APPENDIX TO THE CUBIT PAPER

A PROGRAMMING SEMANTICS OF CUBIT

CUBIT complies with the classic specification for secondary indexes in DBMSs, and its API provides Query, Update, Delete, and Insert operations.

- Query(value or range) takes a (point or range query) predicate as parameter, and returns the result in the form of either a bitvector or an array of pointers to the matching tuples.
- Update(row, val) retrieves the current value of the specified *row*, updates the value to *val*, and returns *true* if succeeds.
- Delete(row) retrieves the current value of the specified *row*, deletes this row, and returns *true* if succeeds.
- Insert(val) appends the value *val* to the tail of the bitmap index, increments global variables like *N_ROWS*, and returns *true* if the insertion succeeds.

B LATCH-FREE CUBIT

Concurrency Issues with CUBIT-lk. CUBIT-lk employs a read-write latch to serialize concurrent UDIs that attempt to append *ULEs* at the tail of the per-index Delta Log. This design stems from the classic MVCC mechanisms in which concurrent updates use latches to avoid modifying the same portion of data (e.g., tuples or pages) simultaneously. Experimentally, we found that this latch may incur long-tail UDI latency because (1) UDIs on bitmap indexes lock in the granularity of bitvectors, which is typically significantly fewer than tuples or pages that are locked by typical MVCC mechanisms, and (2) skewed UDIs concentrate around a few hot-spots bitvectors leading to even higher contention. Further, CUBIT faces severe time constraints inherent in indexing in DBMSs.

Solution. We address the above-described challenges from two angles. First, when UDIs and merge operations conflict (i.e., attempting to append their *ULEs* to the tail of Delta Log simultaneously), they consolidate their *ULEs* and delegate committing them to subsequent UDIs. Second, instead of busy-waiting, suspended UDIs *help* the other make progress until completion, and then retry. The resulting algorithm, termed CUBIT-lf, provides non-blocking (latch-free) UDIs that never block each other and the system is guaranteed to always make progress (no suspension nor deadlock). This resolves the major cause of unexpectedly long tail latency of UDIs with MVCC.

Helping Mechanism Basis. Under the hood, we choose Michael and Scott's classic lock-free first-in-first-out linked list [7], denoted *MS-Queue*, as the implementation of Delta Log. MS-Queue provides latch-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 [6], 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 operation B then restarts from scratch. Delete operations synchronize with each other similarly.

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

Our Helping Mechanism for CUBIT. We propose a *helping* mechanism to atomically update a group of variables, inspired by recent latch-free designs [1, 3]. Specifically, each UDI and merge operation records the old values and the new values of the variables to be updated in its *ULE* before appending it to the tail of Delta Log by using a *CAS* instruction. Once this step O_1 succeeds (i.e., this UDI operation linearizes), the *ULE* becomes reachable to other threads via the next pointers of the *ULEs* in Delta Log. Another UDI or merge operation O_2, which failed to append its *ULE*, first helps O_1 complete. Specifically, for each variable to be updated, O_2 retrieves the old and the new values, and changes the variable to the new value by using *CAS* instructions. Once the *CAS* fails, which indicates that this variable has been updated by either O_1 or other helpers, O_2 simply skips updating this variable. After helping update all variables in O_1's *ULE*, O_2 starts over.

Correctness. CUBIT-lf is a concurrent data structure (set). We can prove its correctness from the following aspects [6].

Immune to ABA problem. In theory, the ABA problem [6] may arise in updating the variables 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, 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.

No-Bad-Thing-Happen (Correctness) Property. We use the term *shared variables* to describe the global variables updated by UDI and merge operations. CUBIT-lf is correct because of the following facts. (1) Shared variables can only be updated after a *ULE* has been successfully appended to the tail of Delta Log. (2) How shared variables are updated is pre-defined in this *ULE* 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 UDI and merge operation owning a *ULE* completes, each shared variable (a) has been updated to the specified new value, and (b) has been updated only once.

Good-Thing-Always-Happen (Liveness) Property. The arguments on linearization points (see above) suggest that CUBIT-lf provide wait-free queries and latch-free UDIs, that is, they never block.

C SIZE OF HUDS

In general, a HUD has only 0s. As UDIs accumulate, the number of 1s in a HUD increases, so does the size of the HUD (which is stored compactly as a list of positions). We now study the operation sequences that increase the size of a HUD, and then show that it is very unlikely that a HUB contains more than two positions.

FSM of HUDs. Conceptually, a HUD is a bit-array with a length equal to the cardinality of the domain, and the i^{th} bit in this array, denoted U_i, is associated with the corresponding bit of the i^{th} VB, denoted V_i. We study the transition of the HUD by using a Finite-State Machine (FSM), in which each node records the $<U_i, V_i>$ pairs for all possible i, denoted $<U, V>_i$. For ease of presentation, 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 are removed. Each arrow is labeled with the operation that triggers the transition. For example, an insert operation allocates a new HUD and changes its state from <0, 0> to <0, 1>, indicating that the corresponding bit of the HUD has been set to 1. An update may change a HUD 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 1. We make the following observations.

(1) Except for the bottom-right state, the number of 1s in each HUD is zero to two with high probability.

(2) The only operation sequence that increases the number of 1s of a HUD (assume <0, 1>$_{i1}$ initially) is as follows.

- **A1**: A merge happens on the VB i_1, resulting in the state <1, 0>$_{i1}$.
- **A2**: An update changes this row to value i_2, resulting in the state <1, 1>$_{i1}$<0, 1>$_{i2}$.
- **A3**: A merge happens on the VB i_2, resulting in the state <1, 1>$_{i1}$<1, 0>$_{i2}$.
- **A4**: A subsequent update changes this row to any values except i_1, denoted as i_3, resulting in a HUD with three 1s: <1, 1>$_{i1}$<1, 1>$_{i2}$<0, 1>$_{i3}$.
- This resulting HUD is <R, 3, 1, 2, 3>.

In summary, if (a) updates always change a row to new values, (b) update and merge operations happen alternatively and each merge always happens on the new value of the preceding update, and (c) no deletes happen, the number of 1s in a HUD can grow. Assume that there is no delete and that updates and merges happen uniformly on all possible values. The probability of A1 is $1/c$, where c is the cardinality, the probability of A2 is $(c-1)/c$, and the probability of A3 and A4 are $1/c$ and $(c-2)/c$, respectively. Overall, a HUD will contain n 1s with a probability less than $1/c^{n-1}$. For example, when $c = 128$, the probability that a HUD contains seven 1s is $1/128^6 = 1/2^{42}$, which happens extremely unlikely.

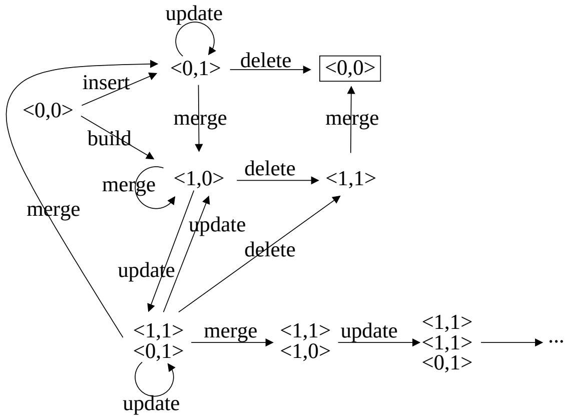

Figure 1: Finite-State Machine of the HUDs of a row by only recording the $<U_i, V_i>$ pairs that contain 1s. Except for the bottom-right state, the number of 1s recorded in a HUD is zero to two. The only sequence of operations that increases the number of 1s in HUDs is a sequence of interleaved update and merge operations, which takes place with low probability.

D EXISTING UPDATABLE BITMAP INDEXES

In-place. The most straightforward approach, denoted *In-place* [8], directly updates the underlying bit-matrix. In order to update the k^{th} row from value v_1 to v_2, *In-place* applies the *decode-flip-encode* procedure on both bitvectors of v_1 and v_2. To delete the k^{th} row, *In-place* applies the same procedure on the bitvector of the old value and sets its k^{th} bit from 1 to 0. An insert operation appends bit 1 at the tail of the bitvector of the corresponding value, and then appends bit 0 to the others. *In-place*'s inferior performance comes from the time-consuming decode-flip-encode procedure.

UCB. To alleviate the performance issue of In-place, UCB [4] introduces an extra bitvector, denoted *existence bitvector* (EB), that indicates whether a given row is valid or not. Initially, all bits in EB are 1s. A delete operation is performed by setting the corresponding bit in EB to 0. An insert operation appends a 1 to the tail of EB, and increments the global variable *N_ROWS* that indicates the number of rows in UCB. An update operation is transformed to delete-then-append operations; that is, the new value is appended at the tail of the bitvector, and a mapping between the invalidated row ID and the new-appended row ID is kept. By avoiding decoding and then encoding the value bitvectors, UDIs of UCB are supposed to be more efficient than In-place. The efficiency of UCB is predicated on EB being highly compressible. In practice, however, its performance deteriorates sharply as the total number of UDIs performed increases and EB becomes less compressible [2]. Meanwhile, UCB does not provide a standard *set* abstract data type, and programmers must maintain an additional level of indirection to map each row ID from users' perspective to UCB's, which incurs considerable performance penalties as the number of updates increases.

UpBit. To address the above-discussed issues, the state-of-the-art solution, UpBit [2], associates an additional *update bitvector* (UB) with each *value bitvector* (VB) in the domain of the indexed attribute. UBs keep track of updates to VBs; that is, UDIs flip bits in UBs that are merged back to VBs in a lazy and batch manner. Therefore, UBs are highly compressible, resulting in reduced decode-flip-encode overheads.

E PARALLELIZING BITMAP INDEXES

UpBit. We parallelize UpBit, the state-of-the-art updatable bitmap index, by using a fine-grained locking mechanism. Specifically, the <VB, UB> pair of every value v is protected by a reader-writer latch, denoted $latch_v$. Global variables like N_ROWS are protected by a global latch $latch_g$. Update and delete operations first acquire the $latch_v$ of all values in shared mode to retrieve the current value of the specified row. Then, they upgrade $latch_v$ of the corresponding bitvectors to exclusive mode in order to flip the necessary bits. An insert operation acquires $latch_g$ and the corresponding $latch_v$ in exclusive mode. Consequently, a query operation acquires $latch_g$ and the corresponding $latch_v$ in shared mode.

UCB. UCB's UDIs update the only EB, and queries read this EB simultaneously. Therefore, we parallelize UCB by using a global reader-writer latch to serialize concurrent queries and UDIs. An insert operation holds this latch before updating the global variable N_ROWS.

In-place. One way to parallelize In-place is to use fine-grained reader-writer latches, the same as in UpBit. However, with this mechanism, an insert operation needs to acquire $cardinality$ latches before appending bits to the tail of all the VBs, dramatically reducing the overall throughput. Therefore, we parallelize In-place by using a global reader-writer latch, the same as for UCB. Surprisingly, the parallelized In-place outperforms UCB for high concurrency (see the evaluation results in our paper).

F TPC-H

We implemented a TPC-H workload with batched updates on DBx1000. In particular, we implemented the New Sales Refresh Function (RF1) and Old Sales Refresh Function (RF2) transactions that modify the dataset and indexes in batch mode to keep the dataset fresh. With prior bitmap indexes, these two classes of transactions must be performed in a scheduled time period, during which indexes are unavailable. In contrast, CUBIT does not introduce any maintenance downtime.

Dataset. The 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].

Workloads. We use the Forecasting Revenue Change Query (Q6) as the representative selective query. The SQL code for Q6 is listed in Algorithm 1. 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].

We assign a worker thread, termed *RF Thread*, to execute New Sales Refresh Function (RF1) and Old Sales Refresh Function (RF2) transactions periodically. Other worker threads perform analytical transactions simultaneously. For each run, the RF thread invokes RF1 or RF2, which modifies 1,500 tuples in the table *ORDERS* and the corresponding about ~4,500 tuples in the table *LINEITEM*, and then updates the indexes, in batch. After each run, the RF thread waits until the ratio of the overall workload of Q6 to that of RF1/2 reaches 98:2, before starting the next RF transaction. The SQL code for RF1 and RF2 is listed in Algorithms 2 and 3.

Algorithm 1: 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];
```

Algorithm 2: TPC-H RF1.

```
LOOP 1,500 times
  INSERT a new row into the ORDERS table
  LOOP random[1, 7] times
    INSERT a new row into the LINEITEM table
  END LOOP
END LOOP
```

Algorithm 3: TPC-H RF2.

```
LOOP 1,500 times
  DELETE from ORDERS where o_orderkey = [VALUE]
  DELETE from LINEITEM where l_orderkey = [VALUE]
END LOOP
```

CUBIT Instances. The DBMS creates three CUBIT instances, respectively on the attributes *l_shipdate*, *l_discount*, and *l_quantity*. 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 of $\frac{1}{7} \times \frac{3}{11} \times \frac{24.5}{50} \approx 2\%$. 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 allocates a new bitvector and then performs bitwise OR/AND operations among 5 (1+3+1) or 6 (1+3+2) bitvectors to get the resulting bitvector. It then fetches the matching tuples to calculate the final revenue result.

G CH-BENCHMARK

We implemented the CH-benCHmark [5] that consists of a full version of the TPC-C benchmark and a set of TPC-H-equivalent analytical queries on the same tables.

Selectivity. In our evaluation, we found that many attributes in CH-benCHmark cover a narrow scope, such that the queries unreasonably select almost all of the tuples. We thus modified the propagated values and the query predicates to provide a reasonable selectivity. For example, we set the values of the *ol_delivery_d* attribute in the *ORDER-LINE* table in the range of [1983, 2023), and the values of the *ol_quantity* attribute in the range of [1, 25000), both in a uniform distribution. As a consequence, each CH-benCHmark Q1 selects rows on years (16 out of 40) and delivery state (9 out of 10), leading to an average selectivity of $\frac{16}{40} \times \frac{9}{10} \approx 36\%$, and each Q6 selects rows on years (20 out of 40), quantities (1000 out of 25,000), and delivery state (9 out of 10), leading to an average selectivity of $\frac{20}{40} \times \frac{1}{25} \times \frac{9}{10} \approx 1.8\%$. The SQL code for the Q1 and Q6 of CH-benCHmark are listed in Algorithms 4 and 5.

Algorithm 4: CH-benCHmark Q1.

```
SELECT ol_number,
        sum(ol_quantity) as sum_qty,
        sum(ol_amount) as sum_amount,
        avg(ol_quantity) as avg_qty,
        avg(ol_amount) as avg_amount,
        count(*) as count_order
FROM      orderline
WHERE     ol_delivery_d > '2007-01-02 00:00:00.000000'
GROUP BY  ol_number ORDER BY ol_number
```

Algorithm 5: CH-benCHmark Q6.

```
SELECT sum(ol_amount) as revenue
FROM orderline
WHERE ol_delivery_d >= '1999-01-01 00:00:00.000000'
        and ol_delivery_d < '2020-01-01 00:00:00.000000'
        and ol_quantity between 1 and 1000
```

REFERENCES

[1] Maya Arbel-Raviv and Trevor Brown. 2018. Harnessing epoch-based reclamation for efficient range queries. *Proceedings of the 23rd ACM SIGPLAN Symposium on PPoPP* (2018).

[2] Manos Athanassoulis, Zheng Yan, and Stratos Idreos. 2016. UpBit: Scalable In-Memory Updatable Bitmap Indexing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. https://dl.acm.org/citation.cfm?id=2915964

[3] Trevor Alexander Brown, William Sigouin, and Dan Alistarh. 2022. PathCAS: an efficient middle ground for concurrent search data structures. *Proceedings of the 27th ACM SIGPLAN Symposium on PPoPP* (2022).

[4] Guadalupe Canahuate, Michael Gibas, and Hakan Ferhatosmanoglu. 2007. Update Conscious Bitmap Indices. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*. 15–25. https://doi.org/10.1109/SSDBM.2007.24

[5] Richard L. Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi A. Kuno, Raghunath Othayoth Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. 2011. The mixed workload CH-benCHmark. In *Proceedings of the International Workshop on Testing Database Systems (DBTest)*. 8. https://doi.org/10.1145/1988842.1988850

[6] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[7] Maged M. Michael and Michael L. Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96*.

[8] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. 2020. *Database System Concepts, Seventh Edition*.