HERMIT in Action: Succinct Secondary Indexing Mechanism via Correlation Exploration

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. However, we find that there indeed exist many opportunities to save storage space by exploiting column correlations. We recently introduced 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. In this demonstration, we showcase HERMIT's appealing characteristics. we not only demonstrate that HERMIT can significantly reduce space consumption with limited performance overhead in terms of query response time and index maintenance time, but also explain in detail the rationale behind HERMIT's high efficiency using interactive online query processing examples.

1. INTRODUCTION

Modern relational database management systems (RDBMSs) support fast secondary indexes that help accelerate query processing

---

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

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 valuable resource in enabling ultra-fast query execution.

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 [5, 1]. While satisfying the space constraints, these techniques 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. These works either store only a subset of the column entries [4] or use compression techniques to reduce space consumption [9]. However, such solutions save limited amount of space and can cause high overhead for lookup operations.

Given these deficiencies in existing solutions, we introduced **HERMIT** [7], a new secondary indexing mechanism that attempts to address this problem in a third way. The main observation that sparks HERMIT's design 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 data structure to capture the correlation between M and N. HERMIT uses *Tiered Regression Search Tree*, or TRS-TREE, to capture the correlation. TRS-TREE exploits multiple simple statistical regression processes to fit the curve of the hidden correlation function. 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. Like several existing index structures [3, 2, 8], TRS-TREE returns approximate results. HERMIT ensures the correctness by removing any unqualified tuples via base table validation. Different from existing machine learning-based indexing solutions, TRS-
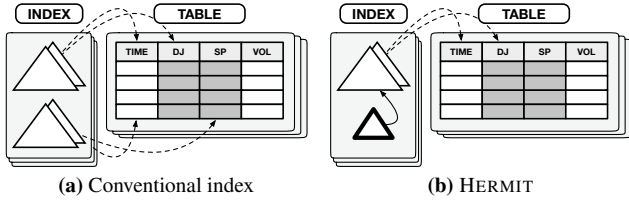
(a) Conventional index **(b)** HERMIT

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

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.

TREE efficiently handles inserts, deletes, and updates, and supports on-demand structure reorganization to re-optimize the index efficiency at system runtime. This ensures that HERMIT can sustain good performance even if data distribution change occurs.

HERMIT yields 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 demonstration 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.

2. HERMIT OVERVIEW

HERMIT is a secondary indexing mechanism that leverages column correlations to accelerate query processing. 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 execute 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.

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 column DJ. Now she decides to create another index on column SP due to the frequent occurrence of queries on this column. 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 DJ, the RDBMS then constructs a TRS-TREE to model the correlation mapping from SP to DJ. Given the query range (S_{min}, S_{max})
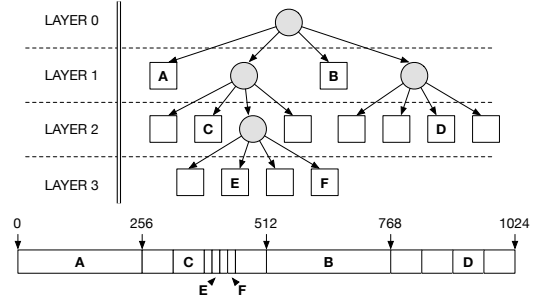
on column SP, HERMIT first inputs this range to the constructed TRS-TREE to fetch the corresponding range (D_{min}, D_{max}) on DJ. Then it searches the host index on DJ with the generated range (D_{min}, D_{max}) to find all the satisfying tuples. To filter out false positives, the RDBMS reads the SP values from the base table and validates the correctness of the result. Figure 1 shows how HERMIT is different from conventional secondary indexing mechanisms when retrieving tuples in the example.

3. HERMIT DESIGN

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 supports both transactional and analytical workloads. The constructed TRS-TREE processes any insert, delete, and update operation 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 very valuable. We now introduce TRS-TREE, the key component of HERMIT, and then present how HERMIT works with TRS-TREE. The detailed description can be found in [7].

3.1 TRS-Tree

TRS-TREE is a k-ary tree structure that maps the values in the target column M to that in the host column N. It 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:

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

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. The confidence interval ϵ can be computed based on a user-defined parameter, called *error_bound*. 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. 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 uncovered 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 tuple's identifier in the form of either a primary key or a tuple location.

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 with 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.

Construction algorithm. The construction algorithm recursively divides M's value range into k uniform sub-ranges until every entry pair (m, n) from M and N covered by the corresponding sub-range can be well estimated by a simple linear regression-based data mapping. 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. This construction algorithm utilizes a FIFO 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.

Lookup algorithm. The lookup algorithm 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 the query predicate. Any overlapping child node will be pushed to the FIFO queue. The lookup algorithm continues iterating until the queue is empty. When reaching leaf nodes, TRS-TREE computes an intersection range between the query predicate and the node's value range. Using this range, the node can then use its linear function to compute the estimated range on N that covers the exact matching.

Maintenance algorithm. At system runtime, TRS-TREE can dynamically support all of the commonly used database operations, including insertions, deletions, and updates. 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. For deletion, TRS-TREE does not perform any computation after locating the
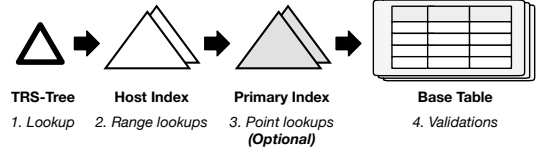

Figure 3: The workflow of HERMIT's lookup mechanism.

leaf node. Instead, it directly checks the outlier buffer and removes the corresponding entry if exists.

Online reconfiguration. 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. For the second case, TRS-TREE checks the node's neighbors to determine whether merging is beneficial. To perform structure reorganization, a 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.

3.2 HERMIT Workflow

TRS-TREE lookup returns only approximate results. To obtain the real matching for the input queries, HERMIT needs to collaborate with the underlying RDBMS and remove all the false positive results. Figure 3 shows the entire workflow of HERMIT's lookup mechanism. We list the key steps as follows: (1) TRS-TREE *lookup* – This step performs a lookup on TRS-TREE. The results are a set of ranges on the host column and a set of tuple identifiers. (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. (3) *Primary index lookup (optional)* – This step occurs only if the RDBMS adopts logical pointers as tuple identifiers [6]. It looks up the primary index with the returned set of tuple identifiers as inputs. The result is a set of tuple locations. (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.

4. DEMONSTRATION PLAN

We have developed a generalized framework to demonstrate HERMIT's high efficiency in terms of both space consumption and query response/index maintenance time. The framework contains two parts: (1) a real-time performance monitoring component that continuously reports HERMIT's runtime performance numbers; (2) an interactive query processing interface that allows users to submit and execute SQL statements. From this demonstration, the audience is expected to understand that (1) HERMIT can be easily configured; (2) HERMIT's TRS-TREE can be constructed within a short period of time; (3) HERMIT can achieve competitive lookup performance; (4) HERMIT can significantly reduce storage consumption; (5) HERMIT can use online structure reconfiguration to reoptimize performance; and (6) how HERMIT works and why HERMIT achieves high efficiency.

Figure 4: HERMIT demonstration framework.

4.1 Configuration

As shown in Figure 4, the users can configure the workload and try out different HERMIT parameters before exploring HERMIT's features. Our demonstration allows users to run HERMIT in two different DBMSs: PostgreSQL, a popular disk-based DBMS, and DBMS-X, an in-memory DBMS prototype built internally in IBM – Almaden. We provide users with three real-world datasets: Stock, Sensor, and Ticket. The detailed descriptions of these datasets can be found in [7]. Besides these datasets, the users may also generate customized dataset by providing user-defined correlation functions as well as other necessary parameters such as variable range and outlier ratio. Given these parameters, our framework can populate a table with two correlated columns generated using the provided correlation function. Before executing online queries, our system needs to populate the table based on the selected dataset and then construct HERMIT's TRS-TREE (as well as a standard B+-tree, for runtime comparison reason) using a number of cores as configured by the users. Afterwards, the system will run mixed queries containing both range lookups and insert/update/delete queries. The users can set the ratio of range lookup operations and the selectivities.

After setting up the workload, the users then need to configure HERMIT. The users can easily tune HERMIT's performance using four parameters: $node\_fanout$, which controls the fanout of TRS-TREE's internal nodes; max_height, which limits the maximum height of TRS-TREE; $outlier\_ratio$, which constrains the maximum capacity of the outlier buffers in TRS-TREE's leaf nodes; and $error\_bound$, which balances the tradeoff between lookup performance and storage consumption. The detailed description of these parameters can be found in [7]. During the demonstration, we plan to provide default parameter values for new HERMIT users.

4.2 Runtime Performance

Once the configuration is ready, our demonstration framework will start executing queries. As shown in Figure 4, our framework can continuously report the lookup throughput and latency numbers achieved by HERMIT. For comparison, it also presents the lookup performance achieved by standard B+-tree and full table scan. The framework further shows the storage space and index construction time consumed by HERMIT's TRS-TREE and standard B+-tree. To help the users understand HERMIT's runtime behavior, the framework can visualize the real-time false positive ratio yielded by HERMIT (recall that HERMIT's tree only generates approximate results). The users can also click "Performance Diagnose" button to further check the performance breakdown. When using customized dataset, the users may experience how HERMIT reacts to dataset changes by clicking the "Online Dataset Update" button to update

the dataset on-the-fly. Once the dataset reconfiguration is ready, our framework will start populating the two correlated columns with newly configured correlation function and parameters. The users are expected to observe how HERMIT's TRS-TREE performs online index reorganization (including node merge and split) to reoptimize its runtime performance.

Our demonstration framework not only showcases HERMIT's excellent efficiency, but also explains the rationale behind it. This is achieved through an interactive query processing interface. Once the users type in a valid SQL statement, our framework will analyze and notify the users whether and why the input statement can or cannot leverage HERMIT to accelerate query processing. The framework then executes this query and reports the performance achieved by HERMIT as well as standard B+-tree and full table scan.

5. CONCLUSION

We plan to demonstrate HERMIT, a novel succinct secondary indexing mechanism that leverages column correlation to significantly reduce space consumption. Our demonstration not only presents HERMIT's high efficiency in terms of both space consumption and lookup performance, but also explains the rationales behind HERMIT via interactive query processing.

6. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes for SQL Databases. In *VLDB*, 2000.

[2] M. Athanassoulis and A. Ailamaki. BF-Tree: Approximate Tree Indexing. *PVLDB*, 7(14):1881–1892, 2014.

[3] H. Kimura, G. Huo, A. Rasin, S. Madden, and S. B. Zdonik. Correlation Maps: A Compressed Access Method for Exploiting Soft Functional Dependencies. *PVLDB*, 2(1):1222–1233, 2009.

[4] M. Stonebraker. The Case for Partial Indexes. *SIGMOD Record*, 18(4), 1989.

[5] 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.

[6] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *PVLDB*, 10(7):781–792, 2017.

[7] Y. Wu, J. Yu, Y. Tian, R. Sidle, and R. Barber. Designing Succinct Secondary Indexing Mechanism by Exploiting Column Correlations. In *SIGMOD*, 2019.

[8] J. Yu and M. Sarwat. Two Birds, One Stone: A Fast, yet Lightweight, Indexing Scheme for Modern Database Systems. *PVLDB*, 10(4):385–396, 2016.

[9] 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.