Transforming Query Sequences for High-Throughput B+ Tree Processing on Many-Core Processors

Ruiqin Tian*
College of William and Mary
Williamsburg, VA 23187, USA
rtian@email.wm.edu

Junqiao Qiu*
University of California at Riverside
Riverside, CA 92521, USA
jqiu004@ucr.edu

Zhijia Zhao
University of California at Riverside
Riverside, CA 92521, USA
zhijia@cs.ucr.edu

Xu Liu
College of William and Mary
Williamsburg, VA 23187, USA
xl10@cs.wm.edu

Bin Ren
College of William and Mary
Williamsburg, VA 23187, USA
bren@cs.wm.edu

***Abstract*—The throughput of B+ tree query processing is critical to many databases, file systems, and cloud applications. Based on bulk synchronous parallel (BSP), latch-free B+ tree query processing has shown promise by processing queries in small batches and avoiding the use of locks. As the number of cores on CPUs increases, it becomes possible to process larger batches in parallel without adding any extra delays. In this work, we argue that *as the batch size increases, there will be more optimization opportunities exposed beyond parallelism, especially when the query distributions are highly skewed*. These include the opportunities of avoiding the evaluations of a large ratio of redundant or unnecessary queries.**

To rigorously exploit the new opportunities, this work introduces a query sequence analysis and transformation framework – *QTrans*. *QTrans* can systematically reason about the redundancies at a deep level and automatically remove them from the query sequence. *QTrans* has interesting resemblances with the classic data-flow analysis and transformation that have been widely used in compilers. To confirm its benefits, this work integrates *QTrans* into an existing BSP-based B+ tree query processing system, PALM tree, to automatically eliminate redundant and unnecessary queries [1]. Evaluation shows that, by transforming the query sequence, *QTrans* can substantially improve the throughput of query processing on both real-world and synthesized datasets, up to 16X.

***Keywords*—B+ tree, many-core processors, latch-free processing, query analysis and transformation**

I. Introduction

As a fundamental indexing data structure, B+ tree is widely used in many applications, ranging from database systems and parallel file systems to online analytical processing and data mining [1], [2], [3], [4], [5]. There have been significant efforts on optimizing the performance of B+ tree, with a large portion of work aiming to improve the concurrency [6], [7], [8], [9], [10], [11]. As the memory capacity of modern servers has increased dramatically, in-memory data processing becomes more popular. Without expensive disk I/O operations, the cost of accessing in-memory B+ trees becomes more critical.

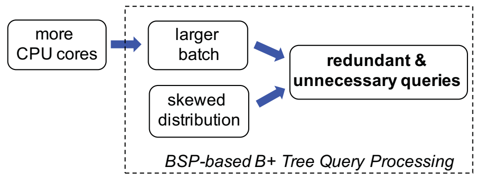

more CPU cores
larger batch
skewed distribution
redundant & unnecessary queries
BSP-based B+ Tree Query Processing

Fig. 1: New Optimization Opportunities

To reduce the tree accessing cost, prior work has proposed latch-free B+ tree query processing [7]. Traditionally, B+ tree query processing requires locks (i.e., latches) to ensure the correctness since queries may access the same tree node and if one of them modifies it (e.g., an insertion query), it would cause conflicts. Latch-free B+ tree query processing avoids the use of locks by adopting a bulk synchronous parallel (BSP) model. Basically, it processes the queries batch by batch, with each batch handled by a group of threads in parallel. By coordinating the threads working on the same batch, the use of locks can be totally avoided (see Section 2). To guarantee the quality of service (QoS), the size of a query batch should be carefully bounded to avoid long delays.

Fortunately, as modern processors become increasingly parallel, the size bound of a batch can be dramatically relaxed without incurring extra delays. For example, the latest Intel Xeon Phi processors equipped with 64 cores can process 1M queries with time cost at only milliseconds (ms) level. In this work, we argue that as the batch size grows, there will be more optimization opportunities exposed beyond parallelism, which are further compounded by the fact that many real-world queries follow highly skewed distributions. The high level idea is abstractly illustrated by Figure 1.

For example, queries to the locations where taxi drivers stop are highly biased in both the time dimension (e.g., rush

*Both authors primarily contributed to the paper.
[1]Artifact available at: https://doi.org/10.5281/zenodo.1486393

hours) and the space dimension (e.g., popular restaurants). As the query batch becomes larger, there will be growing possibilities of redundant queries (e.g., a repeated search of the same location) or unnecessary queries (e.g., a later query "cancel out" the effect of an earlier query).

To identify these "useless" queries, this work proposes a query sequence analysis and transformation framework – *QTrans*, to systematically reason about the relations among queries and exploit optimization opportunities.

QTrans has interesting resemblances with the classic data-flow analysis and transformation, but it targets *query*-level analyses and transformations. Intuitively, *QTrans* treats a query sequence as a "high-level" program, where each query resembles a statement in a regular program. By tracking the queries that "define" values, *QTrans* is able to link search queries to their corresponding defining queries. Based on the analysis, *QTrans* marks all the useful queries in the sequence and sweeps the useless ones, reducing the amount of queries to evaluate. Comparing to a traditional data-flow analysis [12], [13] that iterates over cyclic control flows, *QTrans* only needs to perform acyclic analysis for query sequences with the most basic types of queries—although the algorithm of redundancy elimination is similar regardless of this difference.

To evaluate its effectiveness, we integrate *QTrans* into an existing BSP-based B+ tree processing system, called PALM tree [7]. The integration is at two levels: *QTrans* for each individual batch (i.e., intra-batch integration), and *QTrans* across batches (i.e., inter-batch integration). To minimize the runtime overhead, we also implement the parallel version of *QTrans* and discuss the potential load imbalance issues.

Finally, our evaluation using real-world and synthesized datasets confirms the efficiency and effectiveness of *QTrans*, yielding up to 16X throughput improvement on Intel Xeon Phi processors, with scalability up to all the 64 cores.

In sum, this work makes a four-fold contribution.

- First, this work identifies a class of optimizations for B+ tree query processing, enabled by the increased hardware parallelism and the skewed query distributions.
- It proposes *QTrans*, a rigorous solution to optimizing query sequences, inspired by the conventional data-flow analysis and transformation.
- It integrates *QTrans* into an existing BSP-based B+ tree processing system and the evaluation shows significant throughput improvement.
- The idea of leveraging traditional code optimizations at the query level, in general, could open new opportunities for optimizing query processing systems.

In the following, we will first provide the background on B+ tree and the latch-free query processing (Section 2), then discuss the motivation of this work (Section 3). After that, we will present *QTrans* (Section 4), the integration of *QTrans* into PALM tree (Section 5), and the evaluation results (Section 6). Finally, we discuss the related work (Section 7) and conclude this work (Section 8).

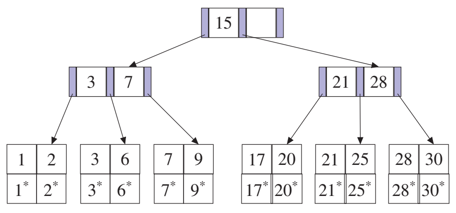


Fig. 2: A 3-order B+ tree, where key-value pairs are stored only in leaf nodes (i.e., last level).

II. Background

This section introduces B+ tree, its basic types of queries, and the high-level idea of latch-free query evaluation.

A. B+ Tree and Its Queries

A B+ tree is an N-ary index tree. It consists of internal nodes and leaf nodes. In contrast to B trees, B+ trees only maintain the keys and their associated values in their leaf nodes, and their internal nodes are merely used to hold the comparison keys and pointers for tree traversals. The maximum number of children nodes for internal nodes is specified by the *order* of B+ tree, denoted as b. The actual number of children for internal nodes should be at least $\lceil \frac{b}{2} \rceil$, but no more than b. Figure 2 shows an example of a 3-order B+ tree. Each internal node contains comparison keys and pointers to the children nodes. The leaf nodes together hold all the key-value pairs. For the 3-order B+ tree, each internal node has at least 2 children nodes, but no more than 3.

The structure of B+ tree dynamically evolves as queries to the tree are evaluated. In general, there are three basic types of B+ tree queries: (i) insertion; (ii) search; and (iii) deletion. Given a B+ tree $\mathcal{T}$, suppose function $\text{FIND}(key_i, \mathcal{T})$ can find the leaf node of key_i if it exists or return $null$ otherwise, then the semantics of queries can be described as follows.

- $I(key_i, v_j)$: if $\text{FIND}(key_i, \mathcal{T}) \neq null$, then update its value to v_j; otherwise, insert a new entry of (key_i, v_j) into $\mathcal{T}$.
- $S(key_i)$: if $\text{FIND}(key_i, \mathcal{T}) \neq null$, return the value of key_i; otherwise, return $null$.
- $D(key_i)$: if $\text{FIND}(key_i, \mathcal{T}) \neq null$, then remove the entry (key_i, v_j) from the B+ tree.

Among the three, only $S(key_i)$ returns results; $I(key_i, v_j)$ and $D(key_i)$ only update/modify the B+ tree. It is important to note that, when multiple queries arrive in a sequence, the order in which the queries are evaluated may affect both the returned results and the tree structure. In other words, there exist dependences among the queries in general.

B. Latch-Free Query Evaluation

When there are multiple threads operating on the same B+ tree, it becomes challenging to evaluate the queries efficiently. First, the workload for each thread is too little to benefit from thread-level parallelism [7]; Second, since different queries may access the same node, threads have to lock the nodes (or even subtrees) that they operate, which essentially serializes the computations, wasting hardware parallelism.

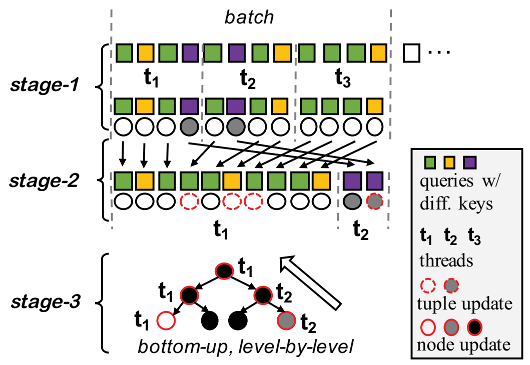


Fig. 3: Latch-Free Query Evaluation

A promising solution to the above issues is *latch-free query evaluation* [7]. Basically, it adopts the bulk synchronous parallel (BSP) model and processes queries batch by batch. Threads are coordinated to process the queries in a batch in parallel without any use of locks. Specifically, each query batch is processed in three stages [2], as illustrated in Figure 3:

Stage-1 ***Partition*** queries to threads evenly; threads then run in parallel to ***find*** the corresponding leaf nodes based on the keys in the queries;

Stage-2 ***Shuffle*** queries based on the leaf nodes such that each thread only handle queries to the same leaf node. ***Evaluate*** queries in parallel, including returning answers to search queries and updating corresponding tuples in the leaf nodes for insert and delete queries;

Stage-3 ***Modify*** tree nodes bottom up:

- ***Update*** tree nodes in parallel and ***collect*** requests for updating the parent nodes (i.e., the upper level);
- ***Shuffle*** modification requests to the parent nodes such that each thread only modifies the same node;
- ***Repeat*** update-shuffle, until the root node is reached and updated as needed.

The shuffling in stages-2 and 3 ensures contention-free operations for each thread, guaranteeing the correctness. Comparing with lock-based schemes, this latch-free scheme can significantly boost the throughput of query evaluation for B+ tree, by up to an order of magnitude [7].

III. Motivation

On top of the promises of latch-free query evaluation, we find new opportunities to further improve the efficiency of B+ tree processing, enabled by modern many-core processors and the highly skewed query distributions.

A. Growing Hardware Parallelism

As the frequency has reached a plateau, modern processors embrace growing parallelism to sustain the performance gain. For example, the latest Xeon Phi processor, Knights Landing [14], owns 64 cores/256 hyper threads. The massive hardware parallelism enables high processing capacity by allowing a larger pool of threads to run in parallel.

In the context of latch-free B+ tree query processing, the availability of more hardware threads allows the use of larger batch sizes while preserving the processing delay. However, this work argues that the benefits of using larger batches are not limited to the parallelism – as the batches become larger, there would be new opportunities exposed, especially when the queries are unevenly distributed.

B. Highly Skewed Query Distribution

In fact, the query distributions of real-world applications are often highly skewed. Take the taxi data of New York City (NYC) as an example. The geolocations where taxi drivers pick up (or drop off) passengers follow a highly skewed distribution, as shown in Figure 4-(a).

The x-axis shows the geolocations and the y-axis indicates the visiting frequencies of each geolocation for a period of one month. The top 1000 geolocations out of 4,194,304 (i.e., 0.02%) covers 68.272% of total visits. In this case, the skewed distribution is caused by the fact that some geolocations are much more likely to be visited by taxis, such as shopping malls or popular restaurants.

In fact, skewed distributions frequently appear in other query processing scenarios, such as BigTable [15], Azure [16], Memcached [17], and among others. Figures 4-(b) and (c) show the key distributions in cloud workloads modeled by Yahoo Cloud Serving Benchmark (YCSB). In these cases, the top 1% keys cover 30% and 56% requests, respectively.

C. Optimization Opportunities

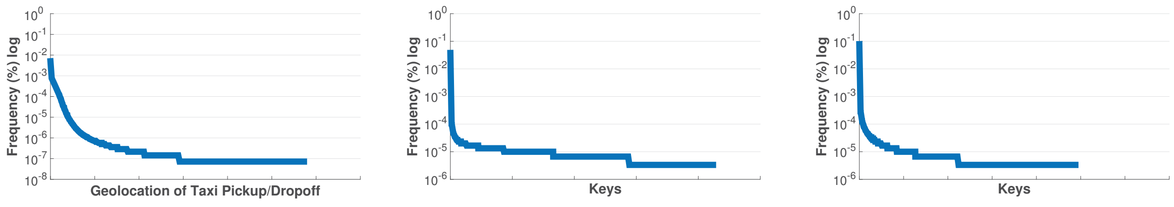
When the distribution becomes highly skewed, queries with identical key tend to appear more frequently. This trend not only results in repetitive queries (i.e., query redundancies), but also queries that might not have to be evaluated.

Next, we use an example query sequence, as shown in Figure 5, to illustrate the optimization opportunities, and informally characterize them into three categories.

- ***Query Redundancy*** ❶. One obvious opportunity is for the repeated search queries like queries 2 and 4 in Figure 5. Since query 3 does not modify key_1, query 4 should return the same value as query 2. Thus, we only need to evaluate one of them, then forward the return value to the other.
- ***Query Overwriting*** ❷. When two queries operate on the same key and both of them are either insert or delete with no search queries on the same key in between, then the second query may "overwrite" the first query. In another word, the first query becomes unnecessary, such as the overwritten queries 3 and 5 in Figure 5.
- ***Query Inference*** ❸. For a search query, by tracing back prior queries in the query sequence, one may find an earlier query carrying the information that the search query needs, thus we may infer its return value without evaluating it, such as query pairs (1, 2), (6, 9), and (7, 8).

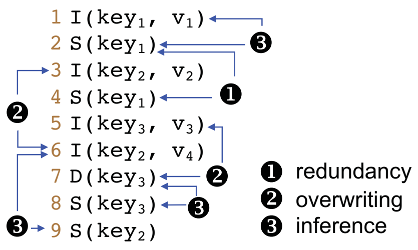
In addition, as existing opportunities are exploited, more opportunities might be uncovered. For example, an earlier

[2]For better illustration, we merged stages 3 and 4 in [7].



a NYC Taxi b YCSB (Zipf Distribution) c YCSB (Latest Distribution)

Fig. 4: Highly Skewed Query Distributions



```
1 I(key₁, v₁)
2 S(key₁)
3 I(key₂, v₂)
4 S(key₁)
5 I(key₃, v₃)
6 I(key₂, v₄)
7 D(key₃)
8 S(key₃)
9 S(key₂)
```

Fig. 5: Optimization Opportunities

removal of a search query may enable a new opportunity of query overwriting. As we will show in the evaluation, the above optimization opportunities frequently appear when dealing with both real-world and synthesized datasets.

IV. Analysis and Transformation

In this section, we present a rigorous way to systematically exploit the new opportunities mentioned above, inspired by the classic data-flow analyses and transformations.

A. *Overview*

Basically, we treat the query sequence as a "program", where each "statement" is a B+ tree query. Then the optimization of query sequence follows the typical procedure of a traditional compiler optimization: it first performs an analysis over the query sequence, based on which, it then transforms the query sequence into an optimized version – a new query sequence that is expected to be evaluated more efficiently. We refer to this new optimization scheme as *query sequence analysis and transformation* or QSAT, in short.

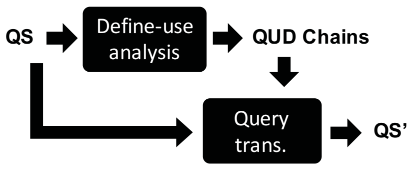


Fig. 6: Conceptual Workflow of QSAT

Figure 6 illustrated the workflow of QSAT. The original query sequence QS is first analyzed to uncover use-define relationships among queries. The output – an intermediate data structure, called QUD chains is then used to guide the query sequence transformation, which yields an optimized query sequence QS'. Next, we present the ideas of QSAT.

B. *Query Sequence Analysis*

The goal of query sequence analysis is to uncover the basic define-use relations among the queries, which will be used to facilitate the later transformation. This resembles the classic *reaching-definition analysis* used in compilers [12], [13]. Basically, it examines the queries in the sequence and finds out which queries "define" the "states" of B+ tree and which queries "use" the "states" correspondingly.

Based on the semantics defined in Section 2.1, the queries that define the state are *insert* and *delete* queries, and the queries that use the state are *search* queries. The define-use analysis matches each *search* query with its corresponding defining query (either an *insert* or a *delete*) based on the keys that the queries carry.

Example. Figure 7-(a) shows the define-use analysis on the running example, where q_i corresponds to the query at line i. Basically, the set e consists of the defining queries that can reach each query. For example, the defining queries q_1, q_6 and q_5 can reach query q_7.

QUD Chain. To represent the results of define-use analysis, we construct a data structure – *query-level use-define chain* (QUD chain). This data structure resembles the UD chain constructed internally by some compilers.

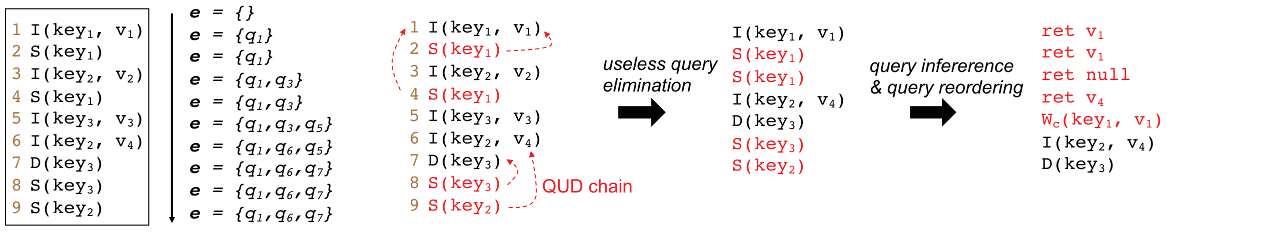
The construction of QUD chains is as follows. Basically, when a use query is met, the construction adds a link from the use query to its corresponding defining query (i.e., the defining query with the same key) if the later exists in current defining query set e. An example of constructed QUD chains is shown in Figures 7-(b).

QUD chains capture the dependence relations among the queries in a query sequence. For the query semantics defined in Section 2.1, the size of a QUD chain is limited to two queries. However, in general, the length of a QUD chain can go beyond two. QUD chains provide critical information for performing query sequence transformation, as shown next.

C. *Query Sequence Transformation*

The purpose of query sequence transformation is to generate an optimized version of query sequence. For clarity, we next describe the transformation with two passes. However, they can be integrated into one pass, as we will show later.

Round-1: Useless Query Elimination. This round is to eliminate queries that do not contribute to the final results of query processing – the returned values of search queries and



```
1 I(key1, v1)        e = {}
2 S(key1)            e = {q1}
3 I(key2, v2)        e = {q1}
4 S(key1)            e = {q1,q3}
5 I(key3, v3)        e = {q1,q3}
6 I(key2, v4)        e = {q1,q3,q5}
7 D(key3)            e = {q1,q6,q5}
8 S(key3)            e = {q1,q6,q7}
9 S(key2)            e = {q1,q6,q7}
                     e = {q1,q6,q7}
```

```
1 I(key1, v1)
2 S(key1)
3 I(key2, v2)
4 S(key1)
5 I(key3, v3)
6 I(key2, v4)
7 D(key3)
8 S(key3)
9 S(key2)
```

QUD chain
useless query elimination
query infererence & query reordering

```
I(key1, v1)
S(key1)
S(key1)
I(key2, v4)
D(key3)
S(key3)
S(key2)
```

```
ret v1
ret v1
ret null
ret v4
Wc(key1, v1)
I(key2, v4)
D(key3)
```

(a) Forward define-use analysis (b) Build QUD Chain (9 queries) (c) Round-1 Trans. (7 queries left) (d) Round-2 Trans. (2 queries left)

Fig. 7: Example of Query Sequence Analysis and Transformation (QSAT)

Algorithm 1 Useless Query Elimination (Mark-Sweep)

```
1: $I = \{\}$          ▷ a list of useful queries
2: QUD()          ▷ QUD($q$) returns the defining query of query $q$
3: for $q_i$ in $\{q_1 \cdots q_n\}$ do
4:     if $q_i$ is a search query then
5:         $I$.add($q_i$)          ▷ mark a search query as "useful"
6:         if QUD($q_i$) $\neq \emptyset$ then
7:             $I$.add( QUD($q_i$) )          ▷ mark defining query "useful"
8: return $I$
```

the key-value pairs stored in the B+ tree. This can be achieved with a *mark-sweep* strategy that has been previously used for garbage collection and dead code elimination.

Algorithm 1 describes the useless query elimination. It first marks all the search queries as useful queries, as they need to return values. Then it traces back the QUD chains to find the corresponding defining queries, and mark them as useful queries as well. Note that the algorithm is customized to QUD chains of length 2, but it can be easily extended to handle QUD chains with arbitrary length.

Example. Figure 7-(c) lists the results after useless query elimination. The number of queries drops from 9 to 7. This round explores query overwriting (see Section 3.3).

Round-2: Query Inference & Reordering. Besides query overwriting, there are two other optimization opportunities: redundant queries and query inference (see Section 3.3). The second round is to explore the latter two.

Basically, for each search query, find its corresponding defining query (if exists), then retrieve the return value and return it. After this optimization, all the search queries with corresponding defining queries (i.e., QUD(q_i) $\neq \emptyset$) will be eliminated, as Figure 7-(d) shown (denoted as `ret` v_i).

Note that, after the optimization, no return operations `ret` v_i depend on any other queries, hence they can be reordered – being moved to the top of the sequence. In this way, the latency of the search queries could be reduced.

An orthogonal optimization is a top-K cache. When the B+ tree is large, performance can be benefited from putting hot key-value pairs (top K pairs) into a small cache. Thus, when an insert query with a top-K key-value pair is left after round 1, we can transform the query into a cache write operation (e.g., $W_c\,(key_i, v_j)$ in Figure 7-(d)).

Finally, after the two rounds of optimizations, there are only 2 queries left that need to be actually evaluated.

Algorithm 2 One-Pass QSAT (for queries of the same key)

```
1: $q_o = null$          ▷ the last defining query
2: $n_s = 0$          ▷ number of search queries
3: $I = \{\}$          ▷ a list of useful queries & operations
4: for $q_i$ in $\{q_n \cdots q_1\}$ do
5:     if $q_i$ is a search query then
6:         $n_s{+}{+}$
7:     else if $q_i$ is an insert or delete query then
8:         if $n_s > 0$ then
9:             $I$.add(INFER_AND_RETURN($q_i$, $n_s$))
10:            $n_s = 0$
11:        if $q_o = null$ then
12:            $q_o = q_i$
13: if $n_s > 0$ then
14:     ▷ no defining query for the last $n_s$ queries
15:     $I$.add(SEARCH_AND_RETURN($q_n$, $n_s$))
16: if $q_o = null$ then
17:     $I$.add($q_o$)
18: return $I$
```

D. Discussion

Comparison with Classic Data-flow Analysis. Despite the similarities between our define-use analysis and the traditional reaching-definition analysis, there are a couple of critical differences. First, the two analyses work at different granularities. The traditional data-flow analysis performs at the instruction level, while ours is applied at the query level. Each query itself may be implemented by a series of low-level instructions. Second, the traditional data-flow analysis operates on the control-flow graph, which may consist of cycles and take several iterations to converge. By contrast, our analysis works on a sequence of queries which imposes no "backward" control flows.

Potential Extension. Note that the ideas of query sequence analysis and transformation are not limited to the basic query semantics. It may benefit other batch-based query processing systems that may involve more complicated query structures as well. Consider a more advance query $I(key_1, S(key_2))$. The query is to insert/update key_1 with the value drawn from key_2. In this case, the length of a QUD goes beyond 2.

E. Implementation

We implemented the above analysis and transformation into a framework, called *QTrans*. For better efficiency, we combine the analysis and transformation into a single pass.

Pre-Sorting. In existing latch-free B+ tree processing [7], the query sequence can be pre-sorted by keys for improved tree search efficiency during stage-1 (see Section 2.2). In this work, we assume this optimization is enabled and leverage the pre-sorting to design an efficient one-pass QSAT.

One-Pass QSAT. Algorithm 2 illustrates the basic idea of one-pass QSAT for a sequence of queries of the same key (after pre-sorted), denoted as $\{q_1 \cdots q_n\}$. Basically, it traverses the query sequence *backwards* from q_n to q_1. If the current query q_i is a search (use) query, it increments the search query counter n_s; If q_i is an insert or delete (defining) query, it first checks the search query counter n_s, and if $n_s > 0$, it adds a INFER_AND_RETURN(q_i, n_s) into the output list I, which extracts the value carried by q_i and emits n_s returns of that value. After that, the algorithm resets n_s. In addition, the algorithm also finds the latest defining query q_o (overwriting all previous defining queries) and adds it to the output list. Note that, a defining query for the last a few search queries may not appear in the sequence (recognized by $n_s > 0$). In these cases, the algorithm adds a SEARCH_AND_RETURN(q_n, n_s) into the output list, which first evaluates q_n by searching its value in the tree, then emits n_s returns of that value. Although one-pass QSAT is designed for acyclic sequences of queries, the processing is similar regardless whether the use-define relations are cyclic or not.

It is not difficult to validate that the one-pass QSAT described above generates the same query and operation sequence as the QSAT introduced in Sections 4.2 and 4.3, but can be performed more efficiently.

Alternative Solution. Note that one-pass QSAT is based on query semantics; it does not evaluate any query (i.e., calling a query processing routine). An alternative solution is simulating the query evaluations, performed on a different data structure (instead of the actual B+ tree). The simulation can also eliminate redundant and unnecessary queries. However, in this case, all the queries still need to be evaluated with the "simulated" query processing routines. In comparison, QSAT does not rely on any implementations of query processing routines. Also, they are able to skip irrelevant queries as needed (e.g., when line 8 in Algorithm 2 fails).

V. Integration

To evaluate the proposed analysis and transformation, we integrate *QTrans* into an existing latch-free B+ tree query processing system – PALM tree [7] (see Section 2.2). To maximize the benefits, this section also describes a parallel implementation of *QTrans* and optimizations across batches.

A. Parallel Intra-Batch Integration

The *QTrans* described in Section 4 applies optimizations sequentially over the sequence of queries. However, in the actual setting of latch-free B+ tree query processing, queries in a batch are processed in parallel for maximum performance on parallel processors. To match with the intra-batch parallel query processing scheme, we next present a parallel design of *QTrans*.

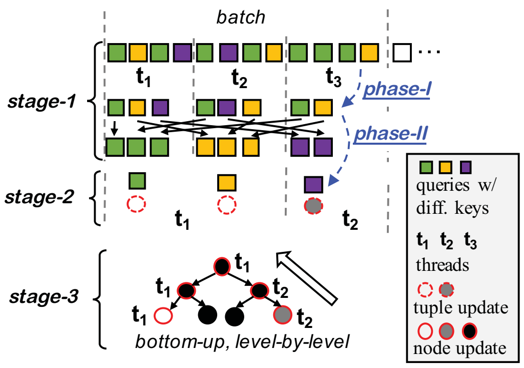


Fig. 8: Latch-Free Query Evaluation w/ *QTrans*

Given a batch of queries, the parallel *QTrans* creates a pool of threads based on the number of available cores N (part of latch-free query evaluation), then performs the query optimizations in two phases:

- **Phase-I**: First, partition the query batch evenly into N *mini-batches*. Then performs sequential QSAT over different mini-batches in parallel.
- **Phase-II**: Shuffle the queries generated by Phase-I based on the keys. Then let each thread perform a sequential QSAT over queries of the same key.

Figure 8 shows the new latch-free query evaluation with the two phases integrated. After Phase-II, there will be at most one (defining) query left for each key. After applying the parallel *QTrans*, the following steps would be the same as the original latch-free query evaluation (see Figure 3).

Advantages. Comparing with the original latch-free query evaluation (Figure 3), the new design (Figure 8) shows several advantages:

- ***Faster sorting***. In the original design, query sorting is at the batch level. While in the new design, query sorting is only performed at the mini-batch level [3].
- ***Reduced leaf searches***. The original design searches for leaf nodes for every query in the batch; In comparison, the new design only searches for leaf nodes for each distinct key in the batch.
- ***Reduced shuffle overhead***. Both the original and the new designs require to shuffle the queries in Stage 2 and Phase-II. However, in the new design, the shuffle overhead is lower, due to the query reduction in Phase-I.

Load Balancing. Despite the above advantages, intra-batch optimization may suffer from workload imbalance at Phase II. After Phase I, the number of remaining queries of different keys might be different. Further, after the query shuffling of Phase II, the number of keys mapped to different leaf nodes might also varies due to the skewed key distribution. Both cases can cause load imbalance among worker threads. Note that the second case also occurs in the original design of query processing. Here, we address them with a lightweight workload balancing strategy.

[3]For generality, query sorting is not shown in Figures 3 and 8.

Basically, our load balancing method leverages the prefix sum algorithm to calculate the starting query index for each thread, so that the number of queries assigned to each thread could be similar, but not necessarily the same. Because the assignment should not assign queries with different keys to different threads, which violates the correctness of BSP.

B. Inter-Batch Optimization

Beside intra-batch optimizations, this work also explores optimization opportunities across batches. However, it is challenging to implement inter-batch QSAT, because the intermediate results of query analysis will grow as more batches are analyzed. For example, a search query's corresponding defining query may appear in a much earlier batch. Keeping tracking all the information will overburden the QSAT, outweighing the benefits.

Instead, we adopt a more scalable strategy that is similar to the alternative solution mentioned in Section 4.4. Basically, it "simulates" the query evaluation at the inter-batch level on a different data structure. In this way, we only need to carry the "state" of key-value pairs from one batch to the next. The key is that the simulation must be faster than the actual query evaluation to bring in potential benefits. We achieve this with a *top-K cache*.

Top-K Cache. This is a small software cache with fixed number of entries – K entries. This design minimizes the costs of read/write operations. The cache can be implemented with a hash table, where the key-value pairs perfectly match with the B+ tree key-value pairs. As the number of entries is fixed, the hash function can be designed in an efficient way so that hashing conflicts can be minimized or even avoided. The entries in the top-K cache can be pre-populated with training data and periodically updated with testing data using various cache replacement policies (e.g., LRU).

To integrate the inter-batch optimization in the query evaluation system, we place the top-K cache operations in Stage 1 right after Phase II (see Figure 8). At this moment, the redundant and useless queries within the batch have been eliminated, hence the cache operations will be reduced to a minimum – only proportional to the number of distinct keys in the batch.

VI. Evaluation

This section evaluates the efficiency and effectiveness of *QTrans* for optimizing the latch-free B+ query processing.

A. Methodology

We use an open-source implementation of latch-free B+ query processing system [4] as the baseline, which follows the design of PALM tree querying system [7]. It supports SIMD operations for key search within a tree node. *QTrans* is implemented in C++ language with the use of Pthread for multicore programming and is then integrated into PALM tree, serving as the optimized querying system.

Platform. We evaluate B+ tree query processing on the latest version of Xeon Phi, Knights Landing. Our Xeon Phi is a 64-core 7210 processor, used as a CPU, running at 1.3 GHz with 1M L2 cache shared between two cores, supporting 512-bit AVX512 instructions.

Datasets. To evaluate our query sequence optimization, we build B+ trees based on the unique keys from four synthetic datasets (with configurations the same as those in [7]) and two realistic datasets:

- gaussian: the keys of queries follow the classic Gaussian distribution with parameters $\mu = N * 0.5$, $\delta = \mu * 0.5\%$, where N is the range of queries;
- self-similar: the keys follow 80-20 rule, which means 80% queries cover 20% range of queries [18];
- zipf: the keys follow Zipfian [18] with θ = 1.0;
- uniform: the keys are uniformly distributed;
- ycsb: Yahoo! Cloud Service Benchmark (YCSB) [19] that is used to evaluate the performance of cloud systems. It includes Zipfian (ycsb-zipf) and latest (ycsb-latest); Note that zipf and ycsb-zipf are different in terms of the parameter settings.
- taxi: NYC taxi data published by New York City Taxi & Limousine Commission, containing the yellow and green trips in New York City at different time[5].

All key distributions except uniform are skewed. The size of our input queries, the configuration of trees, and the input query distributions are summarized in Table I.

TABLE I: Dataset configurations

Dataset	#queries	#uniq-key	parameters
Gaussian	100M	50M	$\mu = N * 0.5$, $\delta = \mu * 0.5\%$
Self-similar	100M	50M	80-20 rule
Zipfian	100M	50M	$\theta = 1$
Uniform	100M	50M	/
YCSB-latest	30M	10M	/
YCSB-zipfian	30M	10M	$\theta = 0.99$
Taxi	13.9M	4.1M	/

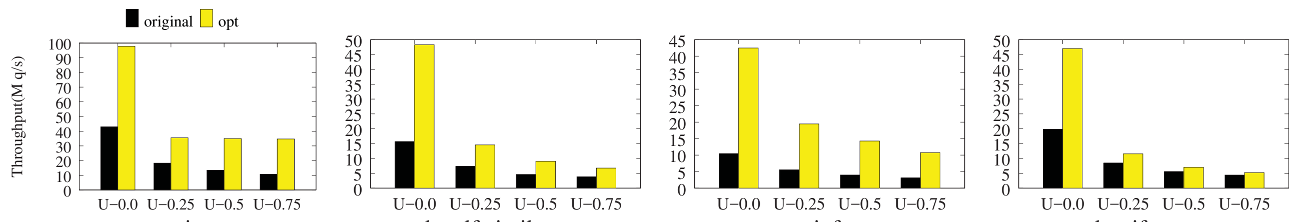
B. Performance and Scalability

Synthesized Data. Figure 9 compares the original B+ tree processing with the one optimized with *QTrans* on four synthetic datasets (i.e., gaussian, self-similar, zipf and uniform) in terms of throughputs. For each distribution, the update ratio (i.e., the ratio of insert and deletion queries) changes from 0% to 75%. For all distributions, the one with *QTrans* (i.e., opt) shows better throughput, with up to $4.05X$ improvement (occurs on zipf dataset).

Specifically, for all datasets, the throughput improvement is higher when the update ratio is lower. Even for the uniform dataset, the throughput improvement reaches $2.37X$. This is because *QTrans* handles all Find queries in stage 1, thus avoiding the time consuming stage 2 in the original design. When the update ratio is greater than 0%, for the skewed datasets, such as gaussain, zipf, and self-similar, the

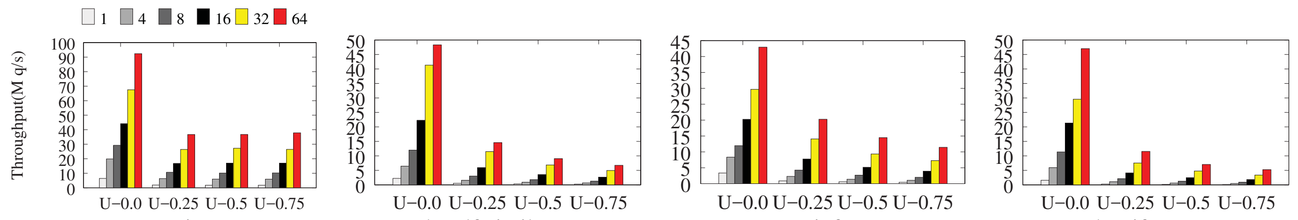
[4]https://github.com/runshenzhu/palmtree

[5]http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml



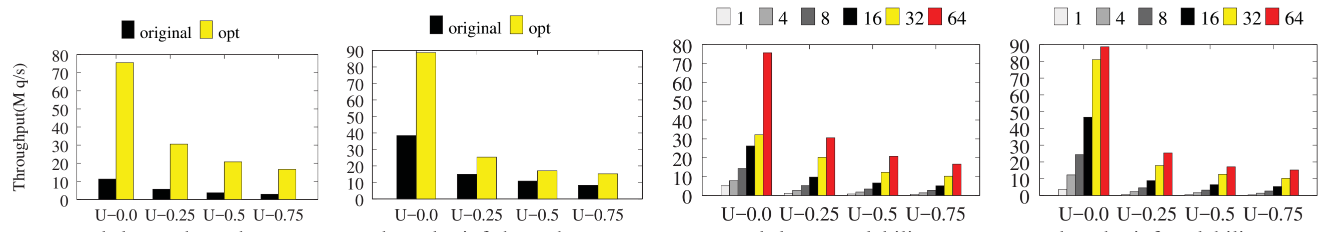
a gaussian b self-similar c zipf d uniform

Fig. 9: Overall throughput improvement. x-axis: update ratios; y-axis: throughput of queries.



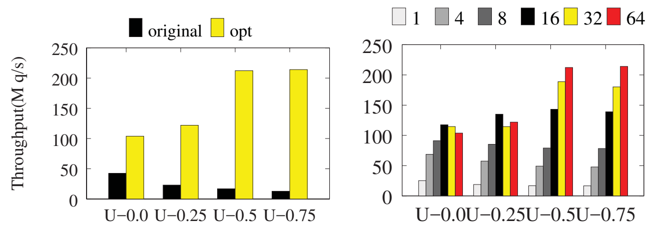
a gaussian b self-similar c zipf d uniform

Fig. 10: Throughput scalability. x-axis: update ratios; y-axis: throughput of queries.



a ycsb-latest throughput b ycsb-zipf throughput c ycsb-latest scalability d ycsb-zipf scalability

Fig. 11: YCSB overall throughput and scalability. x-axis: update ratios; y-axis: throughput of queries.



a taxi throughput b taxi scalability

Fig. 12: Taxi throughput and scalability.

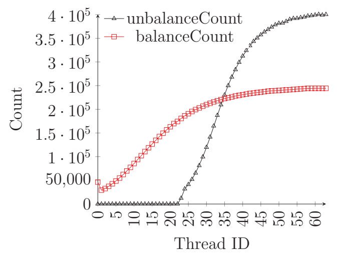


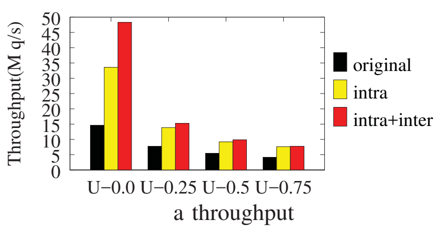
Fig. 13: `self-similar` (U-0.25) leaf operations

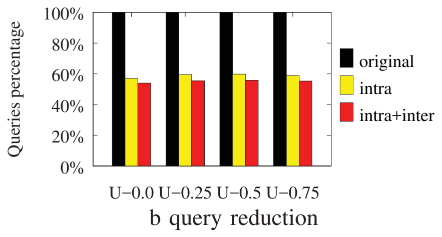
throughput improvement is more significant, from $1.76X$ to $3.59X$. This is because they have higher chances to include queries with identical keys. Interestingly, even for uniform, *QTrans* shows slightly improvement (but much less than other skewed cases) when there are updates, owing to the query transformations.

More specifically, *QTrans* monitors the query types. If no defining queries are found, it will evenly partition the input queries and get rid of the time-consuming workload redistribution. In contrast, such redistribution is always required by the original implementation. Similarly, if the update ratio is low, it only redistributes the update-related queries, leading to better performance.

Realistic Data. Next, we confirm the results with real-world datasets ycsb-latest and ycsb-zipf (Figure 11 (a) and (b)), and NYC Taxi dataset taxi (Figure 12 (a)). Among the three datasets, *QTrans* optimized version achieves higher improvements on ycsb-latest with a nearly $6.71X$ improvement (U-0), and taxi with a nearly $16.60X$ improvement (U-0.75). In comparison, on ycsb-zipf, the *QTrans* optimized version only achieves $2.31X$ improvement. Note that the throughput improvements are different between ycsb-zipf and zipf, due to the parameter setting differences. The former is based on the real-world cloud system characterization; while the latter is chosen from prior work for a direct comparison [7].

Scalability. Figures 10, 11 (c)-(d), and 12 (b) report how the throughput of *QTrans* optimized version changes with the number of threads increasing from 1 to 64. Most cases show strong scalability up to 64 threads. Only taxi scales up to 32 threads. This is because taxi has fewer unique keys than other





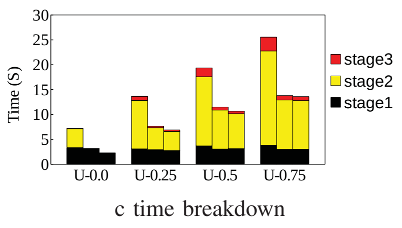


Fig. 14: `self-similar` throughput analysis, three bars in (c) correspond to bars in (a) and (b)

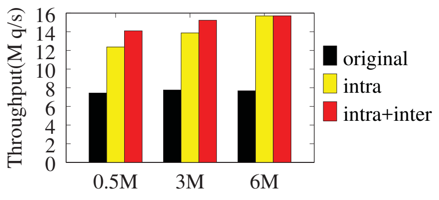


Fig. 15: `self-similar` (U-0.25) throughput

datasets, thus our optimization results in more query reduction. The remaining queries are too few to feed 64 threads. This is proved by its lowest latency in Table II.

C. Performance Breakdown

To better understand the performance improvements, we perform a case study on the self-similar dataset.

Figure 14 compares the intra-batch and inter-batch optimized versions with the original version on throughput, query reduction ratio, and execution time of different stages, with the update ratio ranging from 0% to 75%.

Intra-batch Optimization Benefits. Comparing the original with the one enabled intra-batch optimization (Figure 14a), there is a clear throughput improvement.

The improvement is due to two main reasons. First, intra-batch optimization reduces the number of queries to process, which is reflected by the query reduction ratio, as shown in Figure 14b. However, as mentioned earlier, the query reduction may cause the workload imbalance in the later stage for leaf node searching, which can in turn compromises the reduction benefits to a certain degree. To alleviate this, *QTrans* performs a lightweight load balancing with parallel prefix sum (see Section 5.1). This is the second contributor to the throughput improvement.

In addition, we perform a study on the distribution of workload (counts of operations) on the leaf nodes when all 64 threads are employed, as shown in Figure 13. The counts are for a whole query sequence. The results demonstrate the efficiency of load balance optimization. However, even with our optimization, it is impossible to achieve a perfect load balance. Because there exists data dependencies among update queries that perform on the same tree node, these modifications will always be processed by the same thread. Since the input query is skewed, the number of queries handled by different threads is also skewed.

Inter-batch Optimization Benefits. The last bar in each sub-figure of Figure 14 shows the throughput gain, the task reduction ratio, and the execution time when the inter-batch optimization is applied. In general, the improvement varies because, in some cases, the optimization opportunities have already been explored by the earlier intra-batch optimization, especially for relatively larger batch sizes.

Batch Size Impact. We set batch_size as 0.5M, 3M, and 6M with update ratios of 25% for self-similar distribution, and test the throughput differences under different kinds of optimizations. As shown in Figure 15, the throughput increases as the growing of the batch size, specifically, the benefit from intra-batch redundancy elimination.

Considering the batch size (in Table II) and the absolute throughput after our optimizations together, we observe a strong correlation – a larger batch size leads to a better absolute throughput. In the offline processing case without the latency requirement, we can always select a large batch size to achieve a better throughput. Our work considers a more challenging online processing, and these batch sizes are chosen for a more acceptable latency requirement.

TABLE II: LATENCY FOR EACH DATASET

Dataset	Batch-size	Opt Lat(ms)		Org Lat(ms)	
		U-0.0	U-0.75	U-0.0	U-0.75
Gaussian	5242880	52.26	133.62	122.18	492.09
Self-similar	3145728	65.12	404.88	200.49	818.54
Zipfian	3145728	62.31	253.84	300.37	1011.5
Uniform	2097152	36.50	391.37	105.89	475.99
YCSB-latest	1500000	18.23	112.75	133.16	519.74
YCSB-zipfian	1500000	14.95	99.96	39.02	182.90
Taxi	2081427	14.66	17.65	49.12	161.38

D. Latency

Table II reports the latencies for two scenarios: search-only and 75% update, with the corresponding batch size. For comparison, we also report the original PALM tree's latency with identical batch sizes. Even for the largest batch case, we still maintain our search-only latency lower than 50ms and our update latency lower than 400ms. For the three real-world cases, our search-only latencies range from 14.66ms to 18.23ms, and update latencies range from 17.65ms to 112.75ms. This is smaller than 0.5-1s latency maintained in previous buffering method [20]. In addition, we can always trade our high throughput for faster response time by using a smaller batch size, if it is desired.

VII. Related Work

This section focuses on research related to B+ tree processing, bulk synchronous parallel model, as well as the redundancy elimination in traditional compilers.

B+ Trees and Its Optimizations. As a basic data structure, B+ tree has received significant attentions, especially on improving the concurrency by reducing tree contention.

Prior work can be roughly categorized into three groups. The first group is for improving the lock performance and designing lock-free trees, in asynchronous processing. For instance, Rodeh designed optimized lock-based B+ trees [6], and later, Braginsky and Petrank proposed a lock-free B+ tree to further improve the performance for high contention cases [9]. More recently, new lock-free tree structures are proposed to address the performance challenges brought by contentions, such as [8], [10], [11]. The second group is about leveraging Bulk Synchronous Parallel (BSP) model. PALM tree proposed by Sewall et al. [7] is a representative solution. The third group exploits hardware support like Hardware Transactional Memory (HTM), including the red-black tree implemented by Dice et al. [21] and the Eunomia proposed by Wang et al. [22]. Based on these techniques, it is also effective to apply lazy tree restructuring [23], [24] to further reduce the contention.

The above methods focus either on improving the locking or lock-free policy or on changing the tree structure. By contrast, this work focuses on exploiting the skewed query distribution, the semantic relations among queries, and the high concurrency provided by modern many-core processors, so it is complementary to all of these existing approaches.

In addition, there are techniques to map B+ trees or other similar index trees on many-core processors or other new architectures. For instance, Fix et al. [25] implemented B+ tree on GPU, while Daga et al. [26]'s implementation is for APU. Kim et al. [27] designed and implemented a fast architecture-sensitive search tree on both CPUs with SIMD units and GPUs. A more recent design of B+ tree for heterogeneous platform is given by Shahvarani and Jacobsen [28]. There are also many efforts on improving the cache performance for in-memory trees, such as Cache-sensitive search (CSS) trees [29] and cache-sensitive B+ trees (CSB+-trees) [30].

Bulk Synchronous Parallel (BSP): The BSP model [31] used in latch-free B+ tree query processing has also been commonly used for many other applications. For example, Pregel [32] and Giraph [33], a well-known graph processing model is based on BSP. Many other graph processing engines or libraries are also directly built based on BSP, such as GraphX [34] on distributed cluster and Gunrock [35] on GPU. Moreover, BSP model also serves as a design foundation for many successful programming models in big-data and high-performance computing fields, such as MapReduce [36], Spark [37], and Apache Hama [38].

Redundancy Elimination The key idea of this work is to *eliminate redundant and unnecessary queries by transforming the query sequence*. At high level, it shares the objectives with some traditional compiler optimizations, such as *partial redundancy elimination* (PRE) and *memoization*, which are also designed to eliminate unnecessary code in the programs.

Consider the control-flow graph (CFG) of a function in a program. If a computational statement is evaluated again along a certain path, without any of its operands changed in between, the later evaluation would be (partially) redundant and thus will be removed by PRE. Over the past 30 years, many PRE algorithms [39], [40], [41], [42] have been designed to optimize program performance. Another traditional compiler optimization for redundancy elimination is *memorization* [43], [44], [45], [46], which is heavily used for functional programming languages. The basic idea is to cache the results of frequent yet expensive function calls and returning the corresponding cached result when calls with the same inputs appear again.

The above techniques for code optimizations inspire the design of our query sequence analysis and transformation. In addition, redundancy elimination has also been used to improve the space utilization in storage systems [47], [48], [49] and the integration of relational database schema [50].

Finally, there are some compiler optimization techniques being used to optimize SQL queries [51], [52], [53], where the SQL queries are first transformed into imperative programs, then optimized by conventional compiler techniques. By contrast, our techniques in this paper directly transform the query sequences without any query-to-code transformations.

VIII. Conclusion

This work targets the critical throughput problem of B+ tree query processing. It, for the first time, points out the new optimization opportunities raised by the growing hardware parallelism and the highly skewed query distributions in real-world B+ tree applications. More specifically, this work identifies three categories of optimization opportunities in the B+ tree query evaluation. To systematically exploit these opportunities, it introduces a novel query sequence analysis and transformation (QSAT) framework, inspired by the conventional code optimizations in compilers. For practical use, this work designs a one-pass QSAT, namely *QTrans*, and integrates it into a latch-free B+ tree query processing system, with parallelization and load balancing supports. Finally, our evaluation confirms the efficiency and effectiveness of *QTrans* on both synthetic and real-world datasets with up to 16X throughput improvement.

Artifact Appendix

A. Abstract

This artifact contains the source code of optimized PALM tree integrated with *Qtrans* proposed in our paper. There are in total 4 synthetic datasets (gaussian, self-similar, zipfian and uniform) and 2 realistic datasets (YCSB and Taxi) as mentioned in our paper. The 4 synthetic datasets are generated online and the 2 realistic datasets are included in this artifact. In addition, this artifact also includes the bash scripts to install 3rd-party libraries, to compile the source code, and to generate the results.

As our SIMD implementation requires AVX-512 support, the artifact needs to run on Intel Xeon Phi processor (Knights Landing) with Intel C++ compiler (icc) and Pthreads support. Libraries junction, boost, glog, jemalloc are needed in order to successfully compile and run the source code in this artifact. Moreover, all source code is tested in the environment of Linux CentOS 7.

B. *Artifact Check-List (Meta-Information)*

- **Compilation:** Intel C++ compiler icc.
- **Binary:** The source code of optimized PALM tree integrated with *Qtrans* and the scripts are included to generate binaries.
- **Data set:** There are 4 synthetic datasets and 2 realistic datasets. The four synthetic datasets are gaussian, self-similar, zipfian and uniform. The 2 realistic datasets are YCSB and Taxi. The 4 synthetic datasets are generated online and the 2 realistic datasets are included in this artifact.
- **Run-time environment:** The artifact has been developed and tested on on Linux (CentOS 7) environment. The source code is compiled by Intel C++ compiler icc with Pthreads support. As to extracting the output, Python 2.7 is needed.
- **Hardware:** The artifact is supposed to run on Intel Xeon Phi Processor (Knights Landing/KNL).
- **Execution:** Bash scripts are included for execution.
- **Output:** Results include throughput, count of the queries, queries percentage, break down time and latency.
- **How much time is needed to complete experiments?:** It approximately takes about 25 hours.
- **Publicly available?:** Yes.

C. *Description*

1) *How Delivered*

The source code is available as a public repository on Zenodo (https://zenodo.org/record/1486393) with DOI: 10.5281/zenodo.1486393.

2) *Hardware Dependencies*

The artifact has been developed and tested on the KNL machine.

3) *Software Dependencies*

In order to use Intel C++ compiler, the following script needs to be included in the ~/.bashrc file and reload the .bashrc file with the following command.

```
$source /opt/intel/
    ↪ compilers_and_libraries/linux/bin
    ↪ /compilervars.sh intel64
$source ~/.bashrc
```

The specific path might need to be modified according to the installation path of the icc compiler.

In order to compile and run the source code in this artifact, the following libraries are needed:

- Junction (https://github.com/preshing/junction);
- Boost (https://www.boost.org/);
- Boost Sort Parallel (https://github.com/fjtapia/sort_parallel);
- Glog (https://github.com/google/glog);
- Jemalloc (https://github.com/jemalloc/jemalloc);
- Stx-btree (https://github.com/bingmann/stx-btree).

The bash script to install the above libraries is already included.

4) *Data Sets*

The 4 synthetic data sets are generated at runtime. The 2 realistic data sets are included in this artifact. The root directory of our repository is cgo19, the realistic data sets are included in the directory cgo19/dataset.

D. *Installation*

The bash script cgo19/run.sh is used for installing the 3rd-party libraries as mentioned in section A and generating the binaries palmtree_test and palmtree_test_detail. To run the script, type the following commands:

```
$ chmod +x run.sh
$ ./run.sh
```

The script takes about 15 mins to finish. After that, the generated binaries palmtree_test and palmtree_test_detail can be found in directory cgo19/.

E. *Experiment Workflow*

With palmtree_test and palmtree_test_detail, please navigate to directory cgo19/figure_data to generate the raw data used in the evaluation section of our paper, through the following commands:

```
$ chmod +x get_figure_data.sh
$ ./get_figure_data.sh
```

The raw data generation takes about 24 hours.

F. *Evaluation and Expected Result*

After running the bash script to generate results, the results of each figure will be saved in a corresponding subfolder. Generally, the name of a subfolder corresponds to a figure in our paper. For example, the result of figure9a in the paper is saved in subfolder figure9a. More specifically, under each subfolder, the result is saved in file format_data.txt.

There are two special cases: figure14 and figure4. For figure14, figure14a's data is saved in format_data_figure14a.txt, figure14b's data is saved in format_data_figure14b.txt and figure14c's data is saved in format_data_figure14c.txt in directory figure14. For figure4, the corresponding script is executed with output messages redirected to the file Message.txt. The generated data is stored in the file with suffix .data, e.g., the results shown in figure4 come from the corresponding file Sort*_Distribution.data, and Message.txt contains some details of figure4.

G. Experiment Customization

Optionally, for each subfolder in the directory cgo19/figure_data/, running the python script format_data.py can also generate the data for the corresponding figures. But take a notice, there are dependencies between the results. Figure10a depends on figure9a, figure10b depends on figure9b, figure10c depends on figure9c, figure10d depends on figure9d, figure11c depends on figure11a, figure11d depends on figure11b, figure12b depends on figure12a, and figure13 depends on figure9a, figure9b, figure9c, figure9d, figure11a, figure11b and figure12a. The dependencies among the results limit the order of generating the results. For example, figure10a depends on figure9a, which means one must finish generating data for figure9a before generating data for figure10a.

Acknowledgment

The authors would like to thank the anonymous paper and artifact evaluation reviewers for their valuable and thorough comments. The authors are especially grateful to the shepherd Chen Ding for his extensive feedback and constructive suggestions that help improve this paper substantially. This work was performed in part using computing facilities at the College of William and Mary which were provided by contributions from the National Science Foundation, the Commonwealth of Virginia Equipment Trust Fund and the Office of Naval Research. This work was also supported in part by NSF awards 1565928 and 1618620.

References

[1] R. Elmasri, *Fundamentals of Database Systems*. Pearson Education India, 2008.

[2] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang, "iDistance: An Adaptive B+-tree based Indexing Method for Nearest Neighbor Search," *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 2, pp. 364–397, 2005.

[3] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS File System," in *USENIX Annual Technical Conference*, vol. 15, 1996.

[4] S. Chaudhuri and U. Dayal, "An Overview of Data Warehousing and OLAP Technology," *ACM Sigmod record*, vol. 26, no. 1, pp. 65–74, 1997.

[5] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin, "Fractal Prefetching B+-trees: Optimizing both Cache and Disk Performance," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, 2002, pp. 157–168.

[6] O. Rodeh, "B-trees, Shadowing, and Clones," *ACM Transactions on Storage (TOS)*, vol. 3, no. 4, p. 2, 2008.

[7] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey, "PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors," *Proc. VLDB Endowment*, vol. 4, no. 11, pp. 795–806, 2011.

[8] A. Natarajan and N. Mittal, "Fast Concurrent Lock-Free Binary Search Trees," in *ACM SIGPLAN Notices (PPoPP)*, vol. 49, no. 8. ACM, 2014, pp. 317–328.

[9] A. Braginsky and E. Petrank, "A Lock-Free B+ tree," in *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures (SPAA)*. ACM, 2012, pp. 58–67.

[10] T. Brown, F. Ellen, and E. Ruppert, "A General Technique for Non-Blocking Trees," in *ACM SIGPLAN Notices (PPoPP)*, vol. 49, no. 8. ACM, 2014, pp. 329–342.

[11] D. Drachsler, M. Vechev, and E. Yahav, "Practical Concurrent Binary Search Trees via Logical Ordering," *ACM SIGPLAN Notices (PPoPP)*, vol. 49, no. 8, pp. 343–356, 2014.

[12] K. Cooper and L. Torczon, *Engineering a Compiler*. Elsevier, 2011.

[13] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, "Compilers: Principles, Techniques, and Tools," 2006.

[14] A. Sodani, "Knights Landing (KNL): 2nd Generation Intel® Xeon Phi Processor," in *2015 IEEE Hot Chips 27 Symposium (HCS)*. IEEE, 2015, pp. 1–24.

[15] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[16] M. Copeland, J. Soh, A. Puca, M. Manning, and D. Gollob, "Microsoft azure and cloud computing," in *Microsoft Azure*. Springer, 2015, pp. 3–26.

[17] B. Fitzpatrick, "Distributed caching with memcached," *Linux journal*, vol. 2004, no. 124, p. 5, 2004.

[18] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, "Quickly generating billion-record synthetic databases," in *ACM Sigmod Record*, vol. 23, no. 2. ACM, 1994, pp. 243–252.

[19] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.

[20] J. Zhou and K. A. Ross, "Buffering Accesses to Memory-Resident Index Structures," in *Proceedings of the 29th international conference on Very large data bases-Volume 29*. VLDB Endowment, 2003, pp. 405–416.

[21] D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski, "Early Experience with a Commercial Hardware Transactional Memory Implementation," 2009.

[22] X. Wang, W. Zhang, Z. Wang, Z. Wei, H. Chen, and W. Zhao, "Eunomia: Scaling Concurrent Search Trees under Contention Using HTM," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 2017, pp. 385–399.

[23] T. Crain, V. Gramoli, and M. Raynal, "A Contention-Friendly Binary Search Tree," in *European Conference on Parallel Processing (Euro-Par)*. Springer, 2013, pp. 229–240.

[24] T. Crain, V. Gramoli, and M. Raynal, "A Fast Contention-Friendly Binary Search Tree," *Parallel Processing Letters*, vol. 26, no. 03, p. 1650015, 2016.

[25] J. Fix, A. Wilkes, and K. Skadron, "Accelerating Braided B+ Tree Searches on a GPU with CUDA," in *2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance (A4MMC), in conjunction with ISCA*, 2011.

[26] M. Daga and M. Nutter, "Exploiting Coarse-Grained Parallelism in B+ Tree Searches on an APU," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 2012, pp. 240–247.

[27] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 339–350.

[28] A. Shahvarani and H.-A. Jacobsen, "A Hybrid B+-Tree as Solution for In-Memory Indexing on CPU-GPU Heterogeneous Computing Platforms," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 1523–1538.

[29] J. Rao and K. A. Ross, "Cache Conscious Indexing for Decision-Support in Main Memory," in *VLDB*, vol. 99, 1999, pp. 78–89.

[30] J. Rao and K. A. Ross, "Making B+-Trees Cache Conscious in Main Memory," in *ACM SIGMOD Record*, vol. 29, no. 2. ACM, 2000, pp. 475–486.

[31] L. G. Valiant, "A Bridging Model for Parallel Computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[32] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[33] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One Trillion Edges: Graph Processing at Facebook-Scale," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.

[34] S. Tasci and M. Demirbas, "Giraphx: Parallel yet Serializable Large-Scale Graph Processing," in *European Conference on Parallel Processing*. Springer, 2013, pp. 458–469.

[35] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A High-Performance Graph Processing Library on the GPU,"

in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2016, p. 11.

[36] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[37] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-Memory Cluster Computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[38] K. Siddique, Z. Akhtar, E. J. Yoon, Y.-S. Jeong, D. Dasgupta, and Y. Kim, "Apache Hama: An Emerging Bulk Synchronous Parallel Computing Framework for Big Data Applications," *IEEE Access*, vol. 4, pp. 8879–8887, 2016.

[39] E. Morel and C. Renvoise, "Global Optimization by Suppression of Partial Redundancies," *Communications of the ACM*, vol. 22, no. 2, pp. 96–103, 1979.

[40] D. M. Dhamdhere, "Practical Adaption of the Global Optimization Algorithm of Morel and Renvoise," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 2, pp. 291–294, 1991.

[41] V. K. Paleri, Y. Srikant, and P. Shankar, "Partial Redundancy Elimination: A Simple, Pragmatic, and Provably Correct Algorithm," *Science of Computer Programming*, vol. 48, no. 1, pp. 1–20, 2003.

[42] J. Knoop, O. Rüthing, and B. Steffen, "Lazy Code Motion," *ACM SIGPLAN Notices*, vol. 39, no. 4, pp. 460–472, 2004.

[43] P. Norvig, "Techniques for Automatic Memoization with Applications to Context-Free Parsing," *Computational Linguistics*, vol. 17, no. 1, pp. 91–98, 1991.

[44] A. T. Da Costa, F. M. França *et al.*, "The Dynamic Trace Memoization Reuse Technique," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2000, pp. 92–99.

[45] U. A. Acar, G. E. Blelloch, and R. Harper, *Selective Memoization*. ACM, 2003, vol. 38, no. 1.

[46] A. G. Gounares, Y. Li, C. D. Garrett, and M. D. Noakes, "Selecting Functions for Memoization Analysis," 2013, uS Patent App. 13/671,828.

[47] A. Tolic and A. Brodnik, "Deduplication in unstructured-data storage systems," *Elektroteh Vestn*, vol. 82, no. 5, p. 233, 2015.

[48] D. R. Bobbarjung, S. Jagannathan, and C. Dubnicki, "Improving duplicate elimination in storage systems," *ACM Transactions on Storage (TOS)*, vol. 2, no. 4, pp. 424–448, 2006.

[49] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis, "Dwarf: Shrinking the petacube," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, 2002, pp. 464–475.

[50] J. J. Koh, "Relational database schema integration by overlay and redundancy elimination methods," in *Strategic Technology, 2007. IFOST 2007. International Forum on*. IEEE, 2007, pp. 610–614.

[51] H. Andrade, S. Aryangat, T. Kurc, J. Saltz, and A. Sussman, "Efficient execution of multi-query data analysis batches using compiler optimization strategies," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2003, pp. 509–523.

[52] S. Aryangat, H. Andrade, and A. Sussman, "Time and space optimization for processing groups of multi-dimensional scientific queries," in *Proceedings of the 18th annual international conference on Supercomputing*. ACM, 2004, pp. 95–105.

[53] S. Aryangat, "Optimizing the execution of batches of data analysis queries," Ph.D. dissertation, 2004.