Designing Succinct Secondary Indexing Mechanism by Exploiting Column Correlations

Yingjun Wu
IBM Research - Almaden
yingjun.wu@ibm.com

Jia Yu*
Arizona State University
jiayu2@asu.edu

Yuanyuan Tian
IBM Research - Almaden
ytian@us.ibm.com

Richard Sidle
IBM
ricsidle@ca.ibm.com

Ronald Barber
IBM Research - Almaden
rjbarber@us.ibm.com

ABSTRACT

Database administrators construct secondary indexes on data tables to accelerate query processing in relational database management systems (RDBMSs). These indexes are built on top of the most frequently queried columns according to the data statistics. Unfortunately, maintaining multiple secondary indexes in the same database can be extremely space consuming, causing significant performance degradation due to the potential exhaustion of memory space. In this paper, we demonstrate that there exist many opportunities to exploit column correlations for accelerating data access. We propose Hermit, a succinct secondary indexing mechanism for modern RDBMSs. Hermit judiciously leverages the rich soft functional dependencies hidden among columns to prune out redundant structures for indexed key access. Instead of building a complete index that stores every single entry in the key columns, Hermit navigates any incoming key access queries to an existing index built on the correlated columns. This is achieved through the Tiered Regression Search Tree (TRS-Tree), a succinct, ML-enhanced data structure that performs fast curve fitting to adaptively and dynamically capture both column correlations and outliers. Our extensive experimental study in two different RDBMSs have confirmed that Hermit can significantly reduce space consumption with limited performance overhead, especially when supporting complex range queries.

*Work done during an internship at IBM Research - Almaden.

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3319861

ACM Reference Format:
Yingjun Wu, Jia Yu, Yuanyuan Tian, Richard Sidle, and Ronald Barber. 2019. Designing Succinct Secondary Indexing Mechanism by Exploiting Column Correlations. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3319861

1 INTRODUCTION

Modern relational database management systems (RDBMSs) support fast secondary indexes that help accelerate query processing in both transactional and analytical workloads. These indexes, created either by database administrators or automatically by query optimizers, are built on top of the most frequently queried columns, hence providing an efficient way to retrieve data tuples via these columns. However, managing multiple secondary indexes in the database can consume large amounts of storage space, potentially causing severe performance degradation due to the exhaustion of memory space. This problem is not uncommon especially in the context of modern main-memory RDBMSs, where memory space is a scarce resource.

Confronting this problem, researchers in the database community have proposed various practical solutions to limit the space usage for index maintenance. From the database tuning perspective, some of the research works have introduced smart performance tuning advisors that can automatically select the most beneficial secondary indexes given a fixed space budget [4, 9, 36]. While satisfying the space constraints, these techniques essentially limit the number of secondary indexes built on the tables, consequently causing poor performance for queries that lookup the unindexed columns. From the structure design perspective, a group of researchers has developed space-efficient index structures that consume less storage space compared to conventional indexes [13]. These works either store only a subset of the column entries [35] or use compression techniques to reduce space consumption [40]. However, such solutions save limited amount of space and can cause high overhead for lookup operations.

We attempt to address this problem in a third way. The main observation that sparks our idea is that many columns in the data tables exhibit *correlation* relations, or *soft functional dependencies*, where the values of a column can be estimated by that of another column with approximation. Exploiting such a relation can greatly reduce the memory consumption caused by secondary index maintenance. Specifically, if we want to create an index on a column M that is highly correlated with another column N where an index has been built, we can simply construct a succinct, ML-enhanced data structure, called *Tiered Regression Search Tree*, or TRS-Tree, to capture the correlation between M and N. TRS-Tree exploits multiple simple statistical regression processes to fit the curve of the hidden correlation function. Different from existing machine learning-based indexing solutions, TRS-Tree efficiently handles inserts, deletes, and updates, and supports on-demand structure reorganization to re-optimize the index efficiency at system runtime. To perform a lookup query on M, the RDBMS retrieves a lookup range on N from the newly constructed TRS-Tree and fetches the targeted tuples using N's index. We call this mechanism Hermit.

Hermit achieves competitive performance when supporting range queries, which are prevalent for secondary key column accesses. It also presents a tradeoff between computation and space consumption. While avoiding building a complete index structure remarkably reduces space consumption, Hermit requires any incoming query to go through an additional hop before retrieving the targeted tuples. However, as our experiments will show, this overhead does not substantially affect performance in practice, and it also brings in huge benefits when storage space is valuable and scarce, such as in main-memory RDBMSs.

This paper is organized as follows. Section 2 provides technical background. Section 3 gives an overview of Hermit. Section 4 presents Hermit's TRS-Tree structure, and Section 5 shows how Hermit leverages TRS-Tree to perform tuple retrieval. Section 6 discusses several issues. We report experiment results in Section 7. Section 8 reviews related works and Section 9 concludes.

2 BACKGROUND

In this section, we provide some background about index structures and column correlations in the context of RDBMSs.

Index Structures. Modern RDBMSs use secondary index structures to improve the speed of data retrieval at the cost of additional writes and space consumption. A secondary index is created either by a database administrator or automatically by a query optimizer, and is built on top of one or more frequently accessed columns. An index can be considered as a copy of the corresponding key columns organized in a format that provides fast mapping to the tuple identifiers, in terms of either tuple locations or primary keys [38]. Maintaining multiple secondary indexes can be expensive, especially in the context of main-memory RDBMSs. This is confirmed by a recent study [40] which showed that index maintenance can consume around 55% of the total memory space. Confronting this problem, researchers have proposed various space-efficient index structures to reduce space consumption. In general, these works share two basic ideas: (1) using classic compression techniques such as Huffman encoding or dictionary encoding to reduce the index node size [13]; (2) storing only a subset of entries from the indexed columns to reduce the number of leaf nodes [35]. Despite the limited reduction in memory consumption, these techniques can incur high overhead when processing lookup operations.

Column Correlations. A conventional RDBMS allows database administrators to set integrity constraints using SQL statements to express the functional dependencies among data columns. These explicitly declared functional dependencies can be leveraged by query optimizers to provide a more accurate cost estimation during the query rewriting phase. In addition to these "hard" functional dependencies, modern query optimizers also attempt to explore "soft" functional dependencies to generate better query plans using the column correlation relations. Column correlations capture approximate dependencies, meaning that the value of a column can determine that of another approximately. Following the definition in existing works, we define a column correlation relation as a triple (M, N, Fn), where M and N are data columns in the table exhibiting correlations, and Fn is a *correlation function* specifying how N's values can be estimated from M. Besides simple algebraic computation [7] (e.g., $+$, $-$, $\times$, $/$) and linear functions [11, 15] (e.g., $N = \beta M + \alpha \pm \epsilon$), a correlation function $N = Fn(M)$ can be of any possible form. This will allow us to capture the correlations in many modern database applications, such as environment monitoring (oxygen v.s. carbon dioxide), stock market (Dow-Jones v.s. S&P 500), and healthcare (glucagon v.s. insulin). Existing RDBMSs have already exploited this kind of data characteristics to address system efficiency problems, including data compression, query rewriting, and database tuning [20–22]. In the following sections, we will show how we can leverage correlations to accelerate data access.

3 OVERVIEW

The correlation hidden among different columns in an RDBMS indicates a high similarity of their corresponding index structures. Observing that, we developed a succinct, yet fast secondary indexing mechanism, namely Hermit, which exploits the column correlations to answer queries.

To index a specified column M, Hermit requires two components: a succinct data structure called *Tiered Regression*

Search Tree (abbr., TRS-Tree) on the *target* column M, and a pre-existing complete index called *host index* on the *host* column N. TRS-Tree models the correlation between M and N: it leverages a *tiered regression* method to perform hierarchical curve fitting over the correlation function Fn from M to N, and uses a tree structure to index a set of regression functions each of which represents an approximate linear mapping from a value range of M to that of N.

To process a query, Hermit runs a three-phase searching algorithm: (1) TRS-Tree search; (2) host index search; and (3) base table validation. Specifically, Hermit uses the query predicate to search the TRS-Tree in order to retrieve the range mapping from M to N. It then leverages the host index to find a set of candidate tuple identifiers. We note that this candidate set is approximate, and it contains false positives that fail to satisfy the original predicates. Hermit removes those false positives by directly validating the corresponding values on the base table.

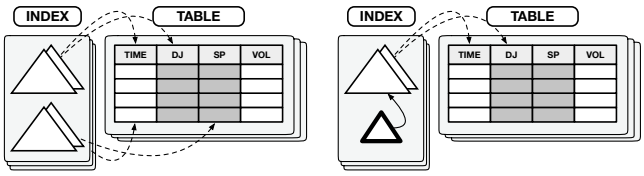
Hermit also works for multi-column secondary indexes. Suppose that two columns A and M on a table are queried together frequently, so an index on (A, M) is desirable. Hermit can utilize a host index on (A, N) and the correlation between M and N, to answer queries on A and M.

In the case where multi-column correlation exists (e.g., (W, X)->(Y,Z), although rarely detected by RDBMSs), Hermit can concatenate multiple keys and build TRS-Tree on them.

We now use a running example to demonstrate how Hermit works. Let us consider a data table `STOCK_HISTORY` recording U.S. stock market trading histories with four different columns: `TIME` (i.e., trading date), `DJ` (i.e., Dow Jones), `SP` (i.e., S&P 500), and `VOL` (i.e., total trading volume). The database administrator has already created an index on (`TIME`, `DJ`). Now she decides to create another index on (`TIME`, `SP`) due to the frequent occurrence of the queries like:

```
SELECT * FROM STOCK_HISTORY
WHERE (TIME  BETWEEN ? AND ?) AND (SP BETWEEN ? AND ?)
```

On receiving the index creation statement, the RDBMS adopting Hermit first checks whether any column correlation involving `TIME` or `SP` has been detected via any correlation discovery algorithms. If observing that the values in `SP` are highly correlated with those in `DJ` and that there is an existing index on (`TIME`, `DJ`), the RDBMS then constructs a TRS-Tree to model the correlation mapping from `SP` to `DJ`. Given the query ranges (T_{min}, T_{max}) on column `TIME` and (S_{min}, S_{max}) on column `SP`, Hermit first inputs the `SP` range (S_{min}, S_{max}) to the constructed TRS-Tree to fetch the corresponding range (D_{min}, D_{max}) on `DJ`. Then it searches the host index on (`TIME`, `DJ`), with `TIME` range (T_{min}, T_{max}) and `DJ` range (D_{min}, D_{max}), to find all the satisfying tuples. To filter out false positives, the RDBMS reads the `SP` values from the base table and validates the correctness of the result.



(a) Conventional index **(b) Hermit**

Figure 1: A comparison between data retrieval via conventional secondary indexes and Hermit. (double triangles denotes conventional secondary index; small single triangle denotes the proposed TRS-Tree structure)

Figure 1 shows how Hermit is different from conventional secondary indexing mechanisms when retrieving tuples in the running example. Unlike existing indexing techniques that provide direct accesses to the tuple identifiers, Hermit requires two-hop accesses. While potentially causing higher access overhead for point queries, Hermit can achieve very competitive performance for range queries that are highly common for secondary indexes (as we will demonstrate in the experiments). And, of course, Hermit can significantly reduce the space consumption for index maintenance.

Hermit can support insert, delete, and update operations with correctness guarantees. Due to its approximate characteristics, Hermit works best for range queries, which are quite common for secondary key columns, especially in data analytics. Furthermore, Hermit is extremely beneficial for main-memory RDBMSs, where memory space is scarce.

4 TRS-TREE

TRS-Tree is a succinct tree structure that models data correlation between a target column M and a host column N within the same data table of a database. It leverages a tiered regression method to *adaptively* and *dynamically* perform the curve fitting over the correlation function $N = Fn(M)$. To be precise, TRS-Tree decomposes the complex curve-fitting problem into multiple simpler sub-problems and uses linear regression method to accurately address these sub-problems. TRS-Tree is adaptive, in the sense that it constructs its internal structures based on the correlation complexity; it is also dynamic, meaning that it reorganizes its internals at runtime to ensure the best efficiency.

In the following section, we first discuss TRS-Tree's internal structure, and then demonstrate its construction, lookup, maintenance, parameter setting, and optimization.

4.1 Internal Structure

TRS-Tree is a k-ary tree structure that maps the values in the target column M to that in the host column N. Its construction algorithm recursively divides M's value range into k uniform sub-ranges until every entry pair (m, n) from M

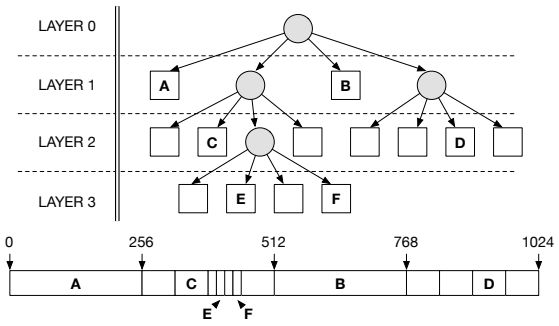


Figure 2: TRS-Tree data structure on a target column with value range from 0 to 1024. The node fanout is set to 4. The boxes represent the leaf nodes and the circles represent the internal nodes. The ruler bar shows how TRS-Tree partitions the range of the target column.

and N covered by the corresponding sub-range can be well estimated by a simple linear regression-based data mapping. As a tree-based data structure, TRS-Tree uses leaf nodes to maintain the detailed data mappings, with its internal nodes providing fast navigation to these leaf nodes. Figure 2 shows a TRS-Tree structure constructed on a target column whose value range is from 0 to 1024.

Leaf node. A leaf node in TRS-Tree is associated with a sub-range r of the target column M. We define that a range r has two elements: a lower bound lb and an upper bound ub. Given a set of column entries M^r from M covered by r (i.e., $\forall m \in M^r, r.lb \leq m \leq r.ub$), the leaf node attempts to provide an approximate linear mapping from M^r to its corresponding set of column entries N^r in the host column N. Such a mapping is represented using a linear function $n = \beta m + \alpha \pm \epsilon$, where m and n represent column values from M^r and N^r, β and α respectively denote the function's slope and intercept, and ϵ denotes the confidence interval.

TRS-Tree computes β and α using the standard linear regression formula [3] listed below:

$$\alpha = \overline{N^r} - \beta\overline{M^r} \qquad \beta = \frac{cov(M^r, N^r)}{var(M^r)}$$

where $\overline{N^r}$ and $\overline{M^r}$ respectively denote the average values of elements in N^r and M^r, $var(M^r)$ is the variance of elements in M^r, and $cov(M^r, N^r)$ presents the covariance of the corresponding elements in M^r and N^r. Based on the above equations, both α and β can be computed with *one scan* of the data in M^r and N^r.

Different from the slope and intercept, the confidence interval ϵ can be computed based on a user-defined parameter, called *error_bound*, as will be elaborated in Section 4.5.

The function $n = \beta m + \alpha \pm \epsilon$ captures an approximate linear correlation between columns M and N under the sub-range r in M. Given an m in M^r, it bounds the corresponding n to be in the range $(\beta m + \alpha - \epsilon, \beta m + \alpha + \epsilon)$. However, not all the entry pairs (m, n) from M^r and N^r are necessarily covered by the computed linear function. We call these entry pairs as *outliers*. The leaf node maintains all these outliers in an *outlier buffer*, which is implemented as a hash table mapping from m to the corresponding tuple's identifier, which can be either a primary key or a tuple location, as we will elaborate in Section 5.

Internal node. An internal node in TRS-Tree functions as a navigator that routes the queries to their targeted leaf nodes. Similar to the leaf nodes, each internal node is associated to a range in the target column M. However, instead of maintaining any mapping to the host column, an internal node only maintains a fixed number of pointers pointing to its child nodes, each of which can be either a leaf node or another internal node. To perform a lookup, an internal node can easily navigate the query to the corresponding child node whose range covers the input value.

4.2 Construction

An RDBMS can efficiently construct a TRS-Tree upon the user's request. Algorithm 1 details the construction steps. The construction algorithm takes as input the base table T, the target column ID cid_M, the host column ID cid_N, and the target column's full range R. The range R contains the minimum and maximum values in the target column and can be easily obtained from the RDBMS's optimizer statistics. The algorithm also requires a set of TRS-Tree's pre-defined parameters for computation.

This construction algorithm utilizes a FIFO (first-in-first-out) queue to build the TRS-Tree in a top-down fashion. Each element in the FIFO queue is a pair that contains a TRS-Tree node and the node's corresponding temporary table. The temporary table for a TRS-Tree node is a sub-table of T, which selects rows with target column in the node's range, and projects only the target and host columns along with each tuple's identifier.

TRS-Tree's construction starts by creating a root node with its range set to the whole range R, and pushing the root node along with its projected temporary table into the FIFO queue. It then does the following steps iteratively until the FIFO queue is empty: (1) retrieve the ($tmpTable$, $node$) pair from the FIFO queue; (2) compute $node$'s β, α, and ϵ and determine whether the generated linear mapping can well cover its corresponding entry pairs; (3) if (2) returns false, then split $node$ and $tmpTable$ respectively into multiple child nodes and sub-tables, then push the corresponding pairs of child node and sub-table back to the FIFO queue.

The `Compute` function scans a node's temporary table to compute the parameters for the linear function. The `Validate` function scans the temporary table again, and validates whether

Algorithm 1: Index construction in TRS-Tree

```
Data: base table T, target column ID cid_M, host column ID
      cid_N, value range R
Result: TRS-Tree's root node root
1  Node root(R);
2  TmpTable fullTmpTable = ProjectTable (T, cid_M, cid_N);
3  FIFOQueue queue;
4  queue.Push (Pair(root, fullTmpTable));
5  while queue.IsNotEmpty () do
6      Pair pair = queue.Pop ();
7      Node node = pair.GetKey ();
8      TmpTable tmpTable = pair.GetValue ();
9      Compute (tmpTable, node);
10     if !Validate (tmpTable, node) then
11         Node[] subNodes = SplitNode (node,
             node_fanout);
12         TmpTable[] subTables = SplitTable (tmpTable,
             subNodes);
13         foreach i in (0 to node_fanout − 1) do
14             queue.Push (Pair(subNodes[i], subTables[i]));
15     DeleteTmpTable (tmpTable);
16 return root;
17 Function Validate(tmpTable, node)
18     foreach entry in tmpTable do
19         if entry.host ∉ node.GetHostRange (entry.target)
             then
20             node.outliers.Add (entry.target, entry.tid);
21         if node.outliers.Size () >
             outlier_ratio*tmpTable.Size () then
22             return false;
23     return true;
```

Algorithm 2: Index lookup in TRS-Tree

```
Data: root node root, predicate P
Result: range set RS, tuple identity set IS
1  Set<Range> RS;
2  Set<TupleID> IS;
3  FIFOQueue queue;
4  queue.Push (root);
5  while queue.IsNotEmpty () do
6      node = queue.Pop ();
7      if node.IsLeaf () then
8          Range r = Intersect (node.range, P);
9          RS.Add (node.GetHostRange (r));
10         IS.Add (node.outliers.Lookup (r));
11     else
12         foreach child in node.children do
13             if child.IsOverlapping (P) then
14                 queue.Push (child);
15 RS = Union (RS);
16 return RS and IS;
```

each pair of target and host column values can be covered by the linear function. Any unqualified entry is inserted to the node's outlier buffer. A node's linear function is determined to be not *good enough* if the outlier buffer exceeds a pre-defined *outlier_ratio*, which is a ratio of the outlier buffer size to the total number of tuples covered by the node. In this case, step (3) is triggered, which drops all the generated content in *node* and splits it into a fixed number (equals to the pre-defined *node_fanout* parameter) of equal-range child nodes. The users can limit the maximum depth of the tree structure by setting the parameter *max_height*.

The user-defined parameters for TRS-Tree can directly control the confidence interval of the linear functions as well as the outlier buffer size, consequently affecting the performance. We discuss the parameters in Section 4.5. We further elaborate several optimization strategies in Appendix D.

We now perform a complexity analysis for Algorithm 1. TRS-Tree uses `Compute` function to scan the tuples covered by every tree node to derive a linear regression model. If the generated TRS-Tree is always a balanced full tree, then running linear regressions for all the tree nodes at the same height takes a full scan of all tuples. A TRS-Tree with height equal to h will perform h full scans in total. As h is bounded by the parameter *max_height*, we can conclude that the average and worst case complexities are $O(N)$.

4.3 Lookup

TRS-Tree allows users to perform both point and range lookups on the target column M to get the corresponding results on the host column N. Instead of returning results that exactly match the query predicates, TRS-Tree' lookup algorithm returns approximate results. Hermit will perform additional lookups on the host indexes and further validate the results and generate exact matchings.

Algorithm 2 lists the details of TRS-Tree's lookup algorithm. The algorithm takes as input TRS-Tree's root node *root* and a query predicate P on the target column M. It generates as output a set of value ranges RS on the host column N as well as a set of tuple identifiers IS. Without losing generality, we consider P to be a value range on M with two elements: lower bound lb and upper bound ub. A point query predicate has its lower bound equals to its higher bound. The lookup starts from *root* and runs a breadth-first search using a FIFO queue. The TRS-Tree iterates every single node in the queue and performs a lookup if the node is a leaf node. On confronting an internal node, TRS-Tree retrieves its child nodes and checks whether each child's range overlaps with P. Any overlapping child node will be pushed to the FIFO queue. The lookup algorithm continues iterating until the queue is empty.

Algorithm 3: Index insertion and deletion in TRS-Tree

```
Data: root node root, target column value m, host column
      value n, tuple ID tid
1  Function Insert(root, m, n, tid)
2      Node node = Traverse (root);
3      if n ∉ node.GetHostRange (m) then
4          node.outliers.Add (m, tid);
5  Function Delete(root, m, n, tid)
6      Node node = Traverse (root);
7      node.outliers.Remove (m, tid);
8  Function Traverse(node, m)
9      if node.IsLeaf () then
10         return node;
11     else
12         foreach child in node.children do
13             if m ∈ child.range then
14                 return Traverse(child);
```

TRS-Tree performs a lookup on a leaf node by taking the following steps. First, it computes an intersection between the query predicate P and the node's value range. The intersection result is a value range r. Using this range, the node can then use its linear function to compute the estimated range on N that covers the exact matching. The estimated range will be either $(\beta \times r.lb + \alpha - \epsilon, \beta \times r.ub + \alpha + \epsilon)$ or $(\beta \times r.ub + \alpha - \epsilon, \beta \times r.lb + \alpha + \epsilon)$, depending on the sign (positive or negative) of the slope β. Not all the matchings are covered by the linear function. Hence, the leaf node further retrieves a set of tuple identifiers from its outlier buffer. These identifiers can be used to directly fetch the corresponding tuples from the RDBMS without looking up the host index. Before terminating the algorithm, TRS-Tree computes a union among all the elements in RS. This is because the returned ranges generated by different leaf nodes can overlap. Computing the union can help reduce the number of elements in RS.

4.4 Maintenance

At system runtime, TRS-Tree can dynamically support all of the commonly used database operations, including insertions, deletions, and updates. This makes TRS-Tree a drastic departure from existing machine learning-based solutions, which rely on a long-running training phase to reconstruct the index structures from scratch. TRS-Tree also reorganizes the TRS-Tree structure at runtime to ensure the best query performance. Algorithm 3 demonstrates how TRS-Tree processes insertions and deletions.

Insertion. Tuple insertions in TRS-Tree are performed swiftly with little runtime change to its internal structures. Given the to-be-inserted tuple's target column value m, host column value n, and tuple identifier tid, TRS-Tree starts the insertion by locating the leaf node containing the range covering m. After that, TRS-Tree checks whether the node's corresponding range of the host column can cover n (using the leaf node's linear function). If not, then TRS-Tree inserts this tuple's m and tid into the outlier buffer. Otherwise, the insertion algorithm directly terminates. The outlier buffer size of certain leaf nodes may grow to be too large, consequently degrading TRS-Tree's lookup performance. In this case, TRS-Tree invokes structure reorganization to further split these nodes, as we shall discuss shortly.

Deletion. The tuple deletion algorithm in TRS-Tree shares a similar routine as the insertion. However, TRS-Tree does not perform any computation after locating the leaf node. Instead, it directly checks the outlier buffer and removes the corresponding entry if exists. Frequent tuple deletion from the index can cause suboptimal space utilization problem, meaning that TRS-Tree can potentially use less number of leaf nodes to accurately capture the column correlations. TRS-Tree also relies on the structure reorganization to handle this issue.

Reorganization. As we have mentioned above, TRS-Tree reorganizes its internal structure on demand to optimize the index efficiency in terms of both lookup performance and space utilization. TRS-Tree detects reorganization opportunities based on two criteria. First, the outlier buffer size of a certain leaf node reaches a threshold ratio compared to the total number of tuples covered in the corresponding range; second, the number of deleted tuples covered by the leaf node's corresponding range reaches a threshold compared to the total number of tuples. For the first case, TRS-Tree directly splits the leaf node into multiple equal-range child nodes, as described in Algorithm 1. For the second case, TRS-Tree checks the node's neighbors to determine whether merging is beneficial.

TRS-Tree uses a dedicated background thread to execute the reorganization procedure, but it offloads the detection of candidate nodes to each insert/delete operation. Specifically, TRS-Tree maintains a FIFO queue to record nodes where merge or split can be made. Once an insert operation finishes its procedure and detects that the outlier buffer size of its visited leaf node has reached the threshold, it then adds the pointer to the leaf node into the FIFO queue. Delete operations proceed in a similar manner, but they add into the FIFO queue the pointer to the parent of the visited leaf node. Every entry in the queue is attached with a flag to identify whether this node is a candidate for split or merge.

To perform structure reorganization, the background thread scans the target column to obtain all the tuples that fall into the affected value range. It then computes the linear functions and populates the outlier buffers before installing the new node(s) into the tree structure. To reduce the latency

of the reorganization procedure, TRS-Tree periodically performs batch structure reorganization, meaning that the background thread can reorganize several candidate nodes in one scan. On confronting drastic workload change, TRS-Tree can reorganize entire subtree at once (as we shall see in Section 7.7). TRS-Tree also supports online structure reorganization, which enables concurrent lookup/insert/delete operations with little interference. TRS-Tree ensures the structure consistency by leveraging a very simple yet efficient synchronization protocol. Due to the space limit, we provide more details in Appendix B.

4.5 Parameters

TRS-Tree requires the users to pre-define some parameters before the index construction, including *node_fanout*, *max_height*, and *outlier_ratio*. There is another important parameter, called *error_bound*, which is used to determine the confidence interval ϵ of each leaf node.

The *error_bound* parameter represents the expected number of host column N values covered by the range returned from searching a TRS-Tree node for a point query on the target column M. So, by setting this parameter, TRS-Tree roughly measures the number of false positives for a point query. For a given leaf node with range r on column M, its linear function returns an estimated range $(\beta \times r.lb + \alpha - \epsilon, \beta \times r.ub + \alpha + \epsilon)$ on column N (We assume slope β to be positive. Our discussion be easily generalized to cases with negative slopes.). For any point $m \in r$ on column M, the linear function returns a range $(\beta \times m + \alpha - \epsilon, \beta \times m + \alpha + \epsilon)$ on column N. Let n be the number tuples covered by the leaf node. Assuming that values on column N are uniformly distributed, then the expected number of values of N for a point query (i.e. *error_bound*) can be estimated as $error\_bound = \frac{2\epsilon}{\beta(r.ub-r.lb)+2\epsilon} \times n \approx \frac{2\epsilon}{\beta(r.ub-r.lb)} \times n$. So, now given a desired *error_bound* parameter value, we can derive $\epsilon \approx \frac{\beta(r.ub-r.lb)\times error\_bound}{2n}$.

In theory, *error_bound* should be set carefully, since larger *error_bound* generates larger ϵ, which subsequently results in larger returned ranges for the upcoming host index lookup. Too small *error_bound* can also cause performance degradation, since more tuples covered by the corresponding range may be identified as outliers, consequently causing node splitting, yielding much deeper tree structure. Fortunately, in practice, the configuration of *error_bound* does not impact the performance too much. This is because database users are more likely to issue range queries on secondary key columns, in which case the amount of false positives brought by large ϵ is negligible compared to the amount of the tuples covered by the range query predicate. Hence, Hermit adopting TRS-Tree can enjoy a very competitive end-to-end performance compared to conventional complete-tree indexes, even with larger *error_bound*.

5 HERMIT

TRS-Tree lookup returns only approximate results. To obtain the real matching for the input queries, Hermit further needs to remove all the false positive results. In this section, we first discuss tuple identifier schemes in existing RDBMSs, and show how Hermit can work with these different schemes and generate accurate query results.

5.1 Tuple Identifiers

A secondary index built on a certain (set of) column(s) provides a mapping from the columns' key values to the corresponding tuples' identifiers. Tuple identifiers can be implemented in two different ways depending on the RDBMS's performance requirement [14, 38]. Hermit's indexing mechanism is general enough to work with both schemes.

An RDBMS adopting *logical pointers* stores the primary key of the corresponding tuple in each secondary index's leaf nodes. The rationale behind logical pointers is that any update to tuple locations will not influence the secondary indexes. However, the drawback of this mechanism is that the RDBMS has to perform an additional lookup on the primary index every time a secondary index lookup happens. Popular RDBMSs like MySQL adopt this mechanism.

An RDBMS adopting the other identifier mechanism, called *physical pointers*, directly stores tuple locations (in the format of "blockID+offset") in each secondary index's leaf nodes. While avoiding traversing the primary index during a secondary index lookup, an RDBMS using physical pointers has to update each index's corresponding leaf node once any tuple location is changed. Several DBMSs such as PostgreSQL employ this scheme.

5.2 Lookup in Hermit

Hermit can generate accurate lookup results for both tuple identifier schemes. Figure 3 shows the entire workflow of Hermit's lookup mechanism. We list the key steps as follows:

Step 1. *TRS-Tree lookup* – This step performs a lookup on TRS-Tree as described in Section 4.3. The results are a set of ranges on the host column and a set of tuple identifiers.

Step 2. *Host index lookup* – This step performs lookups on the host index with the returned host column ranges as inputs. The result is a set of tuple identifiers, which is further unioned with the set of identifiers returned from Step 1.

Step 3. *Primary index lookup (optional)* – This step occurs only if the RDBMS adopts logical pointers as tuple identifiers. It looks up the primary index with the returned set of tuple identifiers as inputs. The result is a set of tuple locations.

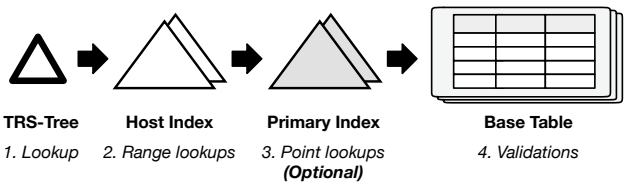


Figure 3: The workflow of Hermit's lookup mechanism.

Step 4. *Base table validation* – This step fetches the actual tuples using tuple locations and validates whether each tuple satisfies the input predicates. This step returns all the qualified results to the input query.

Please note that a primary index can also serve as the host index, in which case the lookup procedure shall be the same.

Compared to conventional secondary index methods, Hermit can bring in additional overhead due to the extra host index lookup phase as well as the base table validation phase. The overhead is exacerbated when using logical pointers as the tuple identifier scheme, as it involves unnecessary primary index lookups for unqualified matchings. However, we argue that such overhead is insignificant when performing range queries, which are prevalent for secondary indexes. This is because the number of false positives for range queries is quite small when compared to that of the qualified tuples. Moreover, as a TRS-Tree greatly reduces the space consumption, it brings huge benefit to modern main-memory RDBMSs, where memory space is precious.

6 DISCUSSION

Tradeoff between space and computation. Compared to conventional indexing mechanisms, TRS-Tree achieves great space saving by sacrificing access performance. While the actual space used by TRS-Tree is dependent on the correlation quality (i.e., how correlate the two columns are), we can indeed strike a balance by tuning the *error_bound* parameter. Let us consider a scenario with *max_height* set to 1, where TRS-Tree becomes a single-node, single-layer structure. Now we tune the *error_bound* parameter and analyze how TRS-Tree behaves. In an extreme case where *error_bound* is set to 0, Hermit shall identify every single data that cannot be perfectly covered by the generated linear function as outlier. In this case, Hermit can consume more memory but achieves optimal lookup performance. The increase of *error_bound* can effectively drop the memory consumption in the expense of reducing lookup performance. This is because TRS-Tree enlarges the returned bound for the lookups, and consequently introducing more false positives. However, as we shall see in our experiments, TRS-Tree's performance is actually not quite sensitive to the value of *error_bound* parameter, as long as it is set to small enough. The key reason is that TRS-Tree navigates lookups based on simple linear function computation, which is much cheaper than chasing pointers and retrieving every single elements from the standard index structure.

Due to the space limit, please refer to Appendix D for detailed discussion on correlation recovery, optimization, complex machine learning models, and several other issues.

7 EVALUATION

7.1 Experiment Setups

We implemented Hermit in two different RDBMSs: DBMS-X, a main-memory RDBMS prototype built internally in IBM – Almaden, and PostgreSQL [2], a disk-based RDBMS that is widely used in backing modern database applications. We performed all the experiments on a commodity machine running Ubuntu 16.04 with one 6-core Intel i7-8700 processor clocked at 3.20 GHz. The machine has a 16 GB DRAM and one PCIe attached 1 TB NVMe SSD.

We use three different applications to evaluate Hermit: Synthetic, Stock, and Sensor. We provide the detailed descriptions of these applications in Appendix A.

Throughout this section, we compare two mechanisms:

◇ **Hermit**: the correlation-based secondary indexing mechanism proposed in this paper.

◇ **Baseline**: the standard B+-tree-based secondary indexing mechanism used in conventional RDBMSs.

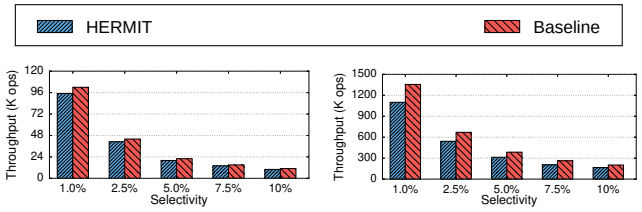
The B+-tree in DBMS-X is fully maintained in memory. It is highly optimized for modern CPUs with many advanced techniques such as cache-conscious layout and SIMD instructions (for numerical keys) applied. The node size is set to 256 bytes. PostgreSQL instead implements a page-based B+-tree backed by buffer pool. In our experiments, we have reconfigured the buffer pool size to ensure that the B+-tree is fully cached in memory.

Without any explicit declaration, we set TRS-Tree's parameters, including *node_fanout*, *max_height*, *outlier_ratio*, and *error_bound*, to 8, 10, 0.1, and 2, respectively. TRS-Tree achieves a good space-computation tradeoff with this configuration, as we shall see later.

7.2 Real-World Applications

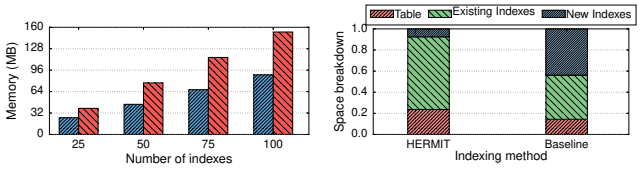
In this subsection, we evaluate Hermit's performance using real-world applications in DBMS-X.

The first experiment uses the Stock application to evaluate Hermit's range query performance. It is simple for Hermit's TRS-Tree to model the correlations between a stock's daily highest and lowest prices, as they form a near-linear correlation. One thing worth noticing is that there does not exist any strict bound between the two prices, and stock price can increase/drop by over 50% in a single day (see PG&E stock (`NYSE: PCG`)). Hermit shall identify and



(a) Logical Pointer (b) Physical Pointer

Figure 4: Range lookup throughput with different selectivities (Stock).



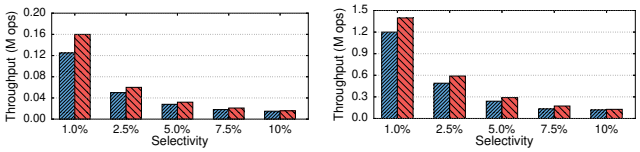
(a) Memory Consumption (b) Space Breakdown

Figure 5: Memory consumption with different numbers of indexes (Stock).

maintain these readings as outliers. Figure 4 shows the range lookup throughput with different selectivities. As we can see, Hermit yields a very competitive performance to the baseline solution, which requires building a complete secondary index on every single column. While Hermit suffers slightly from the overhead caused by false positive removal, its influence to the overall throughput is reduced with the increase of the selectivity.

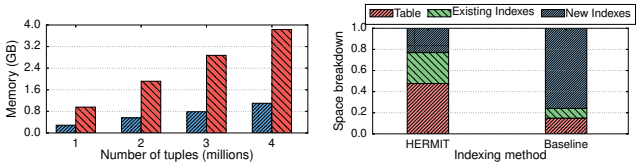
We then measure Hermit's memory consumption by changing the number of stocks stored in the table. Figure 5a shows the result. When setting the number of indexes to 25, it means we store 25 stocks in the table, as we build one index for each stock's lowest price column. The result indicates that Hermit's TRS-Tree takes little memory space, and the RDBMS adopting Hermit consumes only half of the memory compared to adopting the baseline solution, which creates one index for each column. In fact, Hermit in this case spends a great fraction of memory for storing outliers, and this guarantees a small false positive ratio, as we shall see later. Figure 5b provides a space breakdown to confirm this finding.

Next, we test Hermit's performance using the Sensor application. Supporting fast data retrieval in this application is challenging, as each sensor reading column has a non-linear correlation with the average reading columns. Figure 6 shows the throughput with different range lookup selectivities. When setting the selectivity to 1.0%, Hermit yields a throughput that is around 22% lower than the baseline solution. However, the performance gap diminishes with



(a) Logical Pointer (b) Physical Pointer

Figure 6: Range lookup throughput with different selectivities (Sensor).



(a) Memory Consumption (b) Space Breakdown

Figure 7: Memory consumption with different numbers of tuples (Sensor).

the growth of the selectivity. This is because Hermit generates approximate results, and the time ratio of filtering out false positives decreases with the increase of the result size.

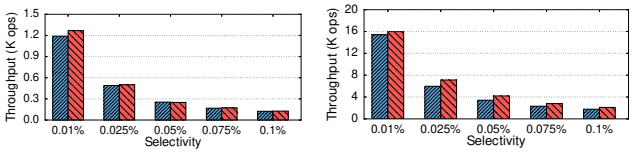
We then use the same application to measure Hermit's memory consumption. As Figure 7a shows, the space consumed by the baseline solution grows much faster than that consumed by Hermit. According to the analytics in Figure 7b, the baseline solution spends most of the memory maintaining newly created secondary indexes. In contrast, Hermit's TRS-Tree takes much less space, and most of the space is used for storing outliers.

7.3 Lookup

In this subsection, we evaluate Hermit's range and point lookup performance using the Synthetic application, with both Linear and Sigmoid correlation functions. We perform all the experiments in DBMS-X.

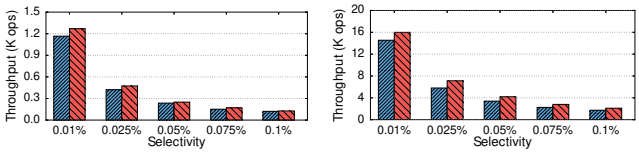
We use both Hermit and the baseline method to index column col_C. Modeling Linear correlation function is trivial using Hermit' TRS-Tree structure. However, it can be challenging to model Sigmoid correlation function, which is polynomial.

Figure 8 depicts the performance of Hermit and the baseline mechanism with Linear correlation function. We set the number of tuples to 20 million and measure the throughput changes of the range lookup queries with different query selectivities. We also adopt different tuple identifier methods to show their impacts on the performance. The result indicates that Hermit's performance is very close to that achieved by the baseline. Using logical pointers, Hermit and the baseline respectively proceed 1.19 and 1.27 K operations per second (K ops) with selectivity set to 0.01%. This gap is reduced with



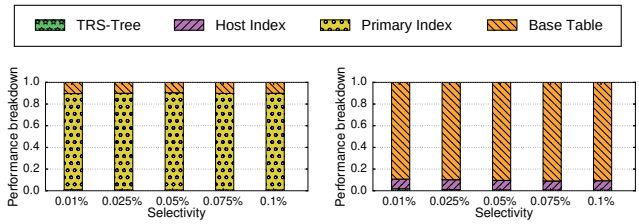
(a) Logical Pointer (b) Physical Pointer

Figure 8: Range lookup throughput with different selectivities (Synthetic – Linear).



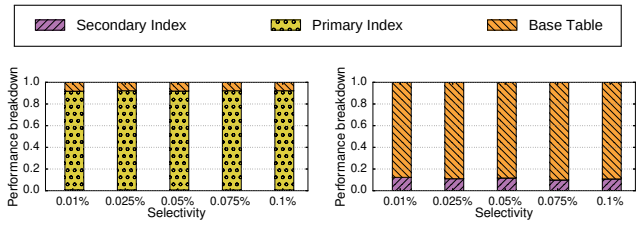
(a) Logical Pointer (b) Physical Pointer

Figure 9: Range lookup throughput with different selectivities (Synthetic – Sigmoid).



(a) Logical Pointer (b) Physical Pointer

Figure 10: Hermit's range lookup performance breakdown with different selectivities (Synthetic – Sigmoid).

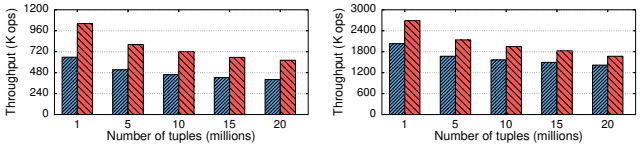


(a) Logical Pointer (b) Physical Pointer

Figure 11: Baseline's range lookup performance breakdown with different selectivities (Synthetic – Sigmoid).

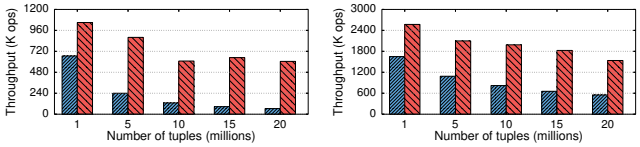
the increase of selectivity. The experiments with physical pointers also demonstrate the same results. One of the reasons is that Hermit's TRS-Tree only needs to use a single leaf node to model the correlation function, yielding optimal performance. We then use Sigmoid function to test whether TRS-Tree can efficiently model complex correlations. The results in Figure 9 show that the performance gap is little changed. Observing this, we decide to perform a detailed analysis to understand where the time goes.

Figure 10 and Figure 11 respectively show the performance breakdown of Hermit and the baseline method. The time



(a) Logical Pointer (b) Physical Pointer

Figure 12: Point lookup throughput with different numbers of tuples (Synthetic – Linear).

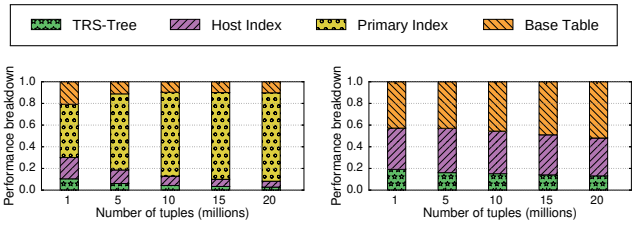


(a) Logical Pointer (b) Physical Pointer

Figure 13: Point lookup throughput with different numbers of tuples (Synthetic – Sigmoid).

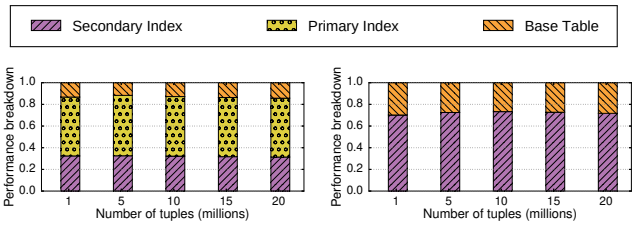
includes both CPU and memory IO. Recall that Hermit performs lookups through the following steps: TRS-Tree lookup, host index lookup, primary index lookup (optional), and base table validation. In contrast, the baseline method only needs to perform secondary index lookup, primary index lookup (optional), and base table access. With logical pointers as tuple identifiers, both methods spend over 90% of their time on the primary index lookup. This is inevitable because an RDBMS using logical pointers does not directly expose the tuple location to any index other than the primary one. When identifying tuples using physical pointers, the major bottleneck of both methods shifted to the base table access. While one-time tuple retrieval from the base table using its tuple location seems to be trivial, the total number of the fetches is actually equivalent to the total number of returned tuples, which can be expensive in range queries.

Despite the high efficiency for range queries, Hermit suffers from some performance degradation on point lookups, due to its introduction of false positives. We now measure the point lookup throughput by increasing the number of tuples in the database. Figure 12 shows the result with Linear correlation function. Using logical pointers for tuple identifiers, Hermit's throughput is 35% lower than that achieved by the baseline method, when the number of tuples is set to 20 millions. This is because Hermit's TRS-Tree lookup results in not only multiple unnecessary lookups on host and primary indexes, but also additional validation phase on the base table. We also observe that such a performance gap is reduced to 15% when switching the identifier method to physical pointers. The key reason is that the absence of expensive primary index lookups helped reduce Hermit's performance overhead. Figure 13 shows the point lookup performance



(a) Logical Pointer (b) Physical Pointer

Figure 14: Hermit's point lookup performance breakdown with different tuple counts (Synthetic – Sigmoid).



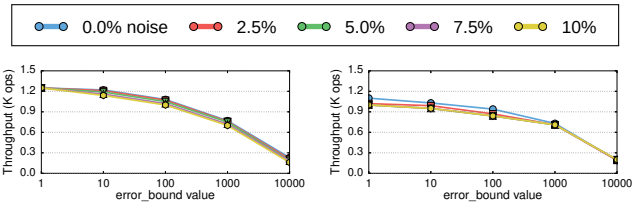
(a) Logical Pointer (b) Physical Pointer

Figure 15: Baseline's point lookup performance breakdown with different tuple counts (Synthetic – Sigmoid).

when using Sigmoid correlation function. Hermit's performance degrades with more tuples. The key reason is that the increasing tuple count makes correlation function more difficult to model, hence Hermit's TRS-Tree can generate more false positives for its subsequent processes, eventually degrading the performance.

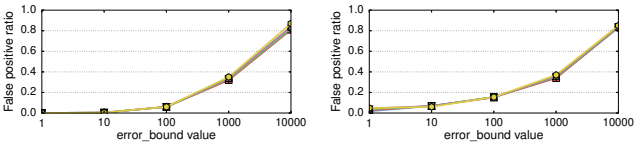
We further perform a performance breakdown to better understand the point queries. Figure 14 and Figure 15 show the results. There are two points worth noticing. First, using logical pointers, Hermit spends an increasing amount of time on primary index lookup when the tuple count increases. As explained above, this is because the larger tuple count indicates more complex correlation relations, and Hermit has to waste more time on retrieving unqualified tuples from the primary index. Second, compared to the baseline method, Hermit spends a larger portion of time on the base table. This is because Hermit has to validate every single tuple fetched from the base table to filter out false positives.

Hermit's performance can be affected by the correlation quality as well as the user-defined parameters. Now we control the percentages of the injected noise as well as the value of *error_bound* to see how Hermit behaves. Figure 16 shows the range lookup (selectivity set to 0.01%) throughput with different percentages of injected noise and different *error_bound* values. We use logical pointers in the experiment. We observe that Hermit's performance drops drastically with the increase of *error_bound*. This is because larger *error_bound* indicates more false positives, and Hermit has to perform redundant secondary index lookups and rely on



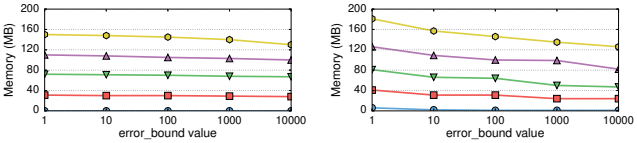
(a) Linear Correlation (b) Sigmoid Correlation

Figure 16: Range lookup throughput with different percentages of injected noises and *error_bound* values (Synthetic).



(a) Linear Correlation (b) Sigmoid Correlation

Figure 17: Range lookup false positive ratio with different percentages of injected noises and *error_bound* values (Synthetic).

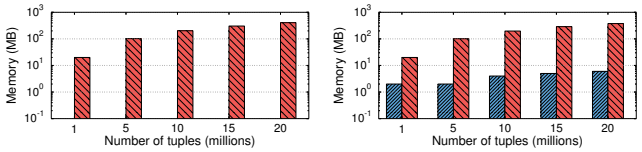


(a) Linear Correlation (b) Sigmoid Correlation

Figure 18: Memory consumption with different percentages of injected noises and *error_bound* values (Synthetic).

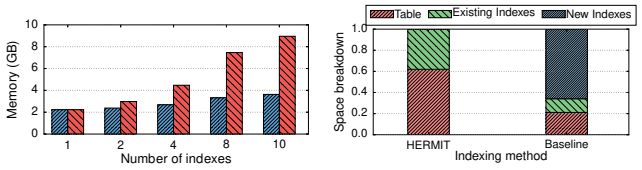
the validation phase to remove unqualified tuples. This is confirmed by Figure 17, which shows that the false positive rate reaches up to 80% when *error_bound* is set to 10,000. An interesting finding is that Hermit's performance remains stable with the increase of noise percentage. The key reason is that Hermit is capable to capture any outlier that fall beyond its generated linear function, and it can effectively find these outliers from the corresponding outlier buffers.

Figure 18 further shows how noisy data and *error_bound* values affect Hermit's memory consumption. Our first finding is that the memory consumption grows linearly with the increase of noise percentage. This is because Hermit stores noisy data in outlier buffers, as explained above. Our second finding is that the memory consumption declines by increasing the *error_bound* values. This is because larger *error_bound* covers more data, and hence Hermit's TRS-Tree can construct less nodes to capture the correlations. One thing worth mentioning is the memory spent for capturing the Linear and Sigmoid correlations tend to be the same (close to 120 MB) when *error_bound* is set to 10,000. This matches our expectation, as Hermit's TRS-Tree spent most



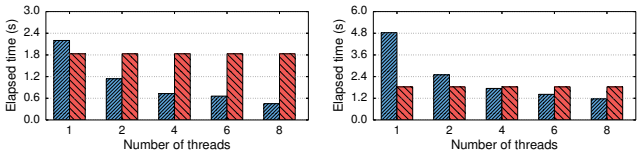
(a) Linear Correlation (b) Sigmoid Correlation

Figure 19: Index memory consumption with different numbers of tuples (Synthetic).



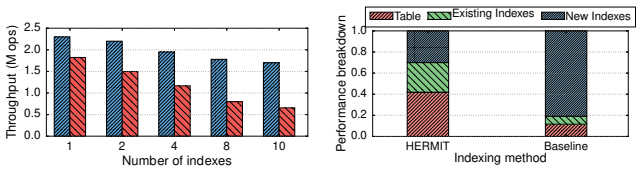
(a) Memory Consumption (b) Space Breakdown

Figure 20: Total memory consumption with different numbers of tuples (Synthetic – linear).



(a) Linear Correlation (b) Sigmoid Correlation

Figure 21: Index construction time with different numbers of threads (Synthetic).



(a) Insertion Throughput (b) Performance Breakdown

Figure 22: Index insertion throughput with different numbers of indexes (Synthetic – Linear).

of the space for storing outliers, and only little space is used for model correlations.

We observe that memory usage for capturing Sigmoid drops a lot when changing *error_bound* from 1 to 10, but we do not really see a noticeable decline in throughput. This is because TRS-Tree can identify too many data as outliers with *error_bound* set to 1. With the increase of *error_bound*, it efficiently captures correlations using linear regression models. It also navigates lookups using computation rather than chasing pointers, hence achieving good performance.

7.4 Space Consumption

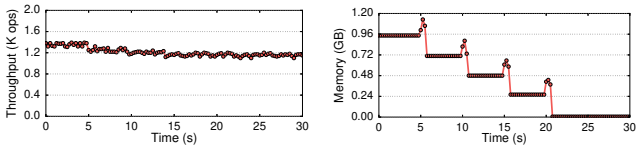
Hermit trades performance for space efficiency. Its goal is to greatly reduce the storage space while achieving "good enough" tuple retrieval speed. In the last subsection, we showed that Hermit yields competitive performance to the conventional secondary index mechanisms when supporting database operations, especially range queries. Now we measure Hermit's space efficiency using the Synthetic application. We still run all the experiments in DBMS-X.

The first experiment measures the amount of memory used respectively by TRS-Tree and conventional secondary index on col_C. Figure 19 shows that, compared to the baseline solution, Hermit takes little space to index the column. An extreme case is to use Hermit's TRS-Tree for capturing Linear correlation function. In this case, TRS-Tree only needs to use a constant amount of memory (a few bytes) to record the linear function's parameters. When modeling Sigmoid function, TRS-Tree consumes more memory (less than 10 MB) as the number of tuples increases, because TRS-Tree needs to construct more leaf nodes to better fit the correlation curve. However, Hermit's memory consumption is still negligible compared to the baseline solution, which takes close to 400 MB.

Next, we measure the overall memory consumption caused by Hermit. Other than the existing indexes on col_A and col_B, we add some additional columns and build one index on each of them. All these newly added columns are correlated to col_B. Figure 20a shows that, when adopting the baseline solution, the amount of memory consumption grows near linearly with the increasing number of newly added indexes. Specifically, the database used up to 8.5 GB memory when supporting 10 secondary indexes. In contrast, when adopting Hermit, the database only consumes 2.4 GB memory, which is a significant gain in memory utilization compared to the baseline solution. Figure 20b further depicts the memory usage breakdown when the number of indexes is set to 10. The space consumed by Hermit's TRS-Tree is negligible compared to that used by the base table and the primary index. However, when adopting the baseline solution, the database application has to use over 70% of the memory to maintain the secondary indexes. This result further confirms Hermit's space efficiency.

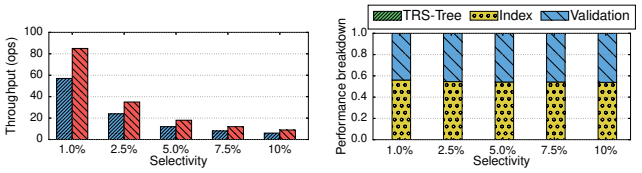
7.5 Construction

The construction of Hermit's TRS-Tree is different from that of the conventional B+-tree-like index structures. In this experiment, we measure the time for constructing TRS-Tree in DBMS-X with different numbers of threads. We compare the results with that obtained by constructing the B+-tree index. The B+-tree is built using single-thread bulk loading, as it currently does not support multithreading mode. We leave the comparison with concurrent B+-tree construction



(a) Lookup Throughput (b) Memory Consumption

Figure 23: Index reorg. performance (Synthetic-Sigmoid).



(a) Lookup Throughput (b) Performance Breakdown

Figure 24: Range lookup performance in PostgreSQL.

as a future work. The results in Figure 21 contain two interesting findings. First, TRS-Tree needs more time to finish the index construction when confronting complex correlation functions, such as Sigmoid. This is because TRS-Tree needs to perform multiple rounds of computations to calculate the leaf nodes' linear functions. Second, TRS-Tree's construction time drops near linearly with the increase of threads. This is because TRS-Tree constructs its internal structures using a top-down mechanism, hence it can be easily parallelized.

7.6 Insertion

Different from existing machine learning based index structures that require expensive retraining in face of data changes, Hermit can dynamically support operations like insertion, deletion, and updates at runtime. In this experiment, we use DBMS-X to compare Hermit's insertion performance with conventional secondary indexes. Figure 22a depicts the overall insertion throughputs with different numbers of indexes. Please note that we take into account the time for updating the primary index and the base table. These results are obtained with Linear correlation function and logical pointers, and we observed the same trend with other configuration combinations. As the result shows, when setting the number of indexes to 10, Hermit can process 1.7 million insert operations per second, which is 2.6 times higher than that achieved by the conventional secondary index scheme. The major reason is that Hermit's TRS-Tree only needs to update the leaf nodes' outlier buffers when necessary, which is pretty lightweight. Figure 22b further explains the result. Using the baseline mechanism, the database application has to spend over 80% of the time for inserting tuples into the secondary indexes. This demonstrates the inefficiency of the conventional indexing mechanism for supporting inserts.

7.7 Maintenance

Hermit's TRS-Tree can support online structure reorganization to re-optimize its efficiency. In this experiment, we show how Hermit's range lookup throughput and memory consumption changes during the process of index reorganization. We first create TRS-Tree on a table with 10 K tuples, and then insert another 19,990 K tuples to the table, yielding 20 million tuples in total. After that, we trigger structure reorganization every 5 seconds, each time reorganizing 1/4 of the structure (given our default *node_fanout* = 8, the reorganization procedure reorganizes 2 first-level subtrees each time). Note that this is an artificial scenario for testing purpose only. In real life scenarios, TRS-Tree can adjust its reorganization frequency based on the update rates, and the reorganization process would happen in parallel with updates. During the test, each partial reorganization takes around 2 seconds to finish. Figure 23 shows a 30-second trace of range lookup throughput (selectivity = 0.01%) and memory consumption. As we can see, the range lookup throughput remains stable during the reorganization. In general, reorganization reduces the sizes of outlier buffers, resulting in less number of direct pointer chasing during query processing. At the same time, it also produces more tree nodes, hence contributing to more precise characterization of the correlation. These two factors balance out during the process. The memory consumption drops significantly thanks to the structure reorganization. However, we also observed instant spike during the start of each reorganization. This is because the background thread needs to perform table scan and materialize corresponding data in order to compute linear function.

7.8 Disk-Based RDBMSs

Now we integrate Hermit into a popular disk-based RDBMS, namely PostgreSQL. We use the Sensor application to compare Hermit's lookup performance with that of PostgreSQL's secondary indexing mechanism (which is denoted as the baseline solution). Please note that PostgreSQL adopts physical pointers for tuple identifiers. We implemented a PostgreSQL client using libpqxx [1] to issue queries. We still keep Hermit's TRS-Tree in memory.

Figure 24a shows Hermit's range lookup performance in PostgreSQL. Similar to what we observed before, the performance gap drops with the increase of the selectivity. When setting the selectivity to 1.0%, Hermit is over 30% slower than the baseline solution. The major reason is that fetching data from secondary storage is more expensive than fetching from main memory. Figure 24b further depicts the breakdown. Not surprisingly, TRS-Tree lookup is negligible

compared to host index lookup in PostgreSQL. Validating false positives also take times, as our implementation materializes and then iterates the result set of the host index lookup. One may optimize the performance by pushing the filter operator down to the index scan.

8 RELATED WORK

Tree index structures. B+-Tree [10] is the textbook index for disk-oriented DBMSs and its structure is well-designed to reduce random disk accesses. With the decrease of main memory prices, researchers and practitioners have developed memory-friendly indexes that can efficiently leverage the larger main memory and fast random access speed. Some pioneering works include T-tree [28] and cache-conscious indexes [33]. All these indexes use the hierarchical tree structure to return accurate query results in a timely manner. However, these solutions can lead to high memory consumptions, causing high pressure to main-memory RDBMSs.

Succinct index structures. Sparse indexes such as column imprints [34] and Hippo [39] only store pointers to disk pages (or column blocks) in parent tables and value ranges in each page (or column block) to reduce the space overhead. Column Sketch [16] indexes tables on a values-by-value basis but compresses the values into a lossy map. The tradeoff is that these structures can introduce false positives in query time. BF-Tree [5] is an approximate index designed for ordered or partitioned data. While generating unqualified results, it can largely reduce the space consumption by only recording approximate tuple locations. The learned index [25] improves space efficiency by exploiting data distribution using machine learning techniques. It yields good performance but requires a long training phase to generate the data structure. Zhang et al. [42] proposed a new range query filtering mechanism for log-structured merge trees.

Stonebraker [35] introduced the partial index that stores only a subset of entries from the indexed columns to reduce the number of leaf nodes. Idreos et al. [18, 19] developed a series of techniques called database cracking to adaptively generate indexes based on the query workload. Specifically, partial sideways cracking [18] introduces an index called partial maps which consists of several self-organized chunks. These chunks can be automatically dropped or re-created according to the remaining storage space such that the maximum available space is exploited. Athanassoulis et al. [6] later proposed the RUM conjecture to capture the relations among read, update, and memory overhead.

Compression techniques [12, 43] drop redundant data information to save storage space. However, these techniques require extra time for compressing data ahead of time and decompressing data at query time. This compromises the query performance and index maintenance speed. In addition, they still store the pointers for tuples such that the amount of saved memory is limited.

Secondary index selection. Several works have also discussed how to select secondary indexes given a fixed amount of space budget. A group of researchers at Microsoft proposed a mechanism that analyzes a workload of SQL queries and suggests suitable indexes [9]. They further presented an end-to-end solution to address the problem of selecting materialized views and indexes [4]. Researchers at IBM modeled the index selection problem as a variant of the knapspack problem, and introduced an index recommendation mechanism into the DB2. Most recently, Pavlo et al. [32] investigated this problem using a machine learning based approach.

Column correlations. BHUNT [7] automatically discovers algebraic constraints between pairs of columns. By relaxing the dependency, CORDS [20] uses sampling to discover correlations and soft functional dependencies between columns. In addition, CORDS recommends groups of columns on which to maintain certain simple joint statistics. Researchers on data cleansing also put lots of efforts on detecting functional dependencies including soft dependency and approximate dependency [8, 26]. CORADD [22] proposes a correlation-aware database designer to recommend the best set of materialized views and indexes for given database size constraints. Correlation Maps [21] (CM) is a data structure that expresses the mapping between correlated attributes for accelerating unindexed column access. While sharing a similar idea of leveraging column correlations to save space, Hermit does not require using clustered columns; more importantly, its ML-enhanced TRS-Tree structure can adaptively and dynamically model both complex correlations and outliers, hence yielding better performance in many cases.

Cardinality estimation. Cardinality estimation plays a crucial role in RDBMS query optimizers. Column correlation is the most common reason that encumbers the estimation. Sample views [27] and PSALM [41] use sampling methods to detect the column correlation. Recent projects [29, 30] start treating column semantics as a black box and use machine learning models to learn cardinalities from query feedbacks. Kipf et al. [23] opt to use deep learning techniques to learn cardinalities for join queries.

9 CONCLUSIONS

We have introduced Hermit, a new secondary indexing mechanism that exploits column correlations to reduce index space consumption. Hermit utilizes TRS-Tree, a succinct, ML-enhanced tree structure to adaptively and dynamically capture complex correlations and outliers. Our extensive experimental study has confirmed Hermit's effectiveness in both main-memory and disk-based RDBMSs.

REFERENCES

[1] libpqxx. http://pqxx.org/development/libpqxx/.
[2] PostgreSQL. http://www.postgresql.org.
[3] Simple linear regression. https://en.wikipedia.org/wiki/Simple_linear_regression.
[4] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes for SQL Databases. In *VLDB*, 2000.
[5] M. Athanassoulis and A. Ailamaki. BF-Tree: Approximate Tree Indexing. *PVLDB*, 7(14), 2014.
[6] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. Designing Access Methods: The RUM Conjecture. In *EDBT*, 2016.
[7] P. G. Brown and P. J. Hass. BHUNT: Automatic Discovery of Fuzzy Algebraic Constraints in Relational Data. In *VLDB*, 2003.
[8] L. Caruccio, V. Deufemia, and G. Polese. Relaxed Functional Dependencies - A Survey of Approaches. *IEEE TKDE*, 28(1), 2016.
[9] S. Chaudhuri and V. R. Narasayya. An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*, 1997.
[10] D. Comer. The Ubiquitous B-Tree. *CSUR*, 11(2), 1979.
[11] P. Godfrey, J. Gryz, and C. Zuzarte. Exploiting Constraint-Like Data Characterizations in Query Optimization. In *SIGMOD*, 2001.
[12] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing Relations and Indexes. In *ICDE*, 1998.
[13] G. Graefe et al. Modern B-Tree Techniques. *Foundations and Trends® in Databases*, 3(4):203–402, 2011.
[14] G. Graefe, H. Volos, H. Kimura, H. A. Kuno, J. Tucek, M. Lillibridge, and A. C. Veitch. In-Memory Performance for Big Data. *PVLDB*, 8(1), 2014.
[15] J. Gryz, B. Schiefer, J. Zheng, and C. Zuzarte. Discovery and Application of Check Constraints in DB2. In *ICDE*, 2001.
[16] B. Hentschel, M. S. Kester, and S. Idreos. Column Sketches: A Scan Accelerator for Rapid and Robust Predicate Evaluation. In *SIGMOD*, 2018.
[17] Hidden. Retracted due to double-blind constraints.
[18] S. Idreos, M. L. Kersten, and S. Manegold. Self-Organizing Tuple Reconstruction in Column-Stores. In *SIGMOD*, 2009.
[19] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Store. *PVLDB*, 4(9), 2011.
[20] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies. In *SIGMOD*, 2004.
[21] H. Kimura, G. Huo, A. Rasin, S. Madden, and S. B. Zdonik. Correlation Maps: A Compressed Access Method for Exploiting Soft Functional Dependencies. *VLDB*, 2(1), 2009.
[22] H. Kimura, G. Huo, A. Rasin, S. Madden, and S. B. Zdonik. CORADD: Correlation Aware Database Designer for Materialized Views and Indexes. *PVLDB*, 3(1-2), 2010.
[23] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR*, 2019.
[24] J. Kivinen and H. Mannila. Approximate Inference of Functional Dependencies from Relations. *Theoretical Computer Science*, 149(1), 1995.
[25] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The Case for Learned Index Structures. In *SIGMOD*, 2018.
[26] S. Kruse and F. Naumann. Efficient Discovery of Approximate Dependencies. *PVLDB*, 11(7), 2018.
[27] P. Larson, W. Lehner, J. Zhou, and P. Zabback. Cardinality Estimation Using Sample Views with Quality Assurance. In *SIGMOD*, 2007.
[28] T. J. Lehman and M. J. Carey. A Study of Index Structures for Main Memory Database Management Systems. In *VLDB*, 1986.
[29] H. Liu, M. Xu, Z. Yu, V. Corvinelli, and C. Zuzarte. Cardinality Estimation Using Neural Networks. In *CASCON*, 2015.
[30] T. Malik, R. C. Burns, and N. V. Chawla. A Black-Box Approach to Query Cardinality Estimation. In *CIDR*, 2007.
[31] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J. Rudolph, M. Schönberg, J. Zwiener, and F. Naumann. Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms. *PVLDB*, 8(10), 2015.
[32] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, et al. Self-Driving Database Management Systems. In *CIDR*, 2017.
[33] J. Rao and K. A. Ross. Making B+-Trees Cache Conscious in Main Memory. In *SIGMOD*, 2000.
[34] L. Sidirourgos and M. L. Kersten. Column Imprints: A Secondary Index Structure. In *SIGMOD*, 2013.
[35] M. Stonebraker. The Case for Partial Indexes. *Sigmod Record*, 18(4), 1989.
[36] G. Valentin, M. Zuliani, D. C. Zilio, G. Lohman, and A. Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *ICDE*, 2000.
[37] A. D. Well and J. L. Myers. *Research design & statistical analysis*. Psychology Press, 2003.
[38] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *PVLDB*, 10(7), 2017.
[39] J. Yu and M. Sarwat. Two Birds, One Stone: A Fast, yet Lightweight, Indexing Scheme for Modern Database Systems. *PVLDB*, 10(4), 2016.
[40] 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 *SIGMOD*, 2016.
[41] H. Zhang, I. F. Ilyas, and K. Salem. PSALM: Cardinality Estimation in the Presence of Fine-Grained Access Controls. In *ICDE*, 2009.
[42] 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 *SIGMOD*, 2018.
[43] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE*, 2006.

A APPLICATIONS IN THE EXPERIMENTS

Synthetic: The synthetic data contains one single table with four 8-byte numeric columns, namely col_A, col_B, col_C, and col_D. Columns col_B and col_C are correlated, as col_B's values are generated by a certain correlation function from col_C, i.e. $col_B = Fn(col_C)$. We use two types of correlation functions: *Linear* function and *Sigmoid* function. We also inject uniformly distributed noisy data to col_B. By default, we inject 1% noises (percent = $\frac{abnormal\ tuples}{cardinality}$). We have already built a primary index on col_A and a secondary index on col_B. The application frequently queries on col_C to retrieve values on col_D. Our experiments build indexes on col_C.

Stock: This application records the market price of 100 stocks in the U.S. stock market over the last 60 years. We store over 15,000 rows containing datetime and daily highest and lowest prices of these 100 stocks in a wide table (201 columns in total). We set the entries to `NULL` if certain readings are not available. Each pair of the highest and lowest price columns forms a simple near-linear correlation. We build a primary index on the datetime column, and a set of secondary indexes on each lowest price column. The application continuously issues queries to those unindexed highest price columns. The queries are like: "during which time periods do Stock `X`'s highest price fall between `Y` and `Z`?". Our experiments build indexes on all the unindexed columns and evaluate the performance.

Sensor: This application monitors chemical gas concentration using 16 sensors. We store 4,208,260 rows containing the timestamp, the 16 sensor readings, and the reading average in a single table (18 columns in total). These 16 sensor reading columns and the average reading column form a non-linear correlation. We have constructed one index on the average reading column. The application continuously queries one of those 16 unindexed sensor reading columns. The queries are like: "during which time period do the readings in Sensor X fall between Y and Z?". Our experiments build indexes on all the unindexed columns and evaluate the performance.

B MAINTENANCE

TRS-Tree can easily support concurrent insertions. As a single insert/delete/update operation only affects at most one leaf node, TRS-Tree can easily guarantee the structure consistency by using concurrent hash tables to implement the leaf nodes' outlier buffers.

TRS-Tree also enables *online* structure reorganization at runtime without incurring much overhead to any concurrent operations. Unlike conventional concurrent tree-based structures, TRS-Tree does not implement latch coupling, which can be overly complicated and expensive for TRS-Tree. Instead, it adopts a coarse-grained latching protocol to maximize concurrency. The intuition behind this decision is that insert/delete/update operations in TRS-Tree never trigger cascading node modifications, and the reorganization happens infrequently and can be processed with low latency.

TRS-Tree uses a flag to identify the reorganization phase. A dedicated background thread starts the reorganization by setting the flag to true. When observing this flag, any concurrent insert / delete / update operations append their modifications to a temporal buffer to avoid phantom problems. The background thread then scans and retrieves all corresponding entry pairs and subsequently creates the new tree nodes. Before installing these nodes to TRS-Tree, the background thread further holds a coarse-grained latch on the entire tree and applies all the changes in a temporal buffer. The latch is released once the new nodes are installed.

C COMPARISON

Compare with Correlation Maps (CM). TRS-Tree in Hermit is a ML-enhanced tree index. CM [21] adopts a map-like structure which stores the bucket mappings between correlated columns. Both CM and HERMIT leverage column correlations to save space. But we find that these two proposals are drastically different.

• TRS-Tree captures correlations using tiered curve fitting and handles outliers. This makes it robust to noisy data which is prevalent in real-world applications. In contrast, CM does not include any scheme to handle outliers, and hence its performance can drop when confronting sparsely distributed noisy data.

• TRS-Tree adaptively constructs its internal structures and automatically decides the partition granularity. It dynamically maintains its internal structures, and performs reorganization in the presence of large amounts of insert/delete/update operations. In comparison, CM relies on its tuning advisor to decide the granularity for its single-layer buckets by building multiple histograms beforehand. It is unknown how CM adapts to dynamic workload where the underlying data drastically change over time.

• Hermit is a general secondary indexing mechanism and can exploit multiple correlations on the same table. In contrast, CM can only exploit correlations when there is a clustered index on the host column. At most one clustered index can exist in a table.

We also empirically compare Hermit with CM in Appendix E.

Compare with BF-Tree. BF-Tree [5] is an approximate index that exploits implicit ordering and clustering in the underlying data to reduce storage overhead. It adopts the same tree structure with B+ Tree but stores a set of Bloom Filters in its leaf nodes. Those Bloom Filters record the approximate physical locations of values. Although both BF-Tree and Hermit tend to reduce index size by introducing false positives, they act very differently.

• BF-Tree exploits implicit ordering and clustering in the underlying data to reduce the index size. Hermit leverages the correlation between the host column and the indexed column to shrink its size on the indexed column.

• BF-Tree requires that the underlying data should be ordered or at least have some clusterings. Hermit does not have any specific requirement for the data distribution. It works for any data order.

• BF-Tree stores Bloom Filters and disk page ranges in its leaf nodes. For every key lookup on the indexed column, these Bloom Filters may return "true" for some non-existing keys and thus result in page scans on the false positive disk pages. The TRS-Tree in Hermit stores ML models (linear regression in our paper) in leaf nodes. This range may include some false positive values which need to be pruned later.

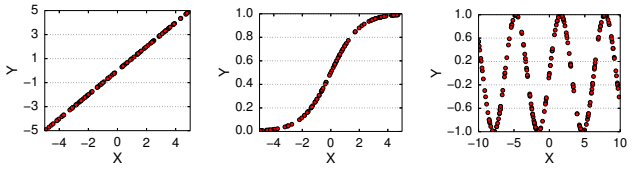
D MORE DISCUSSIONS

D.1 Correlation Discovery

Hermit fully relies on the underlying RDBMS or users to perform correlation discovery. There has been a flurry of systems that addressed the problem of correlation (or functional dependency) discovery in different ways. In the past two decades, researchers extensively studied how to automatically find all functional dependencies (including those among composite columns) in a database. To accelerate the discovery, they [8, 20, 24, 26, 31] opt to leverage specific rules to prune columns or compute approximate coefficients based on samples.

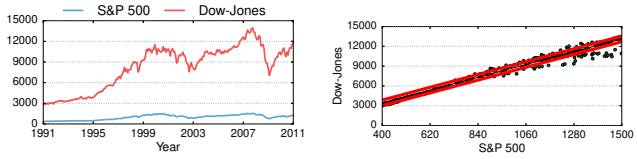
In practice, most commercial RDBMSs still largely rely on "human knowledge" to discover *possibly* correlated columns because of the huge search space of column combinations. A database administrator (DBA) can identify candidate columns that exhibit semantic relationships, then evaluate the correlation using different correlation coefficients, including *Pearson coefficient* and *Spearman coefficient* [37]. Once the coefficient reaches a certain predefined threshold, the DBA can recommend this correlation information to the database optimization module.

Now let us discuss correlation types Hermit may capture. Figure 25 shows three different functions: (1) linear (e.g., $y = x$) (2) monotonic (e.g., $y = sigmoid(x)$) (3) non-monotonic (e.g., $y = sin(x)$). We do not consider noisy data at this moment. TRS-Tree can perfectly index the correlations in both (1) and (2) since it directly navigates a key lookup to a single value on the indexed host column. A DBA can easily capture these two correlations using Pearson coefficient and Spearman coefficient, respectively (coefficient = 1). However, Hermit cannot yield good performance for non-monotonic correlation like *sin* function because a single value



(a) $y = x$ **(b)** $y = sigmoid(x)$ **(c)** $y = sin(x)$

Figure 25: Three different correlation functions.



(a) Stocks over year **(b) Correlation function**

Figure 26: Dow-Jones and S&P 500 values. In (b), Green dots falling beyond red lines are identified as outliers.

on the host column can be mapped to many values on the target column. Consequently, TRS-Tree can generate many false positives, resulting in low performance. The DBA can detect a non-monotonic correlation using Spearman coefficient (coefficient = 0).

Hermit captures correlations and handles outliers. This makes Hermit applicable to cases where nice difference bounding does not exist. Figure 26a shows the value trace of Dow-Jones and S&P 500 during the years 1991 to 2011. While the two indices are correlated in most years, we observe some major shifts in certain months or years. Hermit handles them by directly identifying and maintaining them in outlier buffers, and hence eliminate some false positives.

D.2 Optimization

Sampling-based outlier estimation. During the index construction, a TRS-Tree needs to compute the linear function parameters, namely slope β and intercept α, of the leaf nodes using the standard linear regression formula (see Section 4.1). Sometimes the default construction algorithm shown in Algorithm 1 unnecessarily computes these parameters even if the corresponding node later splits due to too many outliers. We adopt a sampling-based strategy to avoid this problem. Before performing the parameter computation, our optimization algorithm randomly samples the data (by default 5%) covered by the range and runs the simple linear regression on them. Within the sample set, if the number of outliers has already exceeded the pre-defined threshold, then the construction directly partitions the range. This helps us make the decision quickly.

Multi-threaded index construction. Algorithm 1 shows how we construct TRS-Tree using a single thread. Observing the popularity of massively parallel processors, we can now leverage multi-threading to speed up the construction algorithm. Different from B-tree, Hermit constructs its internal and leaf nodes following a top-down scheme. This means that we can parallelize the tree construction without confronting any synchronization points. Assuming that the index fanout is set to k, we can easily parallelize the splitting and computation of TRS-Tree's every single node using k threads. These threads proceed independently without incurring any inter-thread communication.

D.3 Complex Machine Learning Models

Table 1: Training time for different ML models

Number of tuples	1 K	10 K	100 K
Linear regression	0.42 ms	0.81 ms	3.2 ms
SVR (RBF)	0.09 s	4.5 s	> 60 s
SVR (linear)	0.28 s	29 s	> 60 s
SVR (polynomial)	0.29 s	24 s	> 60 s

Hermit performs a linear regression in each TRS-Tree node which costs a scan of corresponding tuples. Actually, TRS-Tree's structure is also flexible enough to adopt more complex models such as Support Vector Regression (SVR) and neural networks. However, although these models may yield less false positives, training these models takes tremendous time (orders of magnitude slower than linear regression) if the table size increases significantly.

To prove that, we run a set of machine learning models on different scales of data and report the training time in Table 1. As depicted in the table, training linear regression models only takes several milliseconds while training SVR models with different kernels including RBF, linear and polynomial is at least 200 times slower. Having said that, we believe that researchers and practitioners still can easily extend Hermit to incorporate other models and fit in their specific scenarios.

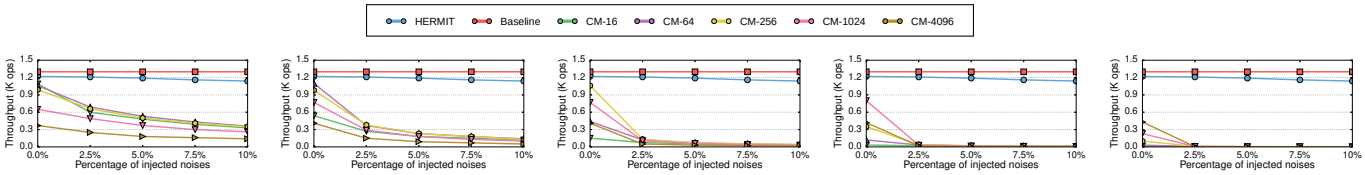
In addition, a recent paper featuring learned indexes [25] discusses the cases of using complex machine learning models such as neural networks and multivariate regression models to predict locations of keys. As opposed to learned indexes, Hermit models the correlation between two columns and leverages the curve-fitting technique to adaptively create simple yet customized ML models for different regions (TRS-Tree tree nodes).

D.4 Hermit on Secondary Storage

Although the storage overhead of an index may not seem too expensive on traditional Hard Disk Drives (HDDs), the dollar cost increases dramatically when the index is deployed on modern storage devices (e.g., Solid State Drives and Non-Volatile Memory) because they are still more than an order of magnitude expensive than HDDs. This is also an important issue when deploying RDBMSs on the cloud, where the customers are charged on a pay-as-you-go basis. Hermit is a generic indexing mechanism designed for both main-memory and disk-based RDBMSs. Deploying it on precious secondary storage such as SSD can save considerable storage budget and still exhibit comparable lookup performance as opposed to B+Tree. Our experiment in Section 7.8 also confirmed this claim.

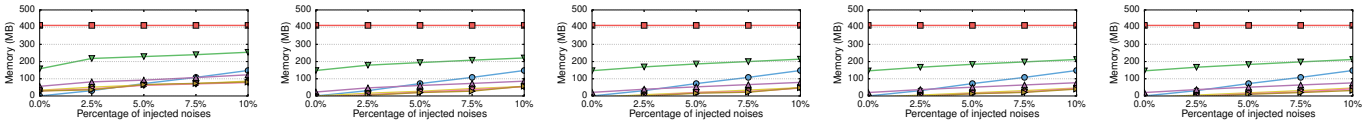
D.5 Complex SQL Queries

As a replacement of classic B+-tree-based secondary indexes, Hermit is general enough and can be used in any complex queries whenever a classic secondary index is used. Even for complex queries like join, RDBMSs can still use Hermit to execute local predicate, consequently accelerating the processing of the entire query plan.



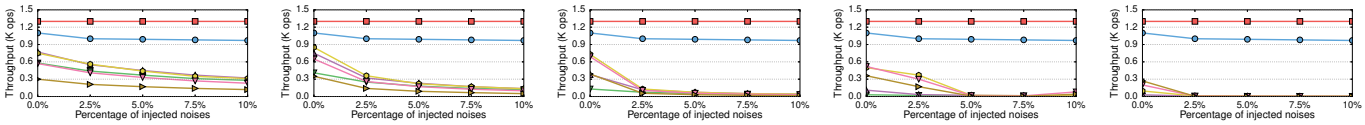
(a) Host bucket size = 2^4 (b) Host bucket size = 2^6 (c) Host bucket size = 2^8 (d) Host bucket size = 2^{10} (e) Host bucket size = 2^{12}

Figure 27: Range lookup throughput with different percentage of injected noises (SYNTHETIC-LINEAR).



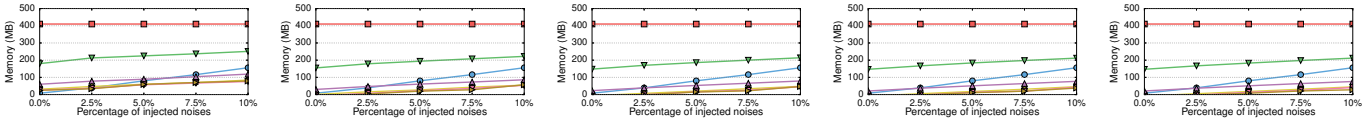
(a) Host bucket size = 2^4 (b) Host bucket size = 2^6 (c) Host bucket size = 2^8 (d) Host bucket size = 2^{10} (e) Host bucket size = 2^{12}

Figure 28: Memory consumption with different percentage of injected noises (SYNTHETIC-LINEAR).



(a) Host bucket size = 16 (b) Host bucket size = 2^6 (c) Host bucket size = 2^8 (d) Host bucket size = 2^{10} (e) Host bucket size = 2^{12}

Figure 29: Range lookup throughput with different percentage of injected noises (SYNTHETIC-SIGMOID).



(a) Host bucket size = 16 (b) Host bucket size = 2^6 (c) Host bucket size = 2^8 (d) Host bucket size = 2^{10} (e) Host bucket size = 2^{12}

Figure 30: Memory consumption with different percentage of injected noises (SYNTHETIC-SIGMOID).

E EXPERIMENTS

We compare HERMIT with an existing solution, namely Correlation Maps (CM). We implemented CM faithfully based on its original paper. Instead of implementing CM's tuning advisor, we performed parameter sweeping and tuned the bucket size in both target and host columns to evaluate the performance. Throughout this section, we use *host bucket size* to refer to the bucket size in host column. We use *CM-X* to denote CM with the bucket size in target column set to X (e.g., CM-16 means the bucket size in target column is 16).

Figure 27 and Figure 28 show the range lookup throughput (selectivity=0.01%) and memory consumption of HERMIT, CM, and the B+-tree-based baseline solution using SYNTHETIC-LINEAR. We change the percentage of injected noises from 0% to 10%. Since CM was designed for disk-based RDBMSs, [21] showed that CM usually performs better with a smaller bucket size. Now CM is adapted to in-memory databases, this does not always hold true any more. The host index look up and base table access are much faster now in memory, thus the overhead of accessing the CM structure itself plays a bigger role in the overall performance. A smaller bucket size means more buckets in CM and accordingly more overhead for accessing the CM structure. We also observed a compute-storage tradeoff in CM. While CM's lookup throughput drops with the increase of bucket size, its memory consumption is actually reduced. Another key observation in these figures is that CM's performance can drop significantly with the increase of percentage of injected noises. This is because CM has to maintain mappings among buckets for every single entry pairs in the target and host columns. Even with small amount of sparsely distributed outliers, CM in the extreme case may have to scan the entire table to remove outliers (just consider the case where the outliers are scattered to every single bucket in the target column). Compared to CM, HERMIT can sustain a high throughput even when the noise percentage is increased to 10%. This is because CM can identify and maintain outliers in leaf nodes' outlier buffers. The tradeoff is that its memory consumption can increase. Overall, HERMIT and CM both can reduce memory consumption compared to B+-tree. However, HERMIT can achieve much better performance in the presence of outliers, and it saves much more memory as the correlation information is well captured by the tiny linear regression models.

Figure 29 and Figure 30 further show the experiment results with SYNTHETIC-SIGMOID. We also observed similar results in this set of experiments. The only difference is that HERMIT needs to spend more memory to capture correlations.