APPENDIX TO CUBIT

A PARALLELIZE PRIOR BITMAP INDEXES

UpBit. We parallelize UpBit by associating a reader-writer latch with each value v of the indexed domain, denoted $latch_v$. A latch $latch_{nor}$ is used to protect the global variables like *number_of_rows*. An insert operation acquires $latch_{nor}$ and the corresponding $latch_v$, before updating the UB of value v and the global variable. An update acquires the latches for both the old and the new values, and a delete acquires the $latch_v$ for the old value v. A query acquires the corresponding $latch_v$ in shared mode, and upgrades it to exclusive mode when a merge operation is required.

UCB. UCB's UDIs update the only EB, and queries read this EB simultaneously. We thus use a global reader-writer latch to synchronize concurrent queries and UDIs.

In-place. One way to parallelize In-place is to use fine-grained reader-writer latches that work the same way as in UpBit. However, with this mechanism, an insert needs to acquire *cardinality* latches before appending bits to the tail of all the VBs, increasing the probability of deadlock. Instead, we parallelize In-place by using a global reader-writer latch, the same as in UCB.

B PROGRAMMING SEMANTICS OF CUBIT

API. CUBIT complies with the standard specification for database indexes and is aware of the standard database transaction semantics [49]. It provides the following API:

- Update(row, val) retrieves the current value of the specified *row*, updates the value to *val*, and returns true.
- Delete(row) deletes the specified *row*, and returns true.
- Insert(val) appends the value *val* to the tail of the bitmap index, and returns true.
- Query(val) finds whether the specified value *val* exists in the bitmap index, and if so, returns a list of matching positions.
- TransBegin() marks the start of a new transaction T by taking a snapshot, on which T's queries and UDIs are performed.
- TransCommit() attempts to commit T. It contains the linearization point [34] at which this transaction's UDIs atomically take effect with respect to other concurrent transactions.
- TransAbort() gives up this transaction.

Further, there is an internal function Merge that propagates changes captured in UBs to VBs and reinitializes the UBs (§4.8).

Write Skew Anomaly. CUBIT satisfies *snapshot isolation* [13] that is not serializable, such that CUBIT permits write skew anomaly [13]. This is the same as most modern MVCC-based database systems [24]. CUBIT can detect write skew anomaly, abort conflicting transactions, and guarantees serializability, by also recording the read set of each transaction and checking read-write conflicts before committing each transaction.

C PSEUDOCODE FOR CUBIT

We now present the key data structures and functions of CUBIT based on its overall architecture shown in Figure 14.

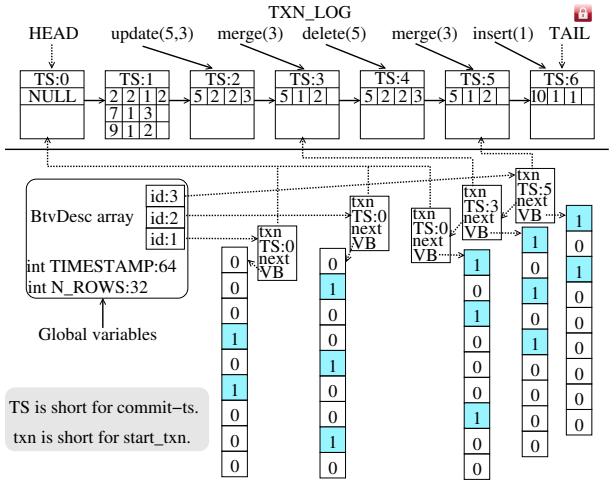

Figure 14: Illustrative architecture of CUBIT. The initial state is equivalent to Figure 2(d), followed by an update operation to change the 5_{th} row to value 3, a merge operation on value 3, a delete to remove the 5_{th} row, another merge on value 3, and an insert operation to append a value 1.

C.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 positions of the 1s. The size of each field in RUB is pre-defined and configurable. For the configuration example 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.

Algorithm 1: Structures and shared variables of CUBIT.

```
1  struct RUB
2    row_id : 32;  n_ones : 8;  pos[];
3  struct TxnDesc
4    int start-ts: 64;  int n_rows: 32;  int commit-ts: 64;
5    RUB rubs[];  TxnDesc *next;
6  struct BtvDesc
7    int commit-ts: 64;  TxnDesc *start_txn;
8    Bitvector *vb;  BtvDesc *next;
9  struct ThreadInfo
10   TxnDesc *txn;
11 Global variables:
12   BtvDesc *btvs[cardinality] ;                  /* Version chains. */
13   int TIMESTAMP : 64 ;                        /* Timestamp counter. */
14   int N_ROWS : 32 ;                         /* Number of rows counter. */
15   TxnDesc *HEAD,  *TAIL ;                     /* Pointers to TXN_LOG. */
16   mutex bitmap_lk ;                       /* Global latch for CUBIT-lk. */
17 Shared variables for thread i:
18   ThreadInfo  *th[i];
```

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.

The structure BtvDesc contains the timestamp indicating when it is committed, a shortcut pointer referencing to the TxnDesc from where to search related RUBs, and a pointer to the underlying VB.

The global array *btvs* contains cardinality pointers, each of which points to a version chain. The global variables TIMESTAMP and N_ROWS record the current timestamp value and the number of rows of the dataset. The pointers HEAD and TAIL points to the head and tail of TXN_LOG. In CUBIT-lk, 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 is set to NULL if this thread currently does not involve in any active transaction.

C.2 Query and UDI

The pseudocode for the query operation is shown in Algorithm 2. Assume that a query is in a transaction T. It first traverses the corresponding version chain and locates the BtvDesc *btv* with the largest *commit-ts* that is less than or equal to *T.start-ts* (Line 22). The query then collects a RUB set by traversing TxnDescs with their *commit-ts* $\in$ (*btv.commit-ts*, *T.start-ts*] (Line 24). If the result set is empty, which indicates that *btv* is the latest version when T took the snapshot, we evaluate using *btv* (Line 27). Otherwise, we make a private copy *vb_new* of *btv* (Line 30), merge RUBs into *vb_new* (Line 32), and then evaluate using *vb_new* (Line 33). If the number of RUBs merged into the private copy is larger than a predefined threshold, the query sends a merge request to the background maintenance threads which are in charge of inserting the newly-generated VB into the version chain (§C.4).

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.pos[]:
41         rubs[rub.row_id] ← rub;
42       else if exists rubs[rub.row_id];
43         rubs.erase(rub.row_id);
44   return rubs;
```

The helper function *rubs_involving_val* checks a list of TxnDescs and returns a RUB set that will be applied to the corresponding VB. By traversing the TxnDescs in the order in which they are committed, *rubs_involving_val* can always select the latest version of a RUB, if there are several versions in different TxnDescs.

UDIs of transaction T do not modify VBs. In contrast, they append RUBs reflecting the modifications to *T's* TxnDesc. UDIs take effect when T is committed, which we will discuss in §C.3.

C.3 Transaction Semantics

Algorithm 3 presents the pesudocode for the transaction semantics. Programmers can explicitly start a new transaction by invoking the function TxnBegin. Otherwise, CUBIT runs in *autocommit* mode. Each TxnBegin allocates an instance of TxnDesc (line 47) and retrieves the current TIMESTAMP and N_ROWS (line 48) which essentially takes a snapshot of the bitmap index.

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) ;          /* For CUBIT-lk. */
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 to 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;
```

The function TxnCommit pairs with TxnBegin. We present the pseudocode for CUBIT-lk in Algorithm 3 and will discuss the nonblocking version in §4.9. CUBIT-lk commits a transaction by appending its TxnDesc to the tail of TXN_LOG and updating the global variables accordingly, which must be atomic with respect to other concurrent transactions. To that end, the function TxnCommit acquires the global latch *bitmap_lk* (line 53) and 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 the TxnDesc'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 the insert operations according to the current value of N_ROWS (Line 59). TxnCommit then appends the TxnDesc of this transaction to the tail of TXN_LOG (line 60), and updates the global variables accordingly (Line 62), before releasing the latch (line 64).

Note that incrementing the global variable TIMESTAMP makes T visible to other transactions. To prevent other queries from retrieving an incorrect <TIMESTAMP, N_ROWS> pair, we place these two variables in an aligned 128-bit word that can be atomically

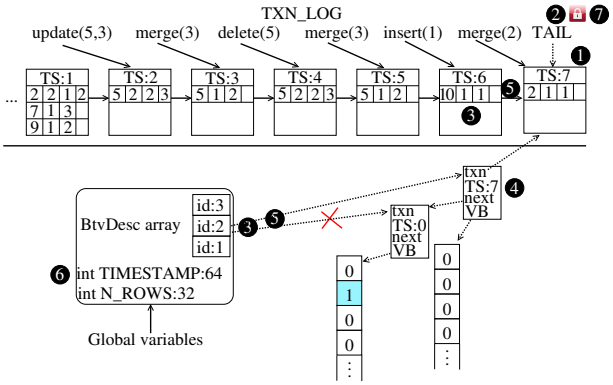

Figure 15: Illustrative example of the merge operation of CUBIT-lk. The initial state is equivalent to Figure 14 (The VBs of values 1 and 3 are omitted.) A merge operation on the VB of value 2 execute the steps 1 - 7 atomically with respect to other transactions.

read (line 48) and written (line 62) on modern 64-bit architectures. Accessing these two variables without using 128-bit words demands a strict access order, which can be found in the source code of CUBIT.

C.4 Merge Operation

A non-blocking first-in-first-out (FIFO) queue is associated with each worker thread. Once a merge operation is required, a query inserts a merge request into its FIFO queue. CUBIT manipulates a group of background maintenance threads that repeatedly check if there are merge requests in the FIFO queues. Once a merge request arrives, a maintenance thread extracts the request out of the FIFO queue and then invokes the function Merge (Algorithm 4).

Algorithm 4: Merge operation.

```
70 function Merge(*th, val, btv_old, btv_new);          /* For CUBIT-lk. */
71   TxnDesc *txn_merge ← generate synthetic TxnDesc;
72   mutex_lock(bitmap_lk);
73     if (btvs[val] != btv_old)    return -ERETRY;
74     tsp_begin ← th->txn->start-ts;
75     tsp_end← TAIL->commit-ts;
76     if (check_conflict(txn_merge, tsp_begin, tsp_end) ≠ 0):
77       return -ERETRY;
78     BtvDesc *btv_merge ← allocate BtvDesc;
79     btv_merge->commit-ts ← TIMESTAMP + 1;
80     txn_merge->commit-ts ← btv_merge->commit-ts;
81     btv_merge->start_txn ← txn_merge;
82     btv_merge->vb ← btv_new.vb;
83     Insert btv_merge to the head of btvs[val];
84     Insert txn_merge to the tail of TXN_LOG;
85     Increment TIMESTAMP by 1;
86   mutex_unlock(bitmap_lk);
```

Merge operations must be synchronized to avoid conflicts with TxnCommit and with each other. To that end, CUBIT-lk use the global latch *bitmap_lk*, and CUBIT-lf leverages a *helping* mechanism and emits this latch (§4.9).

Figure 15 illustrates the main steps of the merge operation of CUBIT-lk on the VB of value 2. A merge operation ❶ creates a synthetic TxnDesc, by making a copy of the received RUB set and cleaning the 1s that have been merged into the newly-generated VB. It then ❷ acquires the global latch *bitmap_lk* and ❸ employs a check-after-locking mechanism to check if (1) other concurrent merge operations have inserted new VBs into the same version chain, or (2) there are any conflicts between the synthetic TxnDesc and the TxnDescs that have been committed by other concurrent transactions. If there is no conflict, the merge operation ❹ sets the *commit-ts* in both the new VB and the synthetic TxnDesc, and refers the new VB's *start_txn* to the synthetic TxnDesc. The merge operation then ❺ inserts the new VB and the synthetic TxnDesc into the corresponding version chain and TXN_LOG, respectively. It finally ❻ increments the global TIMESTAMP which makes its TxnDesc to be visible to other transactions, before ❼ releasing the latch.

D NON-BLOCKING CUBIT

The standard MVCC mechanism notably uses latches to serialize updates to the same portion of the data. For example, CUBIT-lk uses the global latch *bitmap_lk* to serialize TxnCommit and Merge operations updating TXN_LOG. To address this performance limitation, we present a lock-free MVCC mechanism that enables the non-blocking CUBIT-lf.

Standard Helping Mechanism. The core idea behind our lock-free design is a *helping* mechanism [6, 11]. Under the hood, we choose Michael and Scott's classic lock-free first-in-first-out linked list [42], denoted *MS-Queue*, as the implementation of TXN_LOG. MS-Queue provides lock-free insert and delete operations that do not block each other. Specifically, an insert operation A 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. If successful, this is the linearization point of A [33], implying that n has been successfully inserted into the list with respect to other concurrent operations. The operation A then attempts to set the global pointer TAIL to point to n by using another *CAS* instruction. If A is suspended before executing the second *CAS*, other insert operation B, which failed because of the successful insertion of the node n, first *helps* A complete by setting the global pointer TAIL by also using *CAS* instructions. The operations B then restarts from scratch. Delete operations synchronize with each other similarly.

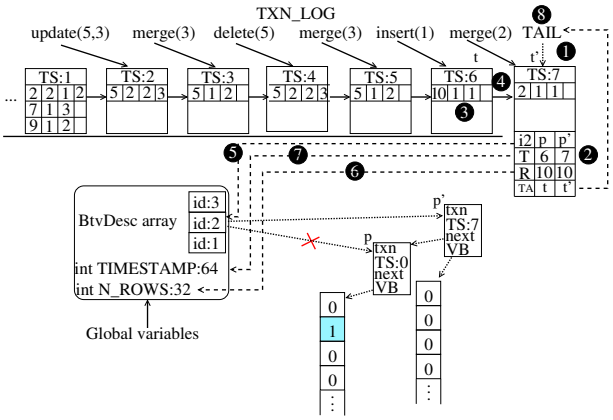

Figure 16: Illustrative example of the merge operation of CUBIT-lf. The initial state is equivalent to Figure 14. A merge operation on the bitvectors of value 2 execute the steps 1 - 8 atomically with respect to other transactions.

Challenge. For MS-Queue, in order to help *A*, all that the operation *B* needs to do is swing the TAIL pointer. In CUBIT, however, the function TransCommit needs to update several global variables including TAIL, TIMESTAMP, and N_ROWS, and the function Merge needs to update TIMESTAMP, TAIL, and the head pointer of the corresponding version chain. We thus extend the standard helping mechanism.

Helping Mechanism in CUBIT. Figure 16 illustrates the main steps of the merge operation of CUBIT-lf on the VB of value 2. A merge operation *A* ❶ creates a synthetic TxnDesc, the same as in CUBIT-lk. It then ❷ records the old and the new values of the variables to be updated. For example, the first row <i2, p, p'> indicates that the second entry in the global BtvDesc array will be changed from p to p', which essentially inserts the newly-generated VB (with TS=7) into the version chain. Similarly, the second row <T, 6, 7> indicates that the global TIMESTAMP will be changed from 6 to 7. The merge operation then ❸ checks if there are any conflicts, the same as in the merge operation of CUBIT-lk. It then ❹ linearizes by appending its TxnDesc to the tail of TXN_LOG (MS-Queue), by using a *CAS* instruction. If successful, *A*'s TxnDesc becomes reachable to other transactions via the next pointers of the TxnDescs in TXN_LOG. Another TransCommit or Merge operation *B*, which failed to append its TxnDesc, helps *A* complete by (1) reading the old and the new values of each variables to be updated in *A*'s TxnDesc, and (2) attempting to change each variable to the new value by using *CAS* instructions (steps ❺ - ❽). Once any *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 recorded in *A*'s TxnDesc, *B* starts over.

Immune to ABA problem. In theory, the ABA problem [33] 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 (e.g., TAIL and the pointers to the version chains) because of the epoch-based reclamation mechanism used in CUBIT (§5), which guarantees that no memory space can be reclaimed (and then, reused) if any worker thread holds a reference to it. We thus get the conclusion that in practice, CUBIT-lf is immune to the ABA problem.

Correctness. 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 to the tail of TXN_LOG. (2) How shared variables are updated is predefined 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 a TxnCommit or Merge owning a TxnDesc completes, each shared variable (a) has been updated to the specified new value, and (b) has been updated only once.

E TPC-H

We now present the details on the experimental setup of running CUBIT over the TPC-H benchmark.

Dataset. Our academic DBMS *DBx1000* 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].

Workloads. We use the Forecasting Revenue Change Query (Q6) as the query workload. The SQL code for Q6 is listed in Algorithm 5. 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].

Algorithm 5: TPC-H Q6.

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

Algorithm 6: TPC-H RF1.

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

Algorithm 7: TPC-H RF2.

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

We use modified New Sales Refresh Function (RF1) and Old Sales Refresh Function (RF2) as the workload of updates. Since our DBMS with CUBIT supports real-time updates to data, it is not

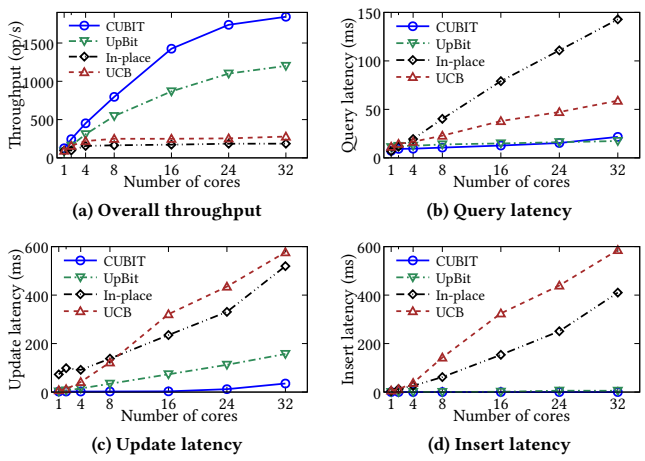

Figure 17: Overall throughput and mean latency of bitmap indexes with Zipfian distribution ($\alpha = 1.5$) as a function of the number of cores.

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 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 6 and 7. In the experiments, the operation distribution of each worker thread is set to 98, 1, and 1 for Q6, RF1, and RF2, respectively.

Indexes. We evaluate the performance of TPC-H Q6 by using CUBIT, hash indexes, B$^+$-trees, and scans.

CUBIT. 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 $\frac{1}{7} \times \frac{3}{11} \times \frac{24.5}{50} \approx 2\%$. 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.

Hash Indexes and B$^+$-Trees. The DBMS builds a Hash index with bucket size $B = 64K$ and a B$^+$-Tree with a fanout of 16. Since the TPC-H Q6 predicate only selects on the *l_shipdate*, *l_discount*, and *l_quantity* attributes, we blend them to generate keys that are used to index into the Hash and B$^+$-Tree indexes. Note that the keys are not unique and each key relates to about 50 entries. Distinct entries with identical key are maintained in a linked list.

Other than when using the Scan, each Q6 first retrieves a set of tuple IDs by leveraging different indexes, and then fetches these tuples to calculate the query result.

Synergistic CC Mechanism. We synergistically build the concurrency control (CC) mechanisms for CUBIT and the underlying tuples to enable wait-free queries. To this end, we choose the optimistic version of Hekaton [24, 36] (an MVCC with timestamping) as the CC mechanism for the underlying tuples, and use CUBIT-lk as the index. The global variables (e.g., TIMESTAMP and N_ROWS) are maintained by Hekaton and are read by CUBIT-lk.

When a transaction T starts, it reads the current TIMESTAMP, which is used by its queries and UDIs to retrieve the corresponding snapshot of the bitmap index 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 $T's$ ID. The *End* fields also serve as the latches for these tuples to prevent write-write conflicts. Corresponding RUBs are then generated.

In order to commit, a transaction that contains UDIs (1) acquires 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 to 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. We address this issue by allowing queries to perform *speculative reads* [36] by proactively (i) checking the transaction's private workspace via the *End* fields of the old versions of the tuples and (ii) reading the new versions if applicable. The whole system implements snapshot isolation.

The primary benefit of the synergistic CC mechanism is that queries are wait-free, even if UDIs are in progress. The reason is that once queries get a start-ts, 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.

F RESULTS WITH ZIPFIAN DISTRIBUTION

The detailed experimental results with Zipfian data distribution (§6.4) are shown in Figure 17. 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. 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 times.

G CORRECTNESS

We now study the operation sequences that increase the number of 1s in RUBs and the probability that these sequences take place.

FSM of RUBs. We study the transition of the RUBs of a row by using a Finite-State Machine (FSM), in which each node records the <VB, UB> pairs of a RUB. Except for the initial state (the top-left node indicating that the row is just allocated) and the final state (the top-right node indicating that the row has been deleted), all the <0, 0> pairs in the RUBs are removed, for ease of presentation and without loss of generality. Each arrow is labeled with the operation that triggers the transition. For example, an insert operation on the row just allocated changes its state from <0, 0> to <0, 1>, indicating that the bit in the corresponding UB has been set to 1.

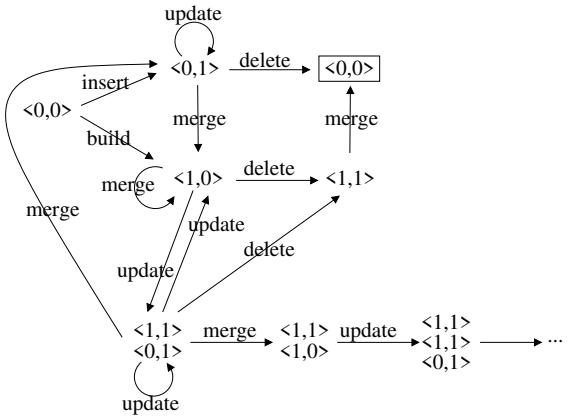

Figure 18: Finite-State Machine of the RUBs of a row by only recording the non-<0, 0> <VB, UB> pairs. Except for the bottom-right state, the number of 1s recorded in a RUB is zero to two. The only sequence of operations that increases the number of 1s in RUBs is a sequence of interleaved special update and merge operations, which takes place with extremely low probability.

An update may change a RUB from '<0,0>,<0,1>' to '<0,1>,<0,0>', leading to a circular arrow starting from and ending at the same node. That is, there is no transition to a new state because the <0, 0> pairs are omitted in the FSM.

The complete FSM is shown in Figure 18. We make the following observations.

- Except for the bottom-right state, the number of 1s in the RUBs of a row is zero to two with very high probability.
- The only sequence of operations that can increase the number of 1s in the RUBs is a sequence of interleaved update and merge operations. Specifically, only if (1) update operations always change a row to new values, (2) update and merge operations take place alternatively, (3) each merge always happens on the new value of the preceding update, and (4) no deletes take place, the number of 1s in the RUBs can grow.
- A RUB contains n 1s with probability less than $1/c^{n-1}$, where $c = cardinality$ (§4.2). When $c = 128$, the probability that a RUB contains three and seven 1s are $1/128^2 = 1/2^{14}$ and $1/128^6 = 1/2^{42}$, respectively, which happens extremely unlikely (§4.2).
- Once the number of 1s in the RUBs exceeds a pre-defined threshold, merge operations can be first executed on the values of the existing <1, 1> pairs to reduce the number of 1s.

Overall, the number of 1s in a RUB is in most cases, zero to two.