT node comprises an array of four 64-bit *tagged pointers* resulting in an overall size of 256 bits per node. The left trie depicted in Figure 4 stores the cell keys {1011, 110011, 1101, 111000} and shows an ACT encoded root node while the remaining levels are encoded in the FST.

Step 3: We connect ACT and FST by inlining the required information in the ACT pointers. Since pointers do not use the entire 64 bits for memory addressing, we can use the two least significant bits to differentiate whether (i) the pointer stores a memory address for an ACT child node, (ii) a referenced polygon id (value) or (iii) an offset into the FST. In the case a label does not exist in a node, we just store a `nullpointer`.

Step 4: We extended the ACT interface by two functions `expand` and `compact` and stored four access counters for each node (cf. ACT-encoded root node in the left part of Figure 4). During runtime, we periodically determine the number of accesses and after a pre-defined number of lookups, we add those nodes whose access counters exceed a given threshold *t* to the *candidate set*. Then, we start expanding the candidate nodes and add them to ACT until the candidate list is empty or a given *memory bound* would be exceeded. E.g., node 2 in Figure 4 gets expanded since its relative access counter 0.63 exceeds the threshold of 0.4.

4 EVALUATION

We applied our framework to the use case of point-polygon joins (cf. Section 2). First, we will evaluate the performance of our hybrid trie and compare it against other data structures. Then, we will discuss the impact of workload adaptation on the overall performance and index structure size. Additionally to our approach, we will show a hybrid trie that compresses the values.

We use a real-world data set containing 289 neighborhoods (polygons) in New York City and join them with 1.23 billion publicly available taxi pickup locations (points) for the years 2009 to 2016 [4]. We calculate the cell coverings for the polygons as described in Section 2. Then, we use different index structures

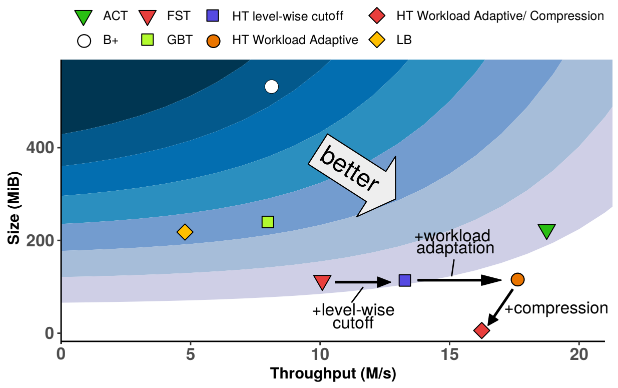

Figure 5: Evaluation of selected index structures considering space and performance. The 'waves' have been added for illustration purposes. Indexes in brighter areas have a better space/performance ratio.

that store the combined covering which contains approximately 14 million cells.

We conduct the experiments on a 14-core Intel Xeon E5-2680 v4 CPUs equipped with 256 GB DDR4 RAM and we compile with GCC 5.4.0 and optimization level 03. Besides ACT and FST, we compare our Hybrid Trie (HT) to the following data structures: the Google B-tree (GBT) [1], the STX B+ Tree (B+) [3], and the `std::lower_bound` algorithm (LB).

Comparing to Other Data Structures. In Figure 5, we show different data structures and their space consumption in MiB for indexing 14 M `uint64_t`-keys on the y-axis. Then, we query 1.23 B keys and denote the performance in M/s on the x-axis.

While LB achieves the lowest throughput (4.78 M/s), the *use-case optimized* ACT allows querying more than 18.75 M entries per second. The most space is used by the B+ tree (535 MiB) and the most space-efficient index structure is the workload-adaptive hybrid trie with enabled *run-length encoding* for the payloads. While GBT uses internal node compression techniques and B+ uses approximately twice the space, they achieve comparable performance (7.98 and 8.10 M/s).

As expected, the performance of the hybrid trie with *level-wise cutoff* (13.27 M/s) is located in between FST (10.09 M/s) and ACT (18.75 M/s), while the required space (113.71 MiB) is increased by a *negligible amount* of 0.007% compared to FST (113.70 MiB, ACT uses 223.65 MiB).

Analyzing Workload Adaptation. In Table 1, we depict the evaluation results for the *workload adaptive* hybrid tries with different level-wise cutoffs and thresholds t. A cutoff level cl means that the upper cl levels are encoded as ACT and the remaining levels are encoded as FST. A node n whose relative accesses *exceed* t gets expanded and added to ACT, whereas n gets removed from ACT if its relative accesses *precede* t. These updates can be performed periodically after a specified amount of time. Entries *without* a given threshold (-) refer to a *non workload-adaptive* hybrid trie. The ACT-encoded part has a *negligible influence* on the total HT size (1.01% for $cl = 11$) and with increasing cutoff levels, the performance impact of the workload adaptation decreases.

For the presented use case, online workload-adaptation works well since taxi pickup locations are skewed (e.g. there are many pickups at the airport and the main train station). The last column of Table 1 shows the percentage of queries that can be answered directly by the refined ACT without entering the FST.

Table 1: Space and performance metrics for the workload-adaptive hybrid tries. With increasing cutoff level and decreasing threshold, the performance increases significantly while the size overhead remains negligible.

Cutoff Level	Refinement Threshold [%]	Size Overhead To FST [%]	TP [M/s] (Perf. of ACT)	ACT-only Lookups [%]
1	-	0.001	10.32 (55%)	0.00
1	1	0.004	14.30 (76%)	31.25
1	0.5	0.008	14.58 (78%)	44.13
6	-	0.007	12.83 (68%)	0.41
6	1	0.009	13.42 (72%)	31.42
6	0.5	0.013	14.65 (78%)	44.30
11	-	1.020	17.28 (92%)	78.13
11	1	1.020	17.27 (92%)	78.13
11	0.5	1.020	17.28 (92%)	78.13

Further Compression Techniques. As discussed in Section 2, most succinct data structures store the payloads in a separate data structure which allows for further compression. We applied *run-length encoding* to the payloads which results in a compression ratio of 19,74 (114856 MiB / 5819 MiB) while the performance of 16.24 M lookups per second is still comparable to ACT. The hybrid trie uses only 2.6% of ACT's space while it retains 86.6% of its performance. By this means, the hybrid trie is *smaller by two orders of magnitude* while it achieves comparable performance to ACT. Note that our approach is not limited to run-length encoding but can of course be combined with any compression technique (e.g. dictionary encoding).

5 CONCLUSIONS AND FUTURE WORK

While our framework achieves good results for the presented use case, it is still an *ongoing work in progress*. As a next step, we will apply the framework to other use cases such as *prefix lookups for strings* and other index structures. We also plan to implement a duplicate-free hybrid trie and apply different compression techniques to the values.

REFERENCES

[1] Google C++ B-tree. https://code.google.com/archive/p/cpp-btree/.
[2] S2 Geometry. https://s2geometry.io/.
[3] STX B+-tree. http://panthema.net/2007/stx-btree/.
[4] *TLC Trip Record Data.* http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.
[5] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. Designing access methods: The RUM conjecture. In *Proc. of EDBT*, pages 461–466, 2016.
[6] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *SEA*, pages 326–337. Springer, 2014.
[7] J. Gray. Tape is dead, disk is tape, flash is disk, ram locality is king. http://research.microsoft.com/en-us/um/people/gray/talks/Flash_is_Good.ppt.
[8] A. Kipf, H. Lang, V. Pandey, R. A. Persa, C. Anneser, E. T. Zacharatou, H. Doraiswamy, P. Boncz, T. Neumann, and A. Kemper. Adaptive main-memory indexing for high-performance point-polygon joins. In *Proc. of EDBT*, 2020.
[9] H. Lang, A. Beischl, V. Leis, P. Boncz, T. Neumann, and A. Kemper. Tree-Encoded Bitmaps. In *Proc. of SIGMOD.* ACM, 2020.
[10] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *Proc. of ICDE*, volume 13, pages 38–49, 2013.
[11] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen. Reducing the storage overhead of main-memory oltp databases with hybrid indexes. In *Proc. of SIGMOD*, pages 1567–1581. ACM, 2016.
[12] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang. In-memory big data management and processing: A survey. *TKDE*, 27(7):1920–1948, 2015.
[13] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. SuRF: Practical range query filtering with fast succinct tries. In *Proc. of SIGMOD*, pages 323–336. ACM, 2018.