

## APPENDIX TO THE CUBIT PAPER

### A PROGRAMMING SEMANTICS OF CUBIT

CUBIT complies with the classic specification for secondary indexes, 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-1k.** CUBIT-1k employs a read-write latch to serialize concurrent UDIs to the global Delta Log. This design stems from the classic MVCC mechanisms in which latches are used to prevent concurrent updates from updating 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.

**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, suspended UDIs help the other make progress until complete, and then retry. The resulting algorithm, denoted as CUBIT-1f, provides non-blocking (latch-free) UDIs that never block each other, resolving the major cause of unexpectedly-long tail latency of UDIs.

**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.

**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*<sub>1</sub> 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*<sub>2</sub>, which failed to append its ULE, first helps *O*<sub>1</sub> complete. Specifically, for each variable to be updated, *O*<sub>2</sub> 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*<sub>1</sub> or other helpers, *O*<sub>2</sub> simply skips updating this variable. After helping update all variables in *O*<sub>1</sub>’s ULE, *O*<sub>2</sub> starts over.

**Correctness.** CUBIT-1f is a concurrent data structure (set) [6] and we can prove its correctness from the following aspects.

**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-1f 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-1f 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-1f 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-If provide wait-free queries and lock-free UDIs (i.e., they never block).

## C SIZE OF HUDS

In general case, a HUD has only 0s. As UDIs accumulate, the number of 1s in the 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.

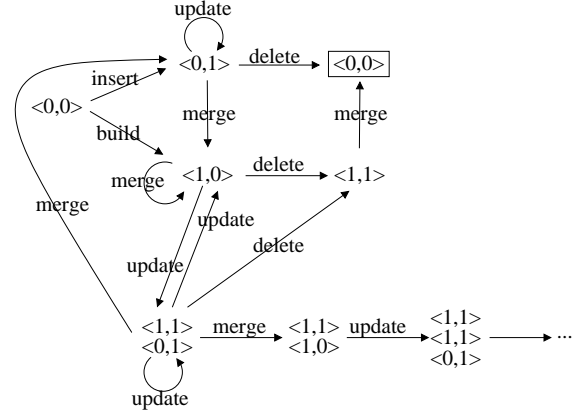
**FSM of HUDs.** Conceptually, a HUD is a bit-string with a length equal to the cardinality of the domain, and the  $i^{th}$  bit in this bitvector, denoted  $R_i$ , is associated with the corresponding bit of the  $i^{th}$  VB, denoted  $V_i$ . We study the transition of the HUD by using a Finite-State Machine (FSM), in which each node records the  $\langle R_i, V_i \rangle$  pairs for all possible  $i$ , denoted  $\langle R, V \rangle_i$ . 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  $\langle 0, 0 \rangle$  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  $\langle 0, 0 \rangle$  to  $\langle 0, 1 \rangle$ , indicating that the corresponding bit of the HUD has been set to 1. An update may change a HUD from ' $\langle 0, 0 \rangle, \langle 0, 1 \rangle$ ' to ' $\langle 0, 1 \rangle, \langle 0, 0 \rangle$ ', 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  $\langle 0, 0 \rangle$  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 the HUDs of a row is zero to two with high probability.

(2) The only operation sequence that increases the number of 1s of a HUD is as follows.

- **A1:** A merge happens on the VB  $i_1$ , resulting in the state  $\langle 1, 0 \rangle_{i_1}$ .
- **A2:** An update changes this row to any possible VBs except  $i_1$ , denoted as  $i_2$ , resulting in the state  $\langle 1, 1 \rangle_{i_2} \langle 0, 1 \rangle_{i_1}$ .
- **A3:** A merge happens on the VB  $i_2$ , resulting in the state  $\langle 1, 1 \rangle_{i_1} \langle 1, 0 \rangle_{i_2}$ .
- **A4:** Finally, another update changes this row to any VBs except  $i_1$  and  $i_2$ , denoted as  $i_3$ , resulting in a HUD with three 1s:  $\langle 1, 1 \rangle_{i_1} \langle 1, 1 \rangle_{i_2} \langle 0, 1 \rangle_{i_3}$ .
- This HUD is now stored as  $\langle R, 3, 1, 2, 3 \rangle$ .

That is, if (a) updates always change a row to new values, (b) updates and merges 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 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  $\langle R_i, V_i \rangle$  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].

**UpBit.** To address the above-discussed issues, the state-of-the-art solution, UpBit [2], maintains a *value bitvector* (VB) and an extra *update bitvector* (UB) for every value 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. UBs are highly compressible, resulting in reduced decode-flip-encode

overheads. Further details can be found in Section Background of the CUBIT paper and the original UpBit paper [2].

## 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  $\langle VB, UB \rangle$  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 synchronize concurrent queries and UDIs. Note that 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 use the TPC-H benchmark in the evaluation. We now present the details on the experimental setup for running CUBIT over the TPC-H benchmark.

**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].

---

### Algorithm 1: TPC-H Q6.

---

```

1 SELECT sum(l_extensdeprice × l_discount) as revenue
2 FROM LINEITEM
3 WHERE l_shipdate >= date'[DATE]'
4   and l_shipdate < date'[DATE]' + interval '1' year
5   and l_discount between [DISCOUNT] ± 0.01
6   and l_quantity < [QUANTITY];

```

---

We use New Sales Refresh Function (RF1) and Old Sales Refresh Function (RF2) as the workload of updates. Each RF1 and RF2 modifies 1,500 tuples in the table *ORDERS* and the corresponding about 4,500 tuples in the table *LINEITEM*, and then updates the

---

### Algorithm 2: TPC-H RF1.

---

```

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

```

---



---

### Algorithm 3: TPC-H RF2.

---

```

11 DELETE from ORDERS where o_orderkey = [VALUE]
12 DELETE from LINEITEM where l_orderkey = [VALUE]

```

---

three CUBIT indexes accordingly, all in one transaction. The SQL code for RF1 and RF2 is listed in Algorithms 2 and 3. According to the TPC-H specification, the operation distribution of each worker thread is set to 98, 1, and 1 for Q6, RF1, and RF2, respectively.

**CUBIT Instances.** 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 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 makes 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.

## 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.

---

```
13 SELECT ol_number,
14        sum(ol_quantity) as sum_qty,
15        sum(ol_amount) as sum_amount,
16        avg(ol_quantity) as avg_qty,
17        avg(ol_amount) as avg_amount,
18        count(*) as count_order
19 FROM   orderline
20 WHERE  ol_delivery_d > '2007-01-02 00:00:00.000000'
21 GROUP BY ol_number ORDER BY ol_number
```

---

---

**Algorithm 5:** CH-benCHmark Q6.

---

```
22 SELECT sum(ol_amount) as revenue
23 FROM orderline
24 WHERE ol_delivery_d >= '1999-01-01 00:00:00.000000'
25        and ol_delivery_d < '2020-01-01 00:00:00.000000'
26        and ol_quantity between 1 and 100000
```

---

## 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*.