VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

Cassandra Storage Engine

Lecturer : Dr. Pavle Mogin

SWEN 432
Advanced Database Design and Implementation

Cassandra The Fortune Teller

Plan for Cassandra Storage Engine

- Table Primary Key and Partitioning
- Storage Engine Rows
- Log Structured Merge Trees (LSM Trees)
- Memtable and SSTables
- Write Paths for Insert and Update
- About Reads
- About Deletes
- Compaction:
  - Size Tiered Compaction
  - Leveled Compaction

- ***Readings:*** Have a look *at Useful Links at the Course Home Page*

Table Primary Key and Partitioning (1)

- The table primary key is defined within the table declaration
- Each table row has a unique primary key value
- A primary key can be:
  - Simple containing just one column name, where the column name is the partition key, or
  - Compound containing more than one column name, where the first column name is the partition key and the remaining column names are the clustering key, or
  - Composite having a multicolumn partition key enclosed in a parenthesis, and a clustering key
- Example:
  - The table `users` has a simple primary key `user_name`,
  - The table `blogs_entry` has a composite primary key `((user_name, date), no)`

Table Primary Key and Partitioning (2)

- All table rows having the same partition key value make a CQL partition
  - A single column primary key makes single row CQL partitions,
  - A compound primary key and a composite primary key have both the partition and clustering keys and produce multi row CQL partitions
  - CQL rows within a partition are sorted according to the clustering key values
- Table rows are assigned to nodes in the consistent hashing ring by
  - A cluster configured partitioner and
  - The replica placement strategy defined for each keyspace
- The partitioner hashes partition keys into tokens
  - Tokens are points on the consistent hashing ring
  - This way are CQL partitions assigned to cluster nodes

Indexes

- The primary index of a table is a unique index on its row key
  - Cassandra maintains the primary index automatically
- Each node maintains this index for data it manages

Storage Engine Row

- A storage engine row is a sequence of table rows having the same partition key value stored on disk
  - For tables having a primary key with no clustering column, a storage engine row contains a single table row
  - For tables having a primary key with clustering columns, a storage engine row contains at least one table row
- As a CQL partition size grows, the storage engine row grows
- As the storage engine row grows, read latency for some queries will increase
- Example:
  - Assume `blog_entries` table key is (user_name, no). As new blogs are added into `blog_entries` table, storage engine rows for particular users may grow very big

Bucketing

- A technique to limit the size of large engine rows is to introduce a sensible time bucket
- Example:
  - In the case of the `blog_entries` table a sensible time bucket may be a `year_month` column that extends the primary key in the following way `((user_name, year_month), no)`
- But, there is no way to change a primary key in Cassandra, as it defines how data is stored physically
- The only work around is:
  - To create a new table with the new primary key,
  - Copy data from the old one, and then
  - Drop the old table
- Bucketing has to be defined at the time of the database design

Reverse Queries

- Often, queries ask for the last entry of a time series
- Example:
  - Retrieve the James's last blog
- One way to satisfy the query is to sort the table for each query:

```
SELECT * FROM blog_entries
WHERE user_name = 'jbond'
ORDER BY no desc
LIMIT 1;
```

- Read performance wise, a more efficient way is to keep CQL partitions sorted in the descending order

Redesigning the `blog_entries` Table

- Table:

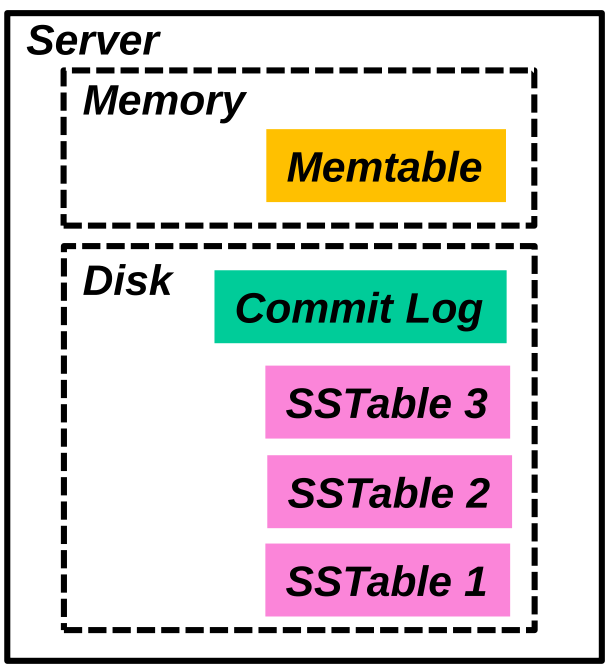
```
CREATE TABLE blog_entries (
user_name text,
year_month int,
body text,
no int,
PRIMARY KEY ((user_name, year_month), no)
)
with clustering order by (no desc);
```

- Query:

```
SELECT * FROM blog_entries WHERE
user_name = 'jbond' and year_month =
201603 LIMIT 1;
```

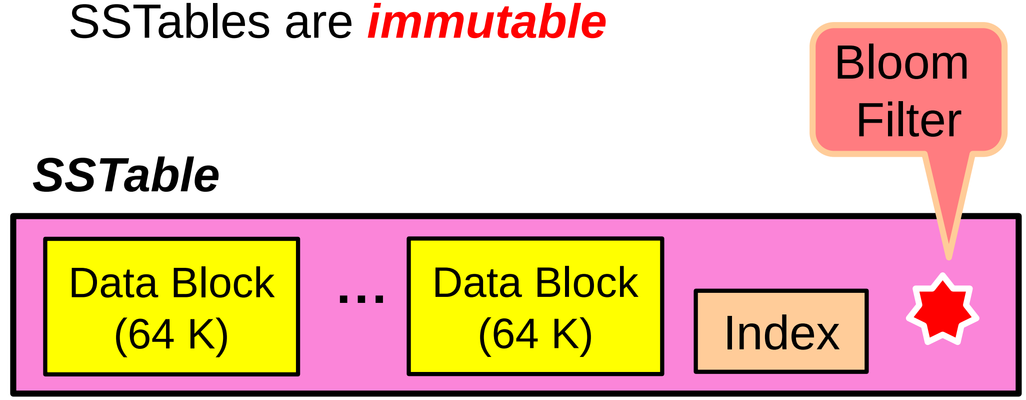
Log Structured Merge Trees

- LSM trees are an approach to use memory and disk storage to satisfy write and read requests in an efficient and safe way
  - In the memory, there is a memtable containing chunks of recently committed data,
  - On disk, there is a commit-log file and a number of SSTables containing data flushed from the memtable



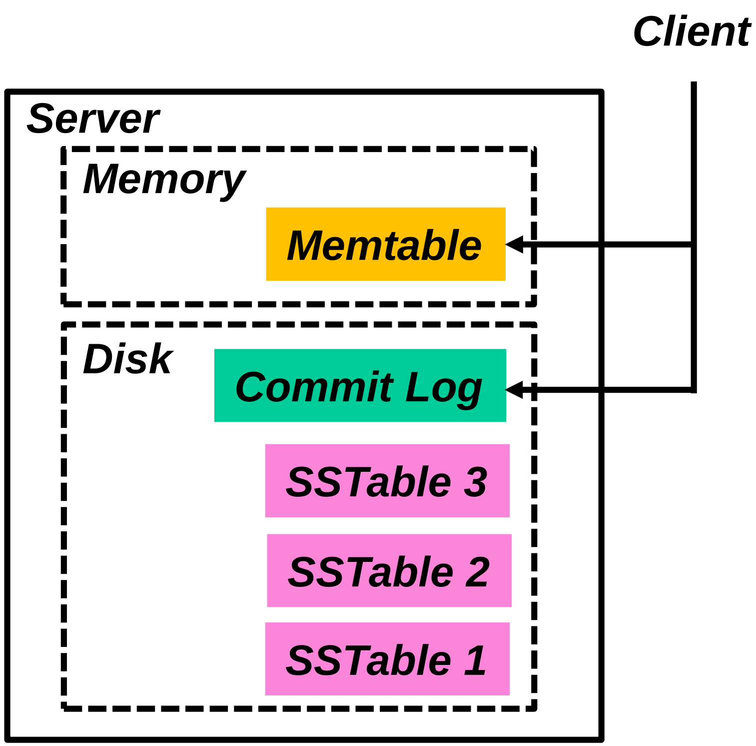
Table’s data reside in the Memtable and SSTable 1, SSTable 2, and SSTable 3

SSTables are ***immutable***



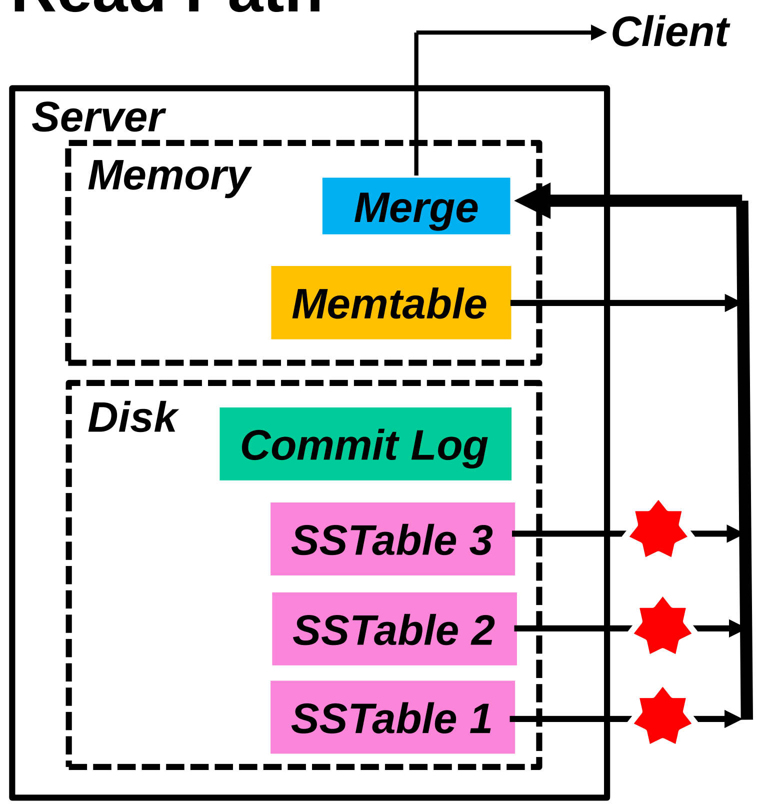
LSM Trees Write and Read Paths

Write Path



LSM trees are optimized for writes since writes go only to Commit Log and Memtable

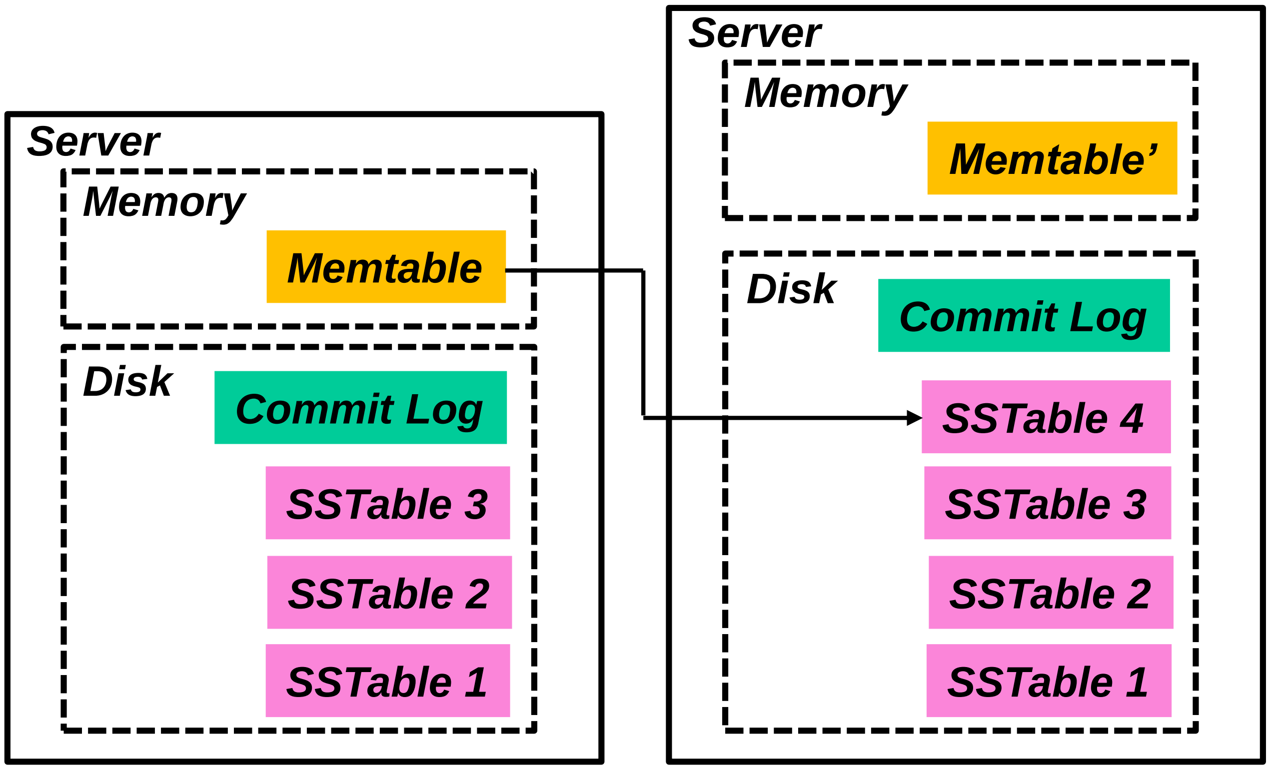
Read Path



To optimize reads and to read relevant SSTables only, Bloom filters are used ( )

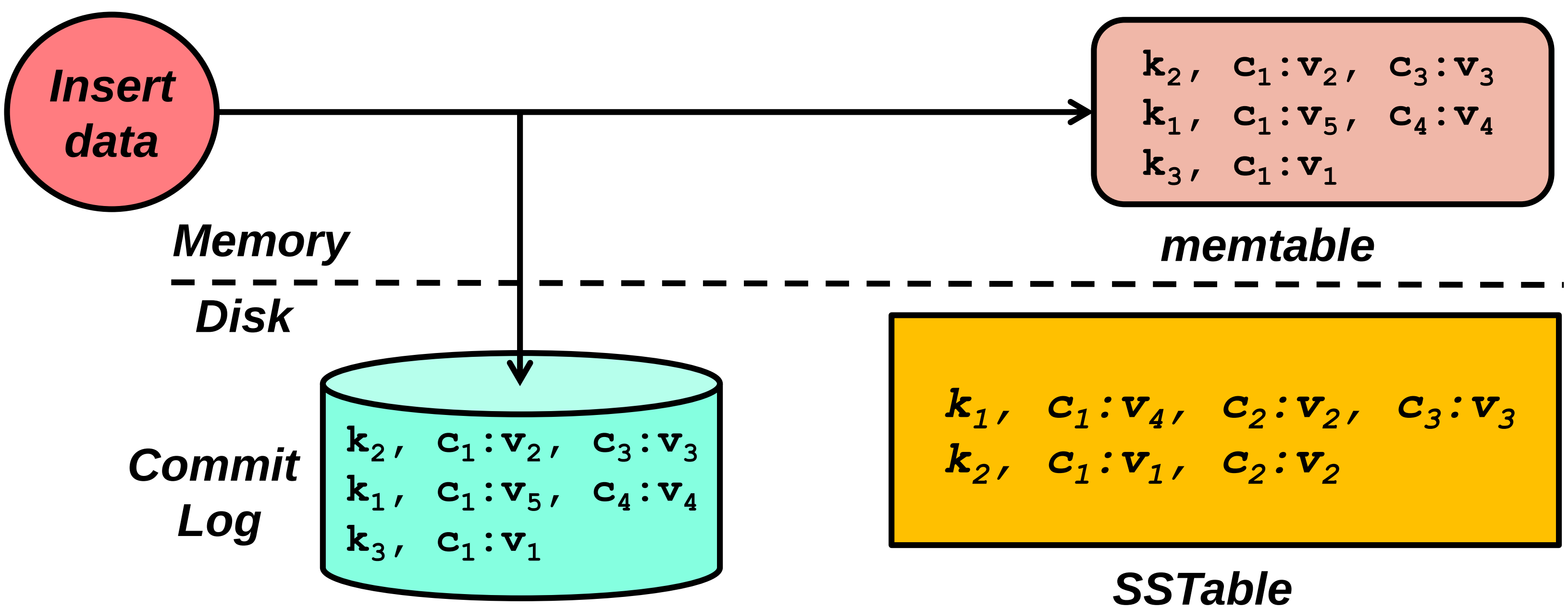
LSM Trees - Flushing

When a Memtable reaches a certain size, it is frozen, a new Memtable is created, and the old Memtable is flushed in a new SSTable on disk



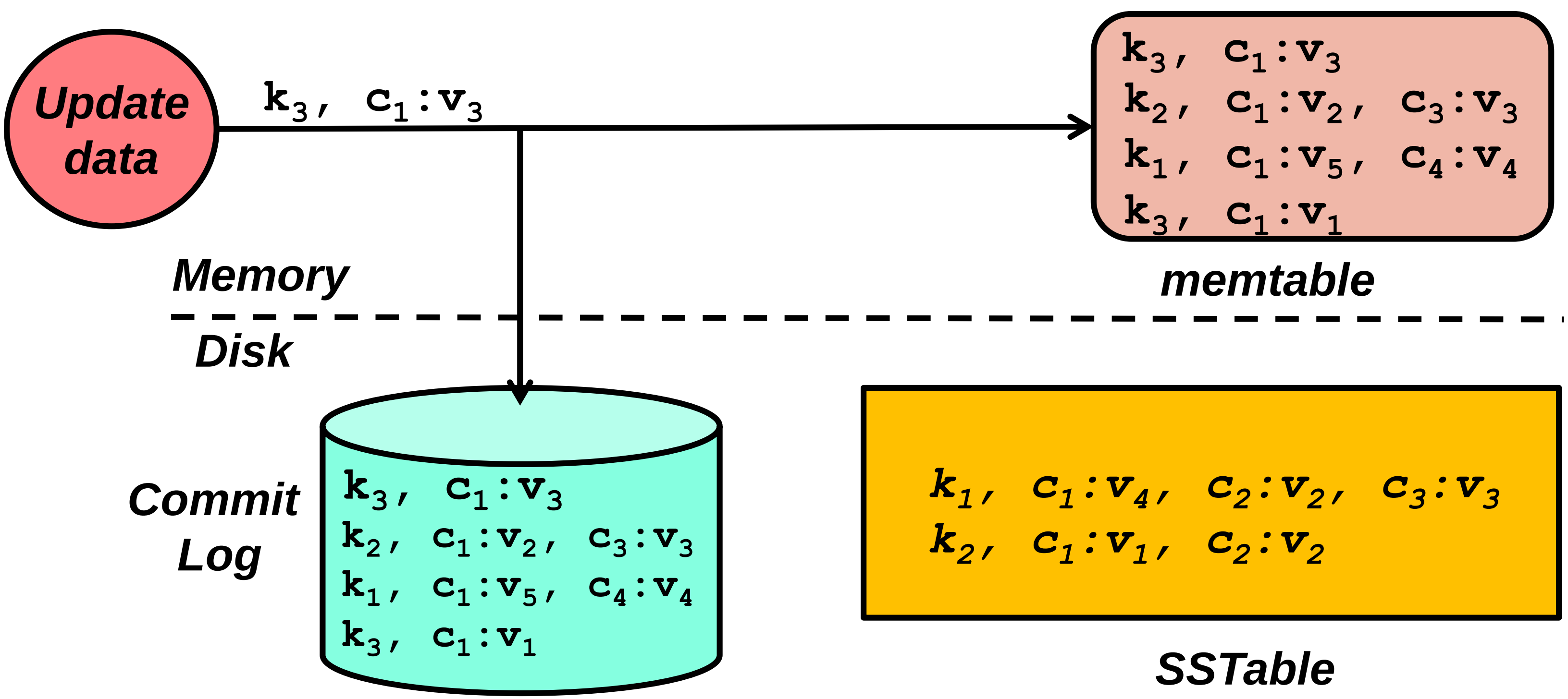
The Write Path of an Insert

- Assume row keys k_1, k_2, and k_3 map to the same partition
- Assume rows (k_1, c_1:v_4, c_2:v_2, c_3:v_3) and (k_2, c_1:v_1, c_2:v_2) are already flushed in a SSTable on disk
- Next are rows (k_3, c_1:v_1), (k_1, c_1:v_5, c_4:v_4) and (k_2, c_1:v_2, c_3:v_3) written into Commit Log and memtable



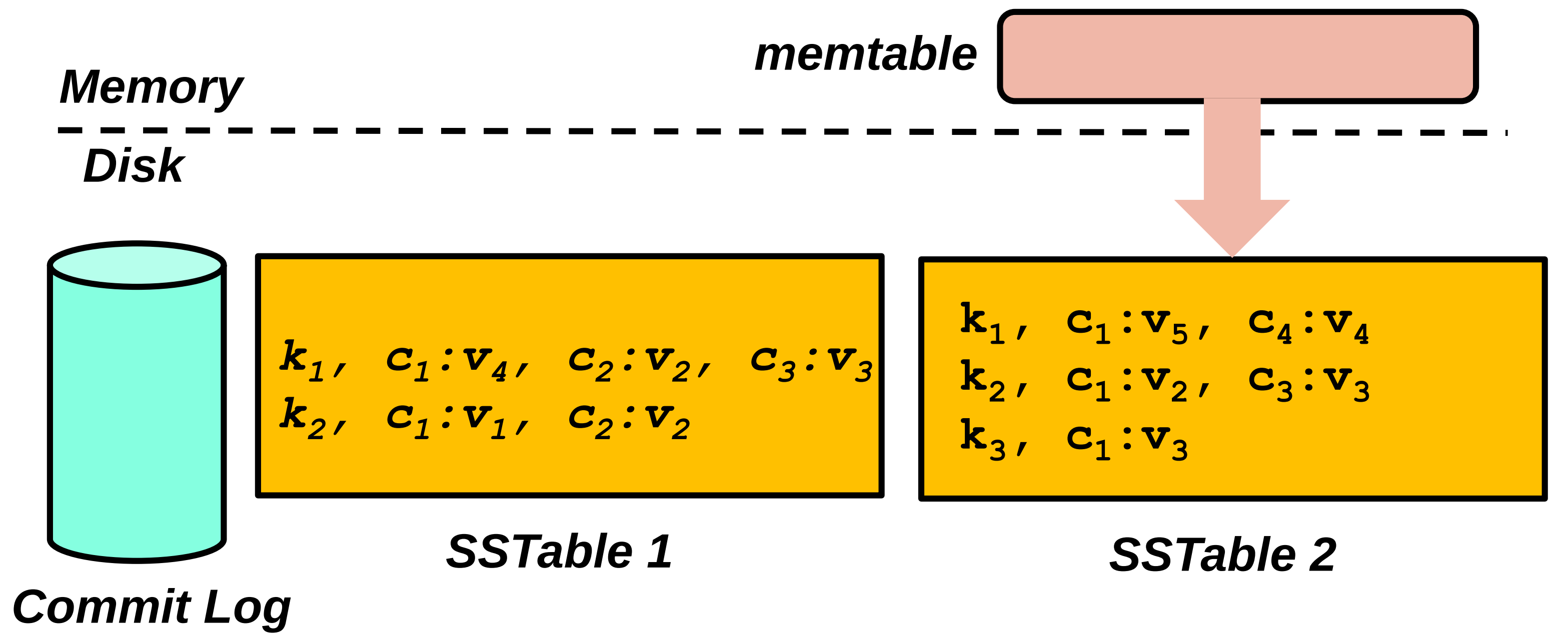
The Write Path of an Update

- The update command works like an upsert:
  - It simply inserts a new row into the commit log and memtable,
  - Cassandra doesn't modify a column value in place



Memtable Flushing

- When a memtable exceeds a configurable threshold, the memtable data are sorted by the primary key and flushed into a SSTable on disk
  - Only the latest value of each column of a row goes to the SSTable, since it has the greatest time stamp value
  - After flushing the memtable, the Commit Log data is also purged



Log Structured Merge Trees (Summary)

- Cassandra uses Log Structured Merge (LSM) Trees in a similar way as BigTable does:
  - Writes are done into a commit log on disk and in a memtable in memory,
  - Each column family (table) has its own memtable, commit log, and SSTables in each partition,
  - When the size of a memtable reaches a prescribed threshold, it is flushed in a SSTable on disk,
  - SSTables are immutable, hence different SSTables may contain different versions of a row column, and updates and deletes are implemented as time stamped writes (there is no in-place updates or deletes),
  - There are no reads before writes and no guarantee of the uniqueness of the primary key (unless special mechanisms are applied),
  - Reads are performed by merging requested data from the memtable and all SSTables,
    - To read only from SSTables containing data requested, Bloom Filters are used,
  - Clients are supplied the latest versions of data read,
  - SSTables are periodically compacted into a new SSTable

Updates and Timestamps

- Cassandra flushes only the most recent value of each column of a row in memtable to the SSTable
  - The most recent column value has the greatest timestamp
- Precise timestamps are needed if updates are frequent
- Timestamps are provided by clients
- Clocks of all client machines should be synchronized using NTP (network time protocol)

About Reads

- Cassandra must combine results from the memtable and potentially multiple SSTables to satisfy a read
- First, Cassandra checks the Bloom filter
  - Each SSTable has a Bloom filter associated with it that checks the probability of having any data for the requested partition key in the SSTable before doing any disk I/O
- If the Bloom filter does not rule out the SSTable, Cassandra checks the partition key cache and takes a number of other actions to find the row fragment with the given key

Bloom Filter (1)

- A Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set
  - False positive matches are possible, but
  - False negatives are not
- An empty Bloom filter is a m bit array, all bits set to 0
- There must also be n ($< m$) different hash functions, each of which maps an element to one of the m array positions
- To add an element into the Bloom filter, its n array positions are calculated
  - Bits at all n positions are set to 1

Bloom Filter (2)

- To test whether an element is in the set, its *n* array positions are calculated
  - If any of the bits at these positions are *0*, the element is definitely not in the set
  - Otherwise, it is probably in the set
- It has been proved that fewer than *10* bits per an element in the set are required for a *1%* false positive probability, independent of the size or number of elements in the set

Bloom Filter Example

- Assume:
  - The set of SSTable keys is {173, 256, 314}
  - Hash functions are: $h_1 = k\ mod7$, $h_2 = k\ mod13, h_3 = k\ mod17$, where k is a SSTable key, and $n = 3$
  - Let $m = 17$
- Array positions:
  - For $k = 173$, array positions are ($5, 4, 3$)

0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

  - For $k = 256$, array positions are ($4, 9, 1$)
  - For $k = 314$, array positions are ($6, 2, 8$)

0	1	1	1	1	1	1	0	1	1	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- Tests:
  - For $k = 105$, array positions are ($0, 1, 3$), so negative
  - For $k = 106$, array positions are ($1, 2, 4$), so false positive

About Deletes

- CQL `DELETE` statement works as a write
  - Deleted column value is written in the memtable and commit log as a (row_key, column_name, TOMBSTONE, time_stamp) tuple
    - TOMBSTONE is the new column value, indicating it has been deleted
- Data in a Cassandra column can have an optional expiration time called TTL (time to live)
- The TTL in seconds is set in CQL
  - Cassandra marks TTL data with a tombstone after the requested amount of time has expired
  - Tombstones exist for a period of time defined by `gc_grace_seconds` that is a table property (10 days by default)
  - After data is marked with a tombstone and `gc_grace_seconds` has elapsed, the data is automatically removed during the normal compaction process

Storage Engine Example (1)

row key user_id	column	SSTable 1	SSTable 2	SSTable 3	merge
jbond	name	James ts 10			James ts 10
	city	London ts 10		Paris ts 30	Paris ts 30
	email		jbond@ecs ts 20		jbond@ecs ts 20
	pet		dog ts 20	tombstone ts 40	tombstone ts 40

Storage Engine Example (2)

- Assume:

```
(row_key: jbond, (name, James, ts10), (city,
                                London, ts10))
```

has been flushed into SSTable 1

- After issuing CQL commands:

```
ALTER TABLE users ADD email text, pet text;
```

```
UPDATE users SET email='jbond@ecs.vuw.ac.nz',
pet = 'dog' WHERE id = 'jbond';
```

the record

```
(row_key: jbond, (email, jbond@ecs.vuw.ac.nz,
ts20), (pet, dog, ts20))
```

has bee later flushed into SSTable 2

Storage Engine Example (3)

- Assume CQL commands:

```
UPDATE users SET city = 'Paris'
WHERE id = 'jbond';
```

```
DELETE pet FROM users WHERE id = 'jbond';
```

induce storing the following records into SSTable 3

```
(row_key: jbond, (city, Paris, ts30))
(row_key: jbond, (pet, tombstone, ts40))
```

- The command

```
READ * FROM users WHERE user_name = 'jbond';
```

returns:

```
id       name     city           email
--------|--------|----------|--------------------------
jbond James   Paris   jbond@ecs.vuw.ac.nz
```

Compaction Strategies

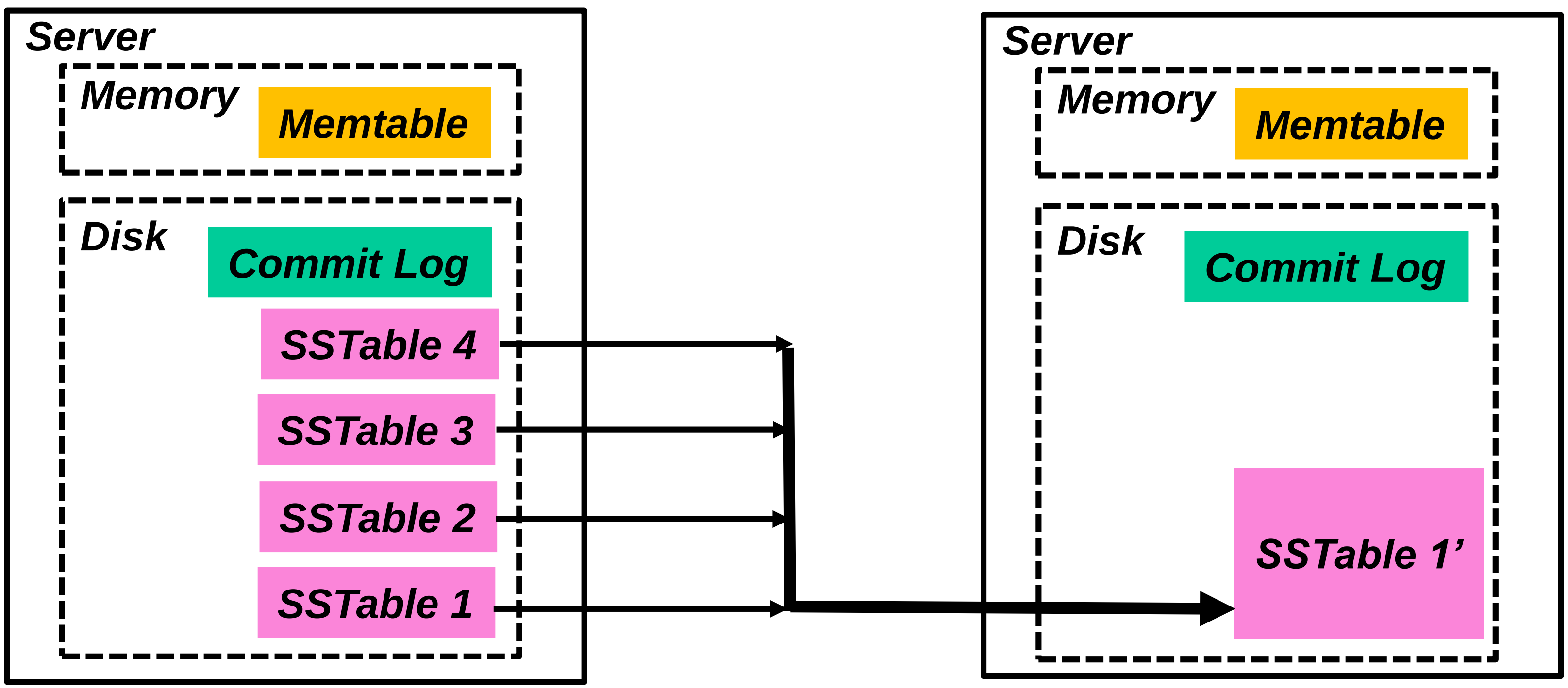
- Compaction is instrumental for attaining fast reads
- Cassandra supports:
  - `SizeTieredCompactionStrategy` designed for write intensive workloads (default),
  - `DateTieredCompactionStrategy` designed for time-series and expiring data, and
  - `LeveledCompactionStrategy` designed for read intensive workloads
- Compaction strategy is defined per column family and applied to its SSTables
- Cassandra performs compaction automatically, or it can be started manually using the `nodetool compact` command

LSM Trees - Compaction

- Since SSTables are immutable, column value updates and deletes are accomplished solely by writes in the memtable:

write(*old_row_key, old_column_key, new_column_value, new_timestamp*)

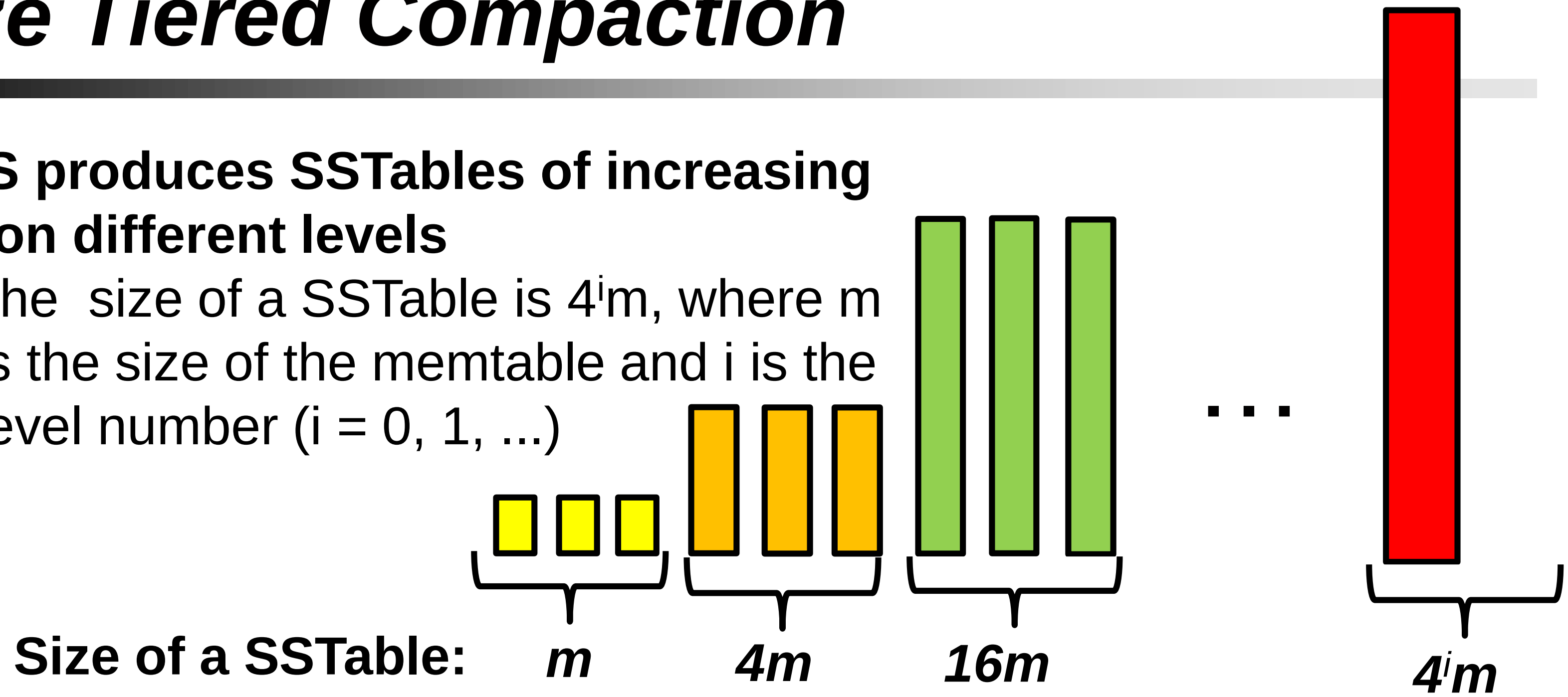
- In the case of deletes, the *new_column_value* is called the ***tombstone***.
- To reclaim the disk space and speed up reads, after a certain number of memtables has been flushed, the compaction of SSTables is undertaken



Size Tiered Compaction

STCS produces SSTables of increasing size on different levels

- The size of a SSTable is 4^im, where m is the size of the memtable and i is the level number (i = 0, 1, ...)

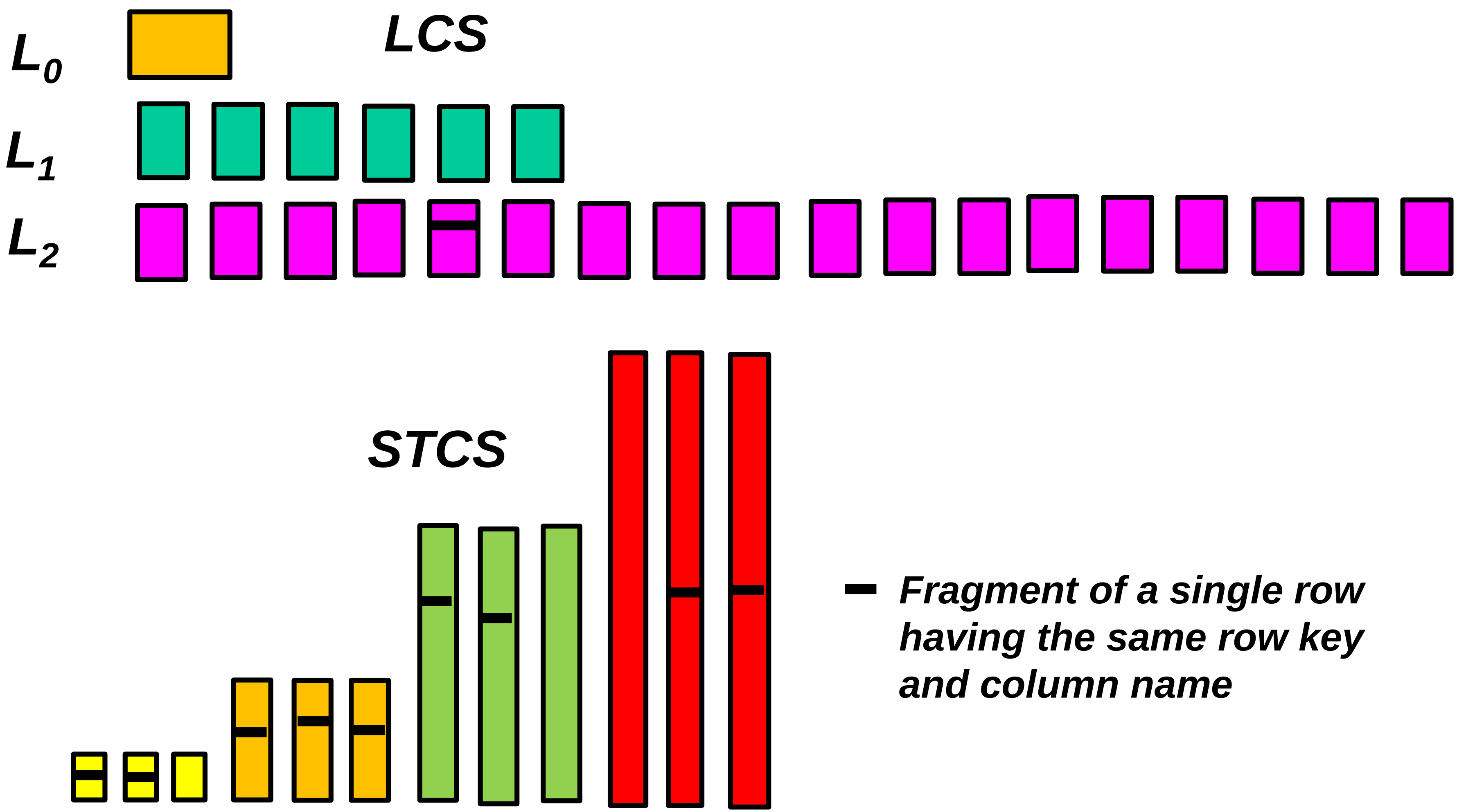


- Problems in an update intensive workload:
  1. Read performance can be inconsistent since a row may have columns in many SSTables,
  2. Removing obsolete columns (particularly deleted ones) is a problem
  3. Time and storage space to do the compaction of bigger SSTables rise

Leveled Compaction Strategy (LCS)

- LCS creates SSTables of a fixed size and groups them into levels L_1, L_2, L_3,...
  - Each level L_i ($i > 1$) has ten times greater size than the previous one and contains accordingly up to 10 times more SSTables
- The LCS compaction produces SSTables of a level by making a long sequential file that is split in a number of fixed size SSTables
- Accordingly, each row key appears on a single level at most once
  - There may exist an overlap of row keys on different levels
- Even more, with LCS:
  - The probability that only one of all SSTables contains a given row key value is *~0.9*, and
  - The expected number of SSTables to contain a given row key value is *~1.1*

LCS versus STCS – SSTables to Read



WHEN to Use LCS

- LCS is a better option when:
  - Low latency reads are needed,
  - High read/write ratio,
  - Update intensive workload
- To declare LCS for a table:

```
CREATE TABLE <table_name> (...)
WITH COMPACTION = {'class' :
'LeveledCompactionStrategy'};
```

- In the `blogs` keyspace:

```
sqlsh.blogs => CREATE TABLE user_subs
(user_name text PRIMARY KEY, no_of_subs
counter) WITH COMPACTION = {'class' :
'LeveledCompactionStrategy'};
```

WHEN to Use STCS

- STCS is default for a column family
- STCS is a better option than LCS if:
  - DISK I/O is expensive,
  - The workload is write heavy, and
  - Rows are written once (and rarely updated)
- In the `blogs` keyspace example, all tables except the `user_subs` and `suscribes_to` tables should have Seize Tiered Compaction Strategy
- Even the `blog_entries` table should be compacted by STCS since it is insert and not update intensive
  - Inserts write new rows
  - Updates write existing rows with a new column value

Summary

- Cassandra's Storage Engine uses Log Structured Merge (LSM) Trees with:
  - A commit log on disk,
  - A per column family memtable in memory
- Memtables are flushed in immutable SSTables on disk
- Updates and deletes are implemented as time stamped writes
- Different SSTables may contain different versions of a row column
- SSTables are periodically compacted into a new SSTable