APPENDIX TO CUBIT

A PARALLELIZE EXISTING BITMAP INDEXES

UpBit. We parallelize UpBit by using fine-grained reader-writer latches. Specifically, the <VB, UB> pair of each value v is protected by a reader-writer latch, denoted $latch_v$. A global latch $latch_{nor}$ is used to protect the global shared variables like *number_of_rows*. An insert operation acquires $latch_{nor}$ and the corresponding $latch_v$ in order, before updating the UB of value v and the variable *number_of_rows*. An update acquires the $latch_v$ for both the old and new values, and a delete acquires the corresponding $latch_v$ before updating the corresponding UB. A query acquires the corresponding $latch_v$ in shared mode, and upgrades it to exclusive mode once a merge operation is required.

UCB. All UDIs of UCB update the only EB, and all queries access this EB simultaneously. We thus choose a global reader-writer latch to synchronize concurrent queries and UDIs.

In-place. It is possible to parallelize In-place by using fine-grained reader-writer latches that work the same as in UpBit. However, when using this mechanism, an insert needs to acquire c latches, where c is the cardinality, before updating the last bits of the bitvectors. This increases the probability of being deadlock. We thus parallelize In-place by using a global reader-writer latch, the same as in UCB. Surprisingly, the parallelized In-place outperforms UCB for high concurrency (§6.2).

B PSEUDOCODE FOR CUBIT

B.1 Data Structure

Algorithm 1 lists the shared variables and core data structures of CUBIT. Structure RUB consists of a 32-bit row ID, an 8-bit counter indicating the number of 1s in this RUB, and an array of 12-bit integers keeping track of the IDs of the virtual UBs containing the 1s. The size of each field in RUB is pre-defined and configurable. For the configuration in Algorithm 1, the number of rows, number of 1s in each RUB, and cardinality of the indexed attribute are in maximum 2^{32}, 2^8 and 2^{12}, respectively.

Each transaction allocates an instance of the structure TxnDesc, which contains the timestamps indicating when the transaction started (*start-ts*) and committed (*commit-ts*), the number of rows of this snapshot, and an array of RUBs generated by the UDIs in this transaction.

Each BtvDesc contains the timestamp value indicating when this VB is inserted into the system, a shortcut pointer referencing to the TxnDesc from where subsequent queries and UDIs start searching related RUBs, and a pointer to the underlying bitvector,

The global array, *btvs*, contains cardinality pointers, each of which references to a version chain. The global variables TIMESTAMP and N_ROWS record the current timestamp value and the number of rows of the dataset. Pointers HEAD and TAIL points to the head and tail of TXN_LOG. In CUBIT-lk, concurrent updates to the global variables are protected by the latch *bitmap_lk*. Note that queries do not acquire this latch, and that in CUBIT-lf, this latch is removed.

Each active thread contains an instance of *ThreadInfo*, which points to the TxnDesc of the active transaction, or to NULL if this thread currently does not involve in any active transaction.

Algorithm 1: Structures and shared variables of CUBIT.

```
struct RUB
    row_id : 32;  n_ones : 8;  ones_idx[];
struct TxnDesc
    int start_ts: 64;  int n_rows: 32;  int commit_ts: 64;
    RUB rubs[];  TxnDesc *next;
struct BtvDesc
    int commit_ts: 64;  TxnDesc *start_txn;
    Bitvector *vb;  BtvDesc *next;
struct ThreadInfo
    TxnDesc *txn;
Global variables:
    BtvDesc *vcs[cardinality] ;      /* Version chains */
    int TIMESTAMP : 64 ;       /* Timestamp counter */
    int N_ROWS : 32 ;    /* Number of rows counter */
    TxnDesc *HEAD,  *TAIL ; /* Pointers to TXN_LOG */
    mutex bitmap_lk ; /* Global latch for CUBIT-lk */
Shared variables for thread i:
    ThreadInfo  *th[i];
```

B.2 Query and UDI

The pseudocode for the query operation is shown in Algorithm 2. Assume a query is in a transaction T. It first traverses the corresponding version chain and locates the BtvDesc btv with the lagest $btv.commit{-}ts$ that is less than or equal to $T.start{-}ts$ (Line 22). The query then collects the RUB set for the corresponding virtual UBs by traversing TxnDescs with their *commit-ts* $\in$ ($btv.commit - ts$, $T.start - ts$] (Line 24). If the return set is empty, which indicates that btv is the latest version when T took the snapshot, Query reuses this bitvector for evaluation (Line 27). Otherwise, Query makes a private copy of the underlying bitvector (Line 30), applies the RUBs to the copy (Line 32), and then evaluate the private copy (Line 33). If the number of RUBs merged into the private copy is larger than a pre-defined threshold, this query sends a merge request to the background maintenance threads which are in charge of inserting the newly-generated bitvector into the version chain (§B.4).

The helper function rubs_involving_val checks a list of TxnDescs and returns all of the RUBs that will be applied to a specified bitvector. By traversing the TxnDescs in the order when they are committed, rubs_involving_val can always select the latest version of the RUB, if there are several versions in different TxnDescs.

UDIs of transaction T do not modify existing bitvectors. In contrast, they build RUBs reflecting the modifications, and append RUBs to T's TxnDesc. UDIs take effect when T is committed, which we discuss §B.3.

B.3 Transaction Semantics

Algorithm 3 presents the pesudocode for the transaction semantics. Programmers can explicitly invoke the function TxnBegin to start a new transaction; otherwise, CUBIT runs in the *autocommit* mode.

Algorithm 3: Transaction semantics of CUBIT.

```
45 function TxnBegin(ThreadInfo *th)
46   if th->txn ≠ null return th->txn;
47   th->txn ⟵ allocate TxnDesc ;
48   th->txn-><start_ts, n_rows> ⟵ <TIMESTAMP,
     N_ROWS> ;
49   return th->txn;
50 function TxnCommit(ThreadInfo *th)
51   if th->txn = null   return -EPERM;
52   if th->txn->rubs[] is empty   return -ENOENT;
53   mutex_lock(bitmap_lk) ;
54     tsp_begin ⟵ th->txn->start_ts;
55     tsp_end ⟵ TAIL->commit_ts;
56     if (check_conflict(th->txn, tsp_begin, tsp_end) ≠ 0):
57       return -ERETRY;
58     th->txn->commit_ts = TIMESTAMP + 1 ;
59     Fill row_id field of RUBs generated by Inserts ;
60     Append th->txn at the tail of TXN_LOG ;
61     TAIL ⟵ th->txn;
62     <TIMESTAMP, N_ROWS> ⟵ <TIMESTAMP + 1,
     N_ROWS + #Inserts in this transaction>;
63     th->txn ⟵ NULL;
64   mutex_unlock(bitmap_lk) ;
65 function check_conflict(txn, tsp_begin, tsp_end)
66   for each TxnDesc T in TXN_LOG(tsp_begin, tsp_end]:
67     if (T->rubs.row_ids ∧ txn->rubs.row_ids) ≠ ∅:
68       return -EINVAL;
69   return 0;
```

Algorithm 2: Query of CUBIT.

```
19 function Query(ThreadInfo *th, int val)
20   BtvDesc *btv ⟵ btvs[val];
21   int tsp_end ⟵ th->txn->start_ts;
22   Traverse version chain until btv->commit_ts ≤ tsp_end ;
23   int tsp_begin ⟵ btv->commit_ts;
24   rubs ⟵ rubs_involving_val(val, tsp_begin, tsp_end) ;
25   Bitvector * vb_new;
26   if rubs is empty:
27     vb_new ⟵ btv.vb ;
28   else:
29     vb_new ⟵ allocate Bitvector;
30     vb_new->copy(btv.vb) ;
31     for each rub in rubs:
32       vb_new[rub.row_id] ⟵ ¬ vb_new[rub.row_id] ;
33   Evaluate(vb_new) ;
34   if (rubs.size() > MERGE_THRESHOLD):
35     register_merge_request(th, val, btv, vb_new, rubs) ;
36 function rubs_involving_val(val, tsp_begin, tsp_end)
37   map<int:32, int:64> rubs;
38   for each TxnDesc T in TXN_LOG(tsp_begin, tsp_end]:
39     for each rub in T->rubs:
40       if val ∈ rub.ones_idx[]:
41         rubs[rub.row_id] ⟵ rub;
42       else if exists rubs[rub.row_id];
43         rubs.erase(rub.row_id);
44   return rubs;
```

TxnBegin allocates an instance of TxnDesc (line 47) and retrieve the current TIMESTAMP and N_ROWS (line 48), which, in effect, takes a snapshot of the bitmap index and allows queries and UDIs in this transaction to work on this snapshot.

The function TxnCommit pairs with TxnBegin. We present the latch-based version of TxnCommit in Algorithm 3, and will discuss the non-blocking version §4.9.

TxnCommit must be atomic, with respect to other concurrent transactions. To that end, TxnCommit first grabs the global latch *bitmap_lk* (line 53). It then checks if the RUBs generated in this transaction conflict with those in the TxnDescs committed by other concurrent threads (line 66). If there is no conflict, TxnCommit sets this transaction's commit-ts to the current value of TIMESTAMP in addition to 1 (Line 58), and fills the *row_id* fields of the RUBs generated by Inserts, according to the current value of N_ROWS (Line 59). Deferring the assignment of *row_ids* prevents conflicts among concurrent transactions that include Inserts. TxnCommit then appends the TxnDesc of this transaction at the tail of TXN_LOG (line 60), and updates the global variables accordingly (Line 62), before releasing the latch (line 64).

Incrementing the global TIMESTAMP makes concurrent queries become aware of T. To prevent concurrent queries from retrieving incorrect <TIMESTAMP, N_ROWS> pair, for ease of presentation, we place TIMESTAMP and N_ROWS in an aligned 128-bit word, which can be atomically read (line 48) and write (line 62) on modern 64-bit architectures. Updating N_ROWS without using the 128-bit word demands strict orders in accessing/updating these two variables, which can be found in the source code of CUBIT.

B.4 Merge Operation

Each active worker threads (i.e., threads performing queries and UDIs) maintains an efficient, non-blocking first-in-first-out queue [30], denoted *FIFO queue*. Query generates a merge request by inserting the request into its FIFO queue (Line 35). CUBIT automatically manages a group of background maintenance threads, each of which performs a while loop that repeatedly checks if there are merge requests in any FIFO queues.

Once a merge request is received, the maintenance thread first retrieves the relevant information out of the FIFO queue. It then creates a synthetic TxnDesc, by making a copy of the received RUB set and cleaning the 1s that were in the virtual UBs and have been merged into the newly-generated VB.

The Merge operations must be synchronized to avoid conflicts with TxnCommit and with each other. In CUBIT-lk, we reuse the global latch bitmap_lk, which is also used to protect TXN_LOG. That is, on entry into the function, Merge first grabs bitmap_lk. In §4.9, we discuss how Merge can synchronize without acquiring latches.

The remaining of Merge is similar to TxnCommit. Specifically, Merge first employs the classic check-after-locking mechanism to check (1) if other concurrent Merges have inserted other new VBs into the same version chain, and (2) if there are any conflicts between the synthetic TxnDesc and the TxnDescs committed by other concurrent transactions, the same as in TxnCommit. If there is no conflict, Merge sets the *commit_ts* in both the new VB and the synthetic TxnDesc, sets the new VB's *start_txn* to point to the synthetic TxnDesc, and then inserts the new VB and synthetic TxnDesc into the corresponding version chain and TXN_LOG, respectively. Merge then increments the global TIMESTAMP, and finally releases the latch.

C LOCK-FREE MVCC

We choose Michael and Scott's classic lock-free first-in-first-out linked list [43], denoted *MS-Queue*, as the implementation of TXN_LOG in CUBIT-lf. MS-Queue provides lock-free Insert and Delete operations that do not block each other. Specifically, an insert operation A first sets the *next* pointer of the last node of the list to a new node n, by using an atomic *compare-and-swap (CAS)* instruction. Once succeeds, this is the linearization point of A [34], meaning that n has been successfully inserted into the list, with respect to all other concurrent threads. The operation A then attempts to set the global pointer TAIL to point to n by using another *CAS* instruction. Note that A can be suspended before executing this CAS. Another insert operation B, which runs concurrently and failed because of the successful insertion of the node n, first *helps* A complete by setting the global pointer TAIL to point to n by using *CAS*, and then restarts from scratch. Concurrent delete operations synchronize with each other similarly. The lock-free property of MS-Queue roots from this helping mechanism.

In CUBIT-lf, a TransCommit and Merge succeeds by appending its TxnDesc at the tail of the MS-Queue, by using a single *CAS* instruction, the same as the original MS-Queue algorithm. This is too the linearization point of a successful TransCommit and Merge. The subsequent works, however, are far more than updating TAIL. Recall that in CUBIT-lk, under the protection of the global latch, TransCommit updates TAIL, TIMESTAMP, and N_ROWS, and Merge updates TAIL, TIMESTAMP, and the head pointer of the corresponding version chain.

We thus extend the *helping* mechanism to atomically update a group of variables, inspired by the recent lock-free designs [6, 11]. Specifically, before appending a TxnDesc at the tail of TXN_LOG, each TransCommit and Merge records the old values and the new values of the variables to be updated in its TxnDesc. Once this TransCommit or Merge A linearizes (i.e., the *CAS* succeeds), its TxnDesc becomes reachable by other threads via the next pointers of the TxnDescs in TXN_LOG). Another TransCommit or Merge B, which failed to append their TxnDescs, helps A complete. Specifically, for each variables to be updated, B retrieves the old and new values, and changes the variable to the new value by using a *CAS* instruction. Once the *CAS* fails, which indicates that this variable has been updated by either A or other helpers, B simply skips updating this variable. After helping update all variables in A's TxnDesc, B starts over. Recall that TIMESTAMP and N_ROWS are placed in an aligned 128-bit long variable, such that CUBIT-lf atomically updates them by using a single *double-words CAS* instruction, which has been widely provided by modern 64-bit architectures.

Correctness. In theory, the ABA problem [34] may arise in updating TIMESTAMP and N_ROWS. However, it takes TIMESTAMP more than one million years to wraparound, if there are 500K UDIs per second. Similarly, N_ROWS monotonically increases and can be 64-bit long. Moreover, no ABA problem can arise in updating other variables (TAIL and the pointer to the version chain), because of the epoch-based reclamation mechanism used in CUBIT, which guarantees that no memory space can be reclaimed (and then reused) if any thread holds a reference to it. We thus get the conclusion that in practice, CUBIT-lf is immune to the ABA problem.

We use the term *shared variables* to describe the global variables updated by TransCommit and Merge. CUBIT-lf is correct because of the following facts. (1) Shared variables can only be updated after a TxnDesc has been successfully appended at the tail of TXN_LOG. (2) How shared variables are updated is pre-defined in this TxnDesc by specifying the old and the new values of each variable. (3) Updating shared variables can be performed by any active threads, such that concurrent threads can help each other complete. (4) Shared variables are updated by only using *CAS* instructions. (5) No ABA problem can arise. Overall, CUBIT-lf guarantees that when TxnCommit or Merge owning a TxnDesc completes, each shared variable has been updated to the specified new value, and has been updated only once.

D TPC-H

In this appendix, we discuss in detail how we generate the TPC-H benchmark and how CUBIT is maintained to support full queries. Our DBMS provides snapshot isolation for transactions including queries and real-time modifications.

Dataset. Our prototype DBMS maintains two tables, *ORDERS* and *LINEITEM*. The dates of the tuples in *LINEITEM* span the range of years [1992, 1998], the discounts are distributed in the range [0, 0.1] with increments of 0.01, and the quantities are in the range [1, 50].

Transactions. We use the Forecasting Revenue Change Query (Q6) as the query workload. The SQL code for Q6 is listed in Algorithm 4. The value of the first parameter *DATE* is the first of January of a randomly selected year in between [1993, 1997], the parameter *DISCOUNT* is randomly selected within [0.02, 0.09], and the parameter *QUANTITY* is randomly selected within [24, 25]. The DBMS creates three CUBIT instances, respectively on the attributes *l_shipdate*, *l_discount*, and *l_quantity*. As a result, each Q6 selects the bitvectors corresponding to 1 of the 7 possible years, 3 of the 11 possible discounts, and 24 or 25 of 50 possible quantities, leading to an average selectivity $\frac{1}{7} \times \frac{3}{11} \times \frac{24.5}{50} \approx 2\%$. Note that we use binning to reduce the number of bitvectors for the attribute Quantity from 25 to 3. Values less than, equal to, and larger than 24 go to one of the three bitvectors, respectively. For each Q6 with CUBIT, the DBMS make a private copy of one bitvector and then performs bitwise OR/AND operations among 5 (1+3+1) or 6 (1+3+2) bitvectors, to retrieve a list of tuple IDs in *LINEITEM*. The Q6 then fetches these tuples to calculate the final revenue result.

Algorithm 5: TPC-H RF1.

```
INSERT a new row into the ORDERS table
LOOP random[1, 7] times
    INSERT a new row into the LINEITEM table
END LOOP
```

Algorithm 6: TPC-H RF2.

```
DELETE from ORDERS where o_orderkey = [VALUE]
DELETE from LINEITEM where l_orderkey = [VALUE]
```

Algorithm 4: TPC-H Q6.

```
SELECT sum(l_extendeprice × l_discount) as revenue
FROM LIMEITEM
WHERE l_shipdate >= date'[DATE]'
    and l_shipdate < date'[DATE]' + interval '1' year
    and l_discount between [DISCOUNT] ± 0.01
    and l_quantity < [QUANTITY];
```

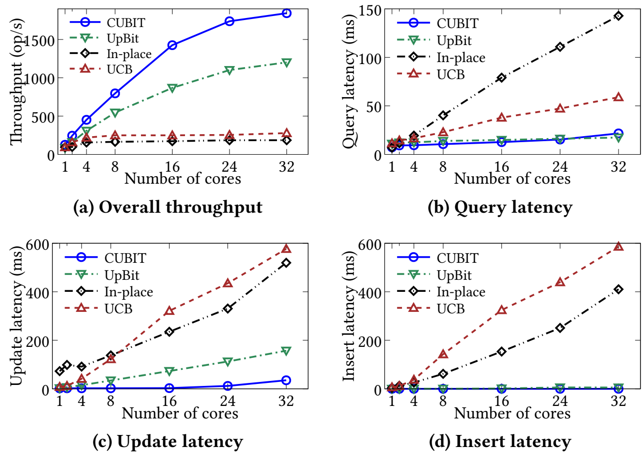
We use a modified New Sales Refresh Function (RF1) and a modified Old Sales Refresh Function (RF2) as the workload of updates. Since our DBMS with CUBIT supports real-time updates to data, it is not necessary for RF1 and RF2 to batch together a number of modifications and then apply them in a batch mode. In contrast, each RF1 and RF2 modifies a single tuple in the table *ORDERS* and the corresponding a few (1-7) tuples in the table *LINEITEM*, and then updates the three CUBIT instances accordingly, altogether in one transaction. The SQL code for RF1 and RF2 is listed in Algorithms 5 and 6. In the experiments, the operation distribution of each worker thread is set to 98, 1, and 1 for Q6, RF1, and RF2, respectively.

Unified CC Mechanism. Integrating CUBIT into a DBMS demands a unified CC mechanism to synchronize concurrent UDIs on both the indexes and the underlying tuples, while allowing wait-free queries. To this end, we choose the optimistic version of Hekaton [24, 37] (an MVCC with timestamping) as the CC mechanism for the underlying tuples, and refer CUBIT-lk to the global variables (e.g. *TIMESTAMP* and the number of rows) maintained by Hekaton. Specifically, on the entry of every transaction, the current *TIMESTAMP* is fetched and is used by queries to retrieve the corresponding versions of the bitmap indexes and the underlying tuples. For UDIs, new versions of the tuples are created, and at the same time, the *End* fields of the old versions are set to the transaction's ID. The *End* fields also serve as the latches for these tuples to prevent write-write conflicts. Corresponding RUBs are then generated. To commit, a transaction containing UDIs (1) grabs a global commit latch, (2) checks if there are write-write conflicts on both the bitmap indexes and the underlying tuples, (3) appends the RUBs at the tail of the TXN_LOGs of the bitmap indexes (Algorithm 3), (4) increments the global *TIMESTAMP*, (5) release the global commit latch, and then (6) sets the timestamp fields of the corresponding tuples. There is a time gap in between the steps 4 and 6, during which concurrent queries may retrieve the latest timestamp but the corresponding versions of the tuples are not ready yet. In this case, queries perform *speculative reads* [37] by proactively (1) checking the transaction's private workspace via the *End* fields of the old versions of the tuples and (2) reading the new versions if applicable. The whole system implements snapshot isolation.

The primary benefit of the unified CC mechanism is that queries are wait-free, even if UDIs are in progress, because once queries get a start timestamp, the corresponding versions of the bitmap indexes and the underlying tuples are guaranteed to be accessible. Concurrent UDIs may contend for the global commit latch, but it is acceptable because the critical section is light-weight and the contention is low for OLAP workloads.

E ZIPFIAN DISTRIBUTION

The distribution of data among bitvectors is critical to the performance of updatable bitmap indexes for the following two reasons. First, biased distributions may lead to a situation where a few *target* bitvectors contain many more 1s than others, such that they are less compressible. Second, the target bitvectors face higher contention levels among concurrent UDIs. We thus evaluate how a non-uniform distribution affects the performance of updatable bitmap indexes. We use the same testbed configuration as in §6.2, except that the dataset follows the Zipfian distribution with the distribution parameter α being set to 1.5, which implies that about 40% of the entries have the two most popular values, and the remaining entries are uniformly distributed in the entire value domain.



(a) Overall throughput (b) Query latency

(c) Update latency (d) Insert latency

Figure 14: Overall throughput and mean latency of bitmap indexes with Zipfian distribution (α = 1.5) as a function of the number of cores.

The results are shown in Figure 14. Both the overall throughput and mean query latency of all algorithms are improved by about 2×, compared to the case with uniform data distribution. The reason is that most bitvectors contain very few 1s, and are thus highly compressible, and queries on these bitvectors are very fast. Since queries dominate 90% of the total operations, the overall throughput of all bitmap indexes also increases by about 2×.