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 [9], 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* [8], 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, 4]. 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 [8].

Immune to ABA problem. In theory, the ABA problem [8] 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.

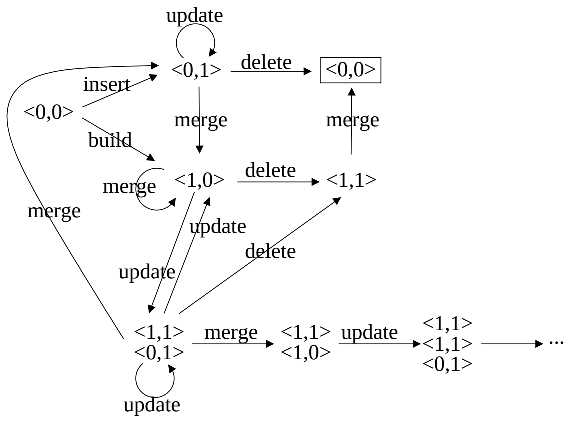


Figure 1: Finite-State Machine of the HUDs of a row by only recording the $<U_i, V_i>$ pairs that contain 1s. Except for the bottom-right state, the number of 1s recorded in a HUD is zero to two. The only sequence of operations that increases the number of 1s in HUDs is a sequence of interleaved update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

D EXISTING UPDATABLE BITMAP INDEXES

In-place. The most straightforward approach, denoted *In-place* [10], 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 [5] 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 integrated CUBIT into DuckDB and implemented 12 of the 22 TPC-H queries, along with the New Sales Refresh Function (RF1) and Old Sales Refresh Function (RF2) transactions. We optimized as many choke points in DuckDB's query engine as possible [3, 7]. For example, in TPC-H Q1, we represented the group-by expression as small integers within a narrow range (instead of using a String class) and utilized an array (rather than a hash table) to store aggregation statistics, which improved query performance by 7%. Additionally, by excluding certain group-by attributes that can be derived from the primary key, we achieved a 46% performance improvement in Q10. We refer to this optimized version of DuckDB as DuckDB$^+$ and use it as the baseline.

In our experiments, we set the Scale Factor (SF) of our workload to 10. *Note that in this section, we present the pseudocode for TPC-H queries for reader's reference; they are identical to those listed in the TPC-H specification.*

Refresh Workloads. The pseudocode for RF1 and RF2 is presented in Algorithms 1 and 2. They update the LINEITEM and ORDERS tables along with the associated indexes. Because of their update-friendly nature, CUBIT instances on attributes of these two tables do not introduce any maintenance downtime when RF1 and RF2 are executed in our experiments. In contrast, with other bitmap indexes, RF1 and RF2 must be performed during a scheduled time period, during which indexes are unavailable.

Algorithm 1: 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 2: TPC-H RF2.

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

We assign a worker thread, termed *RF Thread*, to execute RF1 and RF2 periodically, while other worker threads concurrently perform query transactions. During each run, the RF thread invokes RF1 or RF2, modifying 1,500 tuples in the ORDERS table and approximately ~4,500 tuples in the LINEITEM table, and then updates the indexes, in batch. After each run, the RF thread waits until the ratio of the overall workload of queries to that of refreshes reaches 98:2, before starting the next RF transaction.

Algorithm 3: TPC-H Q1.

```
SELECT
  l_returnflag,
  l_linestatus,
  sum(l_quantity) as sum_qty,
  sum(l_extendedprice) as sum_base_price,
  sum(l_extendedprice*(1-l_discount)) as sum_disc_price,
  sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge,
  avg(l_quantity) as avg_qty,
  avg(l_extendedprice) as avg_price,
  avg(l_discount) as avg_disc,
  count(*) as count_order
FROM
  LIMEITEM
WHERE
  l_shipdate <= date '1998-12-01' - interval '[DELTA]' day (3)
GROUP BY
  l_returnflag,
  l_linestatus
ORDER BY
  l_returnflag,
  l_linestatus
```

Q1: The TPC-H Q1 (Algorithm 3) generates a summary pricing report for all lineitems shipped as of a given date. The value of the parameter DELTA is randomly selected within the range of [60, 120]. The results (price aggregations) are grouped by *l_returnflag* and *l_linestatus* and then listed in ascending order.

To answer Q1, DMBSs like DuckDB typically (1) scan the LINEITEM table and filter out tuples shipped after the specified date, and (2) group the remaining tuples by *l_returnflag* and *l_linestatus* using perfect hashing to generate price aggregations (SUM, AVG, and COUNT). These two steps takes ~770 and ~780ms, respectively, accounting for 97% of the query latency. Our evaluation results show that perfect hashing is the main performance bottleneck. The executor sequentially iterates over all tuples, calculating the group-by category for each and updating the corresponding statistic counters, inevitably leading to branch misses and cache misses.

By maintaining CUBIT instances on the attributes used as group-by factors (*l_returnflag* and *l_linestatus* for Q1), we implemented a new aggregation executor. Our executor first determines the positions of matching tuples for each group-by category by *AND*ing bitvectors from CUBIT instances, and then calculates the aggregations for each category by reading the specified entries—a computation mode amenable to modern SIMD instructions. Additionally, CUBIT instances shield the query engine from scanning the indexed columns.

For Q1, we create three CUBIT instances on the attributes *l_returnflag* (cardinality = 3) and *l_linestatus* (cardinality = 2), and *l_shipdate* (cardinality = 2,526). The new aggregation executor consists of the following steps.

(1) It performs a logical *NOT* on the resulting bitvector of logical *OR*s between bitvectors in the range of [date '1998-12-01' - interval '[DELTA]' day, ENDDATE], generating the resulting bitvector btv_{date} for the *l_shipdate* predicate.
(2) Each group-by category corresponds to a possible bitvector pair (r, l) where r and l are respectively from the CUBIT instances on the attributes *l_returnflag* and *l_linestatus*. The resulting bitvector btv_{gb} of each category is computed by *AND*ing r, l, and btv_{date}.
(3) The executor performs a sequential scan over the necessary columns. For each row group, it aggregates data for each group-by category by reading the specified tuples, whose positions have been recorded in each btv_{gb}. Since the data array fetched and each btv_{gb} naturally fit the "operand" and "mask" registers of AVX-512 instructions, this step is accelerated by AVX-512 instructions, significantly reducing branch misses compared to the standard aggregation operators.

Experimental results show that the (1) and (2) stages together takes 45ms, and the (3) stage takes 315ms. The overall query latency of Q1 using the CUBIT-powered query engine is 604ms, 2.7× faster than DuckDB's native implementation.

Q5: TPC-H Q5 (Algorithm 4) lists the revenue generated through local suppliers. *REGION* is randomly selected within the list of values defined for R_NAME, and *DATE* is a randomly selected year within [1993, 1997].

To execute Q5, DBMSs like DuckDB typically (1) scan the *LINEITEM* and *ORDERS* tables, and (2) join the two tables on the *orderkey* attribute by using a hash table, which is time-consuming. When SF = 10, the (1) and (2) stages respectively take 340ms and 750ms, in total accounting for 83% of the execution time of Q5. Note that we have implemented a BTree-index-based join executor by pre-building Tree based indexes on the *l_orderkey* attribute, which, however, performed worse than DuckDB's native join operator for large workloads; when SF = 10, it is 1.4× slower.

By maintaining a CUBIT instance on *l_orderkey*, we implemented a new join executor that eliminates the need for the query engine to build and query the hash table, the most time-consuming stage in DuckDB's native executor.

Specifically, for Q5, we create a CUBIT instance on the *l_orderkey* attribute of the LINEITEM table, consisting of ~15 million bitvectors (when SF = 10). For the join operation, our executor retrieves the *orderkey* set from the ORDERS table, reads the corresponding bitvectors from the CUBIT instance, and performs logical *OR*s between them. Using the resulting bitvector, the executor scans the LINEITEM table and performs aggregations in one pass. Note that when SF = 10, the *orderkey* set contains ~357 thousand distinct values and our executor performs ~357 thousand logical *OR*s between bitvectors. Since each bitvector only contains ~4 1s and is thus highly compressible (~16-byte long), our merging mechanism can efficiently merge them together.

Our evaluation shows that *OR*ing the ~357 thousand bitvectors takes 534ms, which is 1.6× more costly than scanning the attribute in DuckDB's native executor. Nevertheless, our resulting bitvector eliminates the need to build and query the hash table (~750ms), making the overall query latency with our CUBIT-powered executor 1.3× faster than DuckDB's native implementation.

Algorithm 4: TPC-H Q5.

```
SELECT
    n_name,
    sum(l_extendedprice * (1 - l_discount)) as revenue
FROM
    CUSTOMER,
    ORDERS,
    LINEITEM,
    SUPPLIER,
    NATION,
    REGION
WHERE
    c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and l_suppkey = s_suppkey
    and c_nationkey = s_nationkey
    and s_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and r_name = '[REGION]'
    and o_orderdate >= date '[DATE]'
    and o_orderdate < date '[DATE]' + interval '1' year
GROUP BY
    n_name
ORDER BY
    revenue desc;
```

Q6: TPC-H Q6 (presented in Algorithm 5) reads the fact table, LINEITEM, and quantifies how discount settings affect the revenue in a given year. 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].

To execute Q6, most column-store DBMSs (including DuckDB) scan four columns (i.e., *l_extendedprice*, *l_shipdate*, *l_discount*, and *l_quantity*), filter out mismatched tuples, and then calculate the revenue. The scanning stage accounting for 95% of the overall execution time.

By maintaining CUBIT instances on the attributes involved in the WHERE clauses, we implemented an indexing-based scan executor that reduces the amount of workload of scan.

Specifically, we create three CUBIT instances, respectively on the attributes *l_shipdate* (cardinality = 2,526), *l_discount* (cardinality = 11), and *l_quantity* (cardinality = 50). Each Q6 selects the bitvectors corresponding to 365 of the 2,526 days, 3 of the 11 possible discounts, and 24 or 25 of 50 possible quantities, leading to an average selectivity of $\frac{365}{2,526} \times \frac{3}{11} \times \frac{24.5}{50} \approx 2\%$. To execute Q6, our executor performs bitwise OR/AND operations among 392 (365+3+24) or 393 (365+3+25) bitvectors to get the resulting bitvector, which is then used to probe the *l_extendedprice* and *l_discount* columns to calculate the final revenue, effectively halving the data read from hard drives.

The Q6 query latency of the DuckDB with our CUBIT-powered scan executor is 273ms, 2.1× faster than DuckDB's native approach.

Algorithm 5: 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];
```

Q3: The primary choke points in executing Q3 are (1) the join operation between the LINEITEM fact table and ORDERS, and (2) the scan and filter operation on the *l_shipdate* column. Similar to our approach to Q1, we maintain two CUBIT instances for the attributes *l_orderkey* (cardinality = 15M) and *l_shipdate* (cardinality = 2,526), utilizing our new join and indexing-based scan executors. Evaluation results show that CUBIT-powered DuckDB executes Q3 in 1081ms, making it 1.2× faster than DuckDB's native approach.

Q4: The primary choke points in executing Q4 is the join operation between the LINEITEM fact table and ORDERS. Similar to our approach to Q5, we maintain a CUBIT instance for the attributes *l_orderkey*, utilizing our new join executor. Evaluation results show that CUBIT-powered DuckDB executes Q4 in 798ms, making it 1.4× faster than DuckDB's native approach.

Q10: We build CUBIT instances on the *l_orderkey*, *l_returnflag* attributes. Evaluation results show that CUBIT-powered DuckDB executes Q10 in 875ms, making it 1.4× faster than DuckDB's native approach.

Q12: We build a CUBIT instance on the *l_shipmode* and *l_receiptdate* attributes. Evaluation results show that CUBIT-powered DuckDB executes Q14 in —ms, making it − − −× faster than DuckDB's native approach.

Q14: We build a CUBIT instance on the *l_shipdate* attribute. Evaluation results show that CUBIT-powered DuckDB executes Q14 in 438ms, making it 2.1× faster than DuckDB's native approach.

Q15: We build a CUBIT instance on the *l_shipdate* attribute. Evaluation results show that CUBIT-powered DuckDB executes Q15 in 962ms, making it 1.4× faster than DuckDB's native approach.

Q17: We build a CUBIT instance on the *l_partkey* attribute (cardinality = 2M). Evaluation results show that CUBIT-powered DuckDB executes Q15 in 962ms, making it 1.5× faster than DuckDB's native approach.

Q18: We build a CUBIT instance on the *l_orderkey* and *o_orderkey* attributes. Evaluation results show that CUBIT-powered DuckDB executes Q18 in 2860ms, making it 2.7× faster than DuckDB's native approach.

Q19: We create dictionaries for the attributes *l_shipmode* and *l_shipinstruct*, and then build CUBIT instances for the encoded reference arrays, similarly to that in DuckDB. Our query engine uses the CUBIT instances and consult the dictionaries if necessary. Evaluation results show that CUBIT-powered DuckDB executes Q19 in 909ms, making it 1.5× faster than DuckDB's native approach.

G CH-BENCHMARK

We implemented the CH-benCHmark [6] 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 6 and 7.

Algorithm 6: 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 7: 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
```

H ADDITIONAL EVALUATION ON SENSITIVITY ANALYSIS

Impact of Data Size. As the dataset size increases, the relative behavior of different indexes remains the same. Figure 2a shows the evaluation results with datasets containing 1B entries (cardinality = 100). Figure 2a looks almost identical to Figure **??**, which

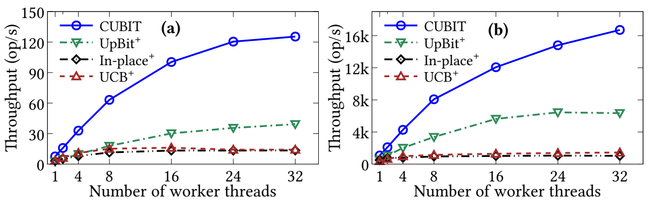


Figure 2: (a) When increasing the dataset size (1B entries), and (b) when querying real datasets (Berkeley Earth dataset with 31M tuples and cardinality of 114), the relative behavior of all approaches remains the same.

demonstrates that data size does not affect the performance trends and the relative behavior of different algorithms. In this evaluation, however, each bitvector contains more bits. Therefore, the absolute performance decreases nearly linearly as the dataset size increases.

Berkeley Earth Dataset. For a real-life application, we evaluate CUBIT and its competitors using the Berkeley Earth dataset. It is an open dataset for a climate study that contains measurements from 1.6 billion temperature reports, each of which contains information including temperature, time of measurement, and location. From the Berkeley Earth data, we extract a dataset containing 31 million entries with cardinality 144. Figure 2b shows that with 32 worker threads, CUBIT's throughput is about 2.6×, 11.5×, and 16.2× higher than that of UpBit$^+$, UCB$^+$, and In-place$^+$, respectively.

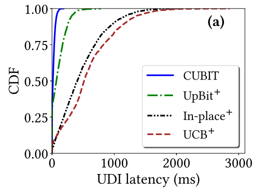
I IMPACT OF DATA SKEW

The distribution of data among bitvectors plays a key role in performance for two reasons. First, biased distributions may lead to few *target* bitvectors containing many more 1s than others, making them less compressible. Second, the target bitvectors face higher contention levels among concurrent UDIs. We thus evaluate the impact of a skewed distribution. We use the same configuration with the highest concurrency level (32 worker threads). The dataset follows the Zipfian distribution with the skew parameter α being set to 1.5, which implies that about 40% of the entries have the two most popular values, and the remaining are uniformly distributed in the entire value domain. We make the following observations.

All Bitmap Indexes Have 2× Faster Queries for Skewed Data. The overall throughput and mean query latency of *all indexes* are improved by about 2×, compared to the case with uniform data distribution. *The reason is that most bitvectors contain few 1s, and are thus highly compressible.* Queries on these bitvectors are very fast. We omit the figures because they have the same trends as Figure **??**.

CUBIT Has Stable UDI Latency for Skewed Data. For all approaches, UDI latency increases for skewed data because 40% of the UDIs involve a few bitvectors, leading to high contention on them. However, CUBIT remains the most stable design (as shown in Figure 3a). Note that UCB$^+$ performance deterioration depends on the total number of UDIs performed; thus, we omit it in the remainder of the analysis.

To better understand the tail UDI latency, we zoom in on Figure 3a and list the corresponding statistics (*-zipf*) in Figure 3b. We also list the statistics of the same experiments with uniform distributions (*-unif*) for comparison. We make the following observations. (A) First, *UpBit$^+$ significantly reduces UDI's tail latency when compared to In-place$^+$ because of its fine-grained locking mechanism.* Figure 3b shows that the P99999 UDI latency of UpBit$^+$ is 2.6× smaller than that of In-place$^+$. (B) Second, *the fine-grained read-write locking mechanism employed by UpBit$^+$ cannot alleviate the contention on the target bitvectors. In contrast, CUBIT's lightweight UDIs reduce contention among hot bitvectors.* For example, the P99999 latency of basic CUBIT is about 4.2× smaller than that of UpBit$^+$. (C) Third, the *helping mechanism* employed by CUBIT-lf reduces the number of latches acquired, further reducing CUBIT's P99 UDI latency by 31% and P99999 UDI latency by 17%.



Index Approach	Median	P99	P99999
CUBIT-lf-unif	11	68	152
CUBIT-lf-zipf	15	76	**153**
CUBIT-unif	11	71	127
CUBIT-zipf	15	111	**185**
UpBit-unif	52	430	656
UpBit-zipf	71	457	**785**
In-place-unif	214	1,008	1,882
In-place-zipf	407	1,523	**2,009**

Figure 3: (a) CUBIT's UDI latency is not affected by data skew (Zipfian, α=1.5), and (table) has superior tail latency.

J CUBIT FOR OLTP

TPC-C. We reuse the DBx1000 framework to implement a full-blown TPC-C benchmark to test CUBIT for a pure transactional workload. We compare the performance of Stock-Level (SL) transactions with a B$^+$-Tree and CUBIT on the *Quantity* attribute. In our evaluation, *#Warehouse* = 100, and the *Stock* table contains about 10M tuples. We make the following two observations.

Update Friendly. Despite that other transactions in TPC-C (e.g., *New-Order*) heavily update the *Quantity* attribute and the associated CUBIT indexes, CUBIT-assisted SL is competitive to other indexes, demonstrating that CUBIT is update-friendly and does not introduce noticeable maintenance overhead.

CUBIT Brings Limited Performance Gains to OLTP. CUBIT-powered DBMS reduces the response time of SL from 0.55ms to 0.54ms by assigning 8 cores to each query. The performance gain is mainly because querying CUBIT is easier to parallelize. However, the improvement is not noticeable because queries in OLTP are very selective (e.g., each SL yields about 200 matching entries), such that other indexes like B$^+$-Tree also perform well. Overall, our conclusion is that CUBIT can be used in OLTP DBMSs without incurring performance penalties, but it is a better fit for OLAP and HTAP with inherent moderate selectivity.

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] Peter A. Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Proceedings of the TPC Technology Conference on Performance Evaluation, Measurement and Characterization of Complex Systems (TPCTC)*. http://link.springer.com/chapter/10.1007%2F978-3-319-04936-6_5

[4] 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).

[5] 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

[6] 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

[7] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H choke points and their optimizations. *Proceedings of the VLDB Endowment* 13 (2020), 1206 – 1220.

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

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

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