Db

wu

April 11, 2024

Contents

1 General

1.1 Databases and Finite-Model Theory

In a very real sense, finite-model theory provides the backbone of database theory. And databases provide a concrete scenario for finite-model theory.

The overlap of database theory with nite-model theory occurs primarily in the area of query languages.

Shit paper

2 Storage

2.1 LSM-based Storage Techniques: A Survey

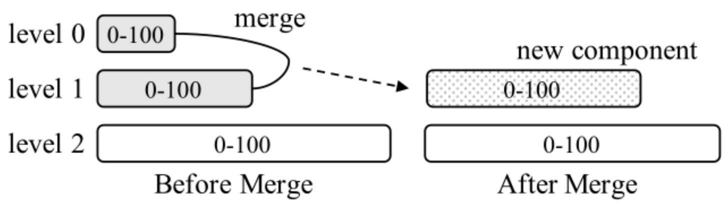
`https://doi.org/10.1007/s00778-019-00555-y` LSM-tree: bigtable, dynamo, hbase, cassandra, leveldb, rocksdb, asterixdb

2.1.1 LSM-tree basics

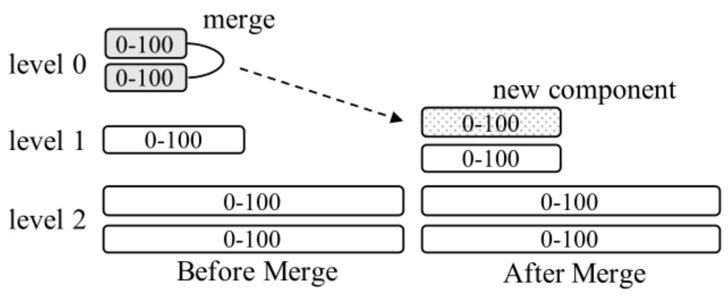
1. Basic Structure

   - Memory component: concurrent data structure, skip-list/B^+-tree
   - Disk component, SSTables: a data block stores key-value pairs ordered by keys, and the index blocks stores the key ranges of all data blocks

   Two types of merge policies:



(a) Leveling Merge Policy: one component per level



(b) Tiering Merge Policy: up to T components per level

Figure 1: LSM-tree merge policies

   - **Leveling merge policy**: each level only maintains one component, but the component at level L is T times larger than the component at level $L-1$. As a result, the component at level L will be merged multiple times with incoming component at level $L-1$ until it fills up, and it will then be merged into level $L+1$, and it will then be merged into level $L+1$.

     Better query performance.

- **Tiering merge policy**: maintains up to T components per level. When level L is full, its T components are merged together into a new component at level $L+1$.

  Better write performance.

2. Well-Known Optimizations **Bloom filter**:

   - built on top of disk components.
   - built for each leaf page for a disk component: a point lookup can first search the non-leaf pages of a B^+-tree to locate the leaf page, where the non-leaf pages are assumed to be small enough to be cached, and then check the associated Bloom filter before fetching the leaf page.

   The false positive rate of a Bloom filter is

   $$\left(1 - e^{-kn}/m\right)^k$$

   where k is the number of hash functions, n is the number of keys, and m is the total number of bits. And the optimal number of hash functions that minimizes the false positive rate is

   $$k = \frac{m}{n} \ln 2$$

   In practice, most systems typically use 10 bits/key as a default configuration, which gives a 1% false positive rate.

   Partitioning: Partitioning is orthogonal to merge policies, both leveling and tiering can be adapted to supported partitioning. But only the partitioned leveling policy has been fully implemented.

   In the partitioned leveling merge policy, pioneered by LevelDB, the disk component at each level is range partitioned into multiple fixed-size SSTables, as in figure 2

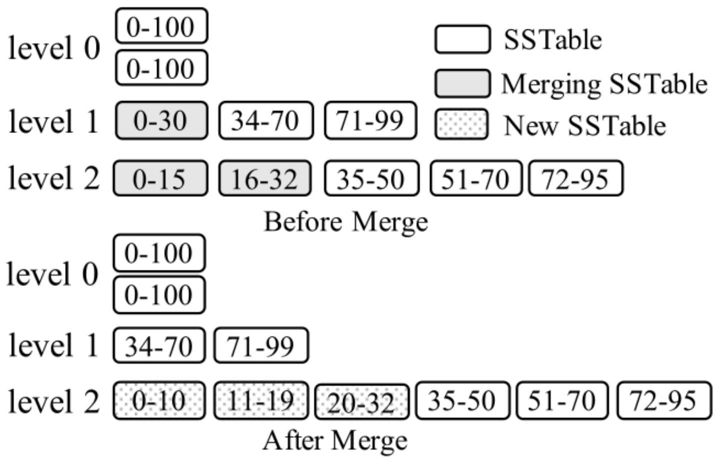


Figure 2: Partitioned leveling merge policy

Each SSTable is labeled with its key range in the figure. To merge an SSTable from level L into level $L + 1$, all of its overlapping SSTable at level $L + 1$ are selected, and these SSTables are merged with it to produce new SSTables still at level $L+1$. Different policies can be used to select which SSTable to merge next at each level.

The partitioned optimization can also be applied to the tiering merge policy. However, one major issue in doing so is that each level can contain multiple SSTables with overlapping key ranges. Two possible schemes can be used to organize the SSTables at each level

(a) **Vertical grouping**: groups SSTables with overlapping key ranges together so that the groups have disjoint key ranges

(b) **Horizontal grouping**: each logical disk component, which is range-partitioned into a set of SSTables, serves as a group directly

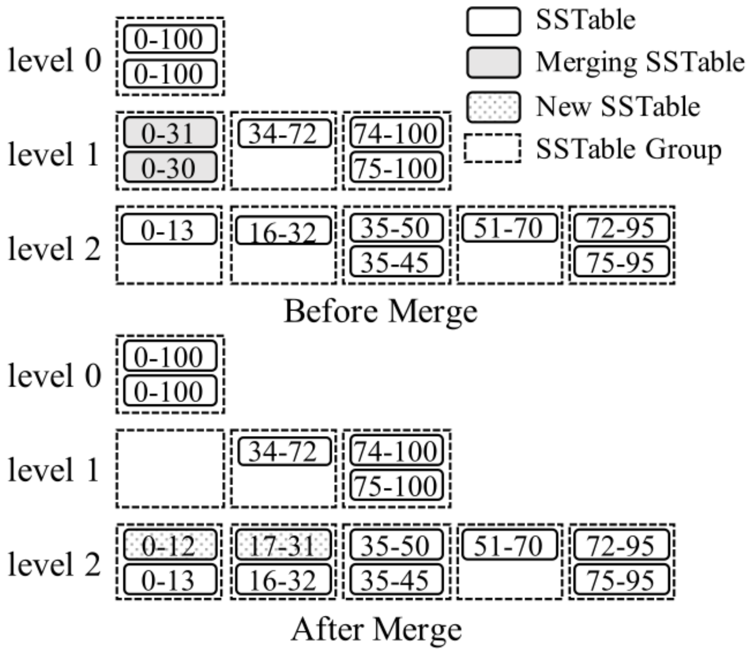


Figure 3: Partitioned tiering with vertical grouping

During a merge operation, all of the SSTables in a group are merged together to produce the resulting SSTables based on the key ranges of the overlapping groups at the next level, which are then added to these overlapping groups.

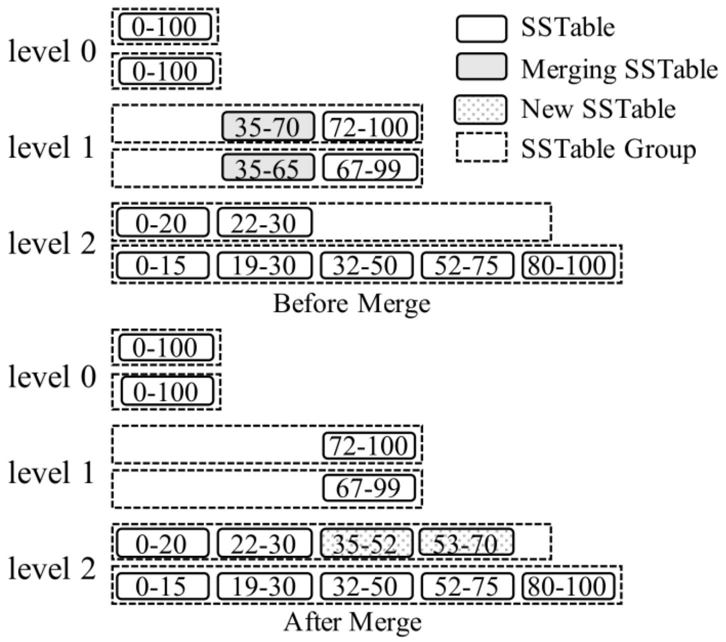


Figure 4: Partitioned tiering with horizontal grouping

Each level L further maintains an active group, which is also the first group, to receive new SSTables merged from the previous level. A merge operation selects the SSTables with overlapping key ranges from all of the groups at a level, and the resulting SSTables are added to the active group at the next level.

3. Concurrency Control and Recovery Depending on the transactional isolation requirement, today's LSM-tree implementations either use a locking scheme or a multi-version scheme. A multi-version scheme works well with an LSM-tree since obsolete can be garbage-collected during merges.

   Concurrent flush and merge operations, however, are unique to LSM-tree. These operations modify the metadata of an LSM-tree, e.g., the list of active components. Thus, accesses to the component metadata must be properly synchronized. To prevent a component in use from being deleted, each component can maintain a reference counter. Before accessing the components of an LSM-tree, a query can first obtain a snapshot of active components and increment their in-use counters.

Since all writes are first appended into memory, write-ahead logging (WAL) can be performed to ensure their durability. To simplify the recovery process, existing systems typically employ a **no-steal buffer management policy**: a memory component can only be flushed when all active write transactions have terminated. During recovery for an LSM-tree, the transaction log is replayed to redo all successful transactions, but no undo is needed due to the no-steal policy.

Meanwhile, the list of active disk components must also be recovered in the event of a crash.

- For unpartitioned LSM-trees, this can be accomplished by adding a pair of timestamps of the stored entries.

  This timestamp can be simply generated using local wall-clock time or a monotonic sequence number. To reconstruct the component list, the recovery process can simply find all components with disjoint timestamps. In the event that multiple components have overlapping timestamps, the component with the largest timestamp range is chosen and the rest can simply be deleted since they will have been merged to form the selected component.
- For partitioned LSM-trees, a typical approach is to maintain a separate metadata log to store all changes to the structural metadata, such as adding or deleting SSTables. The state of the LSM-tree structure can then be reconstructed by replaying the metadata log during recovery.

4. Cost Analysis The cost of writes and queries is measured by counting the number of disk I/Os per operation. This analysis considers an unpartitioned LSM-tree and represents a worst-case cost.

   Define

   $T =$ size ratio of a given LSM-tree
   $L =$ levels of the LSM-tree
   $B =$ number of entries that each data page can store, page size
   $P =$ number of pages of a memory component

   As a result, a memory component will contain at most $B \cdot P$ entries., and level i will contain at most $T^{i+1} \cdot B \cdot P$ entries. Given N total entries, the largest level contains approximately $N \cdot \frac{T}{T+1}$. Thus the number of levels for N entries can be approximated as $L = \lceil \log_T \left(\frac{N}{B \cdot P} \cdot \frac{T}{T+1} \right) \rceil$

The write cost, which is also referred to as **write amplification** in the literature, measures the amortized I/O cost of inserting an entry into an LSM-tree. It should be noted that this cost measures the overall I/O cost for this entry to be merged into the largest level since inserting an entry into memory does not incur any disk I/O.

- For leveling, a component at each level will be merge $T-1$ times until it fills up and is pushed to the next level.
- For tiering, multiple components at each level are merged only once and are pushed to the next level directly.

Since each disk page contains B entries, the write cost for each entry will be $O(T \cdot \frac{L}{B})$ for leveling and $O(\frac{L}{B})$ for tiering.

3 Fault Tolerance

3.1 Chain Replication for Supporting High Throughput and Availability

This paper is concerned with storage systems that sit somewhere between file systems and database systems.

strong consistency guarantees:

1. operations to query and update individual objects are executed in some sequential order
2. the effects of update operations are necessarily reflected in results returned by subsequent query operations.

```
**State is:**
  $Hist_{objID}$ : **update request sequence**
  $Pending_{objID}$ : **request set**

**Transitions are:**
  T1: Client request $r$ arrives:
        $Pending_{objID} := Pending_{objID} \cup \{r\}$
  T2: Client request $r \in Pending_{objID}$ ignored:
        $Pending_{objID} := Pending_{objID} - \{r\}$
  T3: Client request $r \in Pending_{objID}$ processed:
        $Pending_{objID} := Pending_{objID} - \{r\}$
        **if** $r = \mathsf{query}(objId, opts)$ **then**
           **reply** according options $opts$ based
              on $Hist_{objID}$
        **else if** $r = \mathsf{update}(objId, newVal, opts)$ **then**
           $Hist_{objID} := Hist_{objID} \cdot r$
           **reply** according options $opts$ based
              on $Hist_{objID}$
```

Figure 5: Client's View of an Object

Servers are assumed to be fail-stop:

1. each server halts in response to a failure rather than making erroneous state transitions, and
2. a server's halted state can be detected by the environment.

- $Hist_{objID}$ is defined to be $Hist^{T}_{objID}$, the value of $Hist_{objID}$ stored by tail T of the chain,
- $Pending_{objID}$ is defined to be the set of client requests received by any server in the chain and not yet processed by the tail.

1. a server in the chain receiving a request from a client
2. the tail processing a client request

The master distinguishes three cases:

- failure of the head
- failure of the tail
- failure of some other server in the chain.

Let the server at the head of the chain be labeled H, the next server be labeled $H + 1$, etc., through the tail, which is given label T. Define

$$Hist^i_{objID} \preceq Hist^j_{objID}$$

to hold if sequence of requests $Hist^i_{objID}$ at the server with label i is a prefix of sequence $Hist^j_{objID}$ at the server with label j.

Update Propagation Invariant: For servers labeled i and j s.t. $i \leq j$ holds then

$$Hist^j_{objID} \preceq Hist^i_{objID}$$

Inprocess Requests Invariant: If $i \leq j$ then

$$Hist^i_{objID} = Hist^j_{objID} \oplus Sent_i$$