Learning Multi-dimensional Indexes

Vikram Nathan,* **Jialin Ding,*** **Mohammad Alizadeh, Tim Kraska**
Massachusetts Institute of Technology

Abstract

Scanning and filtering over multi-dimensional tables are key operations in modern analytical database engines. To optimize the performance of these operations, databases often create clustered indexes over a single dimension or multi-dimensional indexes such as R-Trees, or use complex sort orders (e.g., Z-ordering). However, these schemes are hard to tune and their performance is inconsistent across different datasets and queries. In this paper, we introduce Flood, a multi-dimensional in-memory read-optimized index that automatically adapts itself to a particular dataset and workload by jointly optimizing the index structure and data storage layout. Flood achieves up to three orders of magnitude faster performance for range scans with predicates than state-of-the-art multi-dimensional indexes or sort orders on real-world datasets and workloads.

1 Introduction

Scanning and filtering are the foundation of any analytical database engine, and the target of performance improvements over the past several years. Most importantly, column stores [7] have been proposed to delay or entirely avoid accessing columns (i.e., attributes) which are not relevant to a query. Similarly, many techniques skip over records that do not match a query filter. For example, transactional database systems create a clustered B-Tree index on a single attribute, while column stores often sort the data by a single attribute. The idea behind both is the same: if the data is organized according to an attribute present in the query filter, the execution engine can quickly narrow its search to the relevant range in that attribute. We refer to both approaches as clustered column indexes.

If data has to be filtered by more than one attribute, secondary indexes can be used. Unfortunately, their large storage overhead and latency due to pointer chasing make them viable only when the predicate on the indexed attribute has a very high selectivity; in most other cases, scanning the entire table can be faster and more space efficient [6]. An alternative approach is to use *multi-dimensional* indexes to organize the data; these may be tree-based data structures (e.g., k-d trees, R-Trees, or octrees) or a specialized sort order over multiple attributes (e.g., a space-filling curve like Z-ordering or hand-picked hierarchical sort). Indeed, many state-of-the-art analytical database systems use multi-dimensional indexes or sort-orders to improve the scan performance of queries with predicates over several columns [1, 5, 13, 6].

However, multidimensional indexes still have significant drawbacks. First, they are extremely hard to tune. For example, Vertica's ability to sort hierarchically on multiple attributes requires an admin to carefully pick the sort order. The admin must therefore know which columns are accessed together, and their selectivity, to make an informed decision. Second, there is no single approach (even if tuned correctly) that dominates all others. As our experiments will show, the best multidimensional index varies depending on the data distribution and query workload. Third, most existing techniques cannot be fully tailored for a specific data distribution and query workload. While all of them provide some tunable parameters (e.g., page size), they do not allow finer-grained customization for a specific dataset and filter access pattern.

To address these shortcomings, we propose Flood, the first learned multi-dimensional in-memory index. Flood's goal is to retrieve or aggregate records over a query filter faster than existing indexes, by automatically co-optimizing the data layout and index structure for a particular data and query distribution.

*Equal contribution

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

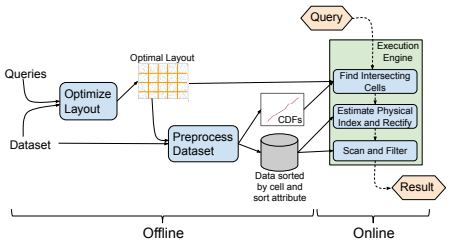

Figure 1: Flood's system architecture.

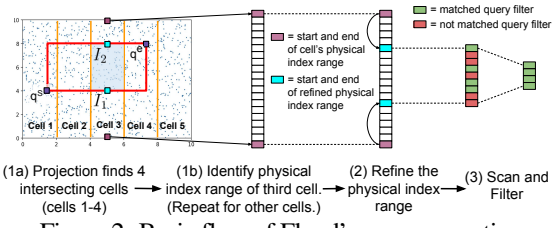

Figure 2: Basic flow of Flood's query execution.

Central to Flood are three key ideas. First, inspired by [17], Flood learns continuous functions to estimate the position of an item based on its attributes, which can improve performance with a significantly reduced index size. However, the techniques proposed in [17] are insufficient since they only work for one-dimensional sorted data. Flood therefore develops the next two key ideas. Second, Flood uses empirical CDF models to project the multi-dimensional and potentially skewed data distribution into a more uniform space. This "flattening" step helps limit the number of points that are searched and is key to achieving good performance. Third, Flood uses a sample query filter workload to understand how often certain dimensions are used, which ones are used together, and which are more selective than others. Based on this information, Flood automatically customizes the entire layout to optimize query performance on the given workload.

While Flood's techniques are general and may potentially benefit a wide range of systems, this paper focuses on improving multi-dimensional index performance (i.e., reducing unnecessary scan and filter overhead) for an in-memory column store. In-memory stores are increasingly popular due to lower RAM prices [16] and the increasing amount of main memory which can be put into a single machine [8, 15]. In addition, Flood is optimized for reads (i.e., query speed) at the expense of writes (i.e., incremental index updates), making it most suitable for rather static analytical workloads, though our experiments show that adjusting to a new query workload is relatively fast. We envision that Flood could serve as the building block for a multi-dimensional in-memory key-value store or be integrated into commercial in-memory (offline) analytics accelerators like Oracle's Database In-Memory (DBIM) [26].

The ability to self-optimize through learning allows Flood to outperform alternative state-of-the-art techniques by up to three orders of magnitude, while often having a significantly smaller storage overhead. More importantly though, Flood achieves *optimality across the board*: it has better, or at least on-par, performance compared to the next-fastest indexing technique on all our datasets and workloads. The extended conference version of this report can be found at [21].

2 Related Work

There is a rich corpus of work dedicated to multi-dimensional indexes, and many commercial database systems have turned to multi-dimensional indexing schemes. For example, Amazon Redshift organizes points by Z-order [20], which maps multi-dimensional points onto a single dimension for sorting [1, 25, 33]. With spatial dimensions, IBM Informix uses an R-Tree [13]. Other multi-dimensional indexes include K-d trees, octrees, R* trees, UB trees (which also make use of the Z-order), among many others (see [23, 29] for a survey). Flood's underlying index structure is perhaps most similar to Grid Files [22], which has many variants [30, 11, 12]. However, none of the existing data structures automatically adjust to the query workload. Additionally, Grid Files tend to have superlinear growth in index size even for uniformly distributed data [9], which Flood avoids.

Flood also differs from adaptive indexing techniques such as database cracking [14, 28] as well as automatic index selection [18, 4, 32]. The main goal of cracking is to incrementally build an index by partitioning the data. However, cracking happens separately per column and does not jointly optimize the layout over multiple attributes. More importantly, each cracker column effectively becomes a secondary index over its attribute with all its overhead. Likewise, automatic index selection focuses on creating secondary indexes; in contrast, Flood optimizes the storage and index together to produce a primary index.

3 Index Overview

Flood is a multi-dimensional clustered index that speeds up the processing of relational queries that select a range over one or more attributes. For example: `SELECT SUM(R.X) FROM MyTable WHERE (a` $\leq$ `R.Y` $\leq$ `b) AND (c` $\leq$ `R.Z` $\leq$ `d)`. Note that equality predicates of the form `R.Z == f` can be rewritten as `f`

$\leq$ R.Z $\leq$ f. Typical selections generally also include disjunctions (i.e. OR clauses). However, these can be decomposed into multiple queries over disjoint attribute ranges; hence our focus on ANDs.

Flood consists of two parts: (1) an offline preprocessing step that chooses an optimal layout, creating an index based on that layout, and (2) an online component responsible for executing queries as they arrive (see Fig. 1). At a high level, Flood is a variant of a basic grid index that divides d-dimensional data space into a $(d-1)$-dimensional grid of contiguous cells, so that data in each cell is stored together.

We form the grid offline as follows. First, we first order the d attributes. The details of how to choose a ordering are discussed in §4, but for the purposes of illustration, we assume it is given. Next, we use the first $d-1$ dimensions in the ordering to overlay a $(d-1)$-dimensional grid on the data, where the ith dimension in the ordering is divided into c_i equally spaced columns between its minimum and maximum values. The regions formed by these intersecting columns are *cells*. Note that the cell is simply a $(d-1)$-tuple, where the jth coordinate is the index of the column in the jth dimension that contains it. The dth dimension, the *sort dimension*, will be used to order points within a cell.

Online, when a query arrives, Flood executes the following workflow.

1. **Projection**: Identify the cells in the grid layout that intersect with the predicate's hyper-rectangle. Suppose that each filter in the query is a range of the form $[q_i^s, q_i^e]$ for each indexed dimension i (or $\pm\infty$ if the dimension is not present). The "lower-left" corner of the query rectangle is $q^s = (q_0^s, ..., q_{d-1}^s)$ and likewise for the "upper-right" corner q^e. Both are shown in Fig. 2. Then, we define the set of *intersecting cells* as the cells that satisfy $\text{cell}(q^s)_j \leq C_j \leq \text{cell}(q^e)_j$, for all dimensions $1 \leq j \leq d-1$. To determine the range of positions in storage, i.e. the *physical index range*, of each intersecting cell, Flood keeps a *cell table* which records the physical index of the first point in each cell.
2. **Refinement**: When the query includes a filter over the sort dimension $R.S$, e.g., $a \leq R.S \leq b$, Flood uses the fact that points in each cell are ordered by the sort dimension to further refine the physical index ranges to scan. This is straightforward to do by performing a binary search for a and b.
3. **Scan**: Not all records within the physical index ranges will match the query filter. This step iterates over all records in the physical index ranges and processes only the records that match the filter.

4 Co-optimizing the Data Layout and Index Structure

Flattening. The simple index in §3 spaces layout columns equally over the range of values in a dimension. However, this type of layout is inefficient when indexing highly skewed data: some grid cells will have a large number of points, causing Flood to scan too many superfluous points.

If we were to have an accurate model of each attribute's distribution, i.e. its CDF, we could choose columns such that for each attribute, each column is responsible for approximately the same number of points. In practice, Flood models each attribute using a Recursive Model Index (RMI), a hierarchy of models, e.g. linear models in our case, that is quick to evaluate [17]. The input to the model is the attribute value v; the output is the fraction of points with values $\leq v$. At query time, suppose that we would like to split the kth dimension into n columns. A point with value v in the kth dimension will be placed into column $\lfloor \text{CDF}(v) \cdot n \rfloor$. Skewness is abundantly present in real-world data; on two of the datasets used in our evaluation (§5), flattening provides a performance boost of $20-30\times$ over a non-flattened layout.

Faster Refinement. The simple index from §3 uses binary search over the sort dimension to refine the physical index range within each cell. In practice, binary search is slow. Instead, Flood builds a CDF model over the sort dimension values for each cell. Flood uses a cell's model to estimate the refined physical index range, and then corrects any misprediction through a local search.

We want a model that can achieve a low average absolute error, while keeping the maximum error bounded to a reasonable value, in order for local search to be fast. Unfortunately, it is difficult to build an RMI with a target error bound. Instead, the model Flood uses is a *piecewise linear CDF*, which models a CDF by partitioning a sorted list of values V into slices, each of which is modeled by a linear segment. Let $D(v)$ be the index of the first occurrence of value $v \in V$. Flood uses a greedy algorithm to partition V into slices: for each $v \in V$ in increasing order, it adds $(v, D(v))$ to the segment for the current slice. If the segment's average absolute error over the values in current slice exceeds a threshold δ, it begins a new slice. The model records the smallest v in each slice and forms a cache-optimized B-Tree over those values. We observed that using $\delta \leq 50$ produced the best performance. At this error range, linear scan is the fastest local search method. In our benchmarks, using the piecewise linear CDF for location provides more than a $5\times$ speedup over binary search.

	records	dimensions	size	source	data	queries
sales	30M	6	1.44GB	major company	real	real
tpc-h	300M	7	16.8GB	[31]	synthetic	synthetic
osm	105M	6	5.04GB	[24]	real	synthetic
perfmon	230M	6	11GB	US university	real	synthetic

Table 1: Dataset Characteristics

Optimizing Layout Parameters. Flood's layout has several parameters that can be tuned. Define a layout over d dimensions as $L=(O,\{c_i\}_{0\leq i<d-1})$, where O is an ordering of the d dimensions, in which the dth dimension is the sort dimension and $\{c_i\}_{0\leq i<d-1}$ is the number of columns in the remaining $d-1$ dimensions that form the grid. We optimize layout parameters O and $\{c_i\}_{0\leq i<d-1}$ using a cost model based approach. Given a dataset D and a layout L, we model the query time of any query q as a sum of three parts, which correspond to the three steps of the query flow from §3:

$$Time(D,q,L)=w_pN_c+w_rN_c+w_sN_s \tag{1}$$

N_c is the number of cells that intersect in the query rectangle; N_s is the number of scanned points; and the weight parameters $w=\{w_p,w_r,w_s\}$ reflect the average time to perform projection on a cell, refinement on a cell, and scan a point, respectively. The weight parameters vary based on the data, query and layout, so Flood uses *weight models*, implemented as random forest regression, to predict w. The features of these weight models are statistics that can be computed without running the query (e.g., the number of dimensions filtered by the query).

Given a cost model, Flood learns a layout that is optimized for a specific dataset and query workload using the following procedure: (1) Iteratively select each of the d dimensions to be the sort dimension. Order the remaining $d-1$ dimensions that form the grid by the average selectivity on that dimension across all queries in the workload. We found that the order of the $d-1$ grid dimensions had no significant impact. This step gives us O. (2) For each of these d possible orderings, run a gradient descent search algorithm to find the optimal number of columns $\{c_i\}_{0\leq i<d-1}$. The objective function is Eq. 1, averaged over all queries in the workload. For each call to the cost model, Flood computes the statistics $N=\{N_c,N_s\}$ and the input features of the weight models efficiently using a sample of the dataset and query workload. (3) Select the layout with the lowest objective function cost amongst the d layouts.

5 Evaluation

We implement Flood in C++ on a custom column store that uses *block-delta* compression: in each column, the data is divided into consecutive blocks of 128 values, and each value is encoded as the delta to the minimum value in its block. Our implementation uses 64-bit integer-valued attributes. We compare Flood to several other indexing approaches, each of which is implemented on the same column store and uses the same optimizations where applicable: Full Scan, in which every point is visited; Clustered Single-Dimensional Index, in which points are sorted by the most selective dimension in the query workload; Grid Files [22]; Z-Order Index (similar to but not identical to [33]); UB-tree [27]; Hyperoctree [19]; k-d tree [3]; and R*-Tree [2, 10].

For a fair comparison, all benchmarks are implemented using a single thread without SIMD instructions. We excluded multiple threads mainly because our baselines were not optimized for it. All experiments are run on an Ubuntu Linux machine with an Intel Core i9 3.6GHz CPU and 64GB RAM. We evaluate indexes on four datasets, summarized in Tab. 1. For each dataset, we generate a train and test query workload from the same distribution, each with around 1000 queries. Flood's layout is optimized on the training set, and we only report results on the test set.

Overall Performance. Fig. 3 shows the query time for each optimized index on each dataset. Flood uses the layout learned using the algorithm in §4, while we tuned the baseline approaches as much as possible per workload (e.g., ordered dimensions by selectivity and tuned page sizes). This represents the best case scenario for the baselines: the database administrator had the time and ability to tune the index parameters.

On three of the datasets, Flood achieves between $2.4\times$ and $3.3\times$ speedup on query time compared to the next closest index, and is always at least on-par, thus achieving *optimality across the board*. However, the next best system changes across datasets. Thus, depending on the dataset and workload, Flood can outperform each baseline by orders of magnitude. For example, on the real-world sales dataset, Flood is at least $43\times$ faster than each multi-dimensional index, but only $3\times$ faster than a clustered index. However,

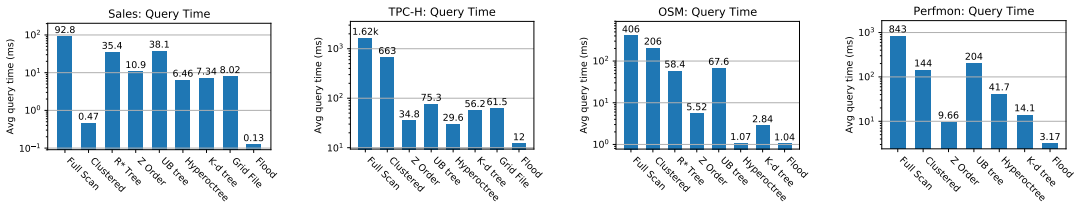

Figure 3: Query speed of Flood on all datasets. Flood's index is trained automatically, while every other index is manually optimized for each workload to achieve the best performance. We excluded the R-tree for cases for which it ran out of memory. Note the log scale.

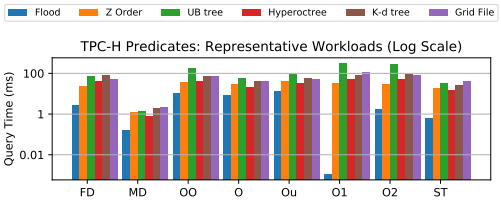

Figure 4: Flood beats alternative indexes on a variety of workloads. Note the log scale.

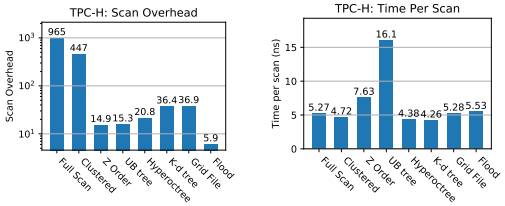

Figure 5: (a) The ratio of points scanned to points in the result (log scale), (b) The time spent by the index on each scanned point.

on the TPC-H derived data, Flood is 187× faster than a clustered index. Flood's index is always smaller than the next fastest index, and can take up to 50× less space.

Different Workload Characteristics. In practical settings, it is unlikely that a database administrator will be able to manually tune the index for every workload change. The ability of Flood to automatically configure its index for the current query workload is thus a significant advantage. We measure this advantage by tuning all indexes for the workloads in Fig. 3, and then changing the query workload characteristics to:

1. Single record lookups using one or two ID attributes, as commonly found in OLTP systems (O1 and O2).
2. OLAP workloads where query types are equally likely (Ou) or skewed (O).
3. An equal split of workloads (1) and (2), i.e., combined OLTP and OLAP queries (OO).
4. A workload with a single type of query, using the same dimensions with the same selectivities (ST).
5. A workload with fewer or more dimensions than indexed by the baseline indexes (FD and MD).

Fig. 4 shows the potential advantages Flood can achieve over more static alternatives. Flood consistently beats other indexes, achieving a speedup of more than 20× on half the workloads.

Performance Breakdown. Where does Flood's advantage come from? One useful comparison is the *scan overhead*, the ratio of total points scanned by the index to the points actually in the result. Since all indexes spend over 80% of their time scanning, the scan overhead is a good proxy for overall query performance. Fig. 5a shows that Flood is able to achieve the lowest scan overhead. Indexes based on Z-order have the closest scan overhead to Flood, but they incur the cost of computing Z-values and thus have a higher time per scanned point (Fig. 5b), which results in higher query time for the same scan overhead.

6 Conclusion

Despite the shift of OLAP workloads towards in-memory databases, state-of-the-art systems have failed to take advantage of multi-dimensional indexes to accelerate their queries. Many instead opt for simple 1-D clustered indexes with bulky secondary indexes that waste space. We design a new multi-dimensional index Flood with two properties. First, it serves as the primary index and is used as the storage order for underlying data. Second, it is jointly optimized using both the underlying data and query workloads. Our approach outperforms existing clustered indexes by $30-400\times$. Likewise, learning the index layout from the query workload allows Flood to beat optimally tuned spatial indexes, while using a fraction of the space. Our results suggest that learned primary multi-dimensional indexes offer a significant performance improvement over state-of-the-art approaches and can serve as useful building blocks in larger in-memory database systems.

References

[1] Amazon AWS. Amazon Redshift Engineering's Advanced Table Design Playbook: Compound and Interleaved Sort Keys. https://aws.amazon.com/blogs/big-data/amazon-redshift-engineerings-advanced-table-design-playbook-compound-and-interleaved-sort-keys/, 2016.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 322–331, New York, NY, USA, 1990. ACM.

[3] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.

[4] S. Chaudhuri and V. Narasayya. An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proceedings of the VLDB Endowment*. VLDB Endowment, 1997.

[5] Databricks Engineering Blog. Processing Petabytes of Data in Seconds with Databricks Delta. https://databricks.com/blog/2018/07/31/processing-petabytes-of-data-in-seconds-with-databricks-delta.html.

[6] A. L. et al. The Vertica Analytic Database: C-Store 7 Years Later. In *Proceedings of the VLDB Endowment*. VLDB Endowment, 2012.

[7] M. S. et al. C-Store: A Column-oriented DBMS. In *Proceedings of the 31st VLDB Conference*. VLDB Endowment, 2005.

[8] Exasol. The World's Fastest In-Memory Analytic Database. https://www.exasol.com/en/community/resources/resource/worlds-fastest-analytic-database/.

[9] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys (CSUR)*, 30:170–231, 1998.

[10] M. Hadjieleftheriou. libspatialindex. https://libspatialindex.org/, 2014.

[11] K. Hinrichs. Implementation of the grid file: Design concepts and experience. *BIT*, 25(4):569–592, Dec. 1985.

[12] A. Hutflesz, H.-W. Six, and P. Widmayer. Twin grid files: Space optimizing access schemes. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 183–190, New York, NY, USA, 1988. ACM.

[13] IBM. The Spatial Index. https://www.ibm.com/support/knowledgecenter/SSGU8G_12.1.0/com.ibm.spatial.doc/ids_spat_024.htm.

[14] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. Conference on Innovative Data Systems Research (CIDR), 2007.

[15] Intel Corporation. Scaling Data Capacity for SAP HANA with Fujitsu PRIMERGY/PRIMEQUEST Servers. Technical report, 2014.

[16] I. Khan. Falling ram prices drive in-memory database surge. https://www.itworld.com/article/2718428/falling-ram-prices-drive-in-memory-database-surge.html, 2012.

[17] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. ACM, 2018.

[18] L. Ma, D. V. Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *SIGMOD*. ACM, 2018.

[19] D. Meagher. Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer. Technical report, 1980.

[20] G. M. Morton. A computer oriented geodetic data base; and a new technique in file sequencing (pdf). Technical report, IBM, 1966.

[21] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska. Learning Multi-dimensional Indexes. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD)*. ACM, 2020. To appear. https://arxiv.org/abs/1912.01668.

[22] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9(1):38–71, Mar. 1984.

[23] B. C. Ooi, R. Sacks-davis, and J. Han. Indexing in spatial databases, 2019.

[24] OpenStreetMap contributors. US Northeast dump obtained from https://download.geofabrik.de/. https://www.openstreetmap.org, 2019.

[25] Oracle Database Data Warehousing Guide. Attribute Clustering. https://docs.oracle.com/database/121/DWHSG/attcluster.htm, 2017.

[26] Oracle, Inc. Oracle Database In-Memory. https://www.oracle.com/database/technologies/in-memory.html.

[27] F. Ramsak1, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer. Integrating the UB-Tree into a Database System Kernel . In *Proceedings of the 26th International Conference on Very Large Databases*. VLDB Endowment, 2000.

[28] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The uncracked pieces in database cracking. *Proc. VLDB Endow.*, 7(2):97–108, Oct. 2013.

[29] H. Singh and S. Bawa. A survey of traditional and mapreducebased spatial query processing approaches. *SIGMOD Rec.*, 46(2):18–29, Sept. 2017.

[30] M. Tamminen. The extendible cell method for closest point problems. *BIT*, 22:27–41, 01 1982.

[31] TPC. TPC-H. http://www.tpc.org/tpch/, 2019.

[32] G. Valentin, M. Zuliani, D. C. Zilio, G. Lohman, and A. Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend its own Indexes. In *Proceedings of the 16th International Conference on Data Engineering*. IEEE, 2000.

[33] Zack Slayton. Z-Order Indexing for Multifaceted Queries in Amazon DynamoDB. https://aws.amazon.com/blogs/database/z-order-indexing-for-multifaceted-queries-in-amazon-dynamodb-part-1/, 2017.