Carnegie Mellon University

Database Systems

15-445/645 SPRING 2026
ANDY PAVLO
JIGNESH PATEL

Lecture #05

Database Storage: Index-Organized + Log-Structured Storage

ADMINISTRIVIA

Project #1 is due Sunday Feb 15th @ 11:59pm
→ Recitation Wednesday Jan 28th @ 6:30pm (**@64**)

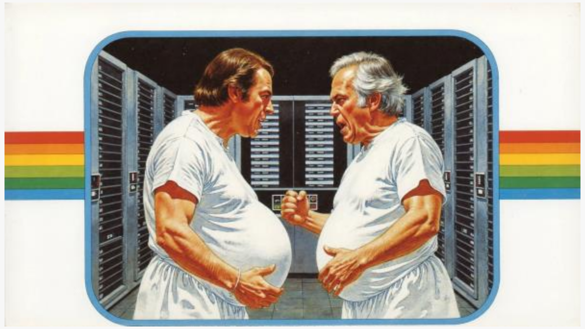
Homework #2 is due Sunday Feb 8th @ 11:59pm

UPCOMING DATABASE EVENTS

CMU-DB Reading Group
→ Tuesdays @ 12:00pm
→ GHC 9115
→ Free food!

Carnegie Mellon Database Group

Database Seminar Series (Zoom)
→ *PostgreSQL vs. The World*
→ Mondays @ 4:30pm
→ Starting Feb 2nd



LAST CLASS

We introduced the buffer pool manager as the location of where the DBMS stores copies of database pages it retrieves from non-volatile storage.

This is for a disk-oriented architecture where the DBMS assumes that the primary storage location of the database is on non-volatile disk.

We then discussed a page-oriented storage scheme for organizing tuples across heap files.

PAGE LAYOUT

For any page storage architecture, we now need to decide how to organize the data inside of the page.
→ We are still assuming that we are only storing tuples in a row-oriented storage model.
→ We will also assume that each tuple fits in a single page.

Approach #1: Slotted Page Storage

Approach #2: Index-organized Storage ← Today

Approach #3: Log-structured Storage

TODAY'S AGENDA

Slotted Page Storage

Index-Organized Storage

Log-Structured Storage

System Catalogs

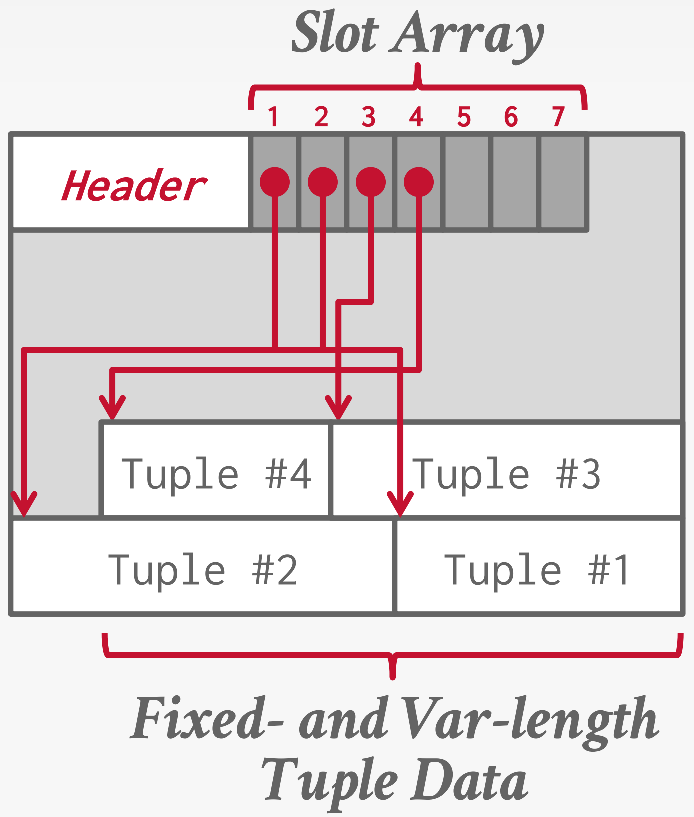
⚡**DB Flash Talk: SingleStore**

SLOTTED PAGES

The most common page layout scheme is called <u>slotted pages</u>.

The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:
→ The # of used slots
→ The offset of the starting location of the last slot used.

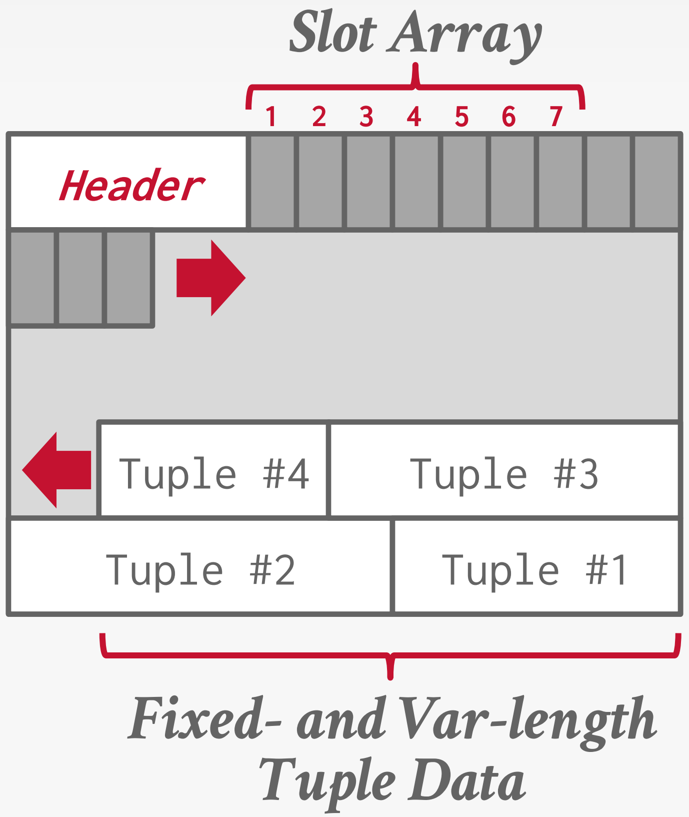


SLOTTED PAGES

The most common page layout scheme is called <u>slotted pages</u>.

The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:
→ The # of used slots
→ The offset of the starting location of the last slot used.



SLOTTED PAGES

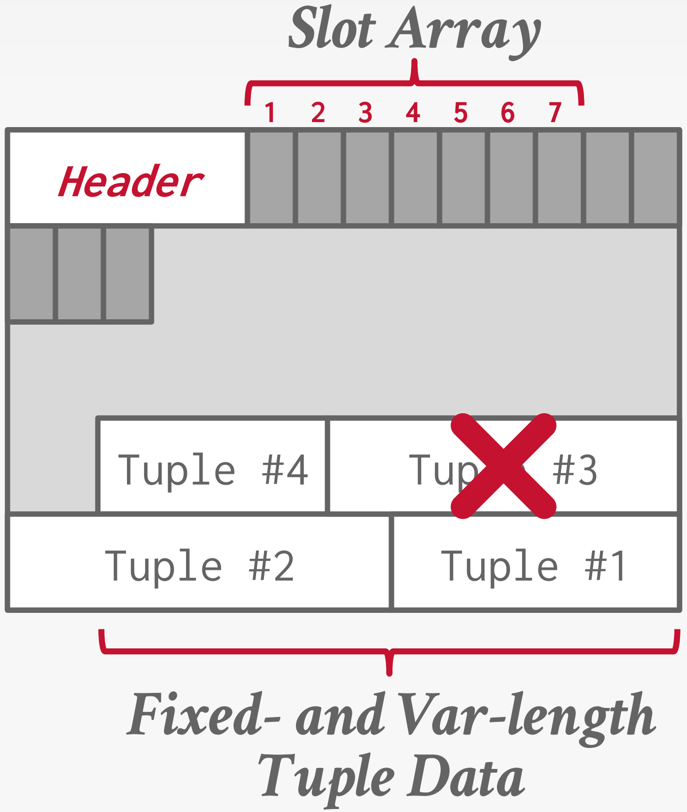
The most common page layout scheme is called slotted pages.

The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:

→ The # of used slots

→ The offset of the starting location of the last slot used.

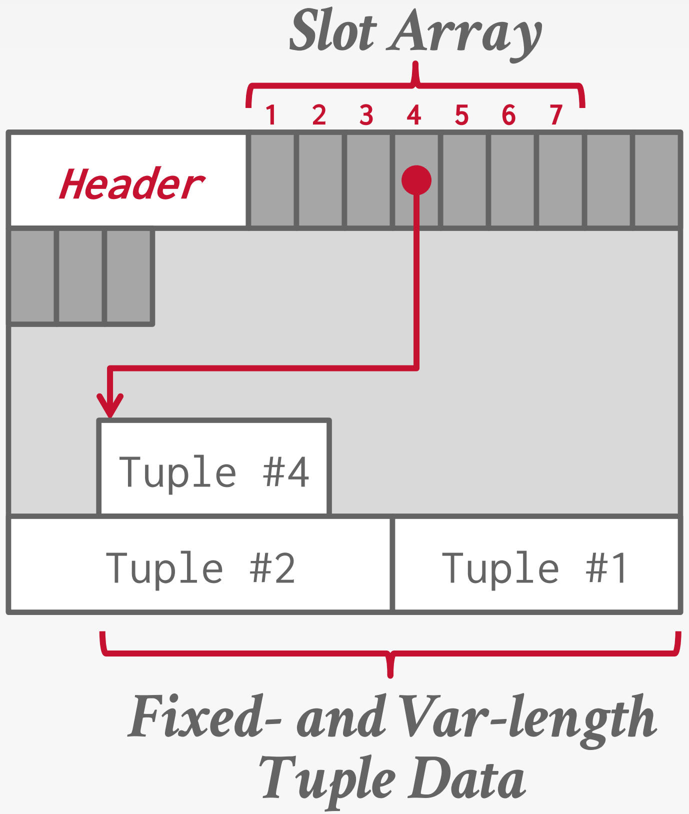


SLOTTED PAGES

The most common page layout scheme is called slotted pages.

The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:
→ The # of used slots
→ The offset of the starting location of the last slot used.

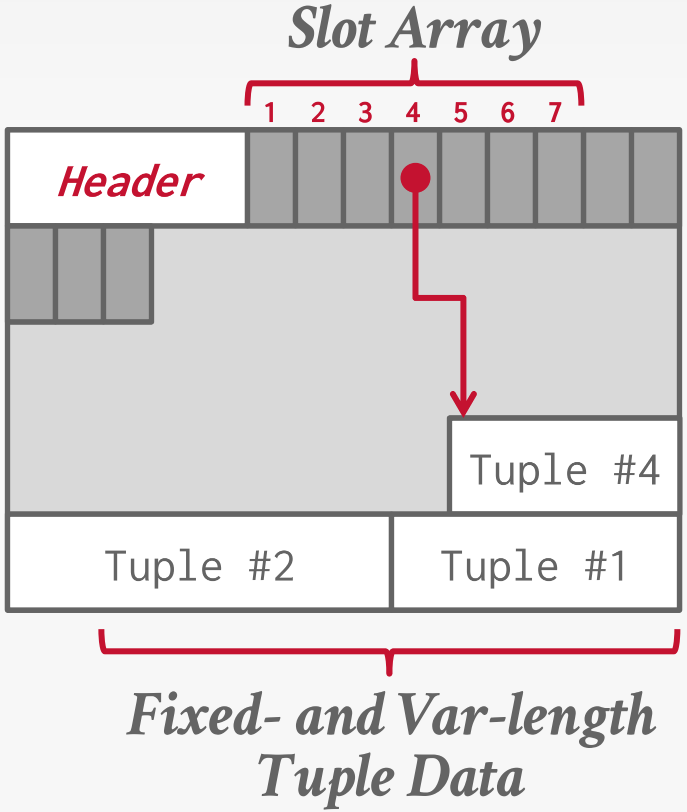


SLOTTED PAGES

The most common page layout scheme is called slotted pages.

The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:
→ The # of used slots
→ The offset of the starting location of the last slot used.



RECORD IDS

The DBMS assigns each logical tuple a unique **record identifier** that represents its physical location in the database.

→ Example: File Id, Page Id, Slot #
→ Most DBMSs do not store ids in tuple.
→ SQLite uses **ROWID** as the true primary key and stores them as a hidden attribute.

Applications should never rely on these IDs to mean anything.

Record Id Sizes

INGRES	TID	*4-bytes*
PostgreSQL	CTID	*6-bytes*
SQLite	ROWID	*8-bytes*
Microsoft SQL Server	%%physloc%%	*8-bytes*
Firebird	RDB$DB_KEY	*8-bytes*
ORACLE	ROWID	*10-bytes*

SLOTTED PAGE STORAGE: READS

Get an existing tuple using its record id:
→ Check page directory to find location of page.
→ Retrieve the page from disk (if not in memory).
→ Find offset in page using slot array.

The DBMS relies on <u>indexes</u> to find individual tuples because the tables are inherently unsorted.

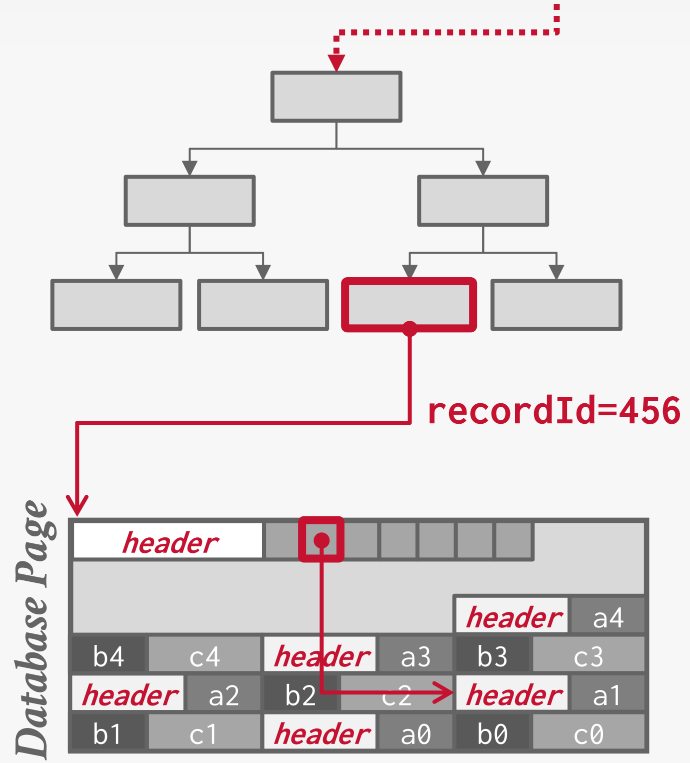
SLOTTED PAGE STORAGE: READS

For a given key(s), an index provides a mapping from that key(s) to one or more tuples with that value via their Record Ids.

The DBMS has to first access the pages for the index then read the page(s) for the tuple(s).

But what if the DBMS could keep tuples sorted automatically using an index?

```
SELECT * FROM A WHERE key = 123
```

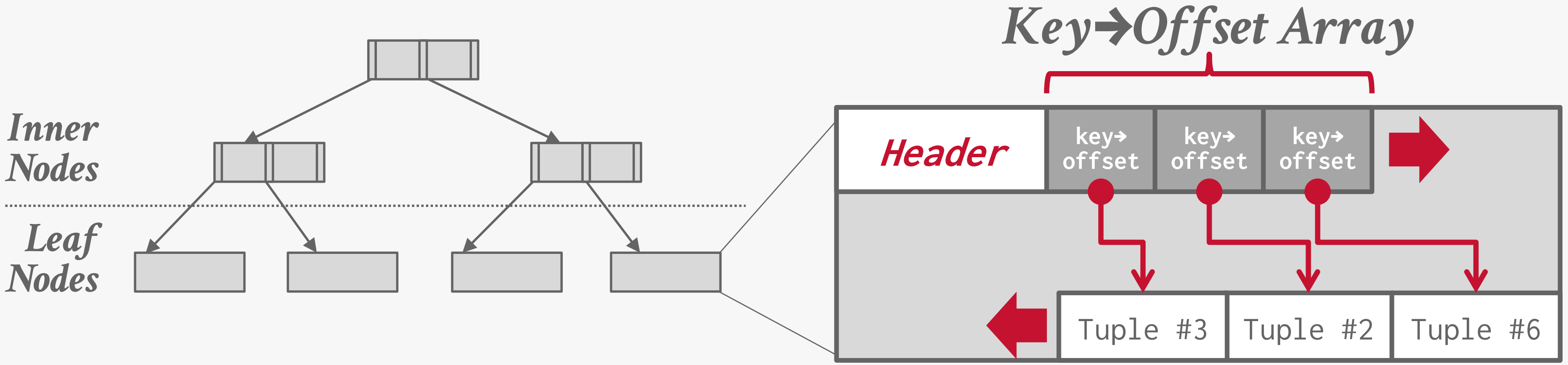


INDEX-ORGANIZED STORAGE

DBMS stores a table's tuples as the value of an index data structure.

→ Leaf nodes page layout are similar to slotted page layout.

→ Tuples are typically sorted in page based on key.



SLOTTED PAGE STORAGE: WRITES

Insert a new tuple:

→ Check page directory to find a page with a free slot.
→ Retrieve the page from disk (if not in memory).
→ Check slot array to find empty space in page that will fit.

Update an existing tuple using its record id:

→ Check page directory to find location of page.
→ Retrieve the page from disk (if not in memory).
→ Find offset in page using slot array.
→ If new data fits, overwrite existing data.
Otherwise, mark existing tuple as deleted and insert new version in a different page.

SLOTTED PAGE STORAGE

Problem #1: Fragmentation

→ Pages are not fully utilized (unusable space, empty slots).

Problem #2: Useless Disk I/O

→ DBMS must fetch entire page to update one tuple.

Problem #3: Random Disk I/O

→ Worse case scenario when updating multiple tuples is that each tuple is on a separate page.

What if the DBMS cannot overwrite data in pages and could only create new pages?

→ Examples: HDFS, Google Colossus, S3 Express

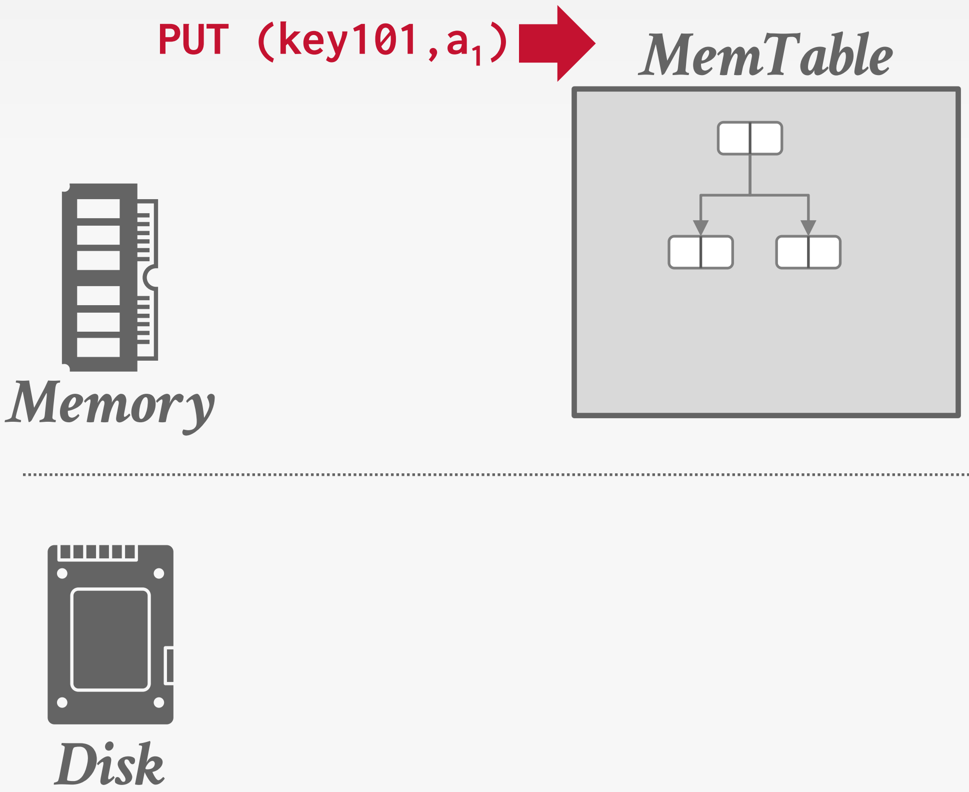
LOG-STRUCTURED STORAGE

Instead of storing tuples in pages and updating the in-place, the DBMS maintains a log that records changes to tuples.

→ Each log entry represents a tuple **PUT**/**DELETE** operation.
→ Originally proposed as log-structure merge trees (LSM Trees) in 1996.

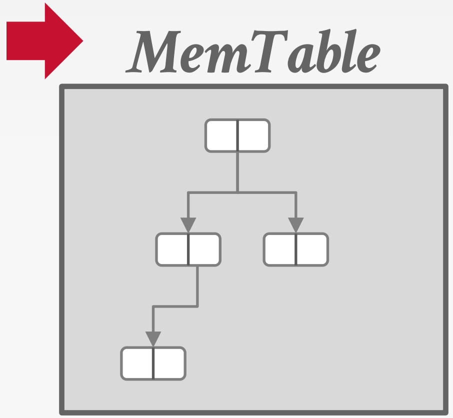
The DBMS applies changes to an in-memory data structure (***MemTable***) and then writes out the changes sequentially to disk as sorted-string tables (***SSTables***).

LOG-STRUCTURED STORAGE



LOG-STRUCTURED STORAGE

`PUT (key101,a₁)`

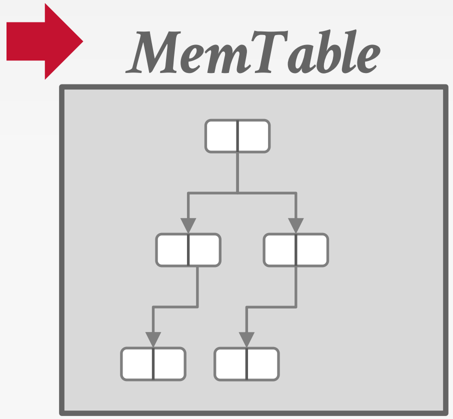


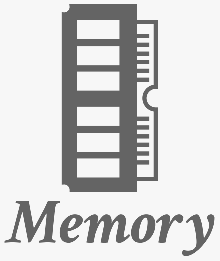
Memory

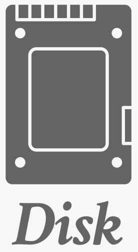
Disk

LOG-STRUCTURED STORAGE

PUT (key102,b_1)

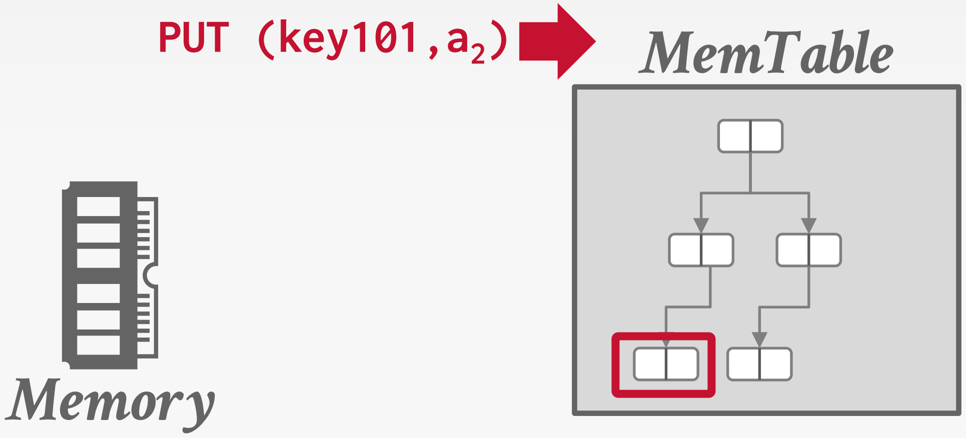


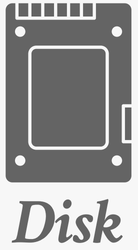




LOG-STRUCTURED STORAGE

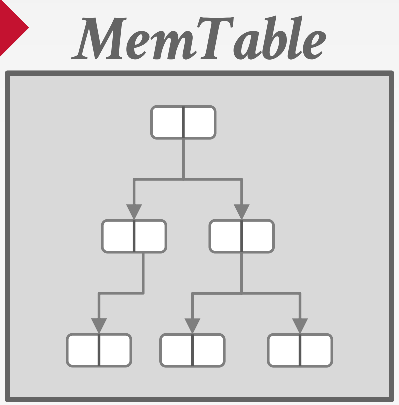
PUT (key101,a_2)





LOG-STRUCTURED STORAGE

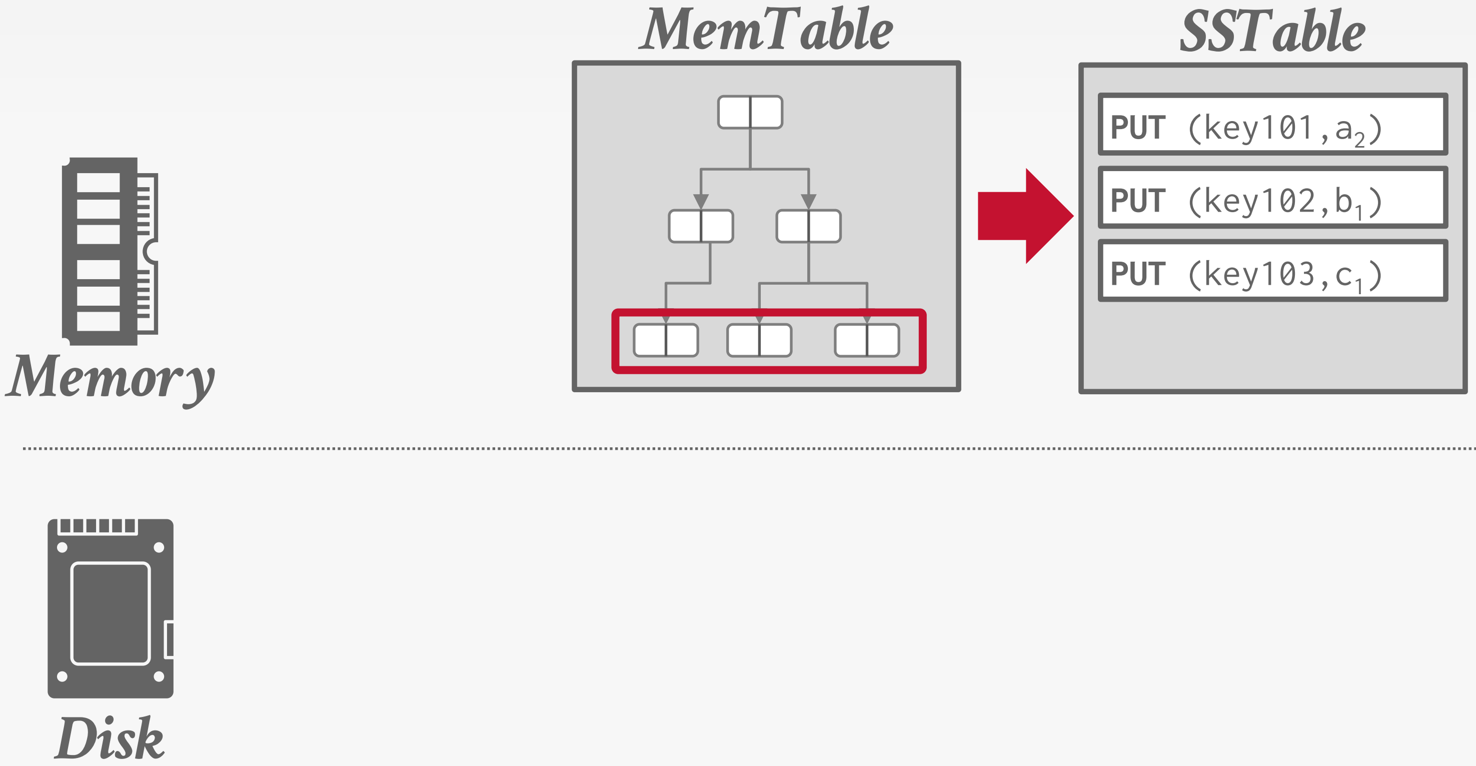
PUT (key103,c_1) →



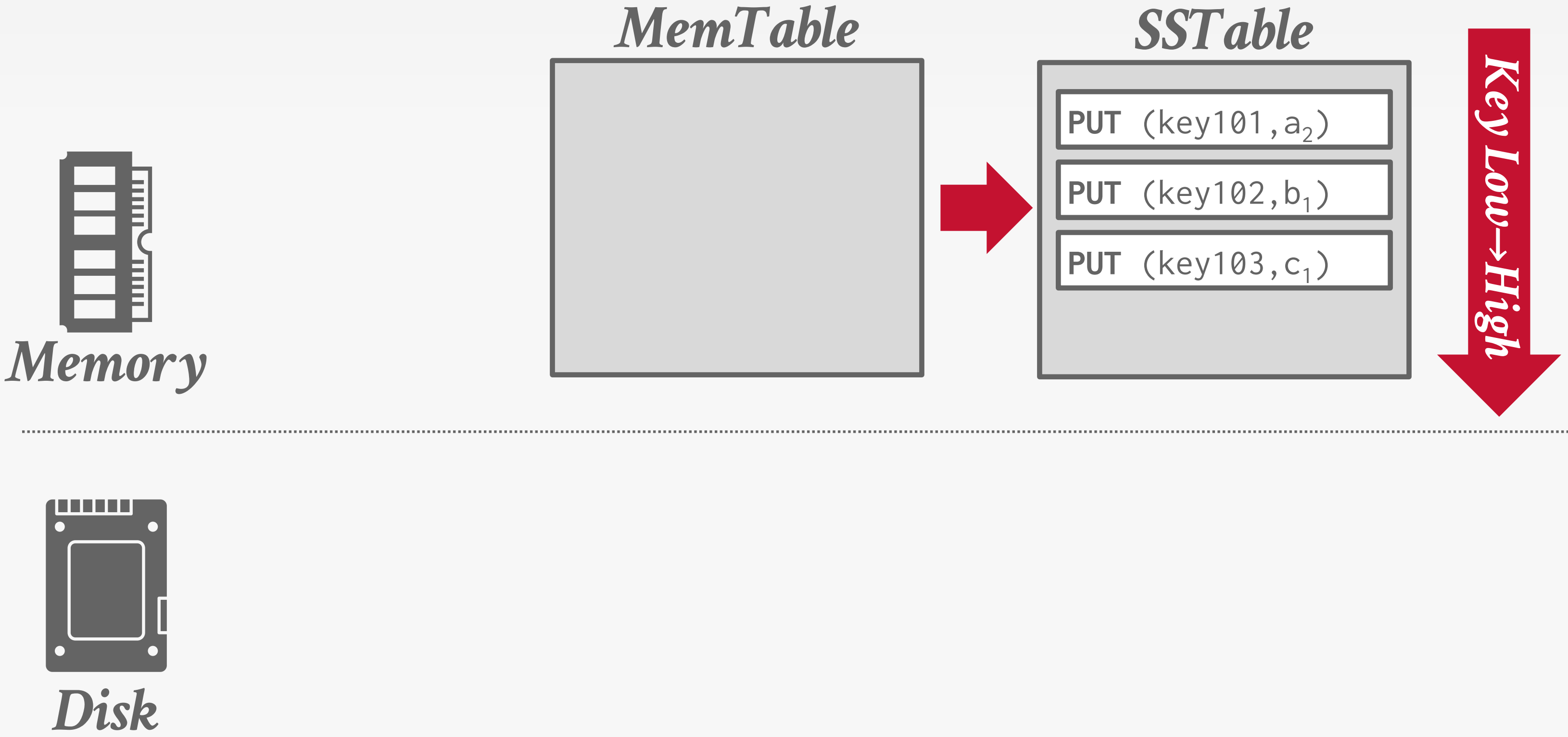
Memory

Disk

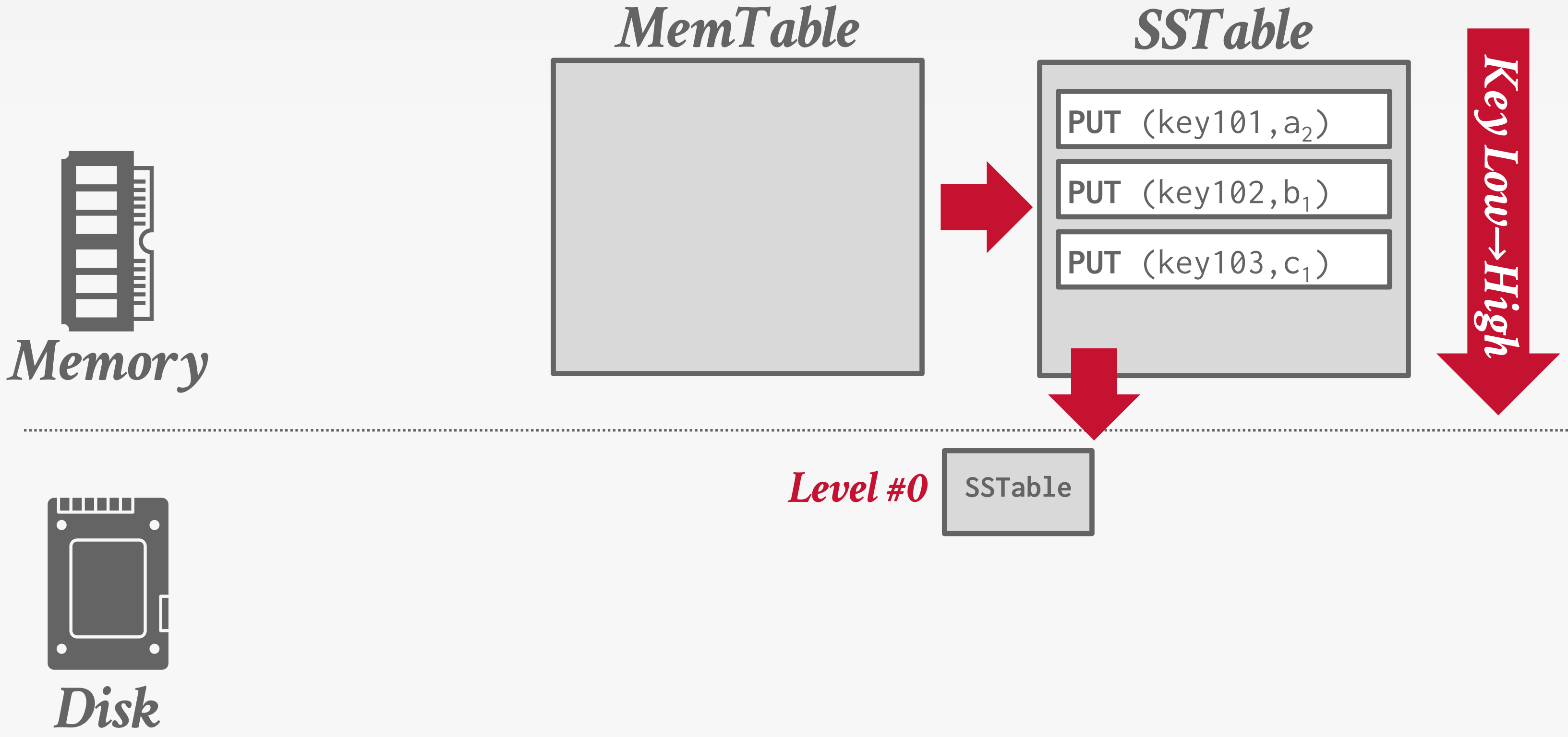
LOG-STRUCTURED STORAGE



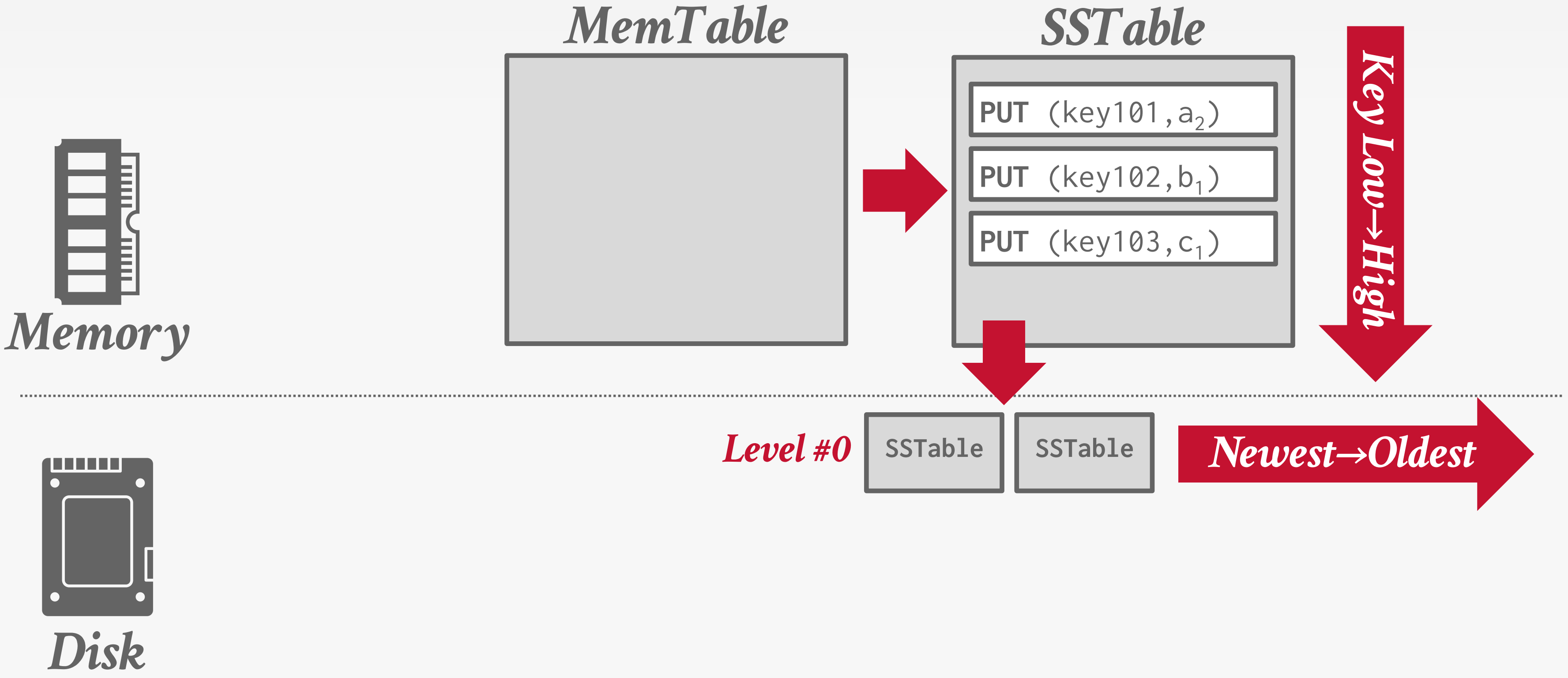
LOG-STRUCTURED STORAGE



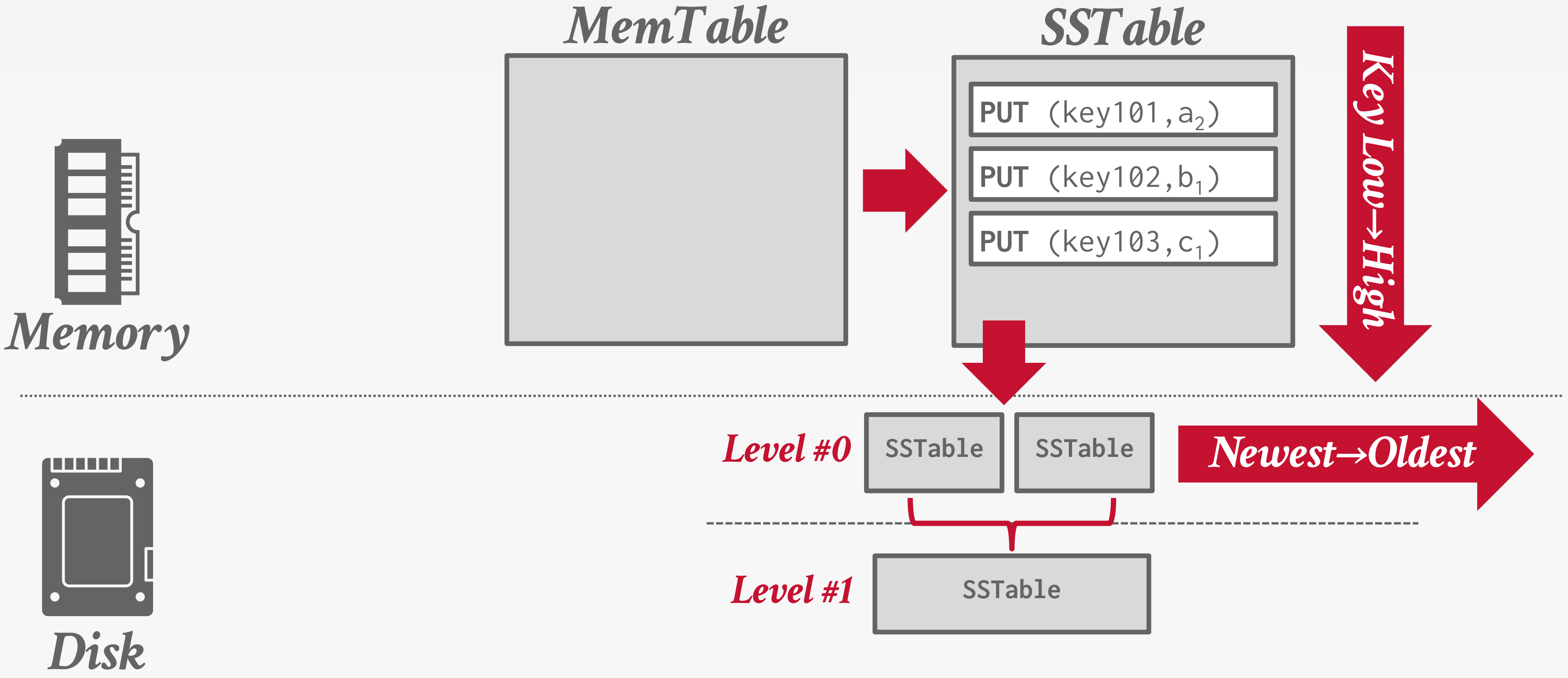
LOG-STRUCTURED STORAGE



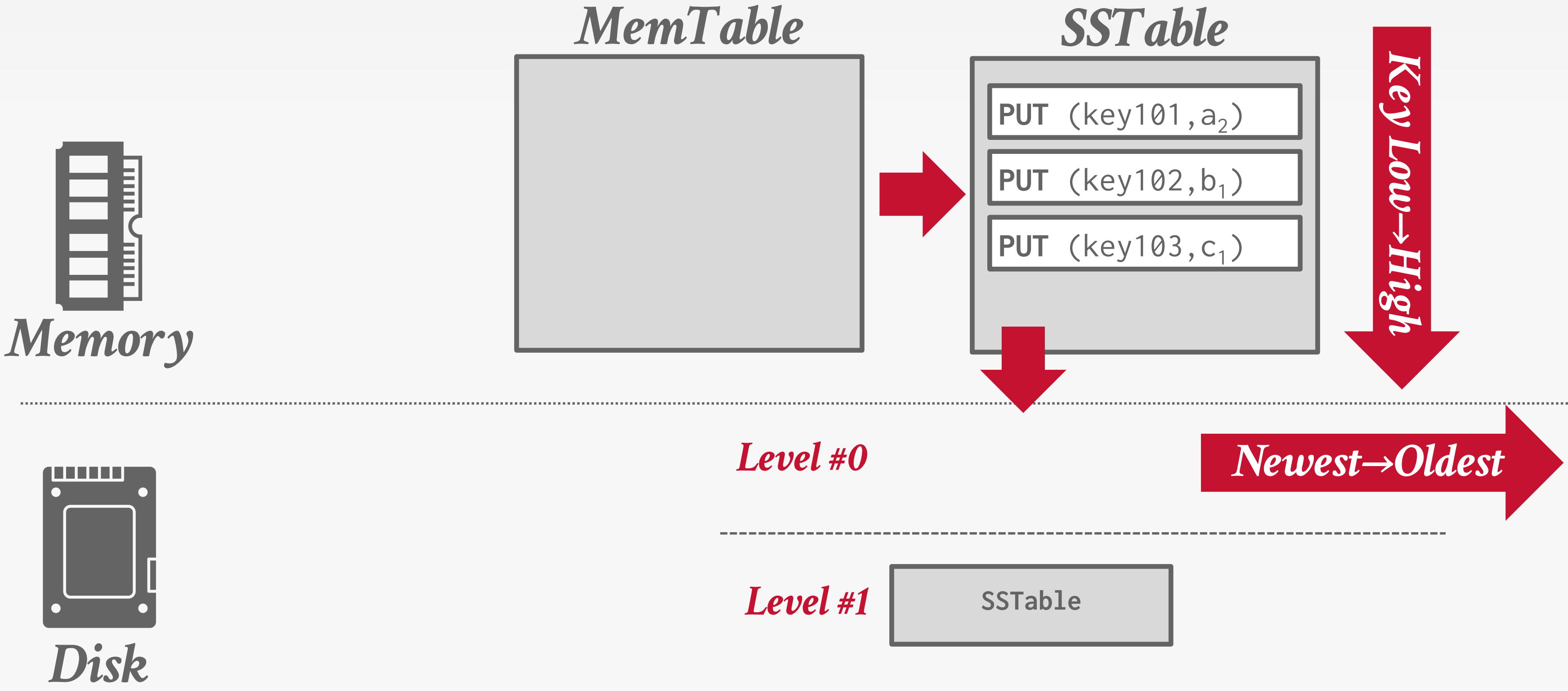
LOG-STRUCTURED STORAGE



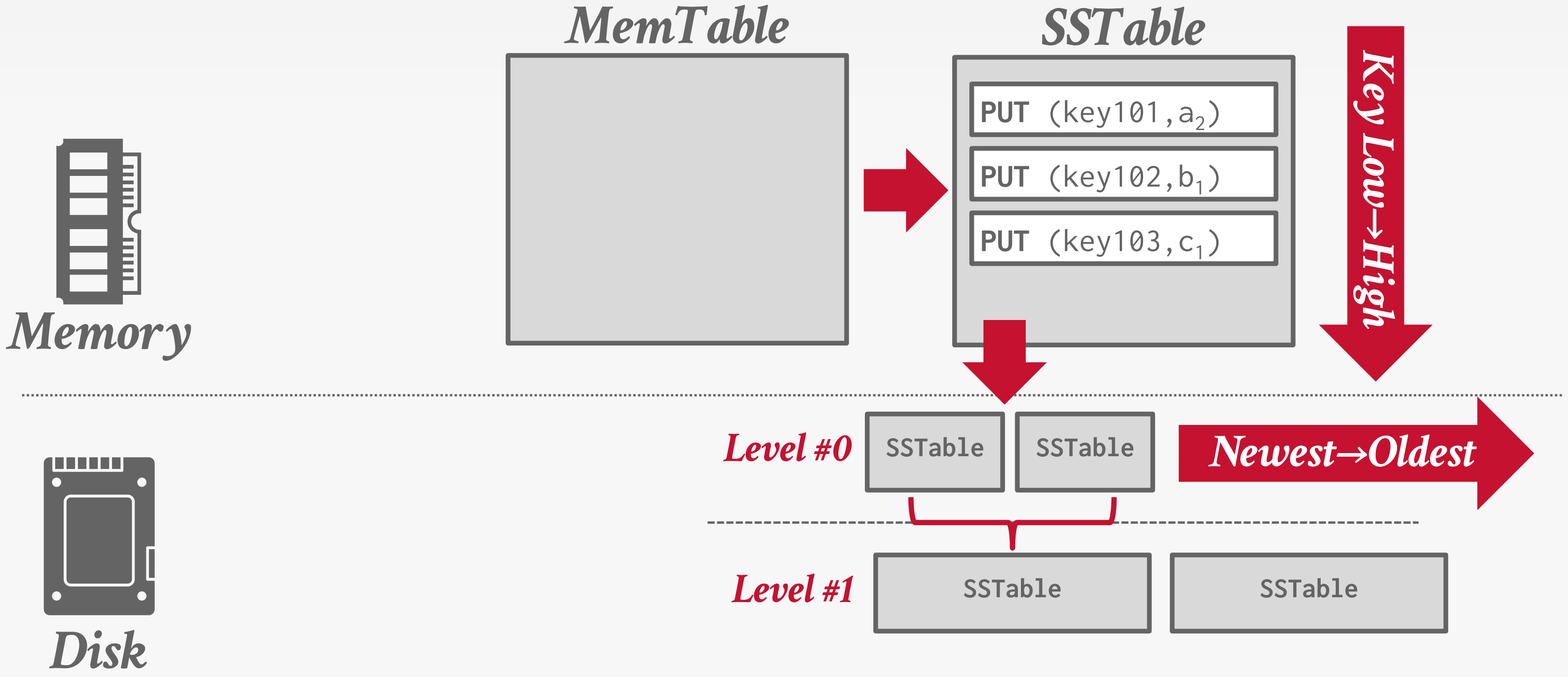
LOG-STRUCTURED STORAGE



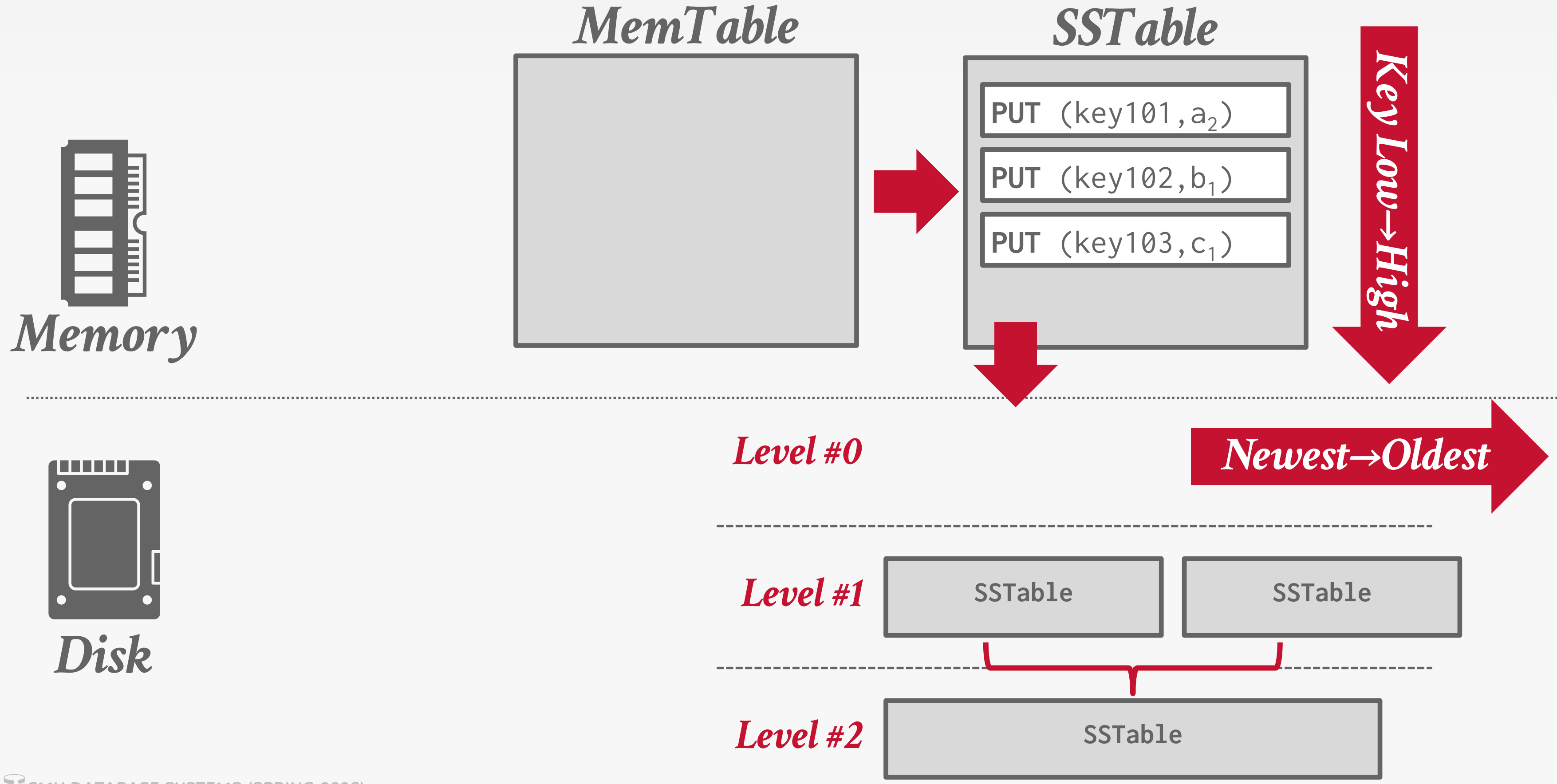
LOG-STRUCTURED STORAGE



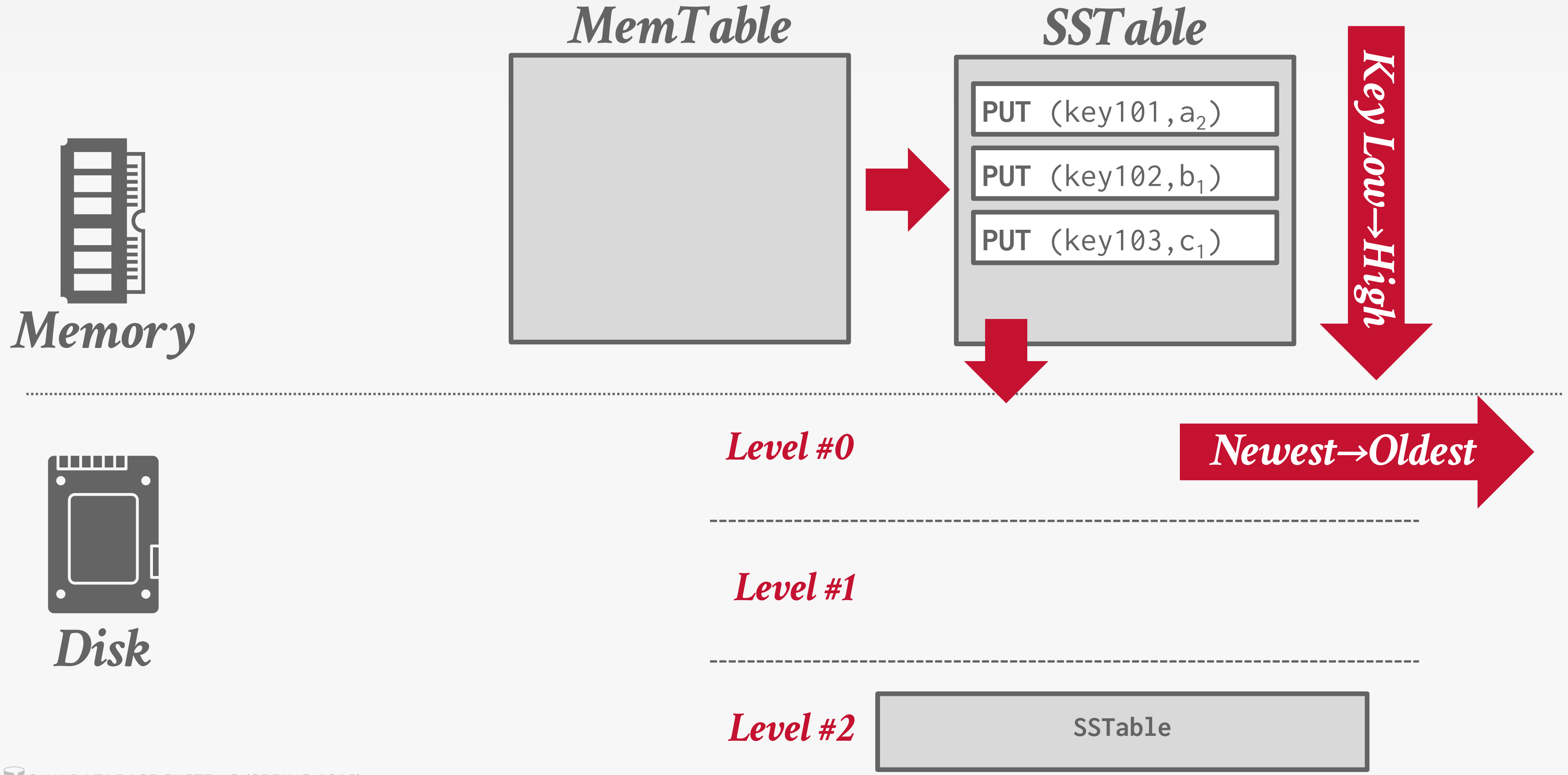
LOG-STRUCTURED STORAGE



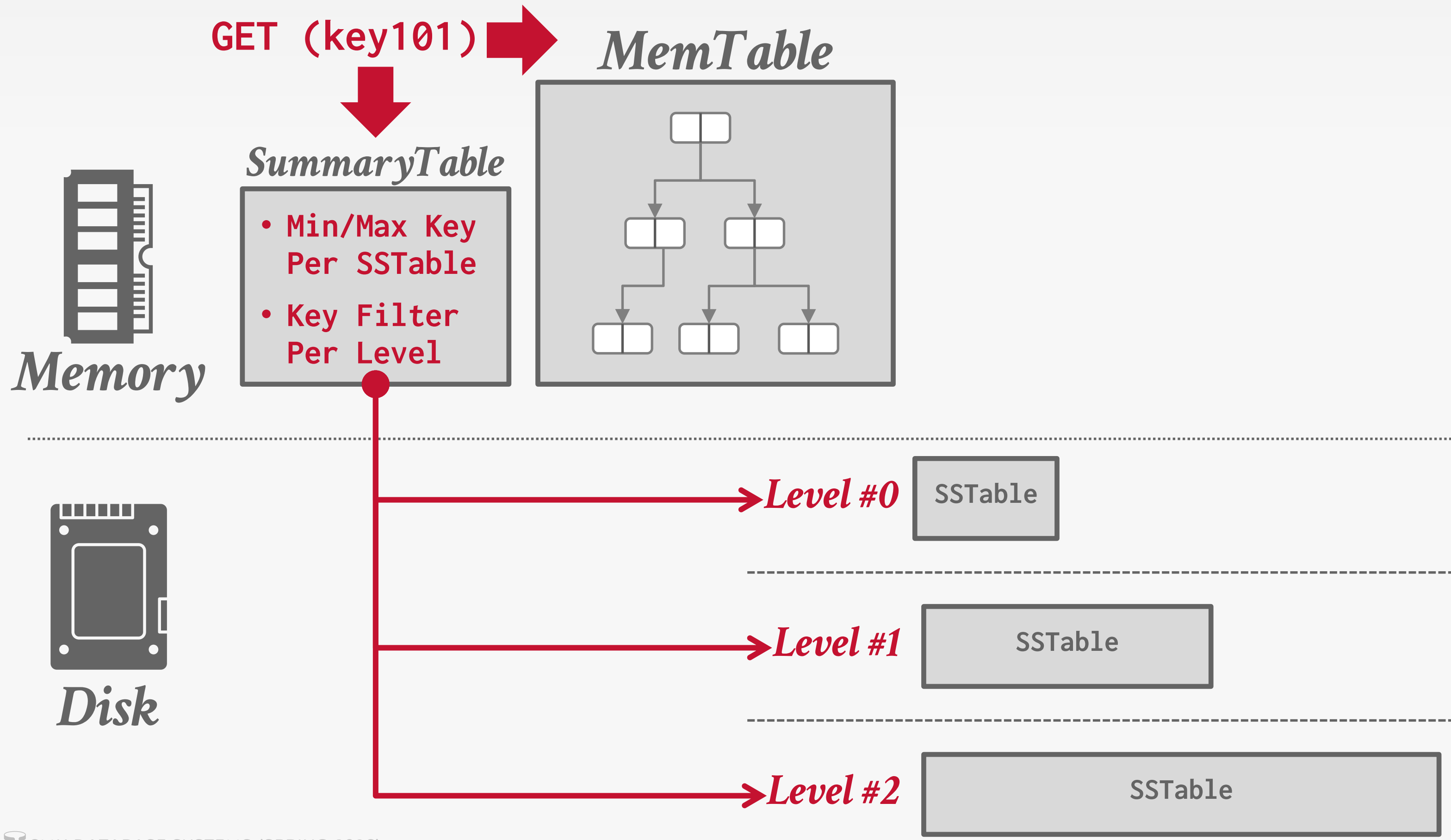
LOG-STRUCTURED STORAGE



LOG-STRUCTURED STORAGE



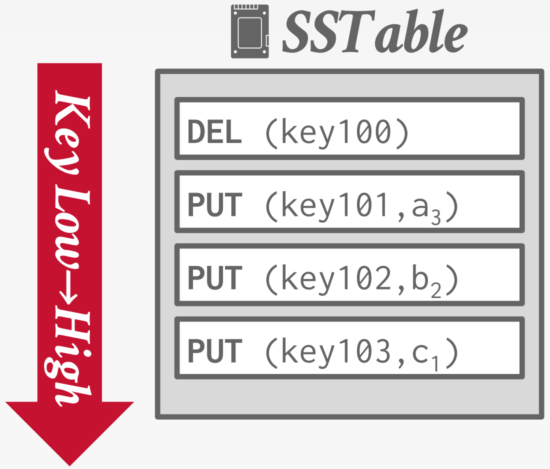
LOG-STRUCTURED STORAGE



LOG-STRUCTURED STORAGE

Key-value storage that appends log records on disk to represent changes to tuples (**PUT**, **DELETE**).
→ Each log record must contain the tuple's unique identifier.
→ Put records contain the tuple contents.
→ Deletes marks the tuple as deleted.

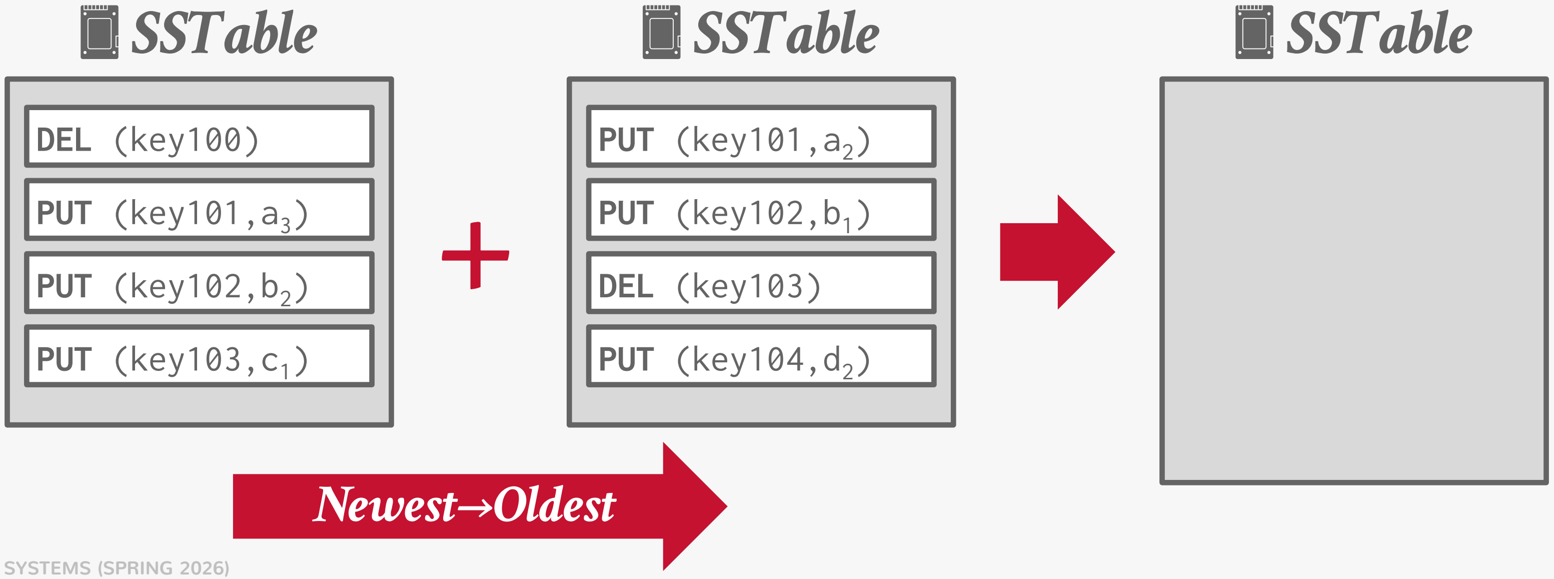
As the application makes changes to the database, the DBMS appends log records to the end of the file without checking previous log records.



LOG-STRUCTURED COMPACTION

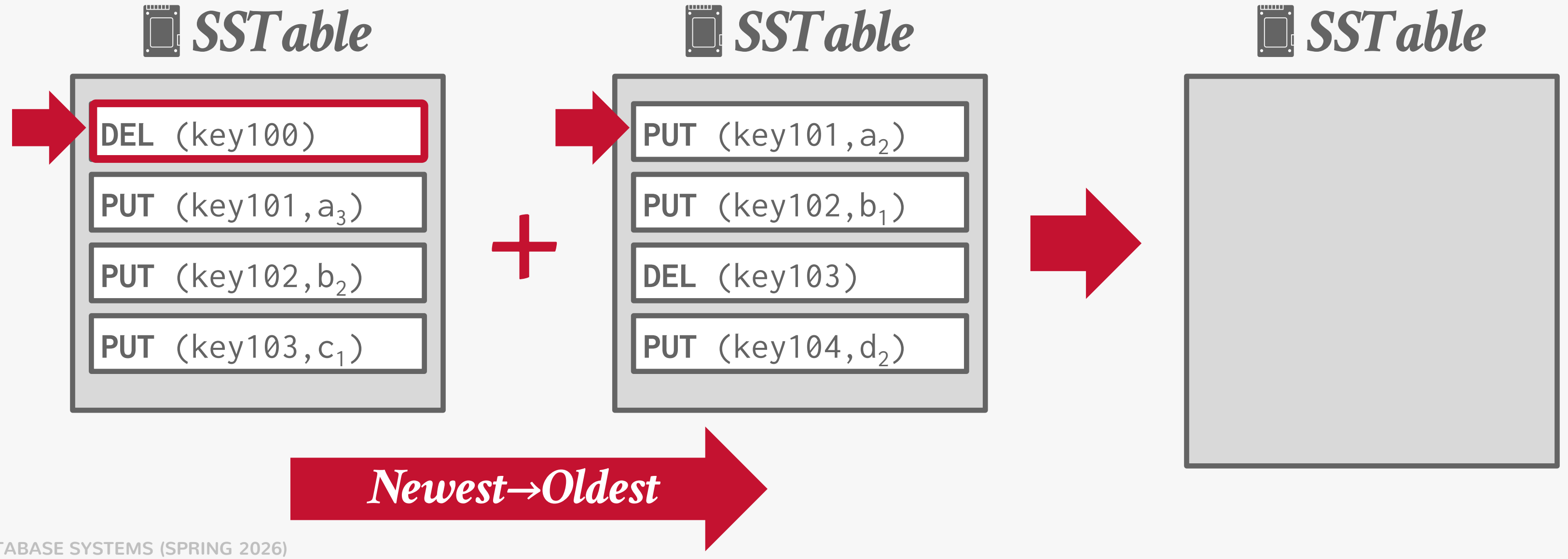
Periodically compact data files to reduce wasted space and speed up reads.

→ Keep "latest" values for each key using a sort-merge algorithm.



LOG-STRUCTURED COMPACTION

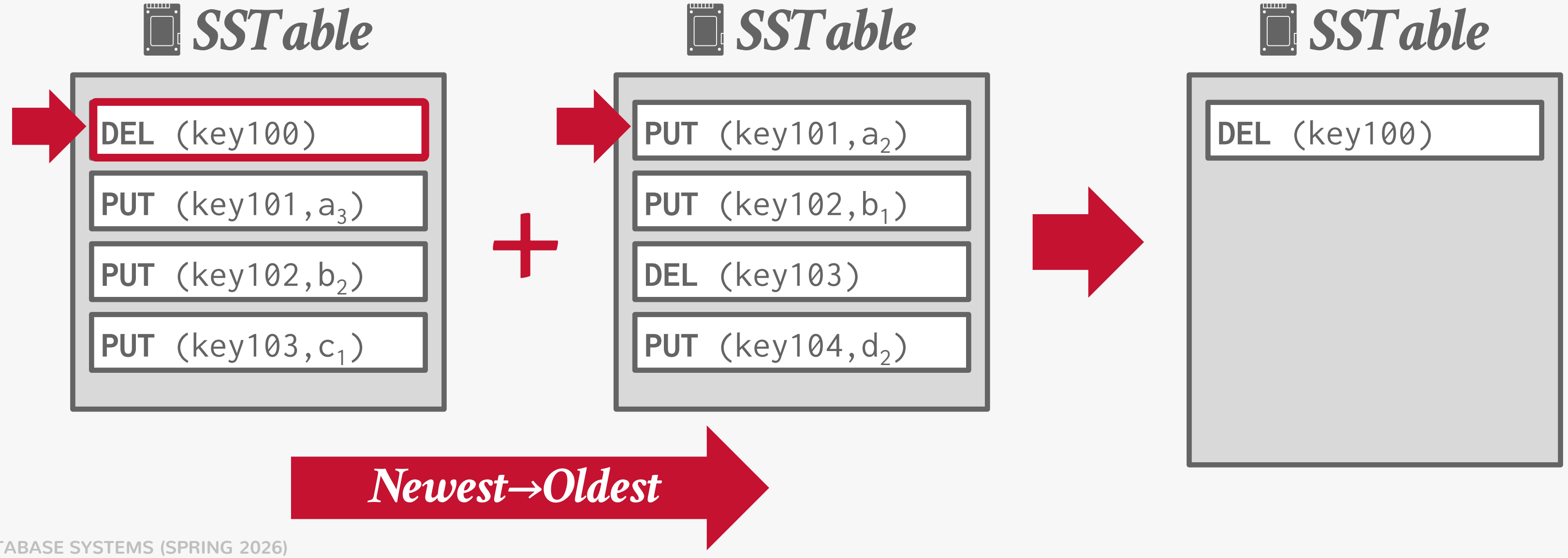
Periodically compact data files to reduce wasted space and speed up reads.
→ Keep "latest" values for each key using a sort-merge algorithm.



LOG-STRUCTURED COMPACTION

Periodically compact data files to reduce wasted space and speed up reads.

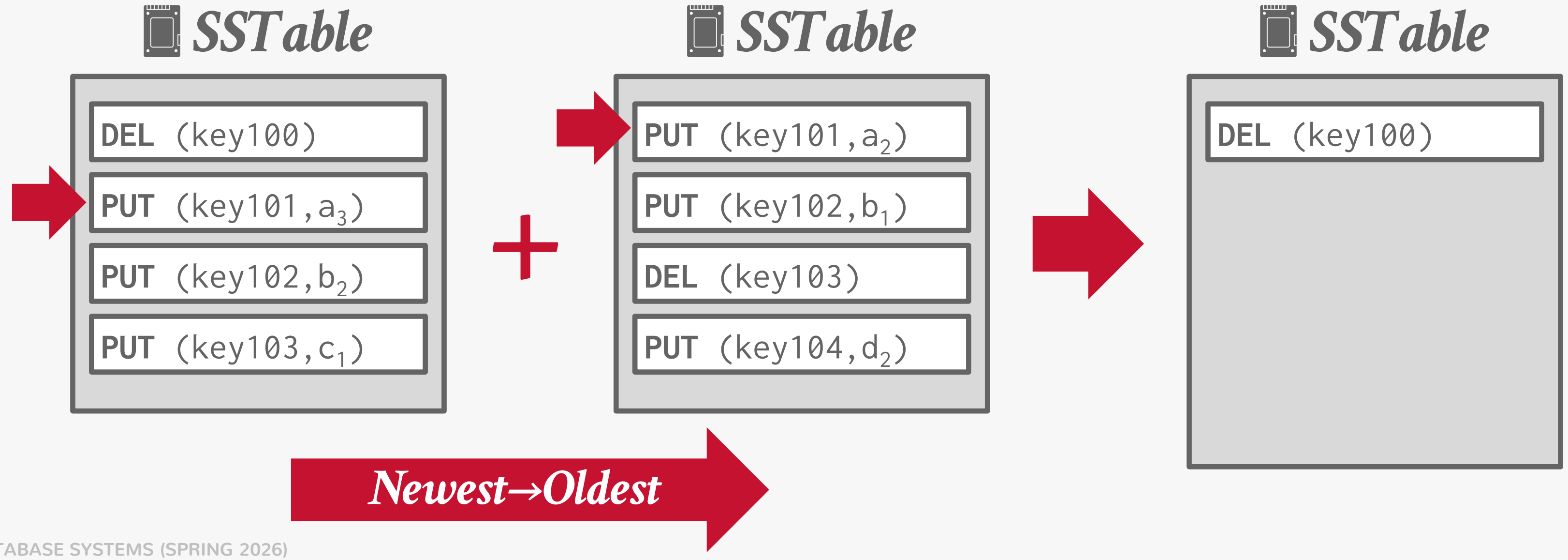
→ Keep "latest" values for each key using a sort-merge algorithm.



LOG-STRUCTURED COMPACTION

Periodically compact data files to reduce wasted space and speed up reads.

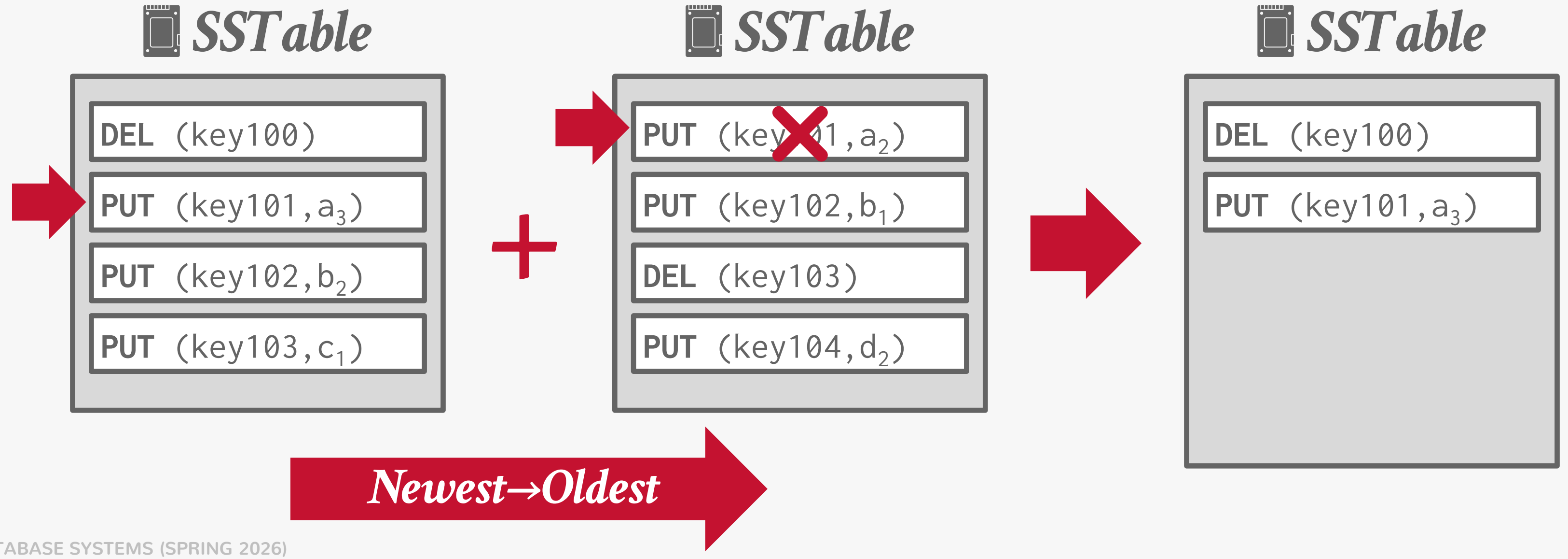
→ Keep "latest" values for each key using a sort-merge algorithm.



LOG-STRUCTURED COMPACTION

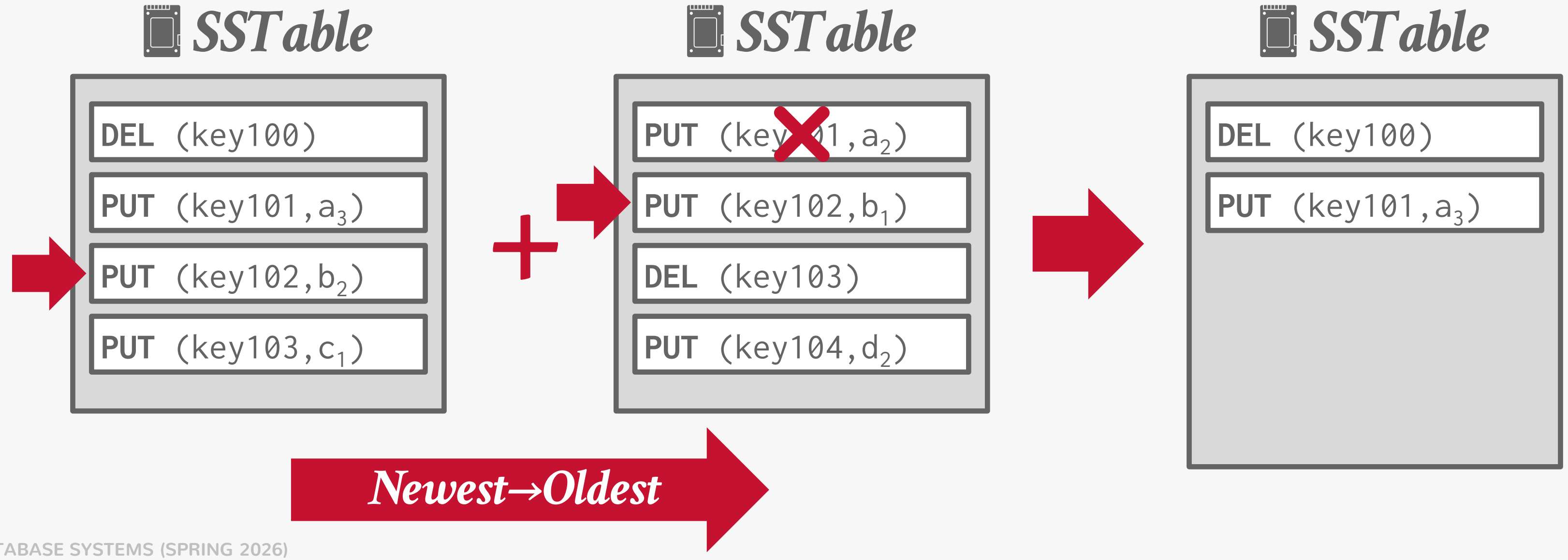
Periodically compact data files to reduce wasted space and speed up reads.

→ Keep "latest" values for each key using a sort-merge algorithm.



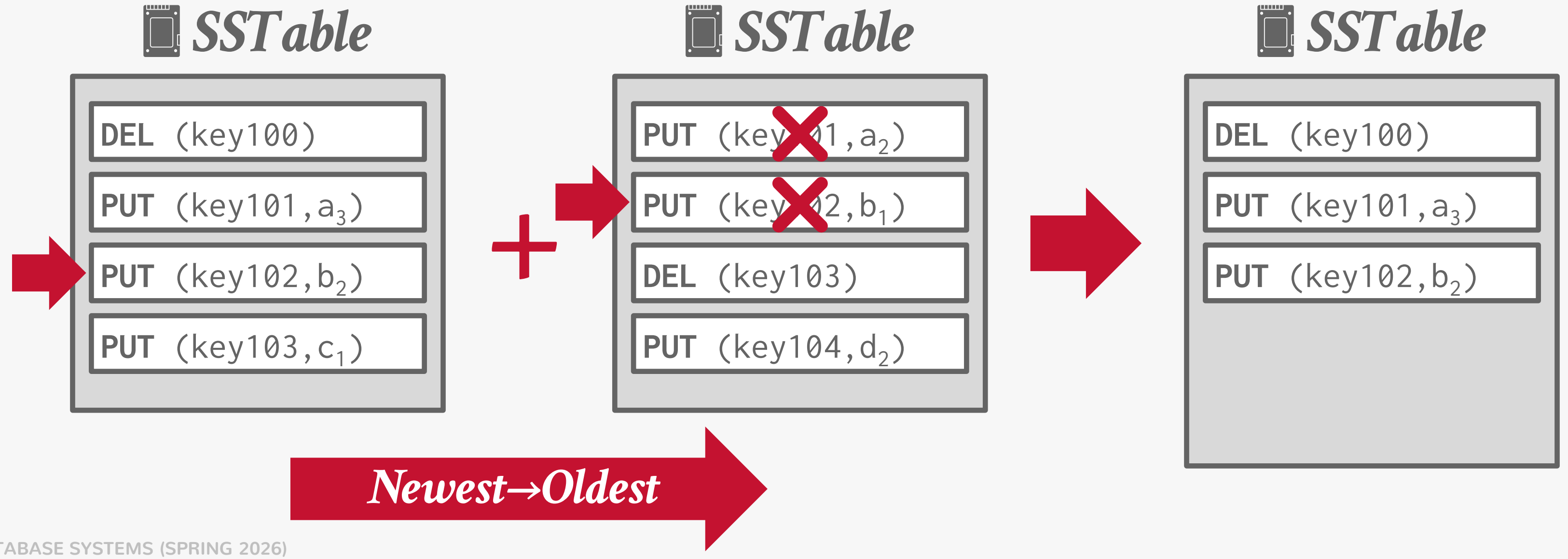
LOG-STRUCTURED COMPACTION

Periodically compact data files to reduce wasted space and speed up reads.
→ Keep "latest" values for each key using a sort-merge algorithm.



LOG-STRUCTURED COMPACTION

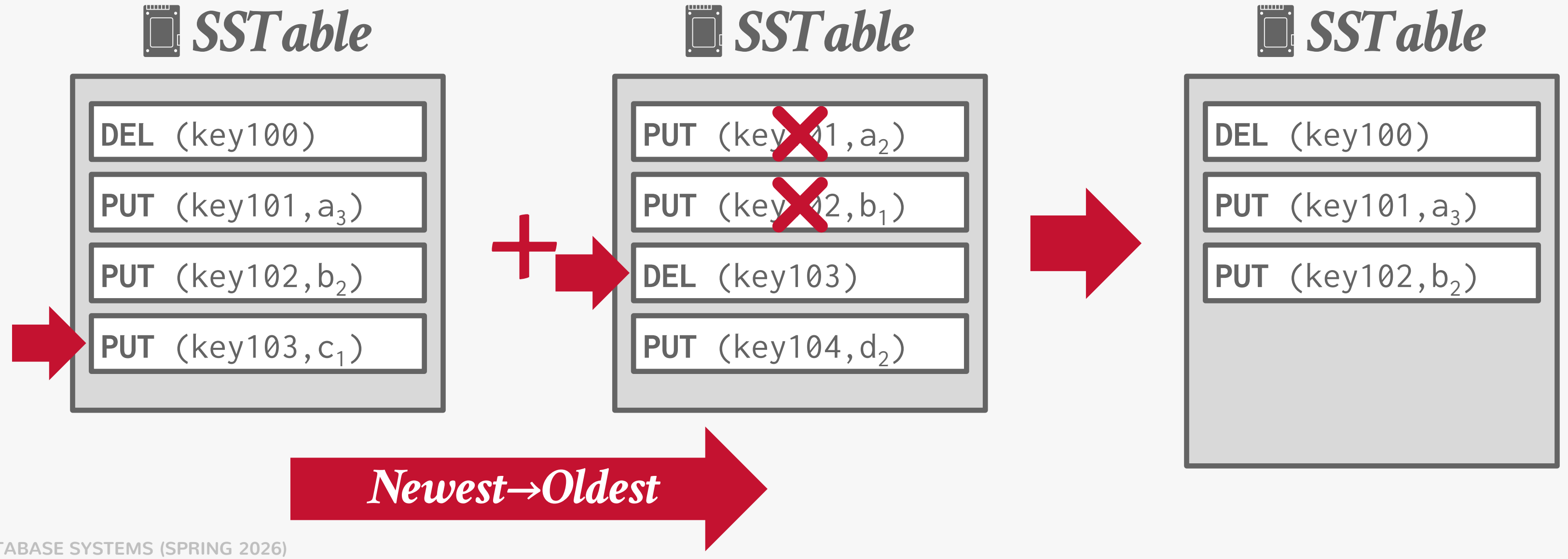
Periodically compact data files to reduce wasted space and speed up reads.
→ Keep "latest" values for each key using a sort-merge algorithm.



LOG-STRUCTURED COMPACTION

Periodically compact data files to reduce wasted space and speed up reads.

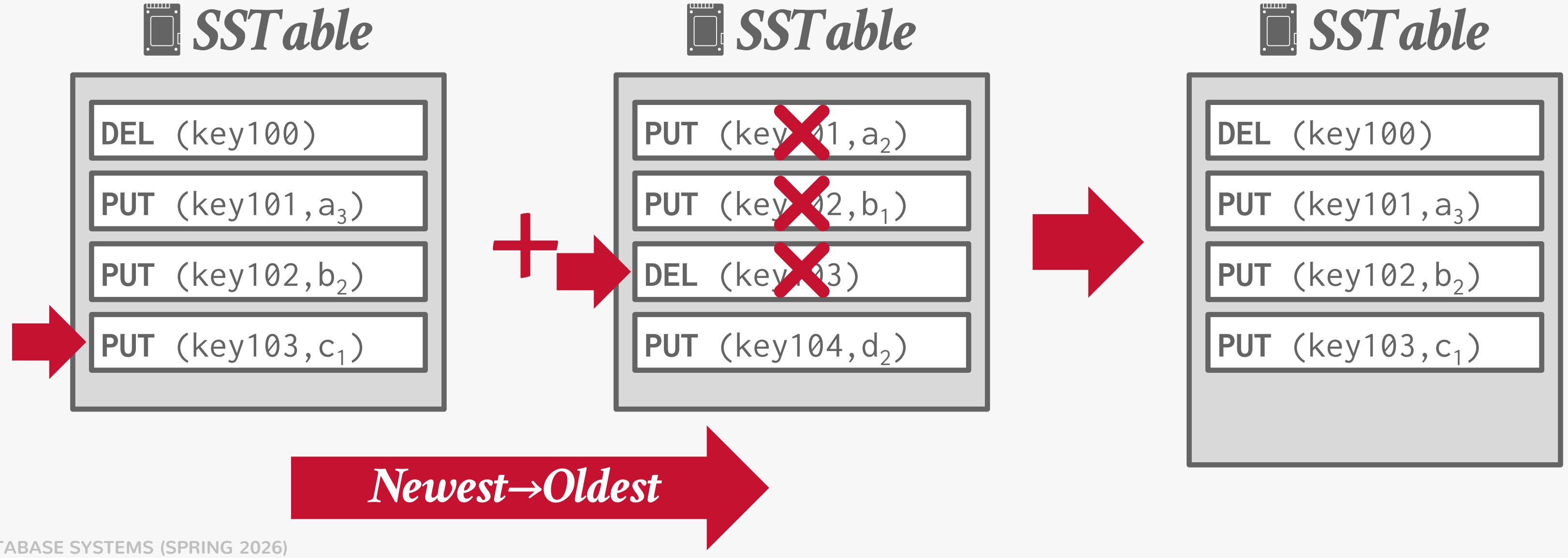
→ Keep "latest" values for each key using a sort-merge algorithm.



LOG-STRUCTURED COMPACTION

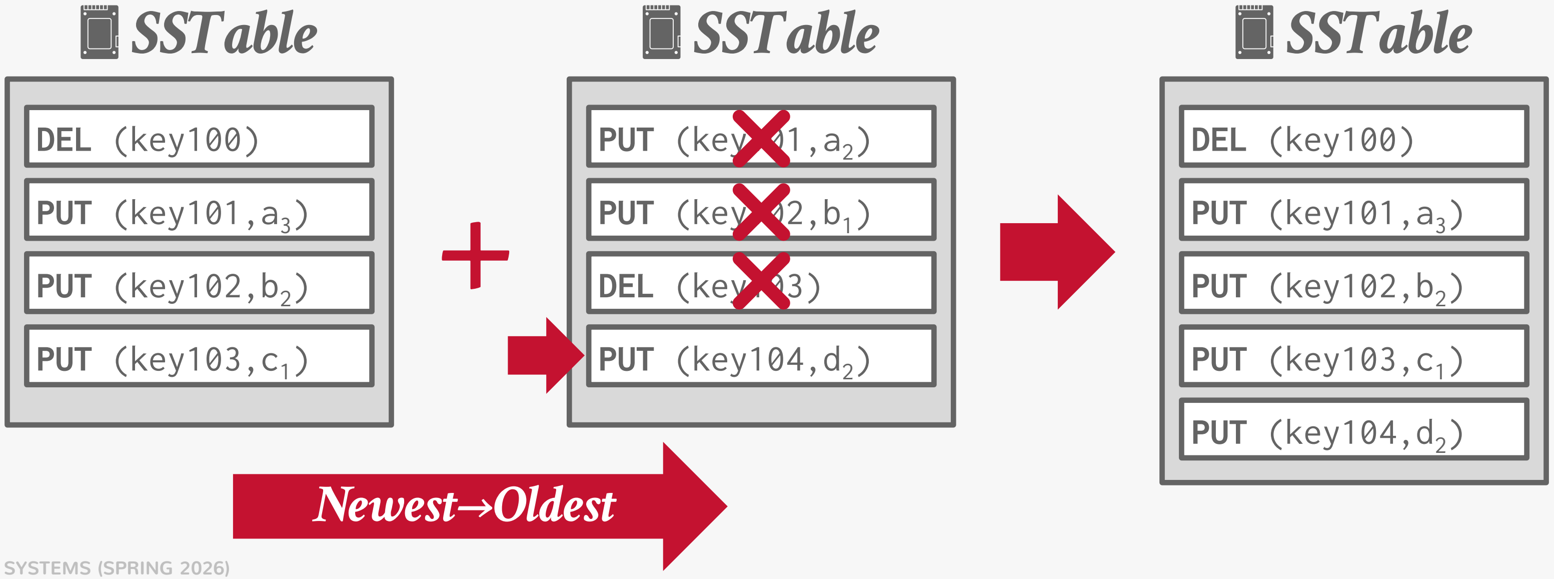
Periodically compact data files to reduce wasted space and speed up reads.

→ Keep "latest" values for each key using a sort-merge algorithm.



LOG-STRUCTURED COMPACTION

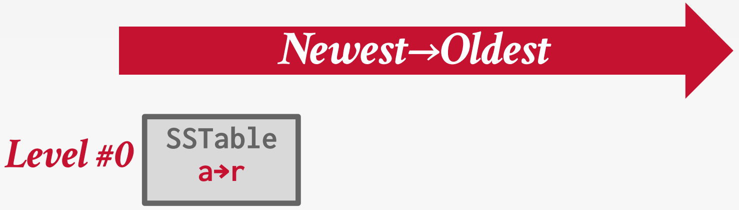
Periodically compact data files to reduce wasted space and speed up reads.
→ Keep "latest" values for each key using a sort-merge algorithm.



LOG-STRUCTURED COMPACTION STRATGEGIES

Approach #1: Leveled Compaction

→ Data is organized into levels with SSTable size limit per level.
→ SSTables in a level are non-overlapping on key ranges (except Level #0).
→ Level #0 contains SSTables recently flushed from memory and contain overlapping ranges.
→ Compactions merge a file from a level into the next lower level, maintaining sorted, non-overlapping key ranges.



Better for read-heavy workloads.

LOG-STRUCTURED COMPACTION STRATGEGIES

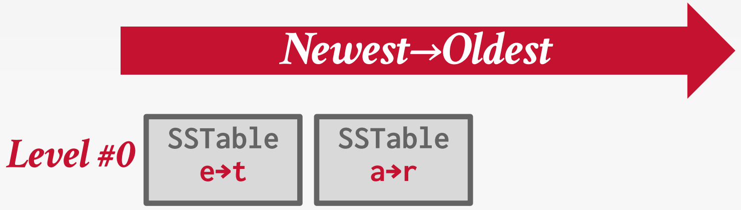
Approach #1: Leveled Compaction

→ Data is organized into levels with SSTable size limit per level.

→ SSTables in a level are non-overlapping on key ranges (except Level #0).

→ Level #0 contains SSTables recently flushed from memory and contain overlapping ranges.

→ Compactions merge a file from a level into the next lower level, maintaining sorted, non-overlapping key ranges.



Better for read-heavy workloads.

LOG-STRUCTURED COMPACTION STRATGEGIES

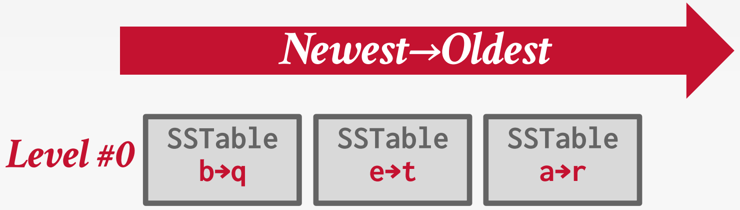
Approach #1: Leveled Compaction

→ Data is organized into levels with SSTable size limit per level.

→ SSTables in a level are non-overlapping on key ranges (except Level #0).

→ Level #0 contains SSTables recently flushed from memory and contain overlapping ranges.

→ Compactions merge a file from a level into the next lower level, maintaining sorted, non-overlapping key ranges.



Better for read-heavy workloads.

LOG-STRUCTURED COMPACTION STRATGEGIES

Approach #1: Leveled Compaction

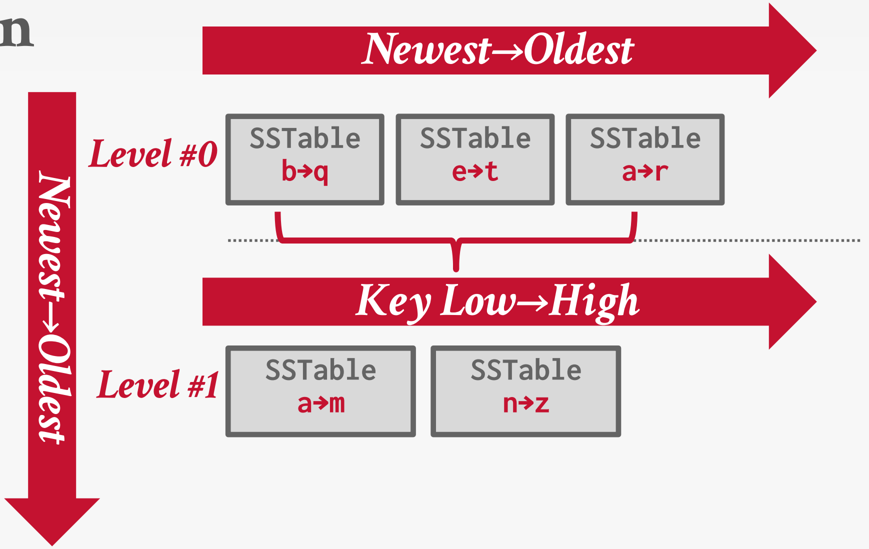
→ Data is organized into levels with SSTable size limit per level.

→ SSTables in a level are non-overlapping on key ranges (except Level #0).

→ Level #0 contains SSTables recently flushed from memory and contain overlapping ranges.

→ Compactions merge a file from a level into the next lower level, maintaining sorted, non-overlapping key ranges.

Better for read-heavy workloads.



LOG-STRUCTURED COMPACTION STRATGEGIES

Approach #1: Leveled Compaction

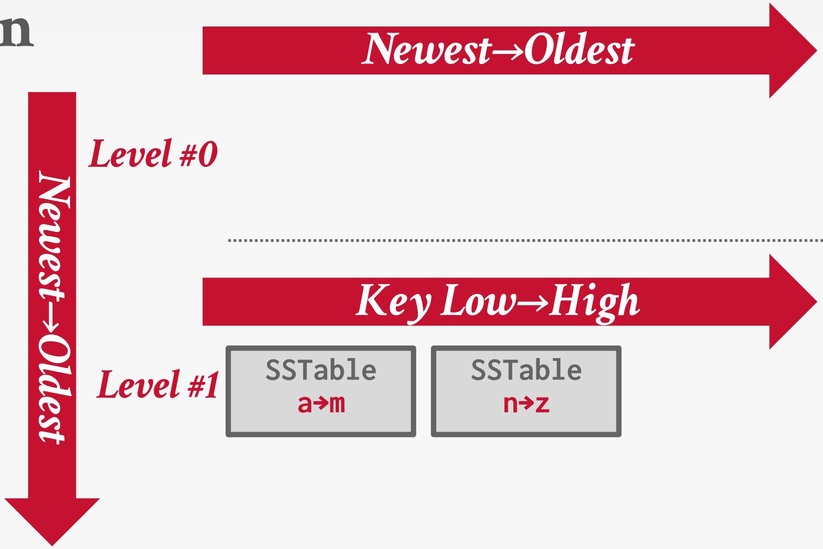
→ Data is organized into levels with SSTable size limit per level.

→ SSTables in a level are non-overlapping on key ranges (except Level #0).

→ Level #0 contains SSTables recently flushed from memory and contain overlapping ranges.

→ Compactions merge a file from a level into the next lower level, maintaining sorted, non-overlapping key ranges.

Better for read-heavy workloads.



LOG-STRUCTURED COMPACTION STRATGEGIES

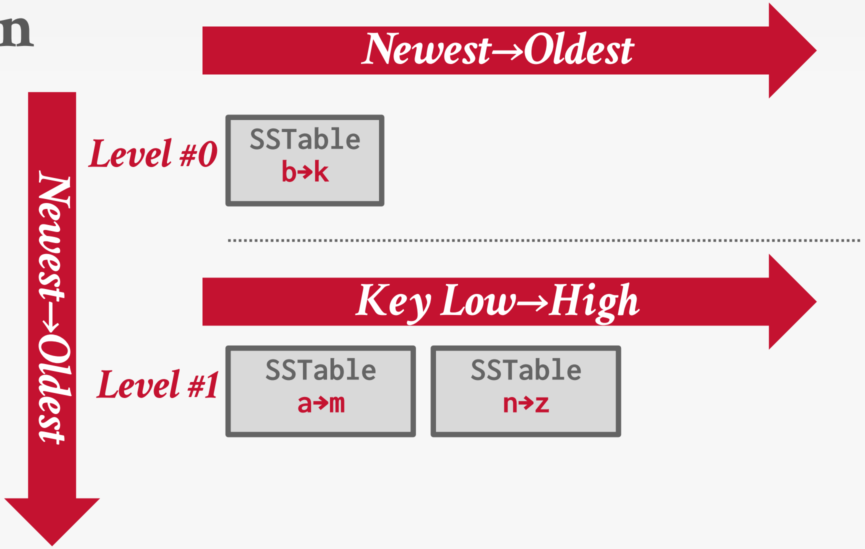
Approach #1: Leveled Compaction

→ Data is organized into levels with SSTable size limit per level.

→ SSTables in a level are non-overlapping on key ranges (except Level #0).

→ Level #0 contains SSTables recently flushed from memory and contain overlapping ranges.

→ Compactions merge a file from a level into the next lower level, maintaining sorted, non-overlapping key ranges.



Better for read-heavy workloads.

LOG-STRUCTURED COMPACTION STRATGEGIES

Approach #1: Leveled Compaction

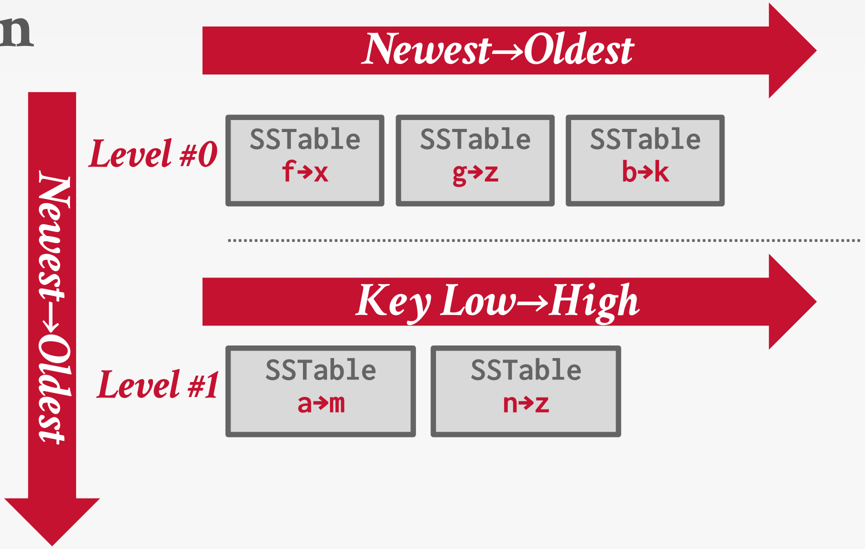
→ Data is organized into levels with SSTable size limit per level.

→ SSTables in a level are non-overlapping on key ranges (except Level #0).

→ Level #0 contains SSTables recently flushed from memory and contain overlapping ranges.

→ Compactions merge a file from a level into the next lower level, maintaining sorted, non-overlapping key ranges.

Better for read-heavy workloads.



LOG-STRUCTURED COMPACTION STRATGEGIES

Approach #1: Leveled Compaction

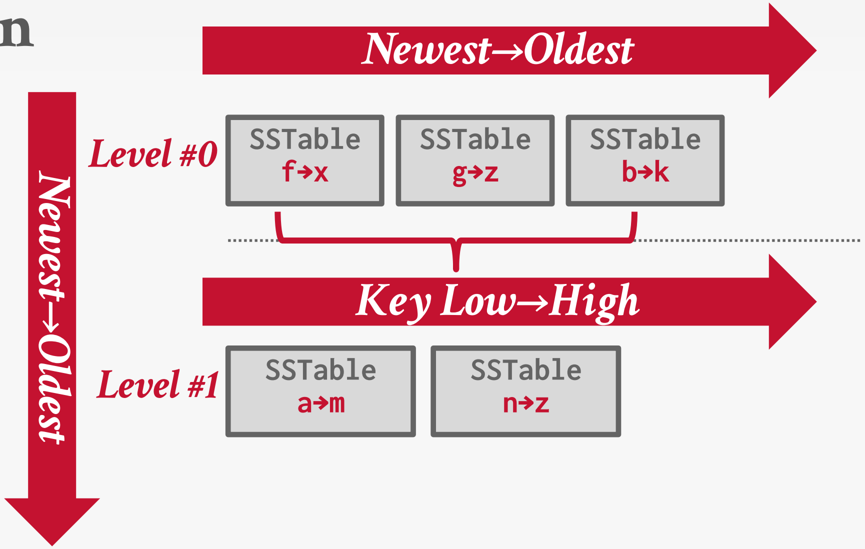
→ Data is organized into levels with SSTable size limit per level.

→ SSTables in a level are non-overlapping on key ranges (except Level #0).

→ Level #0 contains SSTables recently flushed from memory and contain overlapping ranges.

→ Compactions merge a file from a level into the next lower level, maintaining sorted, non-overlapping key ranges.

Better for read-heavy workloads.



LOG-STRUCTURED COMPACTION STRATGEGIES

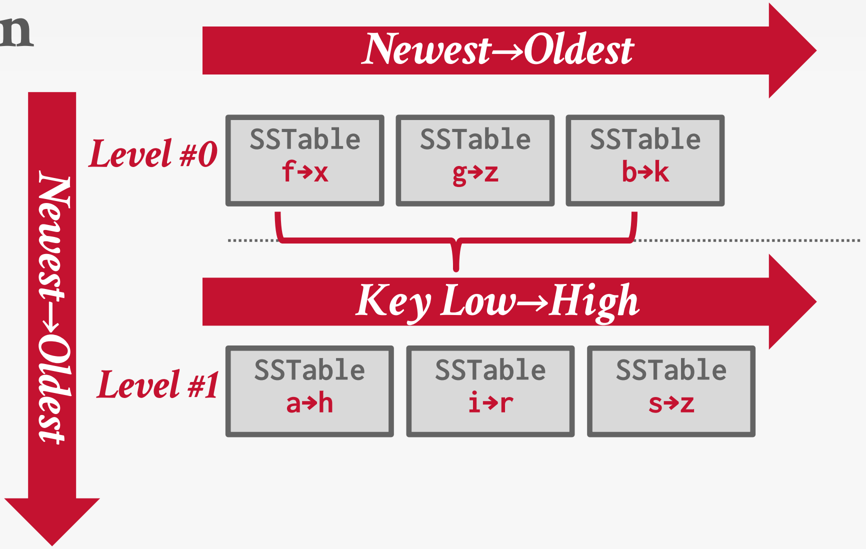
Approach #1: Leveled Compaction

→ Data is organized into levels with SSTable size limit per level.

→ SSTables in a level are non-overlapping on key ranges (except Level #0).

→ Level #0 contains SSTables recently flushed from memory and contain overlapping ranges.

→ Compactions merge a file from a level into the next lower level, maintaining sorted, non-overlapping key ranges.



Better for read-heavy workloads.

LOG-STRUCTURED COMPACTION STRATGEGIES

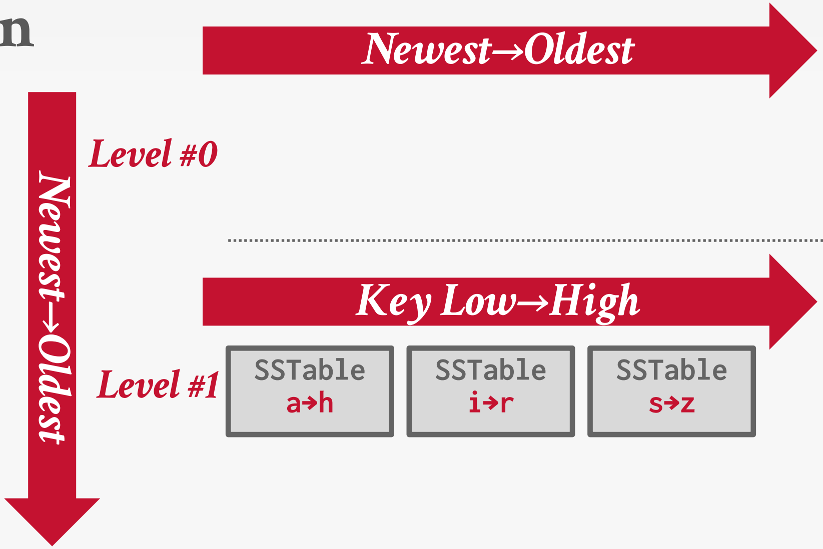
Approach #1: Leveled Compaction

→ Data is organized into levels with SSTable size limit per level.

→ SSTables in a level are non-overlapping on key ranges (except Level #0).

→ Level #0 contains SSTables recently flushed from memory and contain overlapping ranges.

→ Compactions merge a file from a level into the next lower level, maintaining sorted, non-overlapping key ranges.



Better for read-heavy workloads.

LOG-STRUCTURED COMPACTION STRATGEGIES

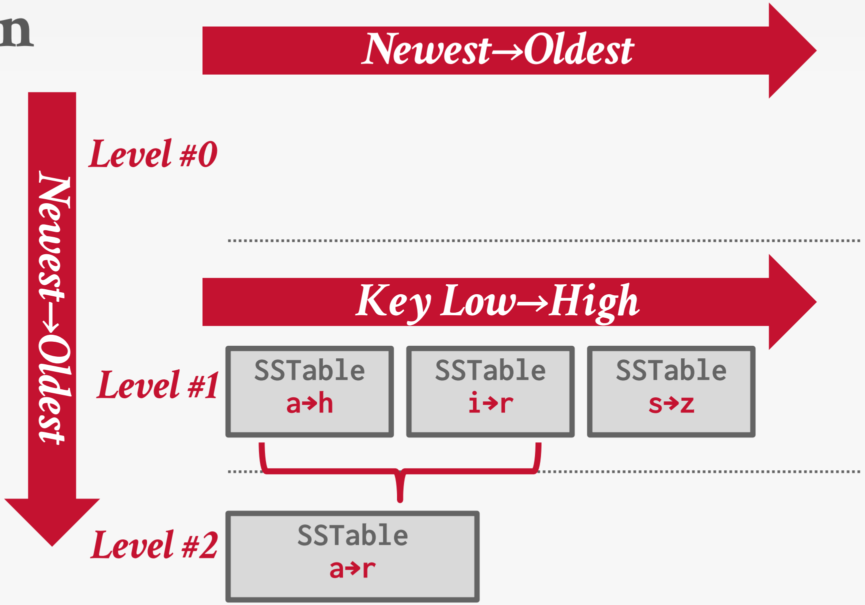
Approach #1: Leveled Compaction

→ Data is organized into levels with SSTable size limit per level.

→ SSTables in a level are non-overlapping on key ranges (except Level #0).

→ Level #0 contains SSTables recently flushed from memory and contain overlapping ranges.

→ Compactions merge a file from a level into the next lower level, maintaining sorted, non-overlapping key ranges.



Better for read-heavy workloads.

LOG-STRUCTURED COMPACTION STRATGEGIES

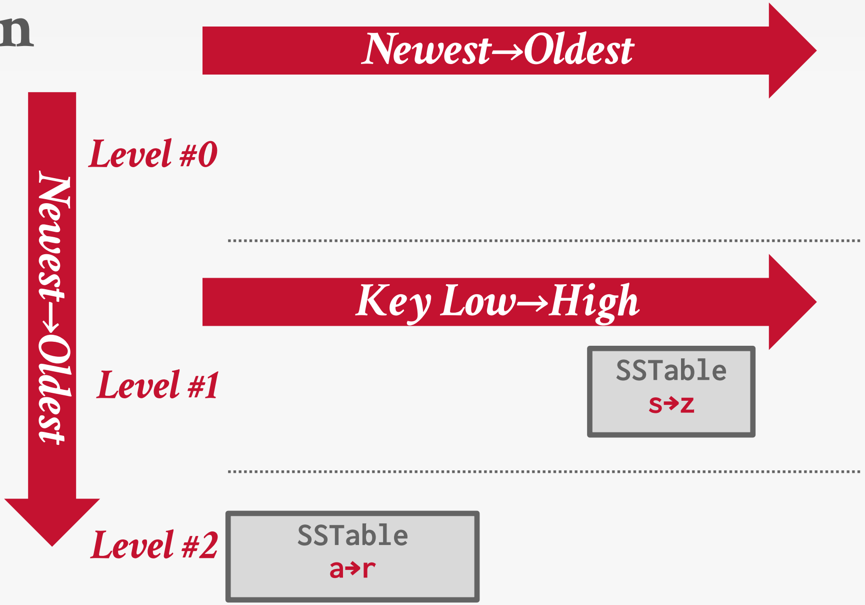
Approach #1: Leveled Compaction

→ Data is organized into levels with SSTable size limit per level.

→ SSTables in a level are non-overlapping on key ranges (except Level #0).

→ Level #0 contains SSTables recently flushed from memory and contain overlapping ranges.

→ Compactions merge a file from a level into the next lower level, maintaining sorted, non-overlapping key ranges.



Better for read-heavy workloads.

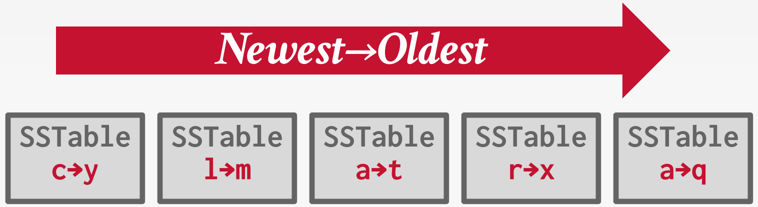
LOG-STRUCTURED COMPACTION STRATGEGIES

Approach #1: Universal Compaction

→ SSTables reside in a single "universal" level (i.e., no multi-level hierarchy).

→ DBMS triggers compaction when too many SSTables overlap in key ranges or exceed size thresholds.

Better for insert-heavy workloads and time-oriented queries

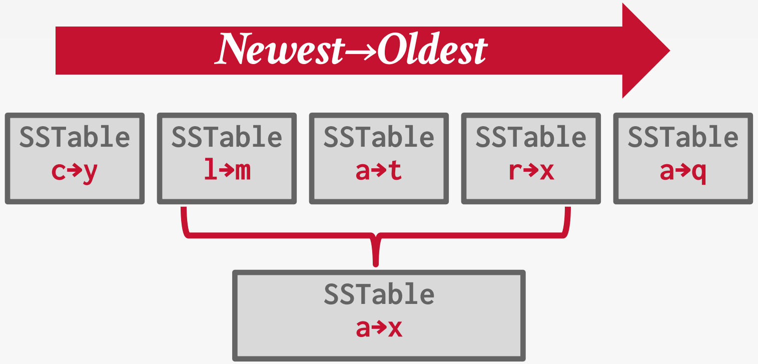


LOG-STRUCTURED COMPACTION STRATGEGIES

Approach #1: Universal Compaction

→ SSTables reside in a single "universal" level (i.e., no multi-level hierarchy).

→ DBMS triggers compaction when too many SSTables overlap in key ranges or exceed size thresholds.



Better for insert-heavy workloads and time-oriented queries

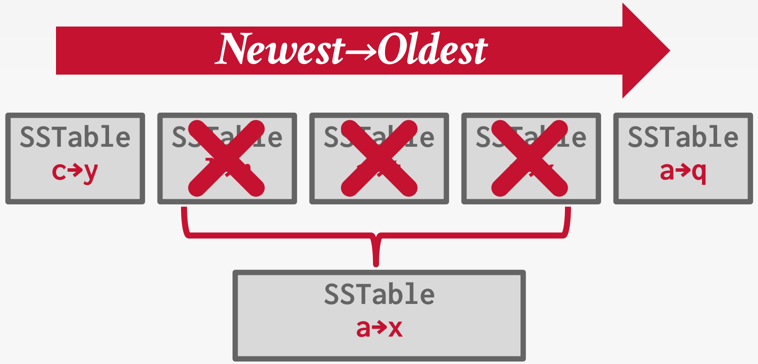
LOG-STRUCTURED COMPACTION STRATGEGIES

Approach #1: Universal Compaction

→ SSTables reside in a single "universal" level (i.e., no multi-level hierarchy).

→ DBMS triggers compaction when too many SSTables overlap in key ranges or exceed size thresholds.

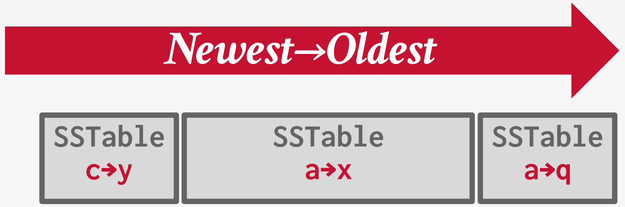
Better for insert-heavy workloads and time-oriented queries



LOG-STRUCTURED COMPACTION STRATGEGIES

Approach #1: Universal Compaction
→ SSTables reside in a single "universal" level (i.e., no multi-level hierarchy).
→ DBMS triggers compaction when too many SSTables overlap in key ranges or exceed size thresholds.

Better for insert-heavy workloads and time-oriented queries

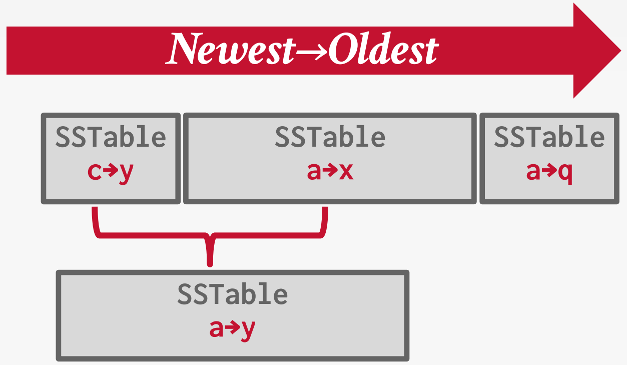


LOG-STRUCTURED COMPACTION STRATGEGIES

Approach #1: Universal Compaction

→ SSTables reside in a single "universal" level (i.e., no multi-level hierarchy).

→ DBMS triggers compaction when too many SSTables overlap in key ranges or exceed size thresholds.



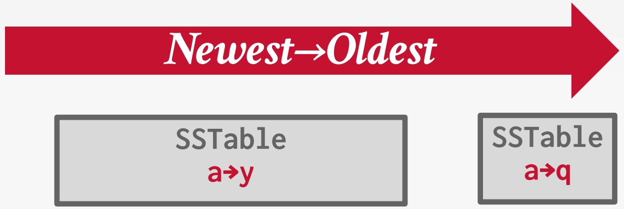
Better for insert-heavy workloads and time-oriented queries

LOG-STRUCTURED COMPACTION STRATGEGIES

Approach #1: Universal Compaction

→ SSTables reside in a single "universal" level (i.e., no multi-level hierarchy).
→ DBMS triggers compaction when too many SSTables overlap in key ranges or exceed size thresholds.

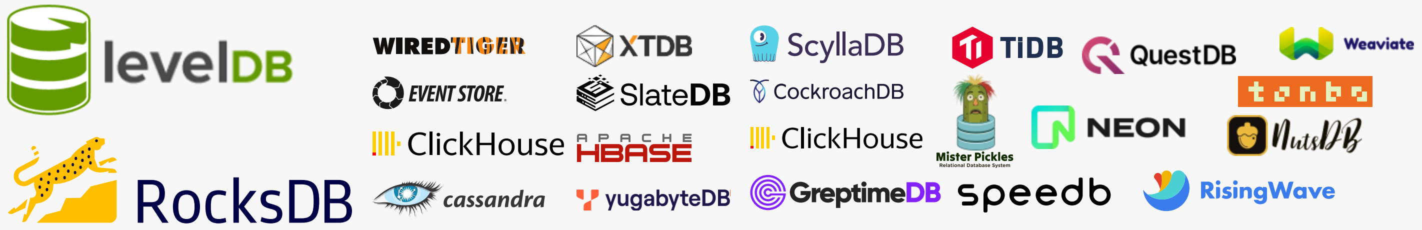
Better for insert-heavy workloads and time-oriented queries



DISCUSSION

Log-structured storage managers are more common today than in previous decades.

→ This is partly due to the proliferation of RocksDB.



What are some downsides of this approach?

→ Write-Amplification.

→ Compaction is expensive.

SYSTEM CATALOGS

A DBMS stores meta-data about databases in its internal catalogs.
→ Tables, columns, indexes, views
→ Users, permissions
→ Internal statistics

Almost every DBMS stores the database's catalog inside itself (i.e., as tables).
→ Wrap object abstraction around tuples.
→ Specialized code for "bootstrapping" catalog tables.

SYSTEM CATALOGS

You can query the DBMS's internal **INFORMATION_SCHEMA** catalog to get info about the database.

→ ANSI standard set of read-only views that provide info about all the tables, views, columns, and procedures in a database

DBMSs also have non-standard shortcuts to retrieve this information.

ACCESSING TABLE SCHEMA

List all the tables in the current database:

SQL-92
```
SELECT *
  FROM INFORMATION_SCHEMA.TABLES
 WHERE table_catalog = '<db name>';
```

Postgres
```
\d;
```

MySQL
```
SHOW TABLES;
```

SQLite
```
.tables
```

ACCESSING TABLE SCHEMA

List all the tables in the student table:

SQL-92

```
SELECT *
  FROM INFORMATION_SCHEMA.TABLES
 WHERE table_name = 'student'
```

Postgres

```
\d student;
```

MySQL

```
DESCRIBE student;
```

SQLite

```
.schema student
```

SCHEMA CHANGES

ADD COLUMN:
- → **Slow**: Copy tuples into new pages and modify to add column.
- → **Fast**: Record meta-data on default value and incrementally update tuples to include new column.

DROP COLUMN:
- → Mark column as "deprecated", clean up later.
- → New tuples omit the dropped column.

CHANGE COLUMN:
- → **Rename**: Update meta-data in catalog.
- → **Type**: Check whether new type is binary compatible with existing values. If no, rewrite tuples.
- → **Constraints**: Scan entire table to see if it violates constraint.
- → **Default**: Update meta-data in catalog.

INDEXES

CREATE INDEX:
- → Scan the entire table and populate the index.
- → Must record changes made by txns that modified the table while another txn was building the index.
- → When the scan completes, lock the table and resolve changes that were missed after the scan started.

DROP INDEX:
- → Drop the index logically from the catalog.
- → It only becomes "invisible" when the txn that dropped it commits. All existing txns will still have to update it.

CONCLUSION

Log-structured storage is an alternative approach to the tuple-oriented architecture.

→ Ideal for write-heavy workloads because it maximizes sequential disk I/O.

The storage manager is not entirely independent from the rest of the DBMS.

NEXT CLASS

Breaking your preconceived notion that a DBMS stores everything as rows…