Query Dictionary for Frequent Non-Indexed Queries in HTAP Databases

SUCHARITHA SHETTY¹, B. DINESH RAO², AND SRIKANTH PRABHU¹, (Senior Member, IEEE)
¹Department of Computer Science and Engineering, Manipal Institute of Technology, Manipal Academy of Higher Education, Manipal 576104, India
²Manipal School of Information Sciences, Manipal Academy of Higher Education, Manipal 576104, India
Corresponding author: B. Dinesh Rao (dinesh.rao@manipal.edu)

ABSTRACT The increasing demand for the simultaneous transaction and review of the data for either decision making or forecasting has created a need for faster and better Hybrid Transactional/Analytical Processing (HTAP). This paper emphasizes the speedup of Online Analytical Processing (OLAP) operations in an HTAP environment where analytical queries are mainly repetitive and contain non-indexed keys as their predicates. Zone maps and materialized views are popular approaches adopted by more extensive databases to address this issue. However, they are absent in in-memory databases because of space constraints. Instead, in-memory databases load the cache with result pages of frequently accessed queries. Increasing the number of such queries can fill the cache and raise the system’s overhead. This paper presents Query_Dictionary, a hybrid storage solution that leverages the full capabilities of SQLite by retaining less information of repetitive queries in the cache and efficiently accommodating the newly updated data by the end-user. The solution proposes storing page-level metadata query information for a larger result set and row-level information for a smaller result set. It demonstrates Query_Dictionary capabilities on three types of representative queries: single table, binary join, and transactional queries on non-indexed attributes. In comparison with SQLite, the proposed method performs better.

INDEX TERMS Data skipping, materialized views, query rewriting, HTAP database.

I. INTRODUCTION
In the modern computing world, HTAP generally adopts in-memory techniques [1]. Storing the entire database structure in the memory is the signature characteristic of in-memory databases. The main challenge of the in-memory database is to work faster with its limited memory. This issue has led to extensive research in query optimization with the efficient design of index structures as a by-product [2]. A recent study of modern in-memory database systems shows that index lookup can contribute to 94% of query execution time [3], [4]. These indexes can prove helpful if the query is repetitious. In contrast, if the queries contain non-indexed attributes in the WHERE clause, the search time will be longer, as illustrated in Figure 1 and Figure 2, which presents the display of the task manager for 50 indexed searches and 50 non-indexed searches, respectively, in an SQLite setting.

In the marketplace, many of the fastest-growing in-memory databases such as Systems,

The associate editor coordinating the review of this manuscript and approving it for publication was Vlad Diaconita.

Applications & Products - High-Performance Analytical Appliance (SAP-HANA) [5], TimesTen can combine transactional and analytical processing in one system with only one copy of the data. While these systems have a faster speedup, they still have a lower throughput due to the high resource contention between transactional and analytical workloads [6], [7].
Many web-based services have emerged due to the Internet and mobile computing. However, data dispersion and heterogeneity make interrogation necessary but difficult, especially if the data is non-indexed. User studies have shown that high user latency drives customers away [8]. Similarly, queries or calls to stored procedures/user-defined functions are typically called numerous times in a relational model, either within a loop in an application program or from the WHERE/SELECT clause of the outer query [9]. These queries contain either indexed or non-indexed keys in their predicates. Search on non-indexed columns is non-trivial and the most expensive operation in an analytical database. Examples of non-indexed queries include partial character string matching, a join of two tables using non-unique join keys [10].

Clustered and non-clustered indices are among the few methods explored for the above concern in the current state of the technology. The clustered index search goes straight to a specific point in the results, allowing to read in order from there. It reduces page transfers and increases cache hits. On the other hand, it also generates several constant page fragments, including both data and index pages. While non-clustered indexes help store data logically, they do not permit physically sorting the data rows. The lookup method on a non-clustered index becomes prohibitively expensive.

Other popular approaches adopted by more extensive databases are data skipping and materialized views. These approaches are significant but insufficient to make substantial contributions to repetitive queries in in-memory databases. Examples of non-indexed queries include partial character string matching, a join of two tables using non-unique join keys [10].

Clustered and non-clustered indices are among the few methods explored for the above concern in the current state of the technology. The clustered index search goes straight to a specific point in the results, allowing to read in order from there. It reduces page transfers and increases cache hits. On the other hand, it also generates several constant page fragments, including both data and index pages. While non-clustered indexes help store data logically, they do not permit physically sorting the data rows. The lookup method on a non-clustered index becomes prohibitively expensive.

Other popular approaches adopted by more extensive databases are data skipping and materialized views. These approaches are significant but insufficient to make substantial contributions to repetitive queries in in-memory databases. Examples of non-indexed queries include partial character string matching, a join of two tables using non-unique join keys [10].

Clustered and non-clustered indices are among the few methods explored for the above concern in the current state of the technology. The clustered index search goes straight to a specific point in the results, allowing to read in order from there. It reduces page transfers and increases cache hits. On the other hand, it also generates several constant page fragments, including both data and index pages. While non-clustered indexes help store data logically, they do not permit physically sorting the data rows. The lookup method on a non-clustered index becomes prohibitively expensive.

Other popular approaches adopted by more extensive databases are data skipping and materialized views. These approaches are significant but insufficient to make substantial contributions to repetitive queries in in-memory databases. Examples of non-indexed queries include partial character string matching, a join of two tables using non-unique join keys [10].

Clustered and non-clustered indices are among the few methods explored for the above concern in the current state of the technology. The clustered index search goes straight to a specific point in the results, allowing to read in order from there. It reduces page transfers and increases cache hits. On the other hand, it also generates several constant page fragments, including both data and index pages. While non-clustered indexes help store data logically, they do not permit physically sorting the data rows. The lookup method on a non-clustered index becomes prohibitively expensive.

Other popular approaches adopted by more extensive databases are data skipping and materialized views. These approaches are significant but insufficient to make substantial contributions to repetitive queries in in-memory databases. Examples of non-indexed queries include partial character string matching, a join of two tables using non-unique join keys [10].

Clustered and non-clustered indices are among the few methods explored for the above concern in the current state of the technology. The clustered index search goes straight to a specific point in the results, allowing to read in order from there. It reduces page transfers and increases cache hits. On the other hand, it also generates several constant page fragments, including both data and index pages. While non-clustered indexes help store data logically, they do not permit physically sorting the data rows. The lookup method on a non-clustered index becomes prohibitively expensive.

Other popular approaches adopted by more extensive databases are data skipping and materialized views. These approaches are significant but insufficient to make substantial contributions to repetitive queries in in-memory databases. Examples of non-indexed queries include partial character string matching, a join of two tables using non-unique join keys [10].

Clustered and non-clustered indices are among the few methods explored for the above concern in the current state of the technology. The clustered index search goes straight to a specific point in the results, allowing to read in order from there. It reduces page transfers and increases cache hits. On the other hand, it also generates several constant page fragments, including both data and index pages. While non-clustered indexes help store data logically, they do not permit physically sorting the data rows. The lookup method on a non-clustered index becomes prohibitively expensive.

Other popular approaches adopted by more extensive databases are data skipping and materialized views. These approaches are significant but insufficient to make substantial contributions to repetitive queries in in-memory databases. Examples of non-indexed queries include partial character string matching, a join of two tables using non-unique join keys [10].

Clustered and non-clustered indices are among the few methods explored for the above concern in the current state of the technology. The clustered index search goes straight to a specific point in the results, allowing to read in order from there. It reduces page transfers and increases cache hits. On the other hand, it also generates several constant page fragments, including both data and index pages. While non-clustered indexes help store data logically, they do not permit physically sorting the data rows. The lookup method on a non-clustered index becomes prohibitively expensive.

Other popular approaches adopted by more extensive databases are data skipping and materialized views. These approaches are significant but insufficient to make substantial contributions to repetitive queries in in-memory databases. Examples of non-indexed queries include partial character string matching, a join of two tables using non-unique join keys [10].

Clustered and non-clustered indices are among the few methods explored for the above concern in the current state of the technology. The clustered index search goes straight to a specific point in the results, allowing to read in order from there. It reduces page transfers and increases cache hits. On the other hand, it also generates several constant page fragments, including both data and index pages. While non-clustered indexes help store data logically, they do not permit physically sorting the data rows. The lookup method on a non-clustered index becomes prohibitively expensive.

Other popular approaches adopted by more extensive databases are data skipping and materialized views. These approaches are significant but insufficient to make substantial contributions to repetitive queries in in-memory databases. Examples of non-indexed queries include partial character string matching, a join of two tables using non-unique join keys [10].

Clustered and non-clustered indices are among the few methods explored for the above concern in the current state of the technology. The clustered index search goes straight to a specific point in the results, allowing to read in order from there. It reduces page transfers and increases cache hits. On the other hand, it also generates several constant page fragments, including both data and index pages. While non-clustered indexes help store data logically, they do not permit physically sorting the data rows. The lookup method on a non-clustered index becomes prohibitively expensive.

Other popular approaches adopted by more extensive databases are data skipping and materialized views. These approaches are significant but insufficient to make substantial contributions to repetitive queries in in-memory databases. Examples of non-indexed queries include partial character string matching, a join of two tables using non-unique join keys [10].

Clustered and non-clustered indices are among the few methods explored for the above concern in the current state of the technology. The clustered index search goes straight to a specific point in the results, allowing to read in order from there. It reduces page transfers and increases cache hits. On the other hand, it also generates several constant page fragments, including both data and index pages. While non-clustered indexes help store data logically, they do not permit physically sorting the data rows. The lookup method on a non-clustered index becomes prohibitively expensive.
II. RELATED WORK
This section presents a survey of related works adopted by more extensive databases like data skipping and materialized views.

A. DATA SKIPPING
Scans with lightweight data skipping algorithms are used in modern main-memory optimized systems to filter data quickly. The scan uses data skipping when the data order is sorted, semi-sorted, or composed of clustered values. On the other hand, data skipping loses its usefulness when dealing with arbitrary data distributions. Using data skipping techniques on non-sorted data can dramatically increase query performance [11]. Partition pruning facilitates data skipping. Partition pruning (for example, in Oracle and Postgres) permits queries to skip partitions depending on partition key ranges (e.g., date). By extending this work, current systems support data block skipping. Lightweight statistics, such as minima and maxima values, are kept in these systems to describe virtual zones, contiguous sections of a data column. A scan can then use the metadata to determine if a zone is relevant to a query and skip or scan its underlying data. When supplemented with a priori user knowledge about query workloads and datasets, data skipping approaches can considerably improve scan performance than traditional indexing algorithms. They minimize the number of filtered records, reducing space and maintenance costs. Many recent frameworks, such as Netezza, Cloudera Impala, MySQL, and Oracle, have included data skipping strategies [13].

B. MATERIALIZED VIEWS
They can speed up frequently-used queries and views, lowering query processing costs. If the system often performs certain joins, it is always good to create join indexes for those joins [14]. One such technique for holding pre-joined tables are materialized views [15]. Furthermore, the system must maintain the materialized views, which is costly for frequently updated databases. Databases have adopted many incremental view maintenance solutions to reduce maintenance costs. Instead of re-materializing the view, incremental techniques refresh it by applying the view’s delta to the base data changes. Lazy maintenance strategies have also been found to reduce maintenance costs [16]. Materialized views and data skipping are the common approaches for faster execution of frequent non-indexed queries. The identified challenges in the above existing works on faster access of frequent non-indexed queries are change of projection attributes in the queries, space occupied to address frequent non-indexed queries, and handling arbitrary data distribution of non-indexed attributes in these queries.

III. BACKGROUND
We first provide an overview of the regular execution step followed by SQLite with non-indexed keys in the SELECT clause. The pseudo-code outlined in Algorithm 1 is a generalization of the regular query plan of an SQLite system [17].

The algorithm accepts the query string as input. If the predicate of the WHERE clause contains an indexed key, an indexed lookup on two binary trees is performed. First, a binary search using the indexed key is performed on the index table to fetch the rowid. The obtained rowid is used as a key to perform a binary search on the original table. If the predicate contains only rowid, the binary search is limited to the original table alone. The rowid column is a key that uniquely identifies the rows in the table. Finally, if the predicate contains a non-indexed key, the first query plan is executed using an auto-index based on the predicate. The automated indexes will only exist for the duration of a single query, will never persist to the disk, and will only be available for a single database connection. The second plan creates a covering index based on the first query plan on repetitive query execution. This new index includes all the columns used by the query, including the search terms and the output of the original table. Because all the required information is in the covering index, SQLite never consults the original table to find the records.

Algorithm 1 Query Plans Generated by SQLite

```plaintext
1: procedure QueryPlan(Query Q)
2:     if checkKey = predicate(WHERE clause) then
3:         if checkKey = IndexedKey then
4:             Fetch Rowid using B+tree index search
5:         end if
6:     else
7:         if checkKey = Rowid then
8:             Fetch Records using B-tree index search
9:         end if
10:     end if
11:     if checkKey = NonindexedKey then
12:         if Query is executed first time then
13:             Create auto_index on nonindexed key
14:         end if
15:     end if
16: end procedure
```

IV. QUERY_DICTIONARY OVERVIEW
A Query_Dictionary is an efficient and concise data structure for repetitive queries having non-indexed attributes as predicates. It is tailored for relational databases that need to scan many memory-resident or memory-mapped pages repeatedly. Query_Dictionary performs coarse-grain data filtering to save time loading data to the cache. For example, suppose the user queries a repetitive query with a non-indexed column in the WHERE clause. In that case, standard search techniques increase the burden of reading non-relevant memory pages each time. Query_Dictionary serves as a quick way...
to find relevant information. It decomposes a query into a collection of pertinent page numbers or rowids. This solution differs from the traditional materialized views that place complete resultset in memory. An ordered list of pageids or rowids is maintained based on the resultset obtained during the first-time execution of the query. The areas touched upon using non-indexed keys as predicates are point queries, scalar subquery expression, aggregate queries, binary join queries.

**Technical Overview:** The key idea behind working with the resultset obtained during the first-time execution is as follows: From the point of view of the cache, we present experimental results to introduce the two equivalent formulations of Query_Dictionary, one is rowid, and the other is page number. For cache sizes close to 2000Kb, 1000Kb, and 500Kb, Figure 3 demonstrates the curve obtained by running 100 queries with around 25 repetitive queries and shows a total execution time for all the 100 queries. The forthcoming subsection describes how the queries and corresponding data sets are generated. During the execution of the queries, storing details of around 40 page numbers per query gave better execution speedup and storage than keeping on an average 20, 60, 80, and 100 page numbers. As the cache size grows, Query_Dictionary can accommodate more queries. The optimal plan to serve more queries is to consider the average number of pages that can be accommodated per query and remove queries that exceed this number. Hence, an approximation threshold of around 40 page numbers per query is considered, and these queries are named as Scan_20 query type. Consequently, based on the results obtained in Figure 3 and the SQLite architecture, which provides 32-bits for page number and 64-bits for rowid, providing rowid for smaller resultset performed faster than the search on page numbers. Hence, queries with lower resultset, say 20 rowids are considered and termed as Scan_20 query type.

**A. SCHEMATIC OUTLINE OF THE QUERY-DICTIONARY ALGORITHM**

The Query Dictionary algorithm employs the following data structures.

- Query_Dictionary[i] denotes the ith location in the hash table. It contains the list of queries whose hash value of the string is i with the most frequent query at the beginning.
- QUERY-DS denotes the individual data structure of the queries. It contains the physical statistics of the executed queries.

These two data structures are included in the regular SQLite source code for the proposed methodology, keeping in mind: (i) **Backward compatibility** that is no modification to SQL Syntax (ii) **Performance** that is to keep negligible overhead on computational time and database size. Note that the layout ignores extra data structures required to speed up search loops, such as obtaining physical statistics. Despite the lack of such specifications, it is clear that the Query_Dictionary algorithm is straightforward and has minimal bookkeeping overhead.

The HTAP algorithms proposed are classified as single table OLAP queries, relational join OLAP queries, and OLTP commands. All queries including aggregates, sub-queries, GROUP BY and ORDER BY are considered in the single table queries. The modified code implements only binary join queries in relational join queries. The efficiency of the Query_Dictionary, which handles data changes transparently, is expressed in OLTP algorithm.

1) **SINGLE TABLE QUERY-DICTIONARY OLAP**

Algorithm 2 describes the core steps followed for single table query execution:

- The new query is parsed, rewritten, and searched in the Query_Dictionary with all the projection attributes replaced with the * symbol (line 3). If not present, Query_Dictionary stores the rewritten query and the physical statistics like the number of relevant pages or rowids.

- If the rewritten query is present in the Query Dictionary, its physical statistics are retrieved. As mentioned in the above technical overview subsection, two cases can happen in this scenario. The Scan_20 query type fetches the rowids. The retrieved rowids replace the * symbol following the SQL query’s WHERE clause (line 19).

Example 1: Consider the following repeated query:

```
SELECT s_name, s_phone FROM supplier
WHERE s_acctbal BETWEEN 500 AND 550;
```

Table 1 illustrates the VDBE code generated during the execution of the query demonstrated in Example 1. The table presents only a few intermediate codes in the whole VDBE code, those that are pertinent to the discussion in the proposed methodology. Other codes are presently ignored for the simplicity of the explanation. Line 1 of the generated VDBE code initiates an internal search loop for all the records in the SUPPLIER table [rootpageno=p2=5], displaying attributes of only selected rows [s_acctbal BETWEEN 500 AND 550]. In the absence of query rewriting, the same VDBE code is generated during the query’s repeated execution.
TABLE 1. Illustration of the VDBE Code for the query in Example 1.

| addr | opcode  | p1 | p2 | p3 |
|------|---------|----|----|----|
| 0    | Init    | 0  | 11 | 0  |
| 1    | OpenRead| 0  | 5  | 0  |
| 2    | Rewind  | .. | .. | .. |
| ..   | ..      | .. | .. | .. |
| 9    | Next    | 0  | 3  | 0  |
| 11   | Integer | 500| 3  | 0  |
| 12   | Integer | 550| 4  | 0  |

The query returns a single output having rowid=120. On repetitive execution of the query, with query rewriting approach, the query now becomes:

SELECT l_suppkey, l_linenumber
FROM lineitem
WHERE rowid = 120;

Table 2 shows the VDBE code for the preceding query, which skips iterating through the entire SUPPLIER table because the rowid [Integer 120] is provided directly.

TABLE 2. Illustration of newly generated VDBE code for the repetitive execution of the query in Example 1.

| addr | opcode  | p1 | p2 | p3 |
|------|---------|----|----|----|
| 0    | Init    | 0  | 10 | 0  |
| 1    | OpenRead| 0  | 9  | 0  |
| 2    | Rewind  | .. | .. | .. |
| ..   | ..      | .. | .. | .. |
| 8    | Next    | 0  | 3  | 0  |
| 11   | Integer | 587| 2  | 0  |

It is observed that searches on only 14 pages numbered 495,517,529,537,553,556,653,673,689,712,745,826 and 829 is sufficient. Table 4 presents a sample VDBE code with page number 495 referenced.

TABLE 3. Illustration of the VDBE Code for the query in Example 2.

| addr | opcode  | p1 | p2 | p3 |
|------|---------|----|----|----|
| 0    | Init    | 0  | 10 | 0  |
| 1    | OpenRead| 0  | 9  | 0  |
| 2    | Rewind  | .. | .. | .. |
| ..   | ..      | .. | .. | .. |
| 8    | Next    | 0  | 3  | 0  |
| 11   | Integer | 587| 2  | 0  |

Query Rewriting is carried out in two ways here:

1) SQLstmt_check_singlerowid: If Query_Dictionary returns a single rowid for the query, the WHERE clause predicate is replaced directly with the rowid.

2) SQLstmt_check_multiplerowids: The second approach leverages the use of the SQL “IN” operator. When the query contains more rows, the idea is to enclose the valid rowid’s in the “IN” operator, replacing the predicate in the WHERE clause.

During Query rewriting, the non-reductive operators are just push-down without any modifications (line 20). This strategy reduces the cost of executing queries, covering only the appropriate rowids. For a query containing GROUP BY clause, the rowids replace the predicate of the WHERE clause. Hence, the GROUP BY operator now operates on thin tuples.

- For the Scan_40 query type, the proposed method extracts page details from the Query_Dictionary. The loop generated by the VDBE code will execute with selected page numbers, as opposed to SQLite, which iterates over all pages of the query (lines 24-25).

Example 2: Consider the query:

SELECT o_orderkey, o_orderstatus
FROM orders
WHERE o_custkey = 587;

Table 3 shows the working of the VDBE code on the execution of the above query. Line 1 of the generated VDBE code initiates an internal loop search in the Btree table ORDERS [rootpageno=p2=9] using the WHERE clause predicate (o_custkey = 587 is the unindexed key).

In SQLite, the result requires searching all the pages related to the table and entering the page number in the p2 column of the generated VDBE code.

TABLE 4. Illustration of newly generated VDBE code for the repetitive execution of the query in Example 2.

| addr | opcode  | p1 | p2 | p3 |
|------|---------|----|----|----|
| 0    | Init    | 0  | 10 | 0  |
| 1    | OpenRead| 0  | 495| 0  |
| 2    | Rewind  | .. | .. | .. |
| ..   | ..      | .. | .. | .. |
| 9    | Next    | 0  | 3  | 0  |

In our proposed method, as in Table 4, caching the correct page numbers will accommodate more queries in the limited cache and reduce the search time by running the VDBE loop fewer times.

2) BINARY JOIN QUERY_DICTIONARY - OLAP

Binary joins provide significant flexibility and good performance on a wide range of workloads due to decades of optimization and fine-tuning. However, any binary join scheme performs poorly in pathological conditions [14]. The most significant disadvantage of binary joins is that they produce intermediate results that can be substantially larger than the query result. In complicated analytical situations, where non-key attribute joins are widespread, this condition is unfortunately unavoidable. For example, one possible query on the TPC-H schema would be to seek items delivered by the same supplier within the same order. Two non-key joins between LINEITEM and PARTSUPP are required to answer this query, both of which provide significant intermediate results. When dealing with such queries, traditional RDBMS performs poorly. One solution is to create appropriate indexes on all permutations of attributes for a join, which necessitates a significant amount of storage overhead.

The proposed approach to handle binary join, as shown in Algorithm 3, works as follows:

- The new query is parsed, rewritten, and searched in the Query_Dictionary with all the projection attributes
Algorithm 2 Query_Dictionary Algorithm for Single Table Queries

1: procedure SingleTableOLAP(Query Q)  
2: \[ i = \text{hash[RewriteQuery}(Q)] \]
3: if lookup(Query_Dictionary[i]) = NULL then  
4: \[ \Rightarrow \text{New Query} \]
5: Process the query Q as a normal SQLite query.  
6: resultcount = total_relevant_records  
7: if resultcount < 20 then  
8: Add resultcount, rowids and RewrittenQuery(Q) in the QUERY_DS  
9: Add QUERY_DS to Query_Dictionary  
10: else  
11: if resultcount < 40 then  
12: Add resultcount, page numbers and RewrittenQuery(Q) in the QUERY_DS  
13: Add QUERY_DS to Query_Dictionary  
14: else  
15: \[ \Rightarrow \text{Existing Query} \]
16: if QUERY_DS.resultcount < 20 then  
17: Replace the predicate in WHERE clause of query Q with the rowids  
18: Push down the non-reductive operators at the end of the query  
19: Process the replaced query as a normal SQLite query.  
20: Execute sqlite3_VDBEcode_exec()  
21: else  
22: Modify the VDBE code to specify page numbers from the Query_Dictionary[i]  
23: Execute sqlite3_VDBEcode_exec()  
24: end if  
25: end if  
26: end if  
27: end procedure  
28: end procedure

Algorithm 3 Query_Dictionary Algorithm for Binary Join Queries

1: procedure BinaryJoinOLAP(Query Q)  
2: \[ i = \text{hash[RewriteQuery}(Q)] \]
3: if lookup(Query_Dictionary[i]) = NULL then  
4: \[ \Rightarrow \text{New Query} \]
5: Process the query Q as a normal SQLite query.  
6: Add resultcount, resultant page numbers of bigger table and RewrittenQuery(Q) in the QUERY_DS  
7: Add QUERY_DS to Query_Dictionary  
8: else  
9: \[ \Rightarrow \text{Existing Query} \]
10: Modify the VDBE code by replacing relevant page numbers from the Query_Dictionary[i]  
11: Execute sqlite3_VDBEcode_exec()  
12: end if  
13: end procedure

Example 4: Consider the query
SELECT o_orderkey, c_acctbal  
FROM customer c, orders o  
WHERE o_custkey=c_custkey AND c_nationkey='11'  
AND c_mktsegment='building'  
AND c_acctbal=-413.03;

Table 5 demonstrates that in SQLite, both examples generate the same VDBE Code with the innermost loop as the smallest table CUSTOMER [rootpageno=p2=8].

| addr | opcode | p1 | p2 | p3 |
|------|--------|----|----|----|
| 0    | Init   | 0  | 10 | 0  |
| 1    | OpenRead | 0  | 9  | 0  |
| 2    | OpenRead | 0  | 8  | 0  |
| 3    | Rewind  |      |    |    |
|      |         |    |    |    |

Similar to Example 2, Table 6 demonstrates the Binary Join Query_Dictionary generated VDBE code, the outermost loop (i.e.) the larger table [ORDERS] alone will be provided with appropriate page numbers, say x, retaining the innermost loop with the smaller table [CUSTOMER] itself.

Table 6. Illustration of newly generated VDBE Code for the repetitive execution queries in Example 3 and Example 4.

| addr | opcode | p1 | p2 | p3 |
|------|--------|----|----|----|
| 0    | Init   | 0  | 10 | 0  |
| 1    | OpenRead | 0  | x  | 0  |
| 2    | OpenRead | 0  | 8  | 0  |
| 3    | Rewind  |      |    |    |
|      |         |    |    |    |

3) OLTP WITH QUERY_DICTIONARY

Hybrid RDBMS must support inserts, updates, and deletes while simultaneously answering OLAP queries [18]. There have been three main HTAP architectures proposed. The first
is the TCMF (two-copy mixed-format) architecture. TCMF maintains two copies of the data, one in row format for OLTP and the other in columnar format for OLAP. To keep the data constant throughout the OLTP and OLAP components, TCMF uses an intermediate data structure called delta. Delta keeps track of tuples that have been changed or added. TCMF periodically scans the delta to send new/modified tuples from the OLTP to the OLAP side. The second is SCMF (single-copy, mixed-format) architecture. SCMF employs the intermediate data structure delta as OLTP storage and stores the data in a single copy. OLTP transactions only affect the delta, but OLAP queries access both the delta and the primary copy of the data. The data in the delta is in row format, whereas the data in the main copy is in columnar format. Finally, SCSF (single-copy, single-format) architecture keeps a single copy of the data and uses a single format, i.e., only row or columnar, for both OLTP and OLAP workloads. It uses copy-on-write snapshotting or multi-version concurrency management techniques to store numerous copies of the data, allowing analytical queries to access the most recent transactional data. The three systems ensure that data is logically shared [19].

HTAP mechanism outlined in Algorithm 4 solves the problem of data sharing. SQLite, by default, fetches pages in page number ordering during the execution of non-indexed queries. The proposed HTAP system manages a data structure called TABLE_DS. This data structure contains the names of the database tables and the queries present in the cache. For every INSERT, UPDATE, or DELETE transaction, the table name is fetched (line 6, line 17), and all the queries associated with the table name in the TABLE_DS are retrieved. As per the OLTP command, the new/modified page is checked in the QUERY_DS against the extracted queries using the binary_search algorithm (line 8, line 19) since the page numbers are in sorted order. The absence of a modified page number leads to the page number insertion in the QUERY_DS. The advantage of this strategy is the flexible addition of new page numbers and subsequent removal of unwanted page numbers during the execution of the repetitive query after the OLTP operation.

B. QUERY_DICTIONARY MAINTENANCE COST

QM is defined to be the total space required to store the queries in the cache. The Query_Dictionary data structure is an array of query structs QUERY_DS, to support variable-sized queries. Each query struct QUERY_DS stores page numbers for higher resultset, rowids for lower resultset. A single linked list stores the actual page numbers or rowids referred by the query. Struct QELE denotes an element of the singly linked list. The size of the query struct is sizeof(QUERY_DS). The total number of pages or rowids covered by the query is denoted as qsize. Equation (1) depicts the space required to store a single query.

\[
\text{SQ}(Q) = \text{sizeof}(\text{QUERY_DS}) + \text{qsize} \times \text{sizeof}(\text{QELE})
\]  

(1)

Equation (2) presents the total cache occupied by the Query_Dictionary layout \( QM = (q_1, \ldots, q_N) \) of N queries

\[
QM = \sum_{i=1}^{N} SQ(q_i)
\]  

(2)

Let the average query size be \( \text{avg\_query\_size} = QM/N \). If the total cache size is \( X \), we can accommodate approximately \( X/\text{avg\_query\_size} \) number of queries in the cache. One cache page is additionally required to upload a relevant page.

In the case of HTAP, metadata factor \( \mu \) of a query is also an indicator to be considered. For a page \( p \) and a query \( Q \), at least one-row \( r \) in \( P \) should be relevant to \( Q \). Let \( P_r \) denote the set of pages applicable to \( Q \) (i.e., having at least one relevant row), and let \( P_m \) signify the pages considered important by the QUERY_DS related with \( Q \). The number of rows in page \( p \) (or dataset \( D \)) is denoted as \( |p| \) [20]. Equation (3) presents

Algorithm 4: Query_Dictionary Algorithm for Handling Transactional Data With Repetitive Analytical Queries

1: \( \text{procedure HTAPWorking(} \text{Query Q}\text{)} \) \( \text{switch Q do case} \) 
2: \( \text{NEW\_SELECT\_QUERY} \)
3: \( \text{case INSERT} \)
4: \( T_{\text{insert}} = \text{Tablename} \) in \text{INSERT} statement
5: \( n = \text{newly touched page number} \)
6: \( \text{Fetch all the Related Queries(RQ) of } T_{\text{insert}} \in \text{TABLE_DS} \)
7: \( \text{for each } \text{QUERY_DS}[i] \in \text{RQ} \text{ do} \)
8: \( \text{if binary_search(QUERY_DS}[i],n) \text{ then} \)
9: \( \text{break;} \)
10: \( \text{else} \)
11: \( \text{insert(QUERY_DS}[i],n) \)
12: \( \text{end if} \)
13: \( \text{case UPDATE/DELETE} \)
14: \( T_{\text{updel}} = \text{Tablename} \) in \text{UPDATE/DELETE} statement
15: \( u = \text{modified page number} \)
16: \( \text{Fetch all the Related Queries(RQ) of } T_{\text{updel}} \in \text{TABLE_DS} \)
17: \( \text{for each } \text{QUERY_DS}[i] \in \text{RQ} \text{ do} \)
18: \( \text{if binary_search(QUERY_DS}[i],u) \text{ then} \)
19: \( \text{break;} \)
20: \( \text{else} \)
21: \( \text{insert(QUERY_DS}[i],u) \)
22: \( \text{end if} \)
23: \( \text{case RepetitiveQueryAfterOLTP} \)
24: \( \text{Generate the VDBE Code} \)
25: \( \text{Execute the VDBE code with the appended list of page numbers in QUERY_DS} \)
26: \( \text{Remove the unwanted page numbers from the QUERY_DS} \)
27: \( \text{end procedure} \)
the calculation of metadata factor:

$$\mu = \frac{\sum_{p \in P_r} |p|}{\sum_{p \in P_m} |p|}$$

(3)

The metadata factor is closely related to the metadata’s false-positive ratio - a low false-positive ratio gives rise to a high metadata factor. Even after the OLTP transaction, the proposed method has more related pages giving rise to a high metadata factor. The metadata factor $\mu$ is close to 1 for most of the queries.

V. ENHANCEMENTS

While faster execution of repetitive non-indexed queries is undoubtedly the main ingredient of this paper, performance-enhancing components implemented provide an advantage over SQLite.

A. DIFFERENT ATTRIBUTES IN PROJECTION

Identifying similar queries is a difficult task [12]. For example, consider the below queries, i.e., queries Q1 and Q2, which are similar, but their projection attributes are different.

Q1: `SELECT COUNT(o_orderstatus) FROM orders WHERE o_orderstatus=’O’ GROUP BY o_custkey ORDER BY o_custkey asc;

Q2: `SELECT MAX(o_orderstatus) FROM orders WHERE o_orderstatus=’O’ GROUP BY o_custkey ORDER BY o_custkey asc;

QNew1: `SELECT * FROM orders WHERE o_orderstatus=’O’ GROUP BY o_custkey ORDER BY o_custkey asc;

QUERY_DS helps store only related pages for similar OLAP queries. When a query with distinct projection attributes appears followed by the exact substring from the FROM clause as in Q1 and Q2, the query becomes general by replacing the projection attributes with a * symbol. QNew1 represents query Q1 and is relevant for all queries like Q2. This technique also addresses queries that are not identical but have the same subqueries.

B. PARALLEL PREFETCH

Database applications take time in (a) preparing requests, (b) generating intermediate code, (c) actual calculation at the database to serve the request (including processing and main-memory access), (d) preparing responses, and (e) relaying responses back [21]. Latency is caused since the calling program blocks through stages (b) through (e). Parallel computation can reduce latency. One of the techniques is sending asynchronous requests ahead of time while the application is still running. Parallel prefetching of the relevant pages involved in later computation can minimize response time. A few parameters and metrics help characterize prefetchers [22].

Consider the number of cycles per query (CPQ) for a processor system with two caches: an infinite cache ($CPQ_{INF}$) and a finite cache ($CPQ_{FCA}$). $CPQ_{INF}$ represents the processor’s performance in the absence of misses (even compulsory misses). The limiting scenario is when the processor has an indefinitely huge cache without prefetching. So consider $CPQ_{FCA}$, which accounts for the delay due to limited cache. Here it is used to measure the effectiveness of prefetching.

$$CPQ_{FCA} = \frac{\text{misses}}{\text{cycles}}$$

(4)

Equation (4) is given a second term during prefetching. Misses are now classified as prefetches and remaining misses. As a result, some misses (raw misses) are prevented (prefetched) and moved to a prefetch term. Equation (5) presents the breakup term of $CPQ_{FCA}$

$$CPQ_{FCA} = \frac{\text{misses}}{\text{cycles}} \left( \frac{\text{cycles}}{\text{query}} \right) + \left( \frac{\text{prefetches}}{\text{query}} \right) \left( \frac{\text{cycles}}{\text{prefetch}} \right)$$

(5)

After using a prefetch method, the total number of misses is represented by the misses-per-query phrase, while the prefetches-per-query term represents the average rate of prefetches. However, in the case of prefetches, not all prefetches are used, and prefetching causes some new misses to occur. This is demonstrated by Equation (6).

$$CPQ_{FCA} = \frac{\text{misses}}{\text{cycles}} \left( \frac{\text{cycles}}{\text{query}} \right) + \left( \frac{\text{prefetches}}{\text{query}} \right) \left( \frac{\text{cycles}}{\text{prefetch}} \right) > = \frac{\text{rawmisses}}{\text{query}}$$

(6)

Prefetch requests are submitted asynchronously according to our execution model. The metadata obtained from the Query_Dictionary for the query provides information to prefetch the first few pages parallely. Simultaneous parallel prefetch of other pages happens during the query execution related to the previously fetched pages. The pipeline dynamics determine the penalty for each miss. When examining the benefits of parallel prefetching, the proposed model does not consider the impact of adding prefetch instructions to the program. Prefetching only relevant pages results in close to zero cache miss in HTAP and zero misses in OLAP. More than one cache page is required to accommodate prefetched pages.

VI. EXPERIMENTAL EVALUATION

This section contains the results of the experimental study. First, it discusses the experimental configuration used for experiments, followed by the benchmark used to test the model and the test query set. Finally, the experimental results are reported and discussed.

A. HARDWARE AND SOFTWARE SETUP

1) HARDWARE AND SOFTWARE

The proposed methodology ran the experiments on a Windows desktop with Intel ® Core™ i5-7400 CPU with 4 cores, 8GB RAM clocked at 3.00 GHz.
2) DATASETS

The experiments employed the TPC-H industry-standard benchmark for transactional and analytical processing systems. The TPC-H schema contains eight separate and individual tables: PART, PARTSUPP, LINEITEM, ORDERS, SUPPLIERS, NATION, CUSTOMER, and REGION. TPC website [23] publishes this data. DBGen, a database population program, was used with the TPC-H benchmark with a scale factor $= 1$ to generate approximately 1G size of data. Changing the scale factor alters the size of data [24].

3) QUERIES

Multidimensional predicates are used in the tests to examine the impact on performance. Query_Dictionary accepts a set of queries, both OLAP and OLTP, over a single data relation and binary joins. The queries are of the form $attr op constant$, where $attr$ is a relation's non-key attribute, $op$ is a comparison operator, and $const$ is a constant value or a subquery. Consider the two sample examples shown below:

$Q_a$: SELECT count(l_shipmode) FROM lineitem WHERE l_returnflag='A' AND l_shipmode='Truck';

$Q_b$: SELECT ps_availqty FROM lineitem l, partsupp ps WHERE l_partkey=ps_partkey AND l_suppkey=ps_suppkey AND l_returnflag='R';

$Q_a$ refers to queries that have a single table, and $Q_b$ refers to binary joins. Some queries include range predicates, IN clause, aggregates, and ORDER BY clause.

B. EXPERIMENT 1: VARYING OLAP COMPLEXITY LEVELS

The experiment checks the performance of Query_Dictionary under different types of query result loads that are Scan_20, Scan_40, binary join, and CPU intensive queries. Table 7 provides a subset of the test queries having non-indexed keys as predicate conditions used in the proposed approach. Row 1-6 refer to single table queries and deal with Scan_20, Scan_40 query type, row 7-9 refer to binary join queries, and row 10 symbolizes CPU Intensive queries taken from the TPC-H benchmark. The test platform selected a random query from the generated query set for each query load type. The selected query was executed three times with different projection attributes, IN clause, aggregates, and ORDER BY clause.

Figure 4 shows the experiment results when using query execution time as a parameter. The horizontal axis compares SQLite and the proposed Query_Dictionary with these queries. The investigations revealed that the Scan_20 query, when executed repeatedly on the Query_Dictionary, showed a significant performance gain of 78% against SQLite. The query rewriting strategy used in the proposed methodology for lower resultset helped provide the rowid directly, resulting in faster page fetch. The second category Scan_40 queries indicated a performance gain of 35% against SQLite. Compared to the SQLite that searches all the pages in a non-indexed query, the proposed optimized VDBE code only works with the page numbers associated with the search. The third category, which takes into the binary join, demonstrated a 46% improvement over SQLite - this could be attributed to the strategy of preserving only the larger table’s associated pages in Query_Dictionary, leaving the smaller table entirely accessible and optimizing the search process.
in memory. Finally, the results obtained on executing CPU Intensive queries showed an improvement of 19.88% compared to SQLite. CPU Intensive queries rely more on GROUP BY and aggregates, so their execution time in databases is longer. However, the proposed methodology shows the gain due to the limited page search.

Besides query execution time, page requests are an integral part of any database as it is latency-related. Figure 5 shows the number of page requests made by each query type on repetitive execution in SQLite and Query_Dictionary. During the first time execution of the query, the number of pages fetched by Query_Dictionary and SQLite is the same. For subsequent execution of these queries, the Query_Dictionary has significantly fewer page requests than SQLite, about 1.3% in Scan_20, 10% in Scan_40, 10% in binary join queries, and around 74% page requests for CPU Intensive queries. As CPU Intensive queries require many combinations of pages, the number of page requests is more for SQLite.

C. EXPERIMENT 2: HTAP SCHEDULING
The experiment considers three scenarios to evaluate Query_Dictionary’s ability to optimize repetitive non-indexed analytical queries while handling simultaneous transactional operations. The investigations split the queries into two parts: OLAP and OLTP transactions. The proportion of the queries for the three scenarios is 70 reads–35 writes, 70 writes-35 reads, and 49 reads–49 writes, respectively. Table 7 displays a subset of the OLAP queries; some were considered repetitive in this experiment.

Figure 6 presents the results. In the case of 70 reads–35 writes, the number of OLAP queries was double that of OLTP transactions. Compared to SQLite, the proposed technique showed a significant gain of 84%. The most prominent factor for the improvement is fewer page requests by Query_Dictionary on repetitive query execution. Next, in the case of 70 writes–35 reads, the OLTP transactions were twice as many as OLAP reads. The findings showed a 26% performance improvement against SQLite due to the less maintenance cost to update the Query_Dictionary during the OLTP transactions. Incorporating binary_search to search and add the new/modified page in the Query_Dictionary helped achieve the speedup. Finally, the last experiment made was by considering an equal number of analytical and transactional queries (i.e.) 49 reads–49 writes. Query_Dictionary showed an improvement of 30% against SQLite. In all three scenarios, the results showed that Query_Dictionary outperforms SQLite irrespective of the ratio between OLTP transactions and OLAP queries.

D. EXPERIMENT 3: PARALLEL PREFETCH
Figure 7 illustrates the speedup of Query_Dictionary with prefetch configuration against Query_Dictionary without the prefetch enabled. The experiment ran 100 analytical queries containing non-indexed attributes in the WHERE predicate to evaluate the performance. The first bar from the left in Figure 7 shows the performance achieved by SQLite on
executing all the OLAP queries. The results indicate that SQLite did not take advantage of non-indexed query iterations and parallel prefetch. The Query_Dictionary, on the other hand, showed a speedup of 72% against SQLite in the absence of prefetch. This increase is due to the skipping of unnecessary pages on the encounter of a repetitive query. When testing Query_Dictionary with prefetch, parallel prefetch shows a performance gain of 3% on Query_Dictionary without prefetch. The result shows a slight performance improvement because of the Query_Dictionary, written on SQLite code. SQLite, by default, fetches pages sequentially in case of non-indexed queries.

VII. DISCUSSION
The time taken for executing the proposed methodology is directly proportional to the nature of the resultset obtained during the first time execution of the query. The proposed approach can store more repetitive query metadata details than materialized views or normal caching as in SQLite. For a single query, the materialized view may be faster. However, the amortized cost of executing more queries in the proposed method is better than materialized views.

Apart from this, materialized views have a fixed set of attributes. Change of attributes in the query results in either retaining the existing materialized view and creating a fresh one or deleting the current view and re-creating the same with a new set of attributes. However, the proposed approach allows any number of changes in the repetitive queries’ projection attributes, thereby allowing the user to write queries flexibly and with less system overhead. Compared to data-skipping (e.g., zone maps), the proposed methodology works not only for sorted, semi-sorted data but also for the arbitrary distribution of data of any type.

For HTAP, periodic snapshots of transaction data [25], [26] say delta version are taken based on the logical timestamp framework. These snapshots are provided to existing analytical frameworks and add a lot of space and time. However, the proposed methodology adds the newly modified page’s metadata to the Query_Dictionary, thereby simplifying the HTAP process. This approach also omits the delay caused due to the lazy shipping of data from write-optimized to read-optimized data.

VIII. CONCLUSION
From the results obtained, it is clear that the proposed methodology is quite simple to use and efficient in terms of time, space, and bandwidth requirement. The Query_Dictionary created dynamically can take any combination like intersect, conjunction, aggregates, group by, binary joins, and transactional queries. The size of the Query_Dictionary is reasonably small and can contain details of hundreds of queries. Query_Dictionary reduces the time of execution of repetitive queries that would otherwise need materialized view, which would consume more space. This methodology can thus prove to be of good use in applications like mobile computing. One of the possible limitations of the proposed method could be the increase in the size of the resultset. As the size of the resultset increases, metadata information storage would increase, resulting in fewer pages in cache for actual query execution. Hence, it would be desirable to use a resultset scattered around 40-page numbers as it shows significant performance. In addition to this, to maintain the simplicity in the Query_Dictionary data structure, only binary joins have been considered. Further work may include improvisation, where the in-memory database can use Query_Dictionary to make query predictions, resulting in higher page hit rates.

REFERENCES
[1] A. Magalhaes, J. M. Monteiro, and A. Brayner, “Main memory database recovery: A survey,” ACM Comput. Surv., vol. 54, no. 2, pp. 1–36, Apr. 2021.
[2] Y. Izenov, A. Datta, F. Rusu, and J. H. Shin, “COMPASS: Online sketch-based query optimization for in-memory databases,” in Proc. Int. Conf. Manage. Data, China, Jun. 2021, pp. 804–816.
[3] A. Mathew and C. Min, “HydraList: A scalable in-memory index using asynchronous updates and partial replication,” Proc. VLDB Endowment, vol. 13, no. 9, pp. 1332–1345, May 2020.
[4] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, “Meet the Walkers: Accelerating index traversals for in-memory databases,” in Proc. 46th Annu. IEEE/ACM Int. Symp. Micronarchitecture (MICRO), Davis, CA, USA, Dec. 2013, p. 468—479.
[5] F. Färber, S. K. Cha, J. Primosch, C. Bornhövd, S. Sigg, and W. Lehner, “SAP HANA database: Data management for modern business applications,” ACM Sigmod Rec., vol. 40, pp. 45–51, Jan. 2012.
[6] U. Sirin, S. Dwarkadas, and A. Ailamaki, “Performance characterization of HTAP workloads,” in Proc. IEEE 37th Int. Conf. Data Eng. (ICDE), Chania, Greece, Apr. 2021, pp. 1829–1834.
[7] C. Luo, P. Tözün, Y. Tian, R. Barber, V. Raman, and R. Sidle, “Umzi: Unified multi-zone indexing for large-scale HTAP,” in Proc. EDBT, 2019, pp. 1–12.
[8] O. Aziz, T. Anees, and E. Mehmoond, “An efficient data access approach with queue and stack in optimized hybrid join,” IEEE Access, vol. 9, pp. 41261–41274, 2021.
[9] S. Gupta, S. Purandare, and K. Ramachandra, “Agyffy: Lifting the curse of cursor loops using custom aggregates,” in Proc. ACM SIGMOD Int. Conf. Manage. Data, Portland, OR, USA, Jun. 2020, pp. 559–573.
[10] U. Inoue, T. Satoh, H. Hayami, H. Takeda, T. Nakamura, and H. Fukuoka, “Rinda: A relational database processor with hardware specialized for searching and sorting,” IEEE Micro, vol. 11, no. 6, pp. 61–70, Dec. 1991.
SUCHARITHA SHETTY is currently pursuing the Ph.D. degree with the Manipal Academy of Higher Education, Manipal. She is currently working as an Assistant Professor with the Department of Computer Science and Engineering, MIT, Manipal Academy of Higher Education. She has published few papers in international conferences and journals. Her research interests include database and parallelism.

B. DINESH RAO received the M.Tech. degree from the Indian Institute of Technology Kanpur, Kanpur, in 1993, and the Ph.D. degree from the Manipal Academy of Higher Education, Manipal, in 2014. He is currently working as a Professor with the Manipal School of Information Sciences, Manipal Academy of Higher Education. He has around 26 years of teaching experience and has published around 11 research papers in national/international journals/conferences. His research interests include fractals and theoretical computer science.

SRIKANTH PRABHU (Senior Member, IEEE) received the M.Tech. and Ph.D. degrees from the Indian Institute of Technology Kharagpur, Kharagpur, in 2003 and 2013, respectively. He is currently working as an Associate Professor with the Department of Computer Science and Engineering, Manipal Institute of Technology, Manipal Academy of Higher Education, Manipal. He has around 15 years of teaching experience and has published around 35 research papers in national/international journals/conferences. He is also on the editorial board of the some journals. His research interests include biomedical image processing, agricultural image processing, face recognition, and wireless visual sensor networks.

* * *