Carnegie Mellon University

Database Systems

Database Storage: *Tuple Organization*

15-445/645 FALL 2024 » PROF. ANDY PAVLO

ADMINISTRIVIA

~~**Homework #1** is due September 8^{th} @ 11:59pm~~

~~**Project #0** is due September 8^{th} @ 11:59pm~~

Project #1 will be released on September 10^{th}

UPCOMING DATABASE TALKS

Databricks
- → Tuesday Sept 10th @ 6:00pm
- → GHC 4401

Snowflake
- → Thursday Sept 12th @ 12:00pm
- → GHC 9115

Apache DataFusion (DB Seminar)
- → Monday Sept 23rd @ 4:30pm
- → Zoom

UPCOMING DATABASE EVENTS

CMU-DB Industry Affiliates Retreat

→ Monday Sept 16th: Research Talks + Poster Session
→ Tuesday Sept 17th: Company Info Sessions
→ All events are open to the public.

Sign-up for Company Info Sessions (**@61**)

Add your Resume if You Want to Make $$$ (**@92**)

LAST CLASS

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

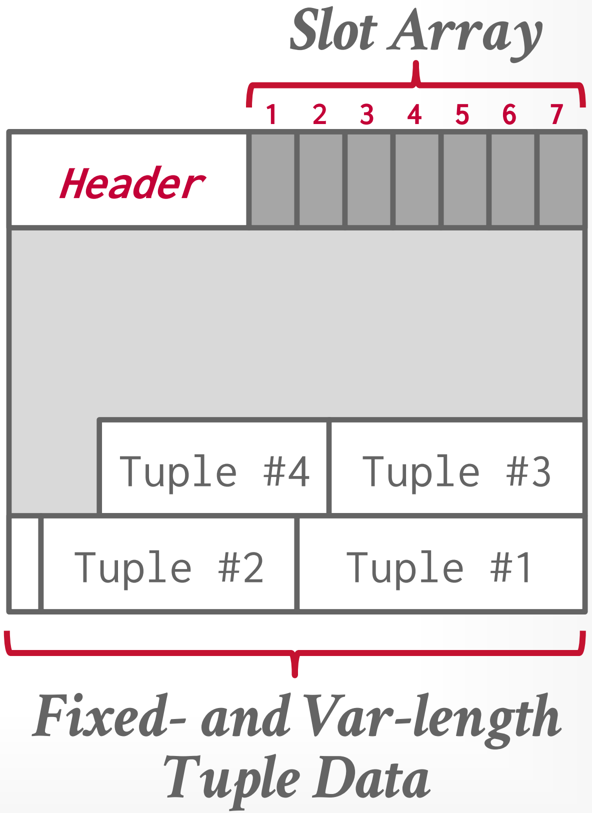
We then discussed a page-oriented storage scheme for organizing tuples across heap files.

SLOTTED PAGES

The most common 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.

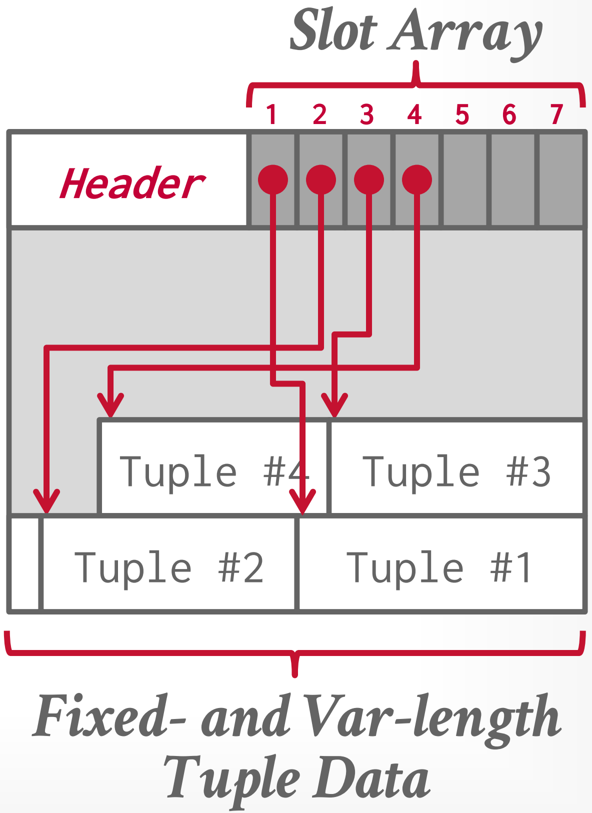


SLOTTED PAGES

The most common 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.

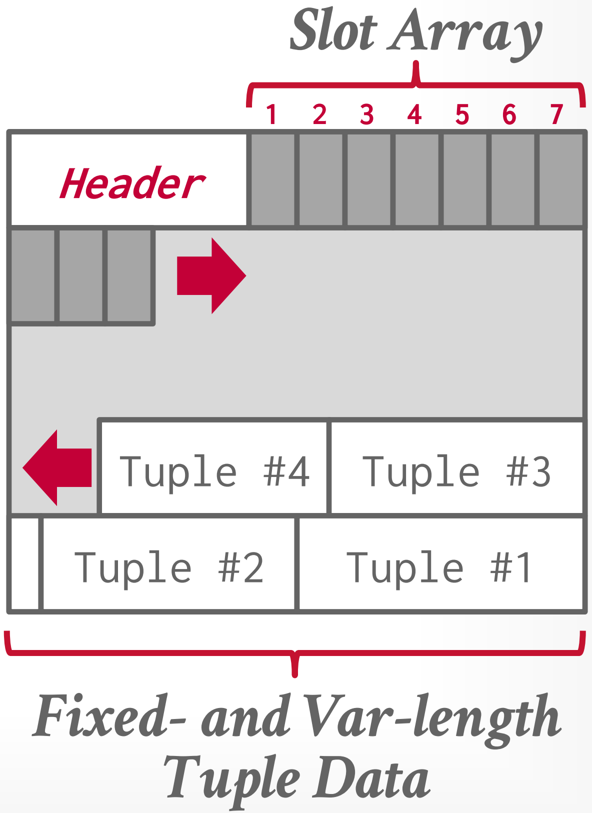


SLOTTED PAGES

The most common 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.

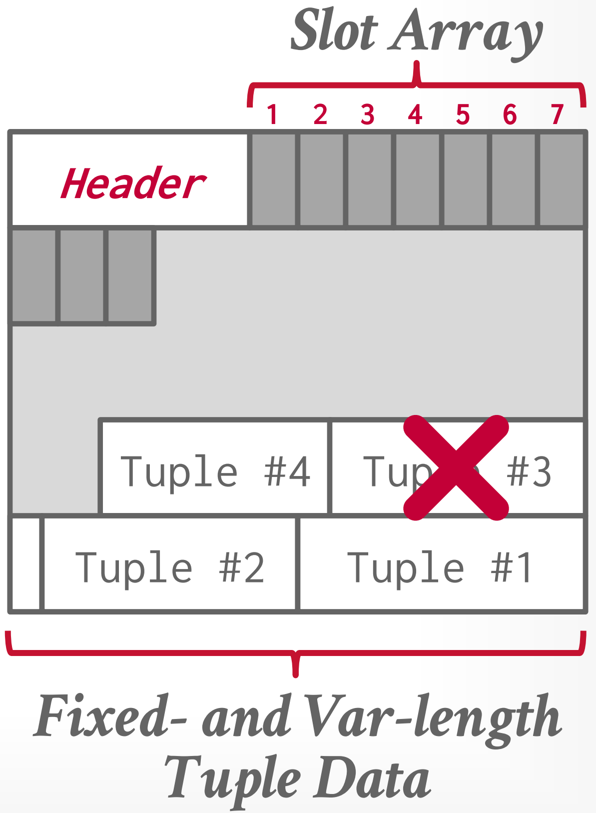


SLOTTED PAGES

The most common 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.

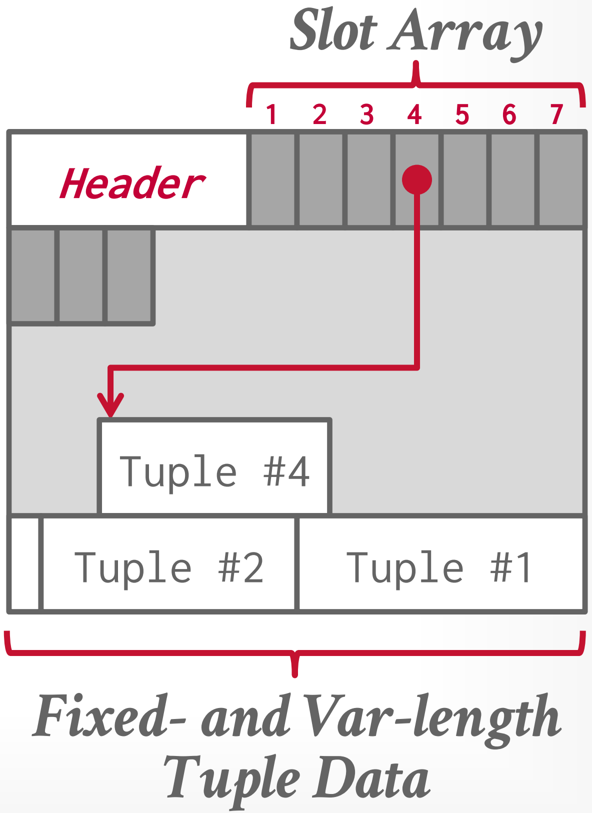


SLOTTED PAGES

The most common 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.

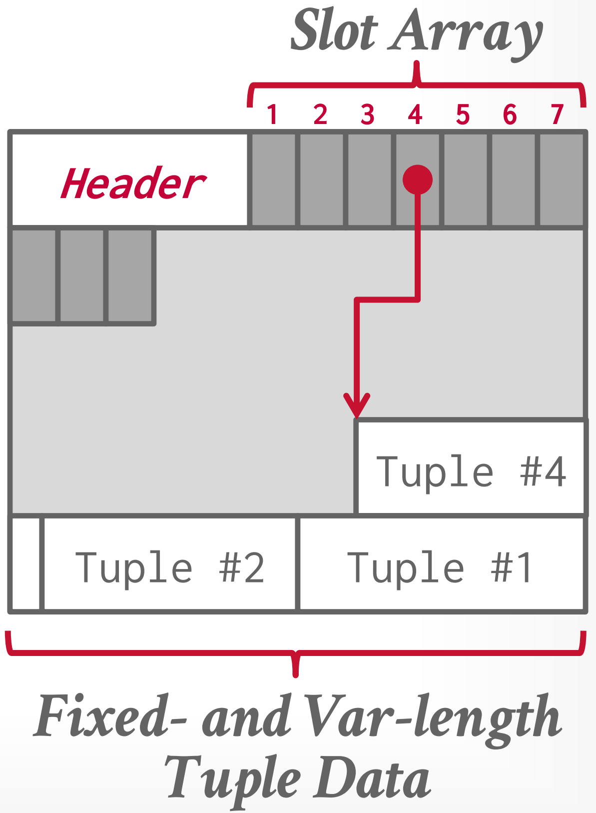


SLOTTED PAGES

The most common 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.

- → 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.

PostgreSQL
CTID (6-bytes)

SQLite
ROWID (8-bytes)

Microsoft SQL Server
%%physloc%% (8-bytes)

ORACLE
ROWID (10-bytes)

TUPLE-ORIENTED STORAGE

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.

TUPLE-ORIENTED 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: Some object stores, HDFS, Google Colossus

TODAY'S AGENDA

Log-Structured Storage
Index-Organized Storage
Data Representation

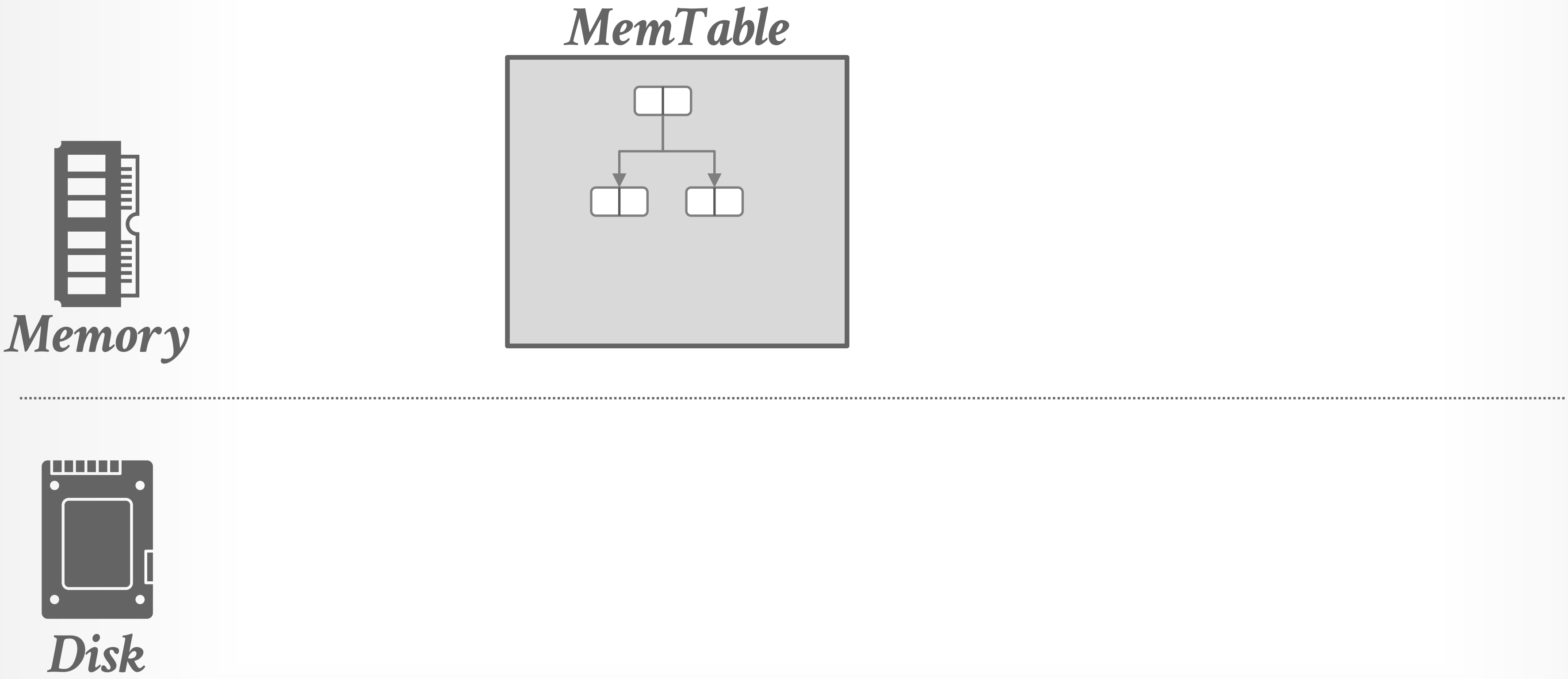
LOG-STRUCTURED STORAGE

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 (***SSTable***).

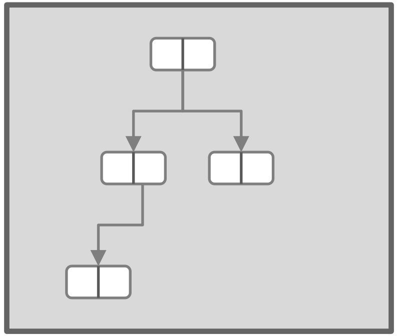
LOG-STRUCTURED STORAGE

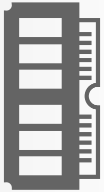


LOG-STRUCTURED STORAGE

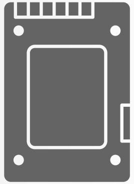
PUT (key101,a_1)

MemTable





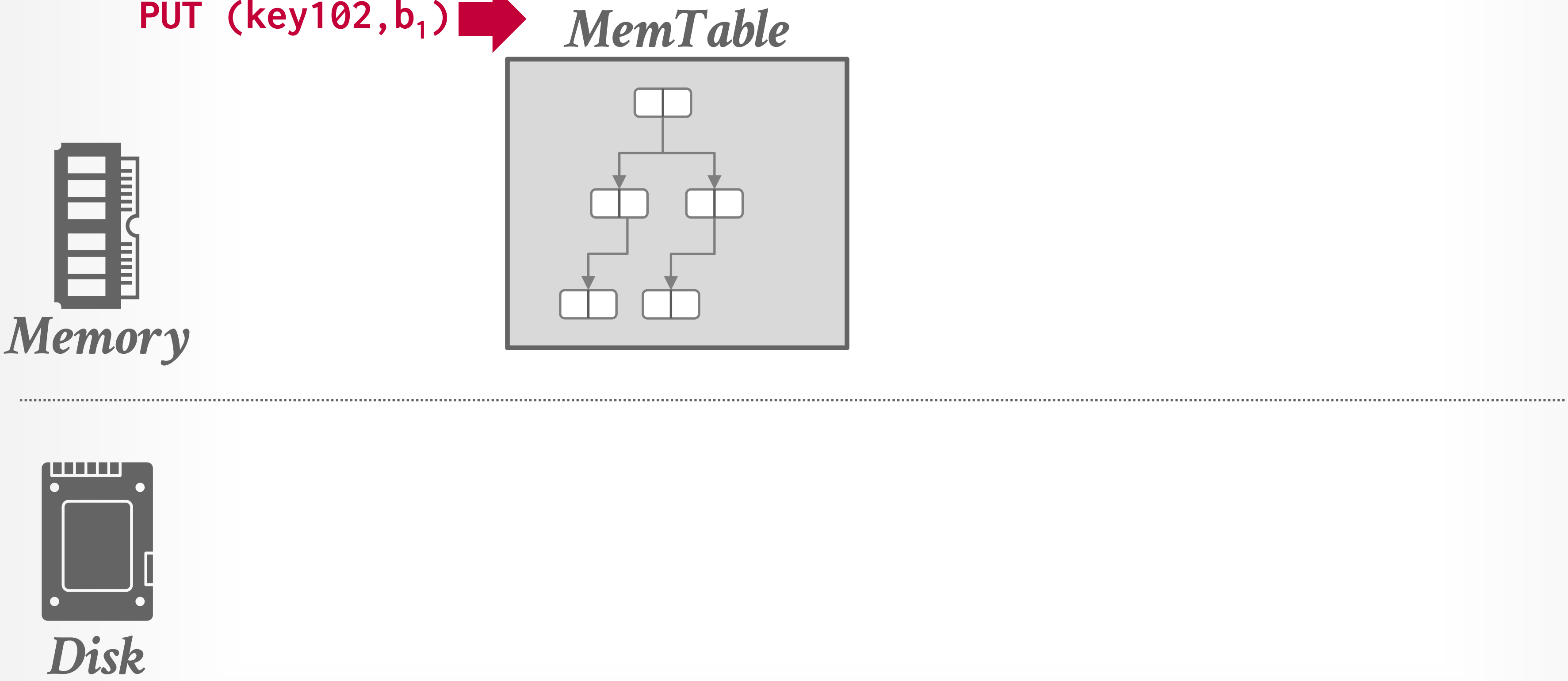
Memory



Disk

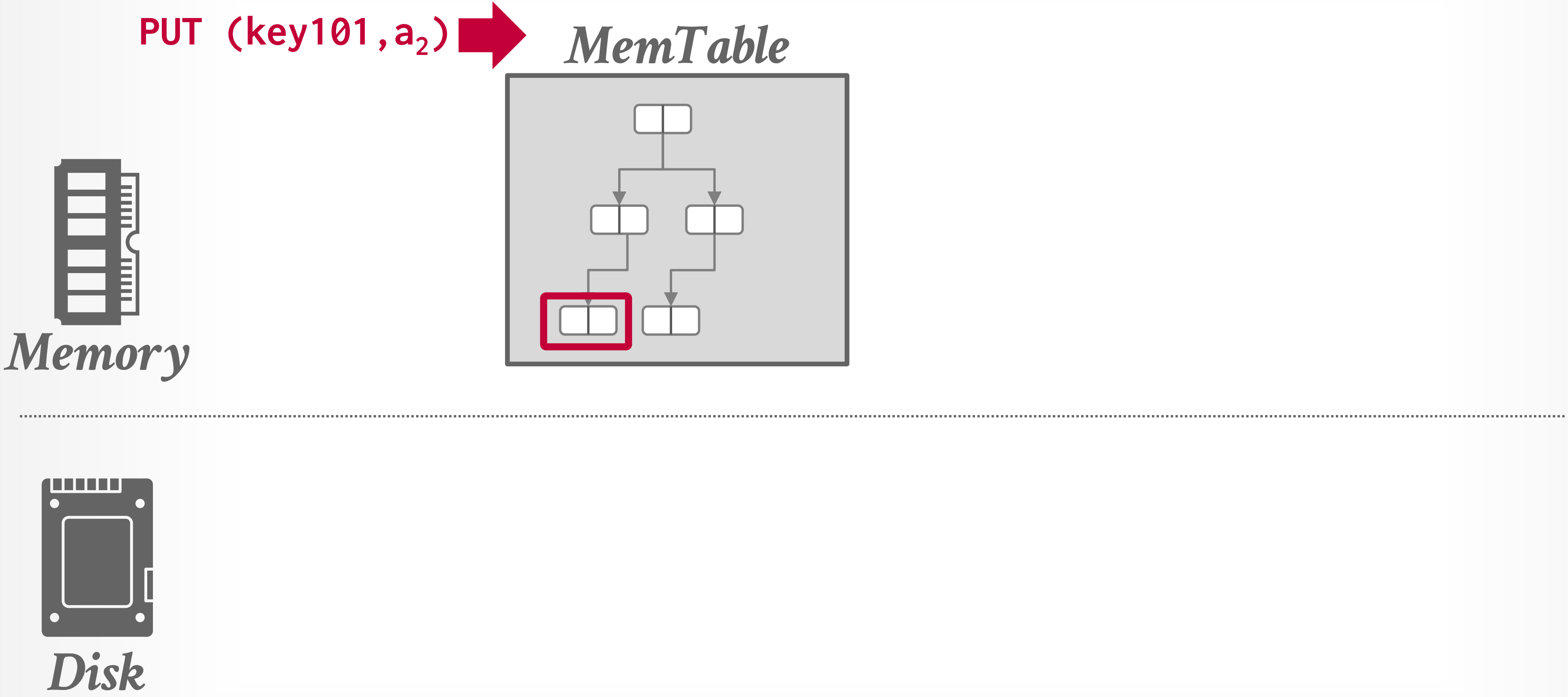
LOG-STRUCTURED STORAGE

PUT (key102,b_1)



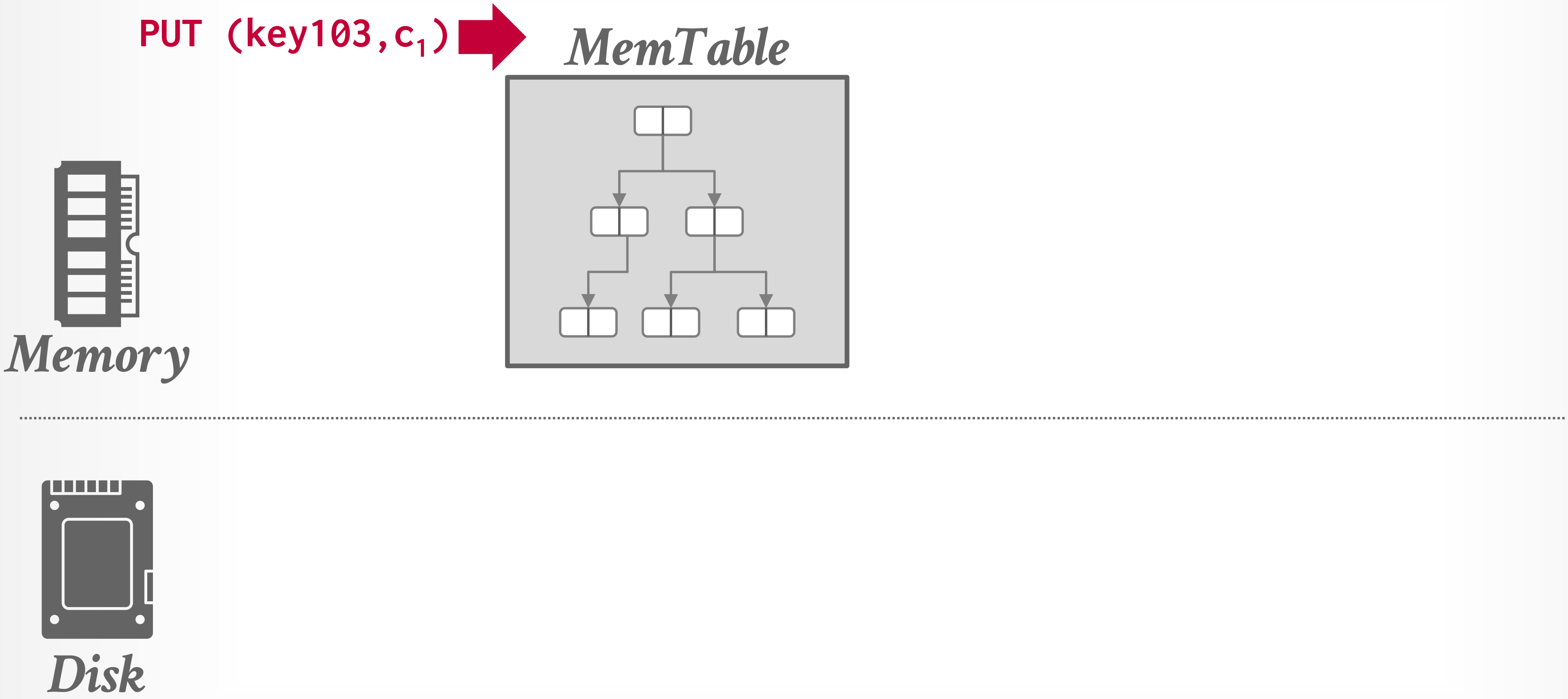
LOG-STRUCTURED STORAGE

PUT (key101,a_2)

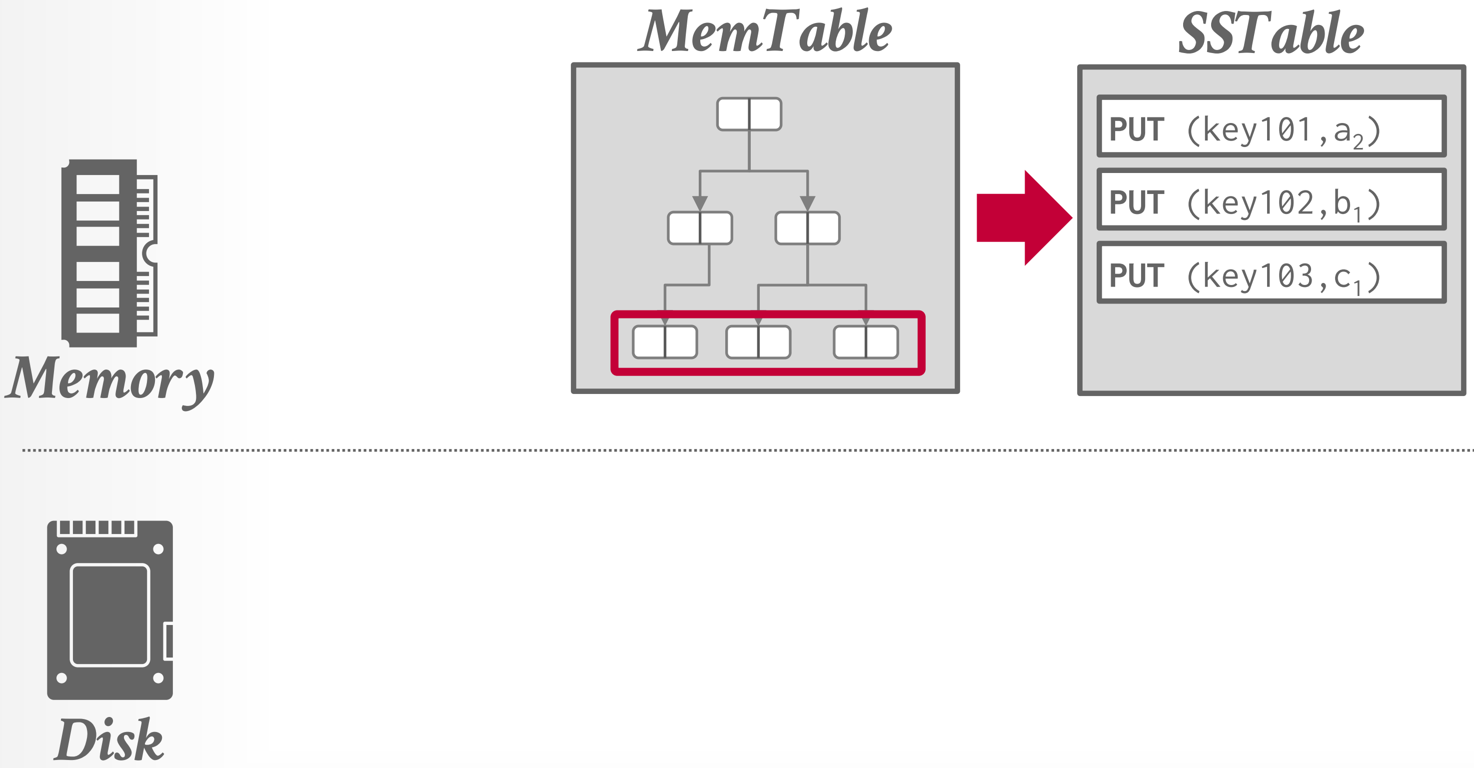


LOG-STRUCTURED STORAGE

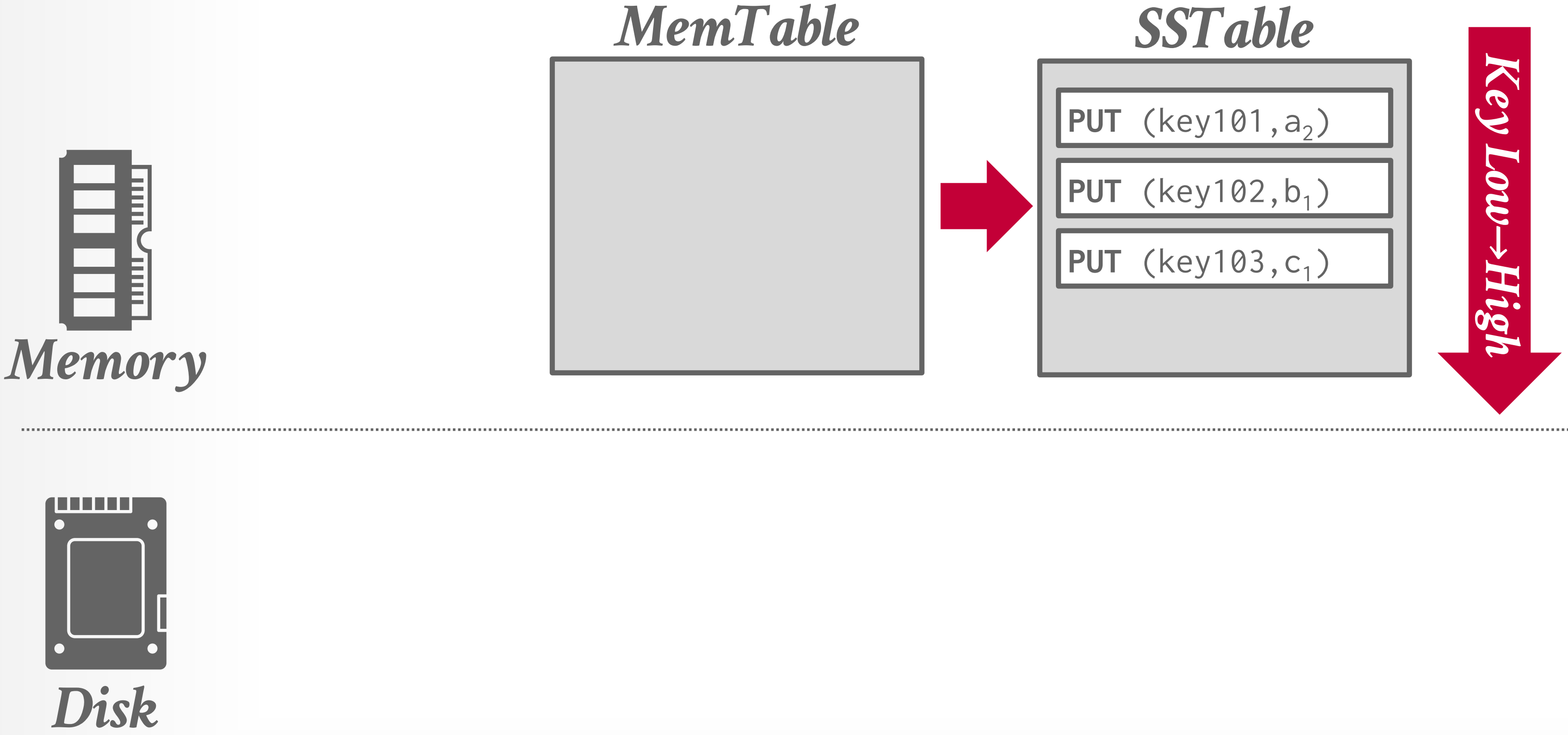
PUT (key103,c_1)



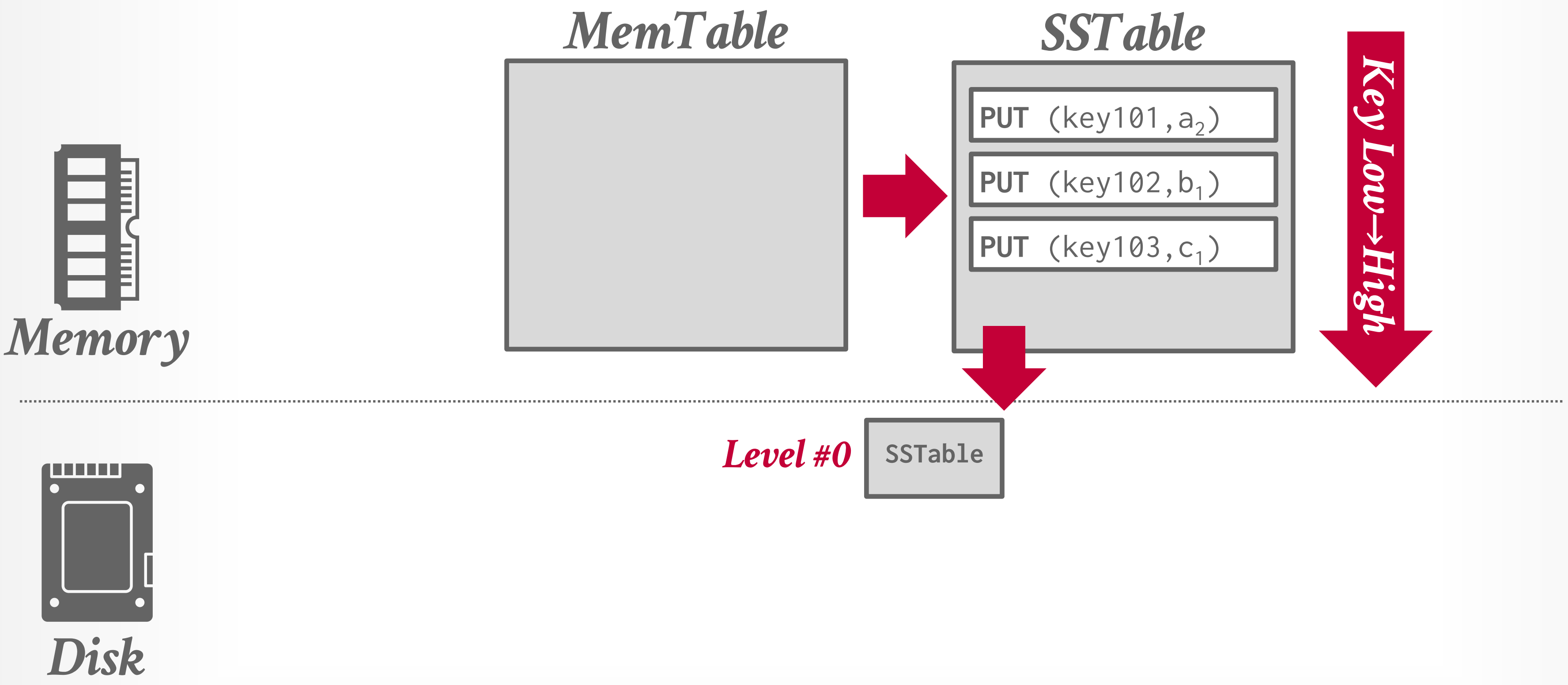
LOG-STRUCTURED STORAGE



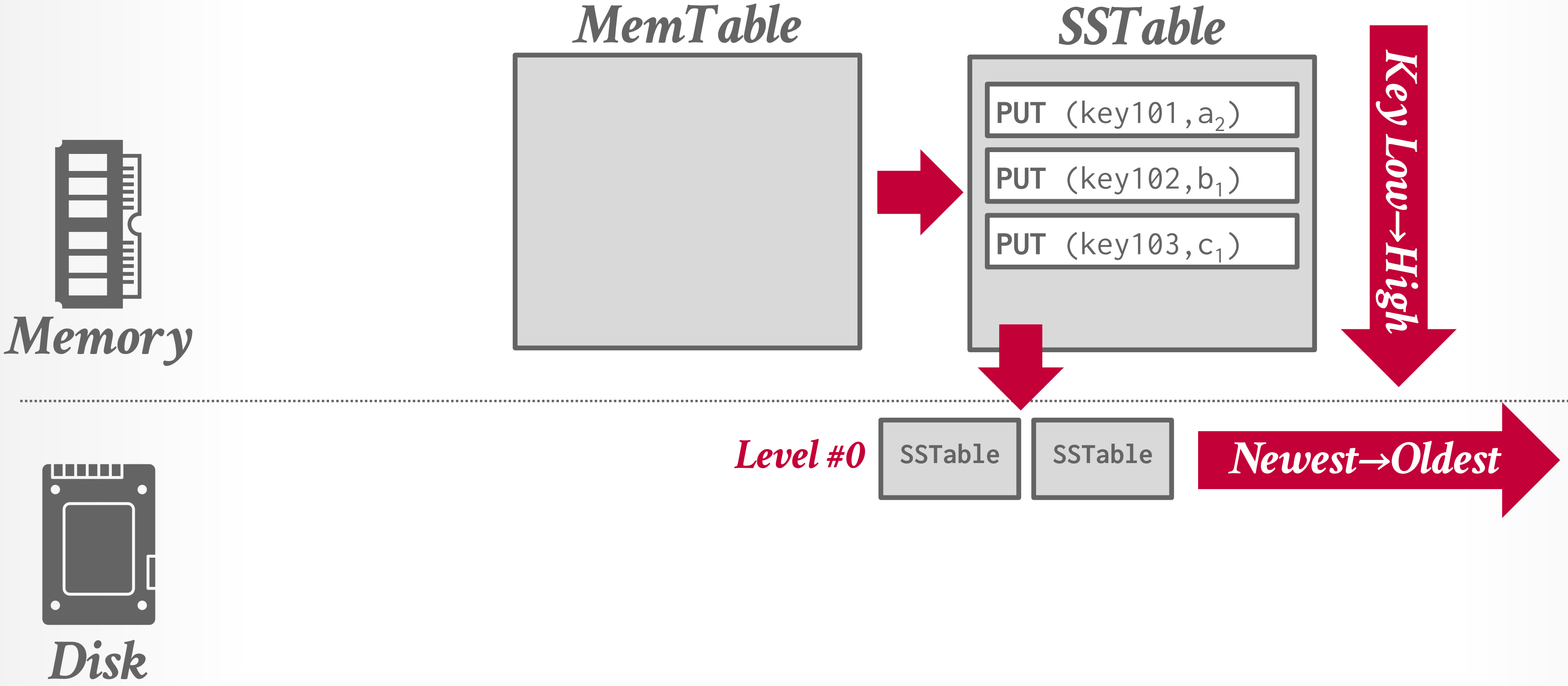
LOG-STRUCTURED STORAGE



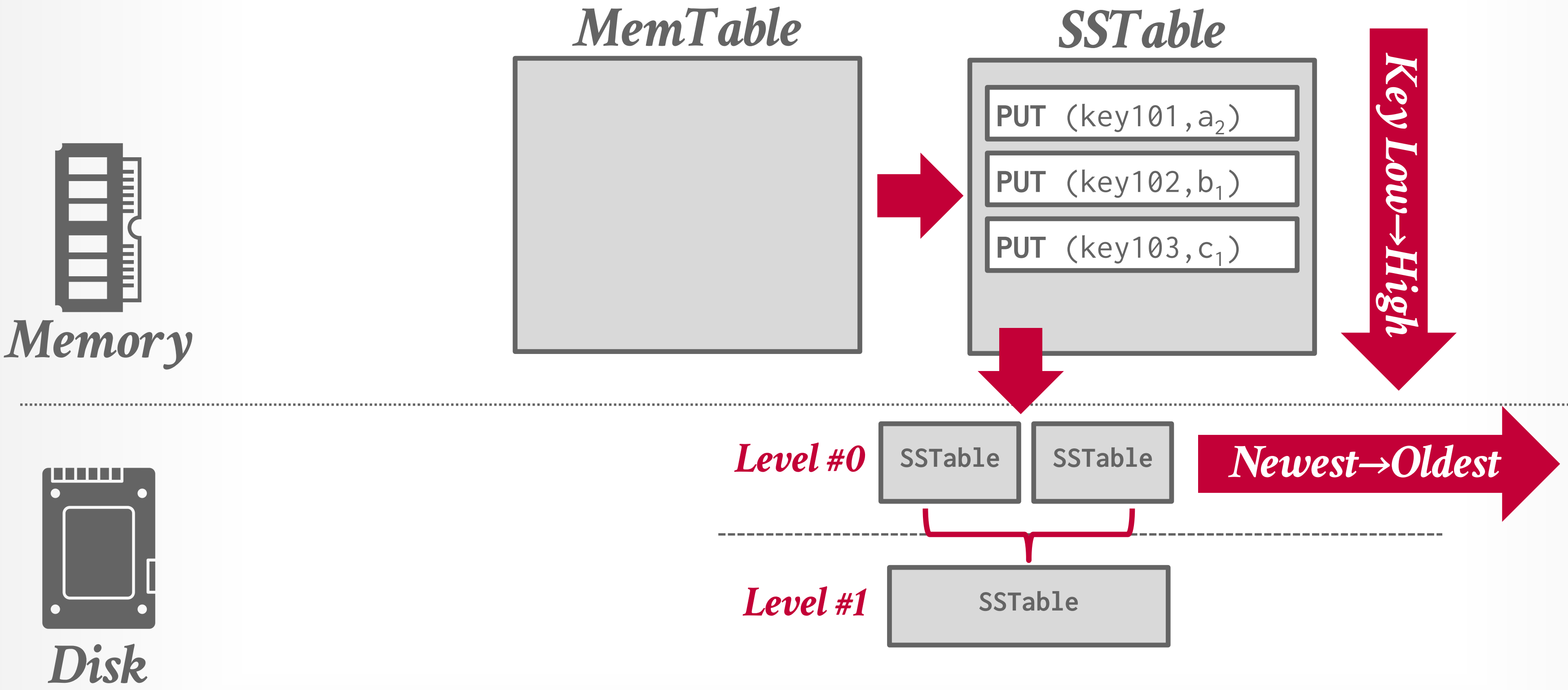
LOG-STRUCTURED STORAGE



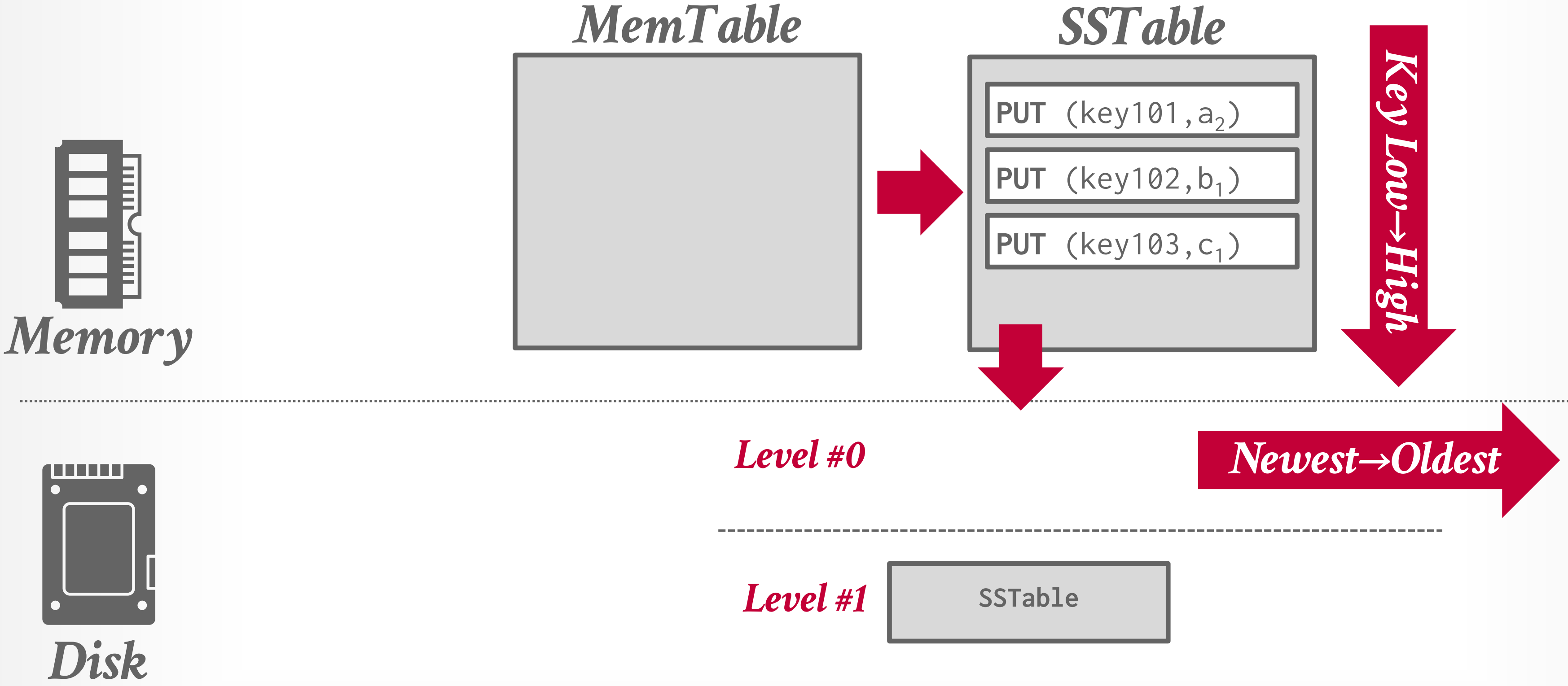
LOG-STRUCTURED STORAGE



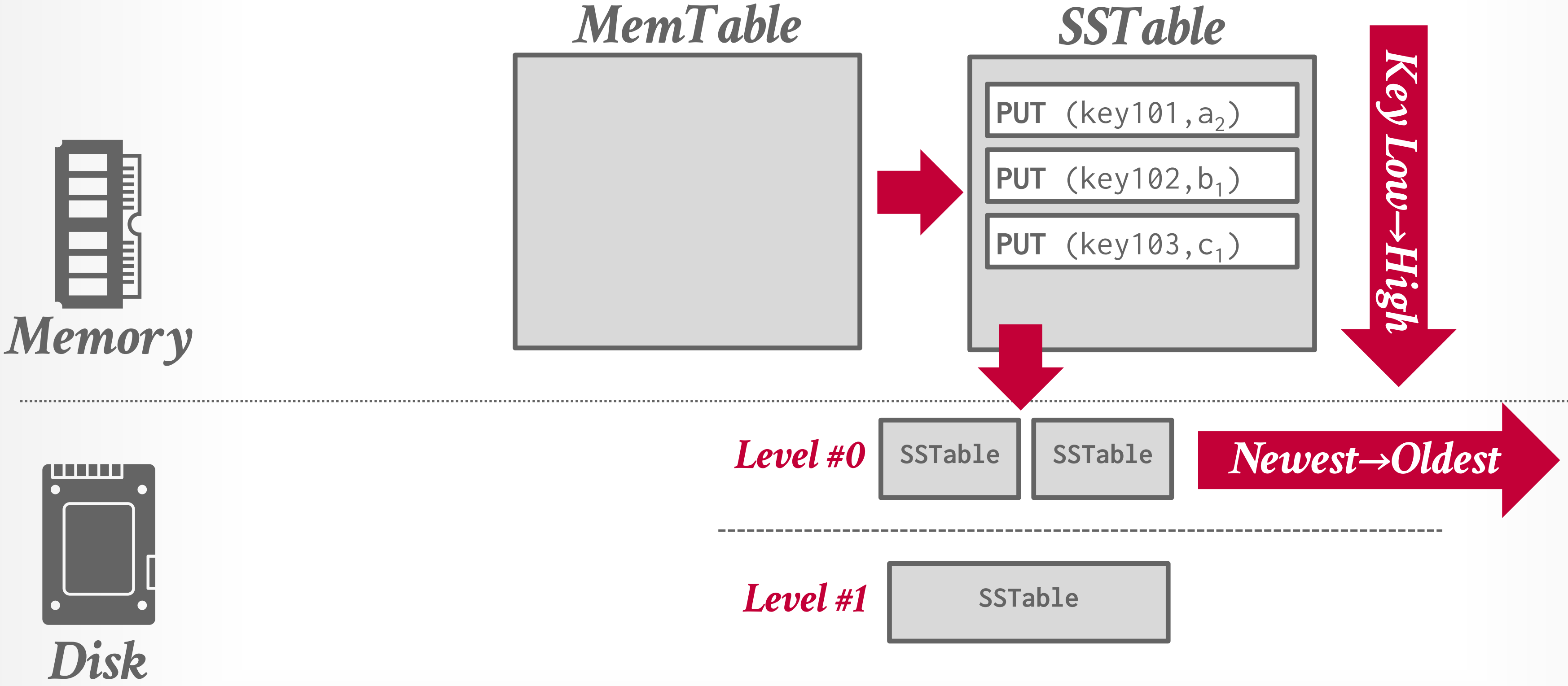
LOG-STRUCTURED STORAGE



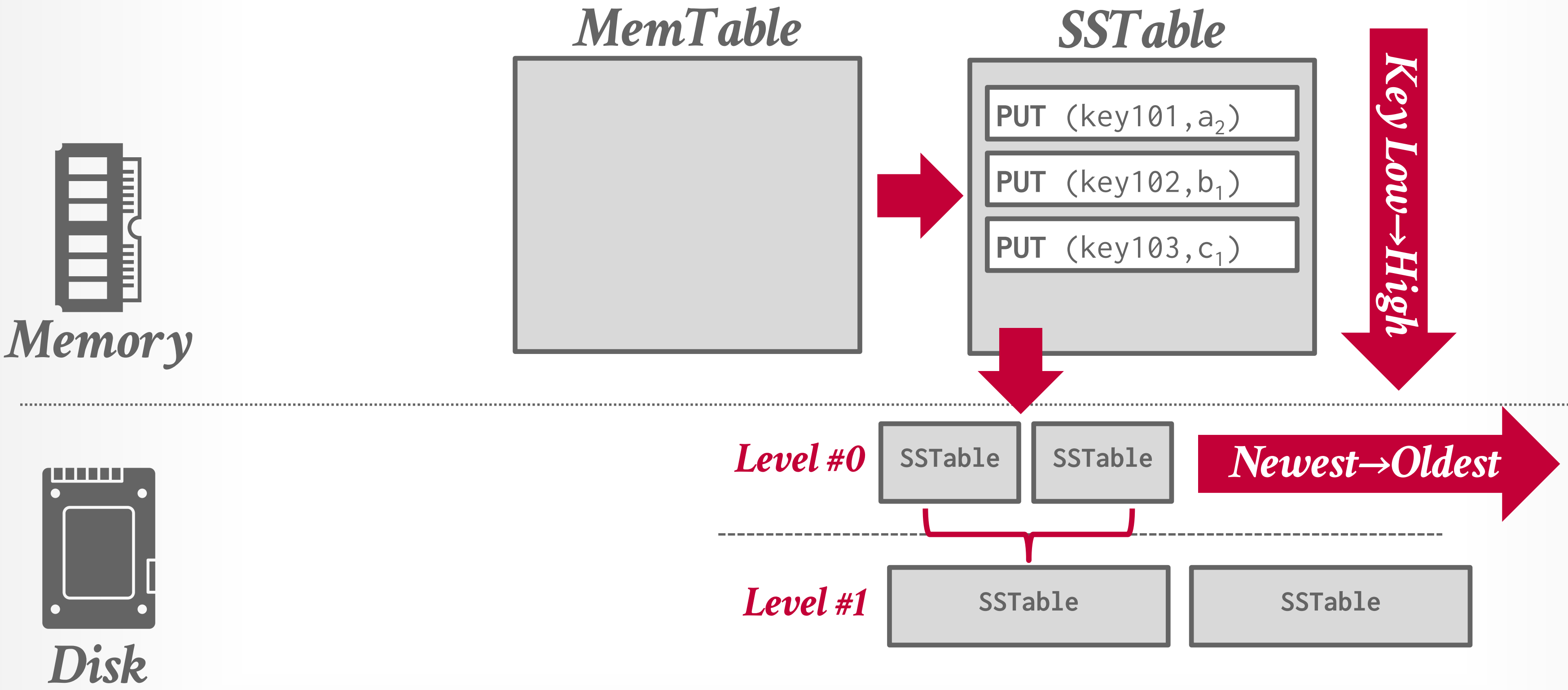
LOG-STRUCTURED STORAGE



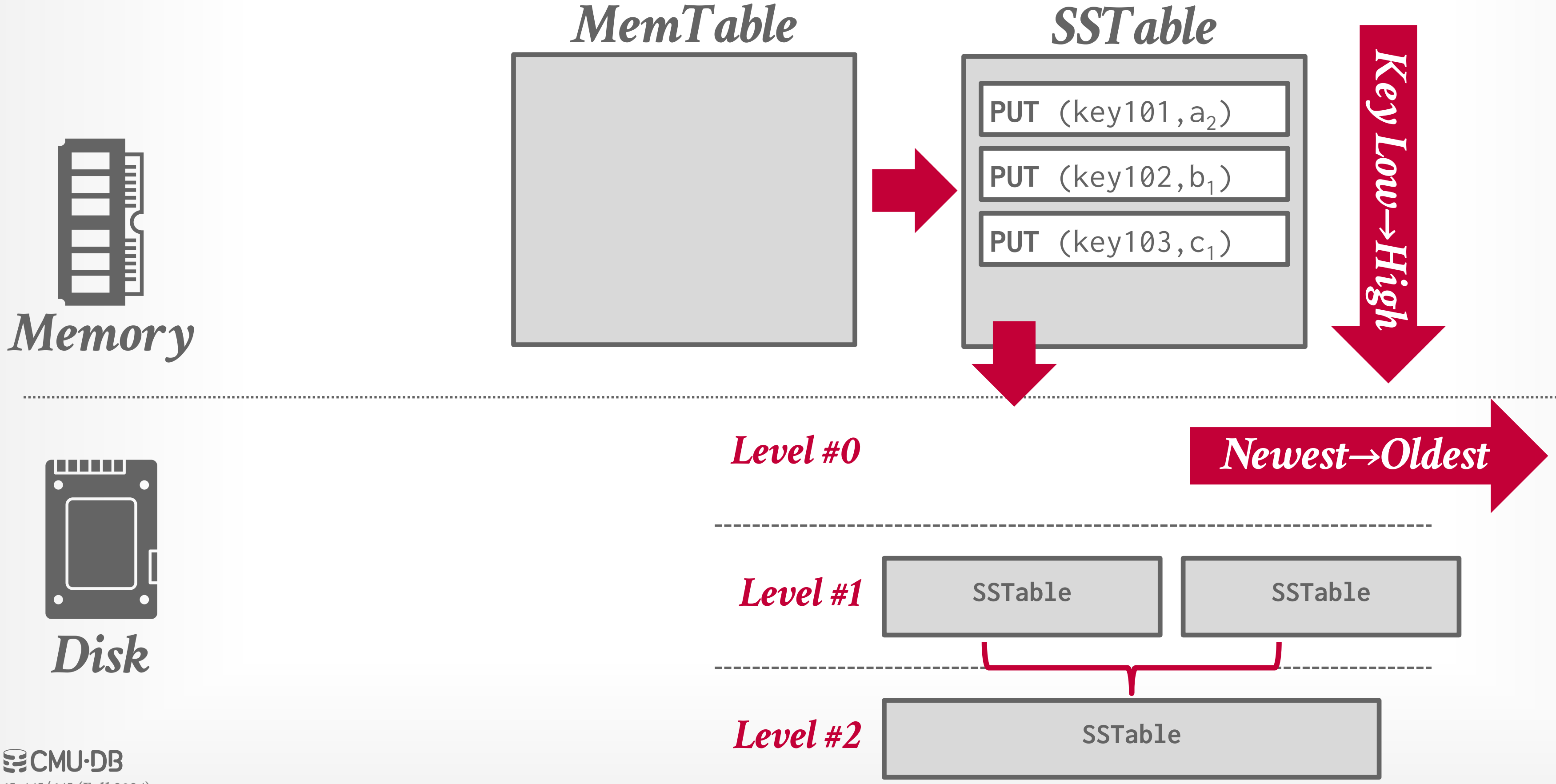
LOG-STRUCTURED STORAGE



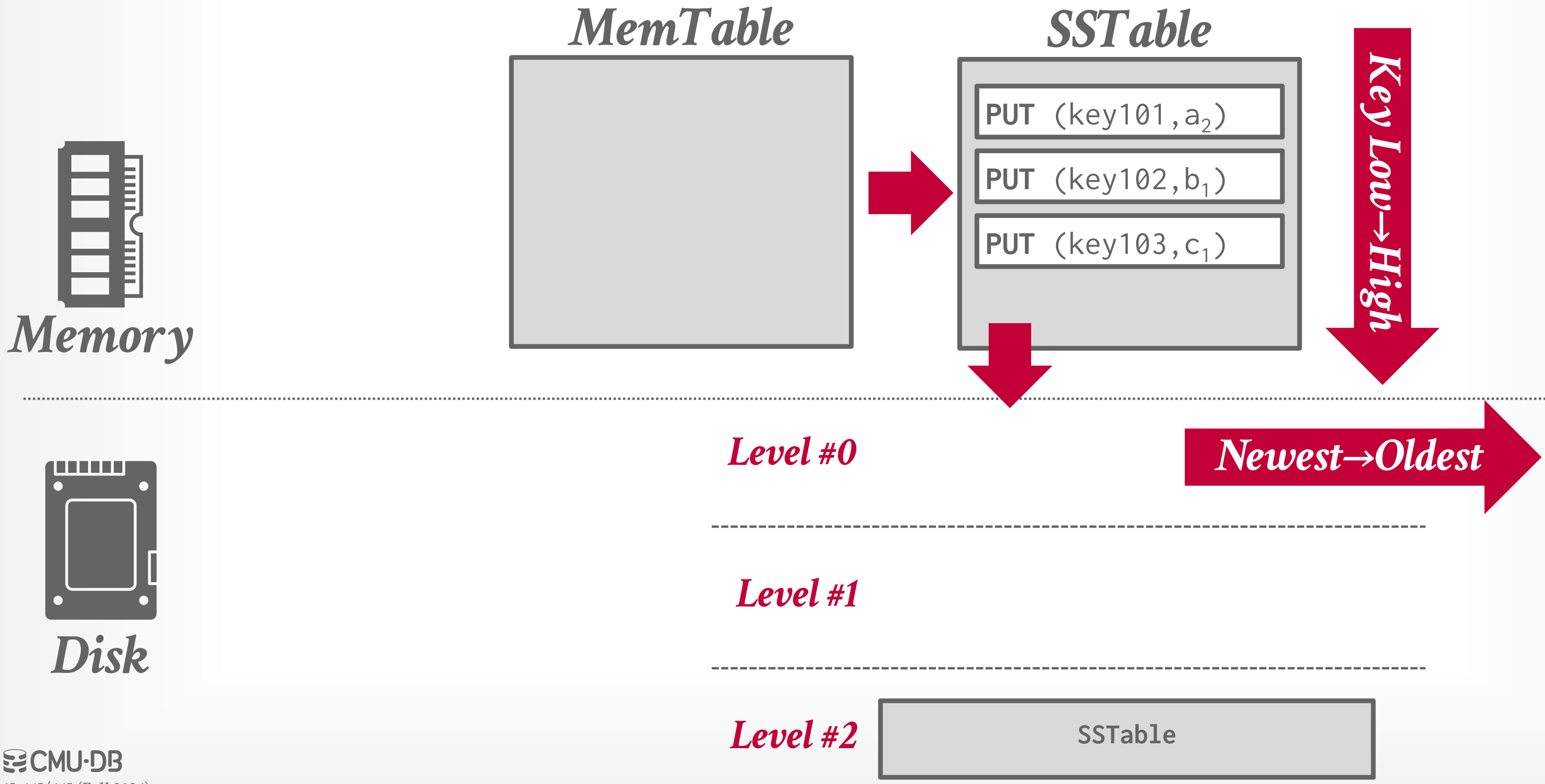
LOG-STRUCTURED STORAGE



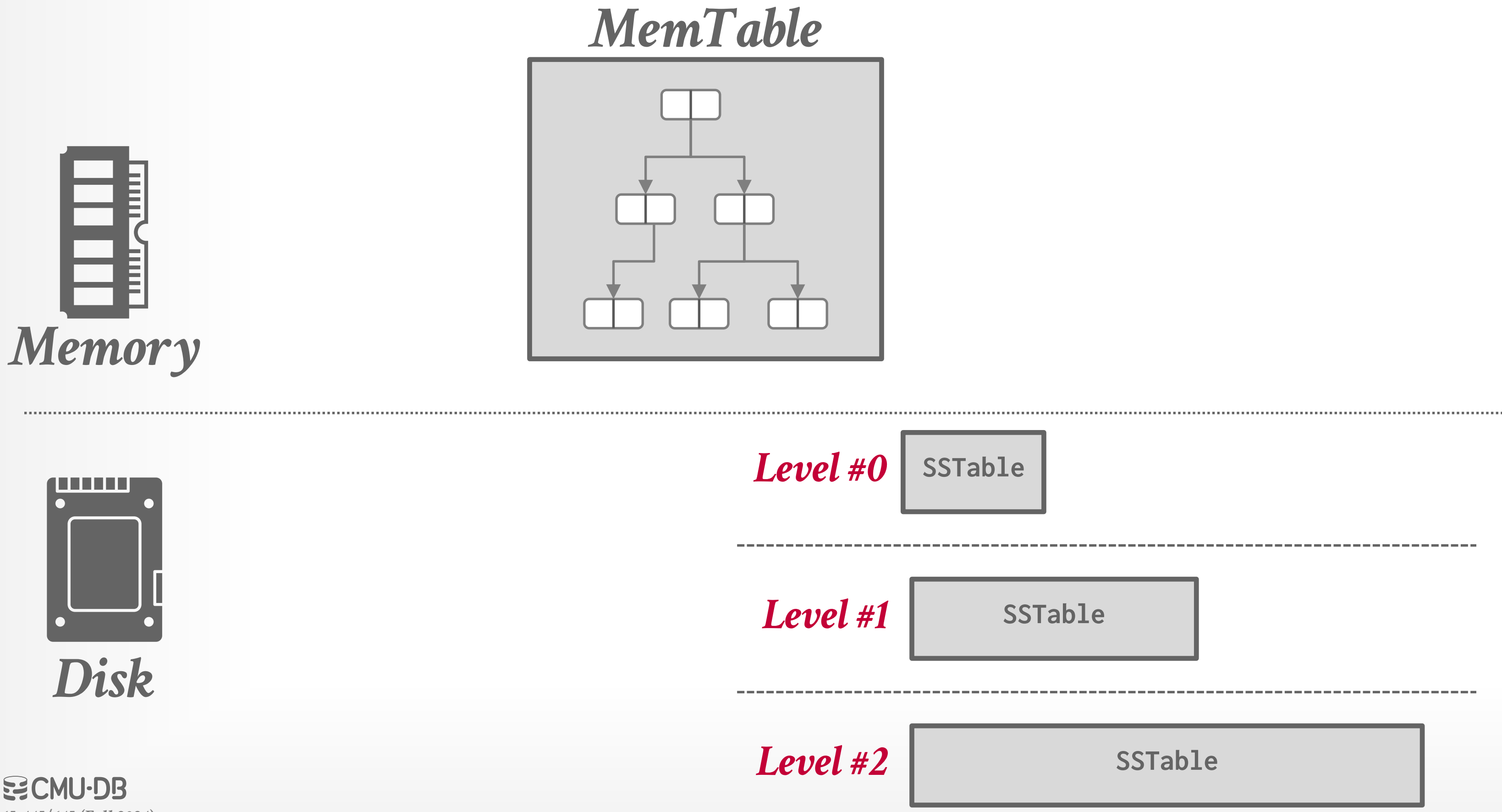
LOG-STRUCTURED STORAGE



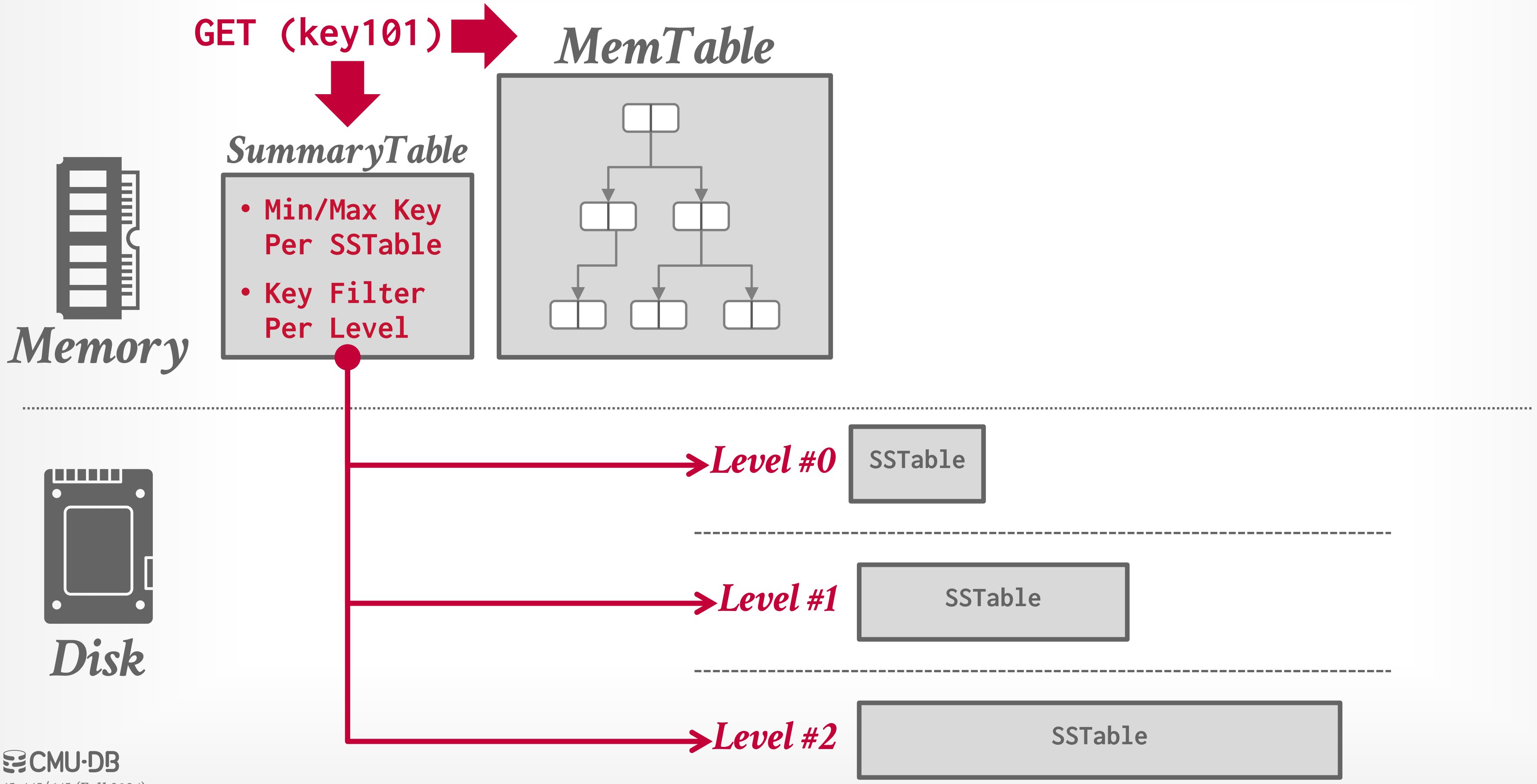
LOG-STRUCTURED STORAGE



LOG-STRUCTURED STORAGE



LOG-STRUCTURED STORAGE



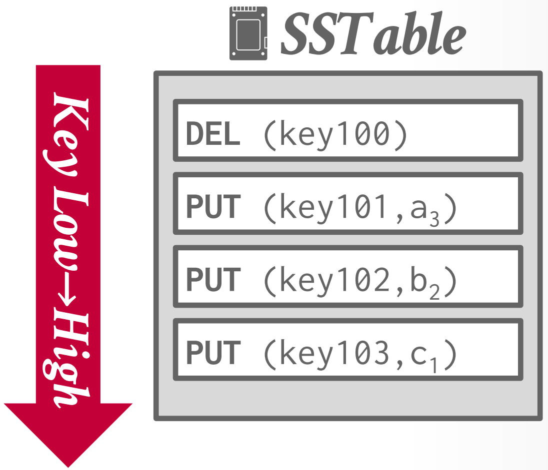
LOG-STRUCTURED STORAGE

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.

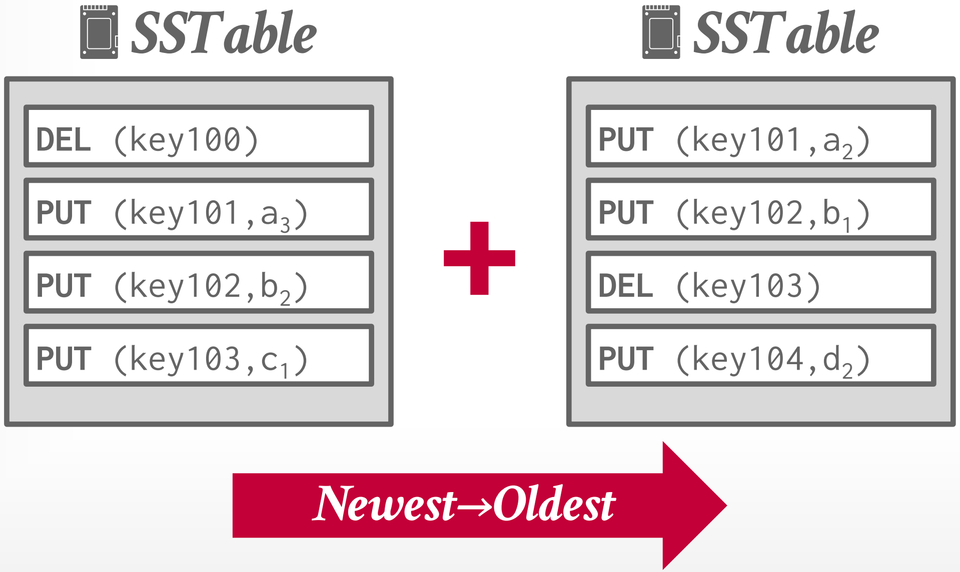


As the application makes changes to the database, the DBMS appends log records to the end of the file without checking previous log records.

LOG-STRUCTURED COMPACTION

Periodically compact SSTAbles to reduce wasted space and speed up reads.

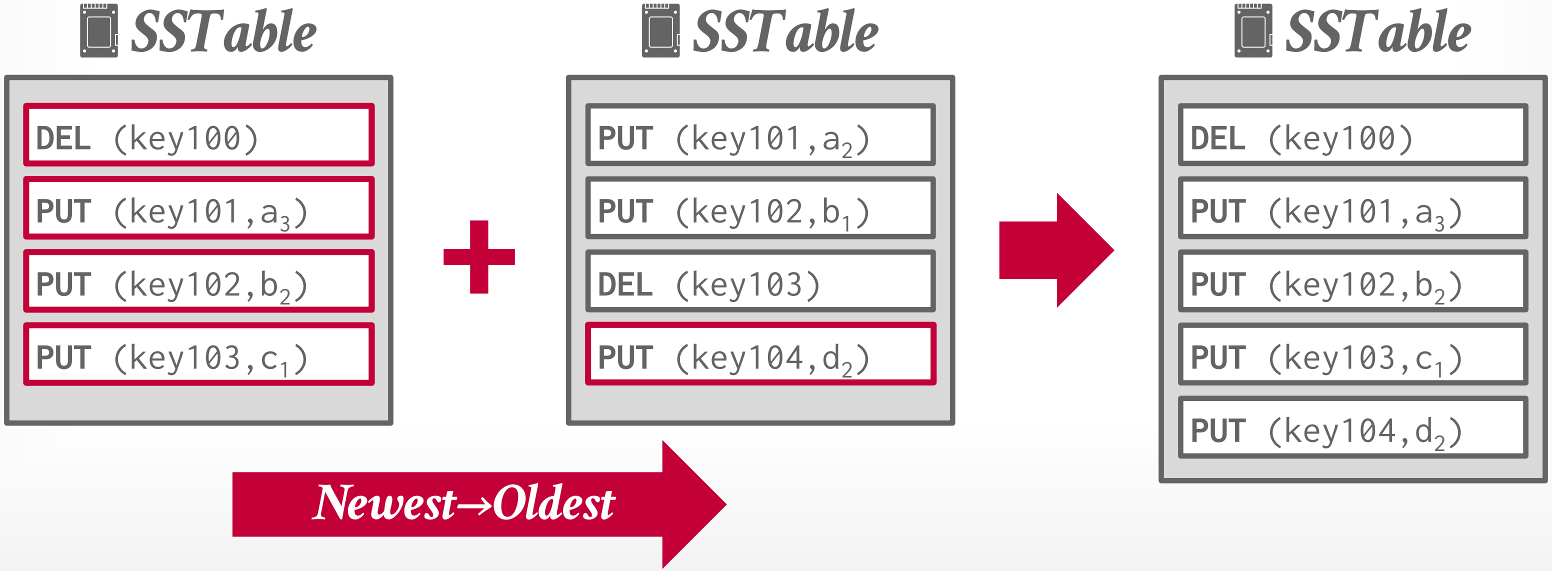
→ Only keep the "latest" values for each key using a sort-merge algorithm.



LOG-STRUCTURED COMPACTION

Periodically compact SSTAbles to reduce wasted space and speed up reads.

→ Only keep the "latest" values for each key using a sort-merge algorithm.



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

OBSERVATION

The two table storage approaches we've discussed so far rely on indexes to find individual tuples.

→ Such indexes are necessary because the tables are inherently unsorted.

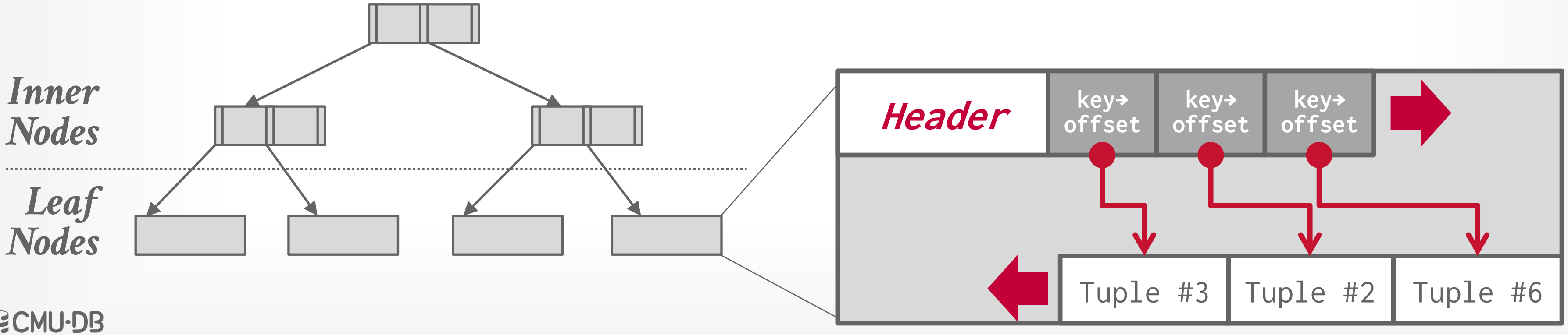
But what if the DBMS could keep tuples sorted automatically using an index?

INDEX-ORGANIZED STORAGE

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

→ Still use a page layout that looks like a slotted page.
→ Tuples are typically sorted in page based on key.

B+Tree pays maintenance costs upfront, whereas LSMs pay for it later.



TUPLE STORAGE

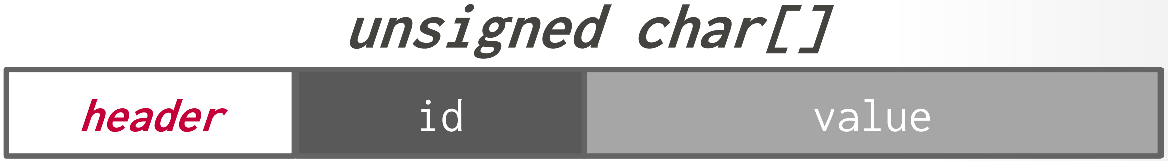
A tuple is essentially a sequence of bytes prefixed with a **header** that contains meta-data about it.

It is the job of the DBMS to interpret those bytes into attribute types and values.

The DBMS's catalogs contain the schema information about tables that the system uses to figure out the tuple's layout.

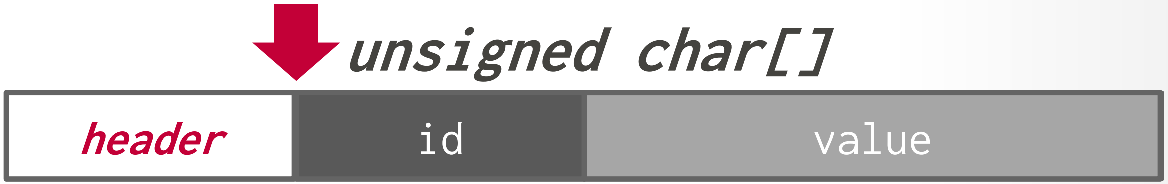
DATA LAYOUT

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  value BIGINT
);
```



DATA LAYOUT

```sql
CREATE TABLE foo (
  id INT PRIMARY KEY,
  value BIGINT
);
```

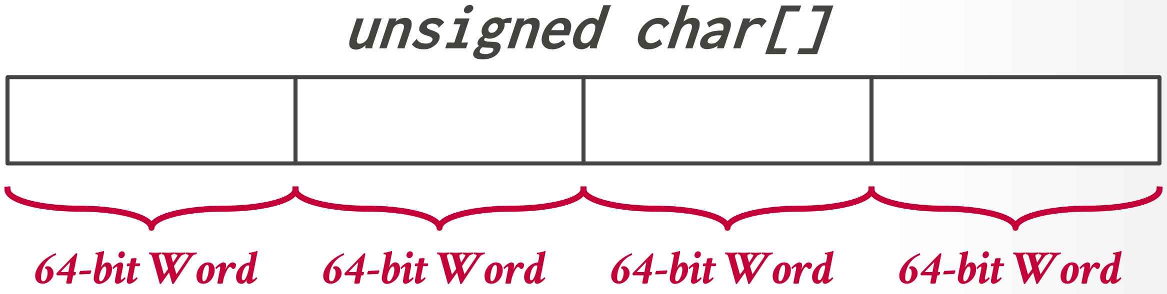



```cpp
reinterpret_cast<int32_t*>(address)
```

WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  cdate TIMESTAMP,
  color CHAR(2),
  zipcode INT
);
```

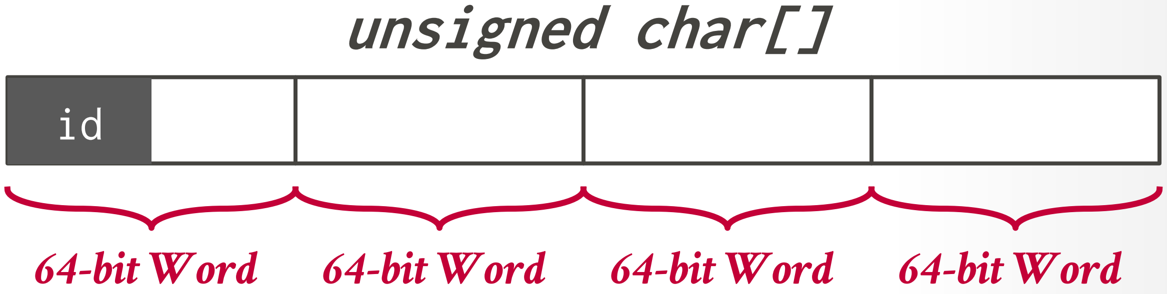


WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

32-bits

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  cdate TIMESTAMP,
  color CHAR(2),
  zipcode INT
);
```

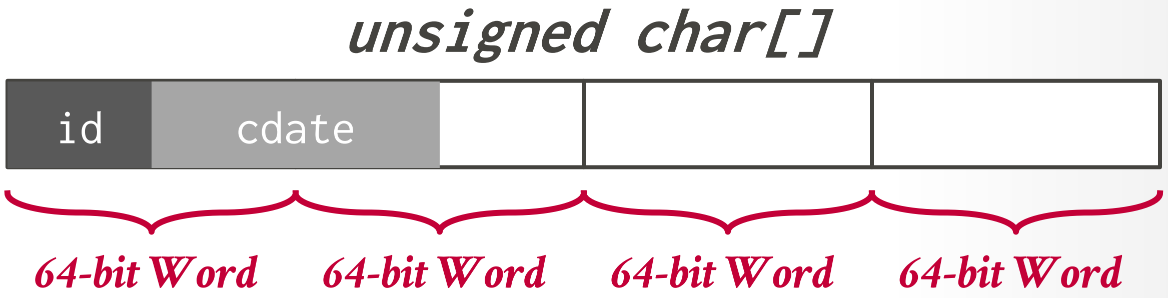


WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  cdate TIMESTAMP,
  color CHAR(2),
  zipcode INT
);
```

32-bits (id)
64-bits (cdate)

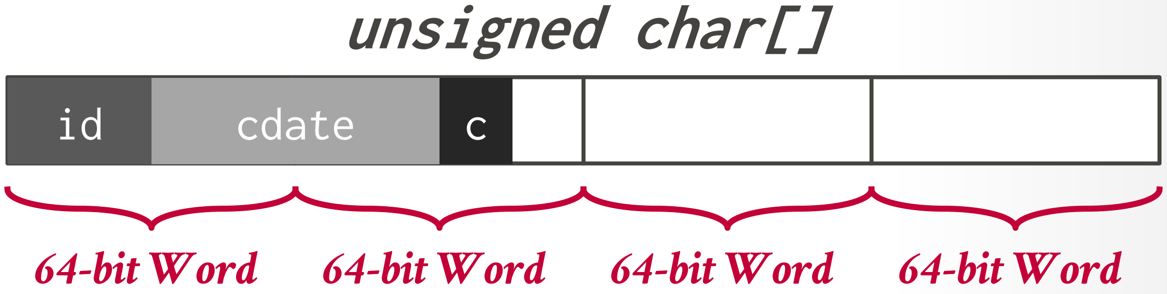


WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  cdate TIMESTAMP,
  color CHAR(2),
  zipcode INT
);
```

32-bits (id), 64-bits (cdate), 16-bits (color)

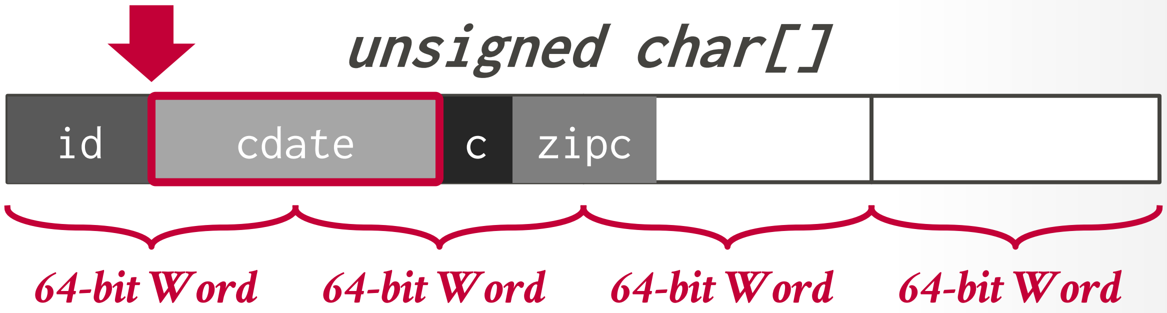


WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  cdate TIMESTAMP,
  color CHAR(2),
  zipcode INT
);
```

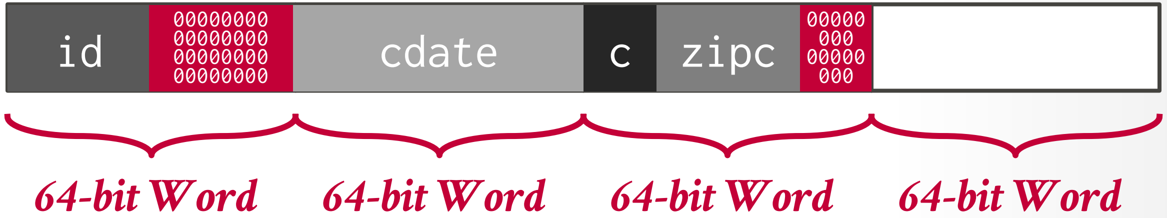
32-bits (id), *64-bits* (cdate), *16-bits* (color), *32-bits* (zipcode)



WORD-ALIGNMENT: PADDING

Add empty bits after attributes to ensure that tuple is word aligned. Essentially round up the storage size of types to the next largest word size.

```
CREATE TABLE foo (
  id INT PRIMARY KEY,      -- 32-bits
  cdate TIMESTAMP,         -- 64-bits
  color CHAR(2),           -- 16-bits
  zipcode INT              -- 32-bits
);
```

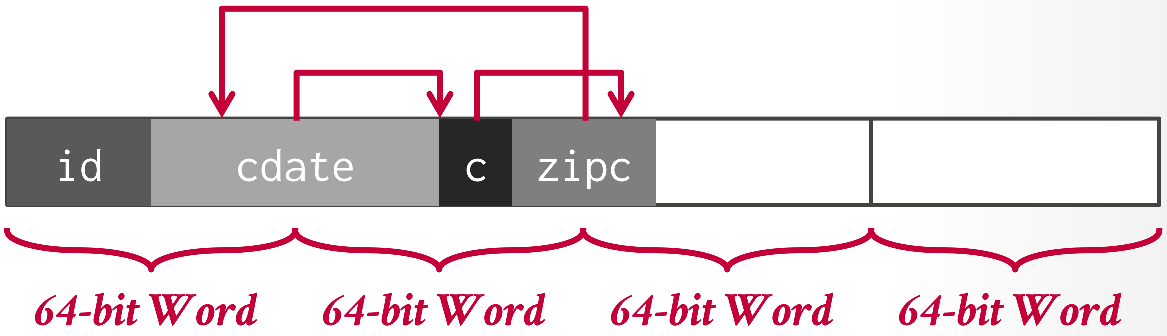


WORD-ALIGNMENT: REORDERING

Switch the order of attributes in the tuples' physical layout to make sure they are aligned.

→ May still have to use padding to fill remaining space.

```
CREATE TABLE foo (
  id INT PRIMARY KEY,       -- 32-bits
  cdate TIMESTAMP,          -- 64-bits
  color CHAR(2),            -- 16-bits
  zipcode INT               -- 32-bits
);
```

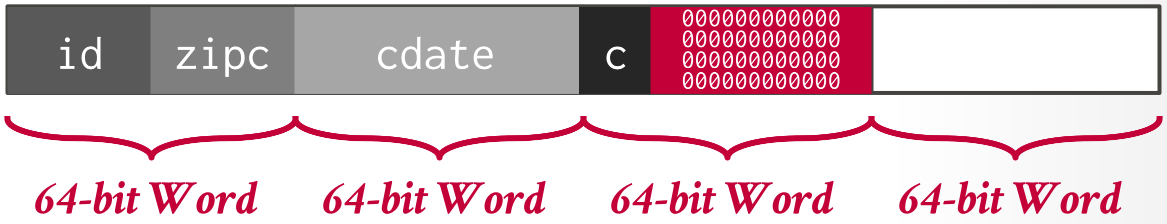


WORD-ALIGNMENT: REORDERING

Switch the order of attributes in the tuples' physical layout to make sure they are aligned.

→ May still have to use padding to fill remaining space.

```
CREATE TABLE foo (
  id INT PRIMARY KEY,      -- 32-bits
  cdate TIMESTAMP,         -- 64-bits
  color CHAR(2),           -- 16-bits
  zipcode INT              -- 32-bits
);
```



DATA REPRESENTATION

INTEGER/BIGINT/SMALLINT/TINYINT
→ Same as in C/C++.

FLOAT/REAL vs. **NUMERIC/DECIMAL**
→ IEEE-754 Standard / Fixed-point Decimals.

VARCHAR/VARBINARY/TEXT/BLOB
→ Header with length, followed by data bytes **OR** pointer to another page/offset with data.
→ Need to worry about collations / sorting.

TIME/DATE/TIMESTAMP/INTERVAL
→ 32/64-bit integer of (micro/milli)-seconds since Unix epoch (January 1st, 1970).

VARIABLE PRECISION NUMBERS

Inexact, variable-precision numeric type that uses the "native" C/C++ types.

Store directly as specified by IEEE-754.

→ Example: **FLOAT**, **REAL**/**DOUBLE**

These types are typically faster than fixed precision numbers because CPU ISA's (Xeon, Arm) have instructions / registers to support them.

But they do not guarantee exact values…

VARIABLE PRECISION NUMBERS

Rounding Example

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    float x = 0.1;
    float y = 0.2;
    printf("x+y = %f\n", x+y);
    printf("0.3 = %f\n", 0.3);
}
```

Output

```
x+y = 0.300000
0.3 = 0.300000
```

VARIABLE PRECISION NUMBERS

Rounding Example


```
#include <stdio.h>

in
}
```


```
#include <stdio.h>

int main(int argc, char* argv[]) {
    float x = 0.1;
    float y = 0.2;
    printf("x+y = %.20f\n", x+y);
    printf("0.3 = %.20f\n", 0.3);
}
```

Output

```
x+y = 0.300000
0.3 = 0.300000
```

```
x+y = 0.30000001192092895508
0.3 = 0.29999999999999998890
```

FIXED PRECISION NUMBERS

Numeric data types with (potentially) arbitrary precision and scale. Used when rounding errors are unacceptable.

→ Example: **NUMERIC**, **DECIMAL**

Many different implementations.

→ Example: Store in an exact, variable-length binary representation with additional meta-data.

→ Can be less expensive if the DBMS does not provide arbitrary precision (e.g., decimal point can be in a different position per value).

POSTGRES: NUMERIC

of Digits

Weight of 1st Digit

Scale Factor

Positive/Negative/NaN

Digit Storage

```
typedef unsigned char NumericDigit;
typedef struct {
  int ndigits;
  int weight;
  int scale;
  int sign;
  NumericDigit *digits;
} numeric;
```

c

Weight of

Sca

Positive/Negat

Digi

```
NumericDigit;
```

```
/* ----------
 * add_var() -
 *
 *  Full version of add functionality on variable level (handling signs).
 *  result might point to one of the operands too without danger.
 * ----------
 */
int
PGTYPESnumeric_add(numeric *var1, numeric *var2, numeric *result)
{
    /*
     * Decide on the signs of the two variables what to do
     */
    if (var1->sign == NUMERIC_POS)
    {
        if (var2->sign == NUMERIC_POS)
        {
            /*
             * Both are positive result = +(ABS(var1) + ABS(var2))
             */
            if (add_abs(var1, var2, result) != 0)
                return -1;
            result->sign = NUMERIC_POS;
        }
        else
        {
            /*
             * var1 is positive, var2 is negative Must compare absolute values
             */
            switch (cmp_abs(var1, var2))
            {
                case 0:
                    /* ----------
                     * ABS(var1) == ABS(var2)
                     * result = ZERO
                     * ----------
                     */
                    zero_var(result);
                    result->rscale = Max(var1->rscale, var2->rscale);
                    result->dscale = Max(var1->dscale, var2->dscale);
                    break;

                case 1:
                    /* ----------
                     * ABS(var1) > ABS(var2)
                     * result = +(ABS(var1) - ABS(var2))
                     * ----------
                     */
                    if (sub_abs(var1, var2, result) != 0)
                        return -1;
                    result->sign = NUMERIC_POS;
                    break;

                case -1:
                    /* ----------
                     * ABS(var1) < ABS(var2)
                     * result = -(ABS(var2) - ABS(var1))
                     * ----------
```

NULL DATA TYPES

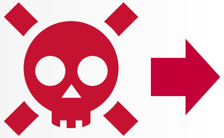
Choice #1: Null Column Bitmap Header

→ Store a bitmap in a centralized header that specifies what attributes are null.

→ This is the most common approach in row-stores.

Choice #2: Special Values

→ Designate a placeholder value to represent **NULL** for a data type (e.g., **INT32_MIN**). More common in column-stores.



Don't Do This!

Choice #3: Per Attribute Null Flag

→ Store a flag that marks that a value is null.

→ Must use more space than just a single bit because this messes up with word alignment.

NULL DATA TY

Choice #1: Null Column Bitma

→ Store a bitmap in a centralized heade attributes are null.

→ This is the most common approach

Choice #2: Special Values

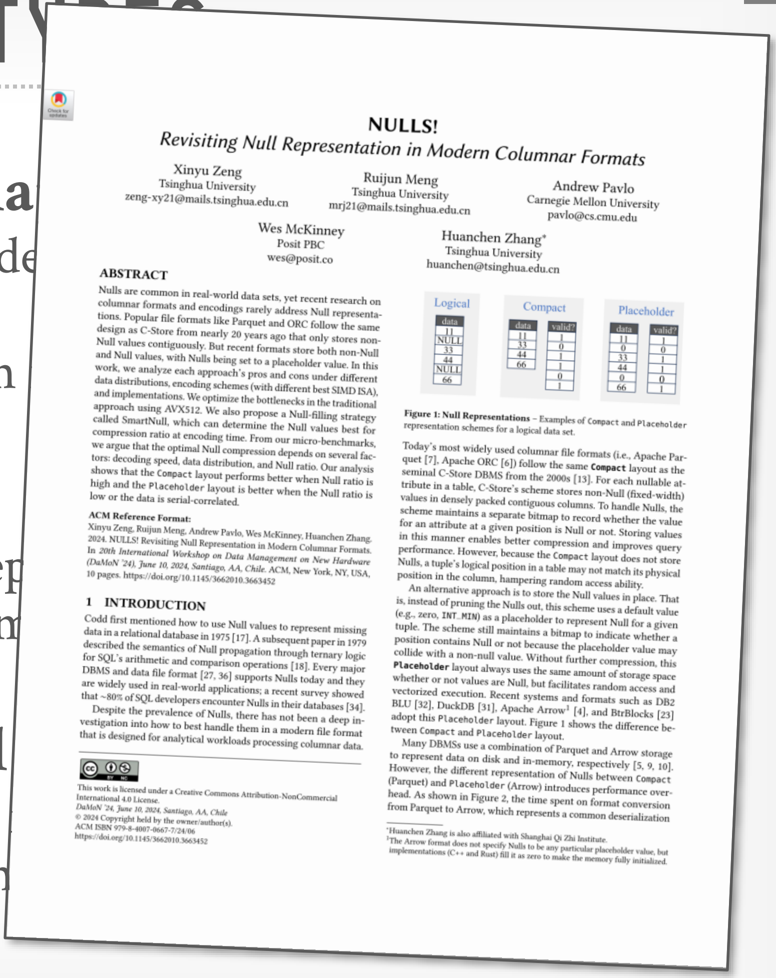
→ Designate a placeholder value to rep type (e.g., **INT32_MIN**). More comm

Don't Do This!

Choice #3: Per Attribute Null

→ Store a flag that marks that a value

→ Must use more space than just a sin messes up with word alignment.



NULLS!

Revisiting Null Representation in Modern Columnar Formats

Xinyu Zeng
Tsinghua University
zeng-xy21@mails.tsinghua.edu.cn

Ruijun Meng
Tsinghua University
mrj21@mails.tsinghua.edu.cn

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

Wes McKinney
Posit PBC
wes@posit.co

Huanchen Zhang*
Tsinghua University
huanchen@tsinghua.edu.cn

ABSTRACT

Nulls are common in real-world data sets, yet recent research on columnar formats and encodings rarely address Null representations. Popular file formats like Parquet and ORC follow the same design as C-Store from nearly 20 years ago that only stores non-Null values contiguously. But recent formats store both non-Null and Null values, with Nulls being set to a placeholder value. In this work, we analyze each approach's pros and cons under different data distributions, encoding schemes (with different best SIMD ISA), and implementations. We optimize the bottlenecks in the traditional approach using AVX512. We also propose a Null-filling strategy called SmartNull, which can determine the Null values best for compression ratio at encoding time. From our micro-benchmarks, we argue that the optimal Null compression depends on several factors: decoding speed, data distribution, and Null ratio. Our analysis shows that the `Compact` layout performs better when Null ratio is high and the `Placeholder` layout is better when the Null ratio is low or the data is serial-correlated.

ACM Reference Format:
Xinyu Zeng, Ruijun Meng, Andrew Pavlo, Wes McKinney, Huanchen Zhang. 2024. NULLS! Revisiting Null Representation in Modern Columnar Formats. In *20th International Workshop on Data Management on New Hardware (DaMoN '24), June 10, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3662010.3663452

1 INTRODUCTION

Codd first mentioned how to use Null values to represent missing data in a relational database in 1975 [17]. A subsequent paper in 1979 described the semantics of Null propagation through ternary logic for SQL's arithmetic and comparison operations [18]. Every major DBMS and data file format [27, 36] supports Nulls today and they are widely used in real-world applications; a recent survey showed that ~80% of SQL developers encounter Nulls in their databases [34].

Despite the prevalence of Nulls, there has not been a deep investigation into how to best handle them in a modern file format that is designed for analytical workloads processing columnar data.

DaMoN '24, June 10, 2024, Santiago, AA, Chile

ACM ISBN 979-8-4007-0667-7/24/06
https://doi.org/10.1145/3662010.3663452

Logical: data	Compact: data	Compact: valid?	Placeholder: data	Placeholder: valid?
11	11	1	11	1
NULL	33	0	0	0
33	44	1	33	1
44	66	1	44	1
NULL		0	0	0
66		1	66	1

Figure 1: Null Representations – Examples of `Compact` and `Placeholder` representation schemes for a logical data set.

Today's most widely used columnar file formats (i.e., Apache Parquet [7], Apache ORC [6]) follow the same **`Compact`** layout as the seminal C-Store DBMS from the 2000s [13]. For each nullable attribute in a table, C-Store's scheme stores non-Null (fixed-width) values in densely packed contiguous columns. To handle Nulls, the scheme maintains a separate bitmap to record whether the value for an attribute at a given position is Null or not. Storing values in this manner enables better compression and improves query performance. However, because the `Compact` layout does not store Nulls, a tuple's logical position in a table may not match its physical position in the column, hampering random access ability.

An alternative approach is to store the Null values in place. That is, instead of pruning the Nulls out, this scheme uses a default value (e.g., zero, `INT_MIN`) as a placeholder to represent Null for a given tuple. The scheme still maintains a bitmap to indicate whether a position contains Null or not because the placeholder value may collide with a non-null value. Without further compression, this **`Placeholder`** layout always uses the same amount of storage space whether or not values are Null, but facilitates random access and vectorized execution. Recent systems and formats such as DB2 BLU [32], DuckDB [31], Apache Arrow[1] [4], and BtrBlocks [23] adopt this `Placeholder` layout. Figure 1 shows the difference between `Compact` and `Placeholder` layout.

Many DBMSs use a combination of Parquet and Arrow storage to represent data on disk and in-memory, respectively [5, 9, 10]. However, the different representation of Nulls between `Compact` (Parquet) and `Placeholder` (Arrow) introduces performance overhead. As shown in Figure 2, the time spent on format conversion from Parquet to Arrow, which represents a common deserialization

*Huanchen Zhang is also affiliated with Shanghai Qi Zhi Institute.
[1] The Arrow format does not specify Nulls to be any particular placeholder value, but implementations (C++ and Rust) fill it as zero to make the memory fully initialized.

LARGE VALUES

Most DBMSs do not allow a tuple to exceed the size of a single page.

To store values that are larger than a page, the DBMS uses separate **overflow** storage pages.

→ Postgres: TOAST (>2KB)
→ MySQL: Overflow (>½ size of page)
→ SQL Server: Overflow (>size of page)

Lots of potential optimizations:

→ Overflow Compression, German Strings

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  data INT,
  contents TEXT
);
```

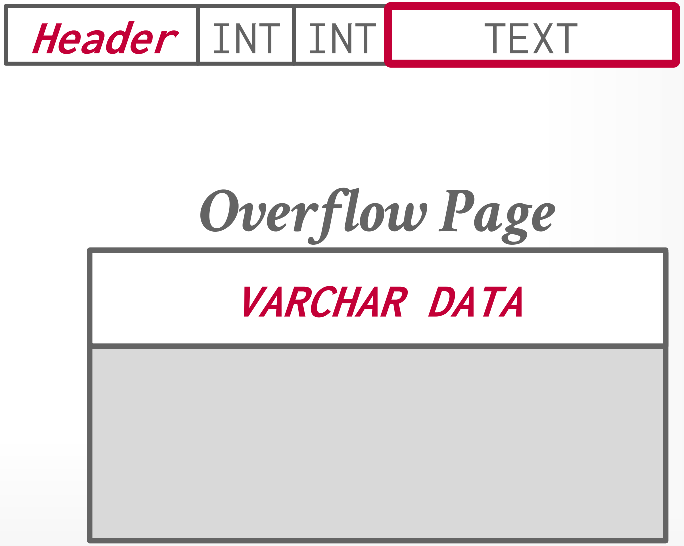
LARGE VALUES

Most DBMSs do not allow a tuple to exceed the size of a single page.

To store values that are larger than a page, the DBMS uses separate **overflow** storage pages.
→ Postgres: TOAST (>2KB)
→ MySQL: Overflow (>½ size of page)
→ SQL Server: Overflow (>size of page)

Lots of potential optimizations:
→ Overflow Compression, German Strings

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  data INT,
  contents TEXT
);
```



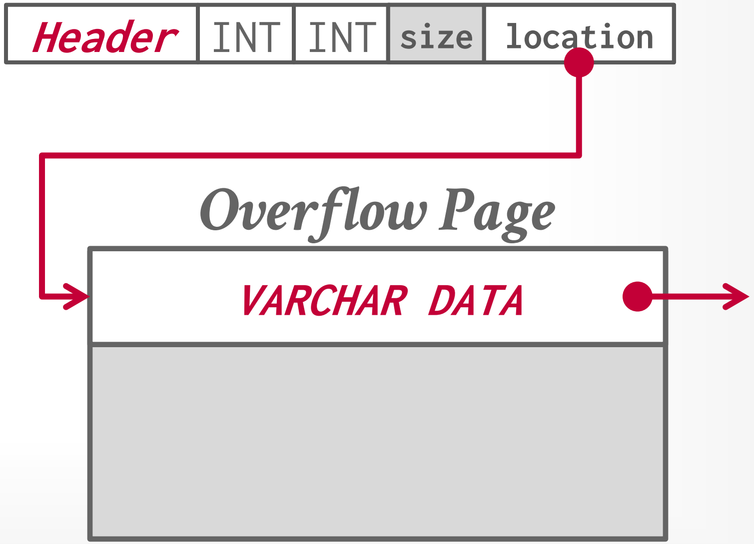
LARGE VALUES

Most DBMSs do not allow a tuple to exceed the size of a single page.

To store values that are larger than a page, the DBMS uses separate **overflow** storage pages.
→ Postgres: TOAST (>2KB)
→ MySQL: Overflow (>½ size of page)
→ SQL Server: Overflow (>size of page)

Lots of potential optimizations:
→ Overflow Compression, German Strings

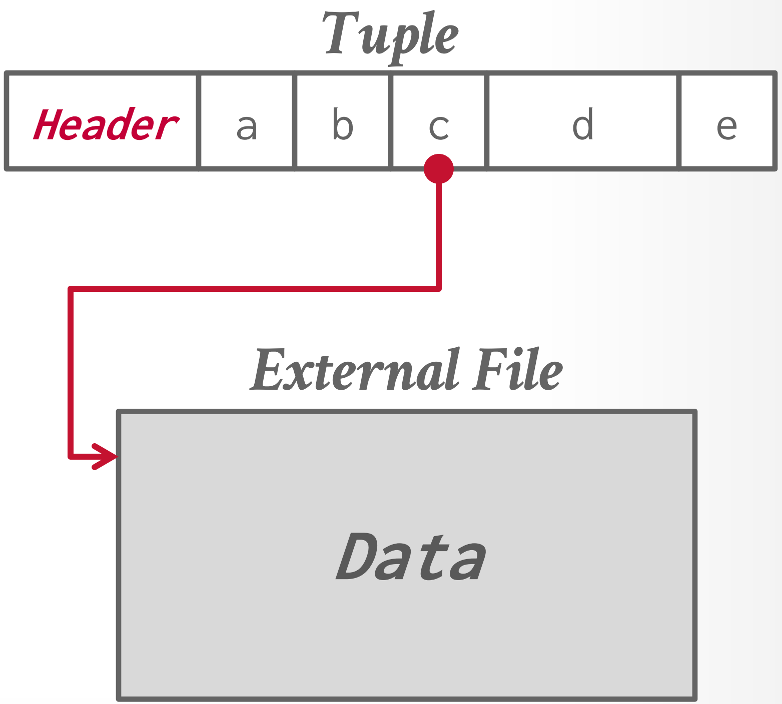
```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  data INT,
  contents TEXT
);
```



EXTERNAL VALUE STORAGE

Some systems allow you to store a large value in an external file.
Treated as a **BLOB** type.
- → Oracle: **BFILE** data type
- → Microsoft: **FILESTREAM** data type

The DBMS **cannot** manipulate the contents of an external file.
- → No durability protections.
- → No transaction protections.



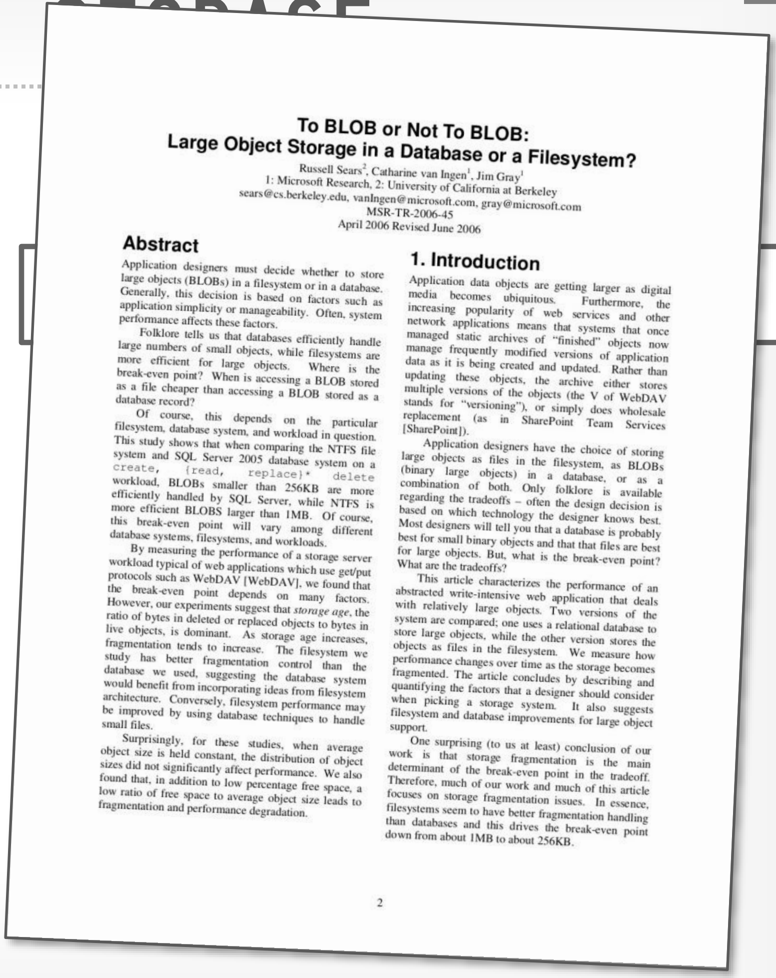
EXTERNAL VALUE

Some systems allow you to store a large value in an external file.
Treated as a **BLOB** type.

→ Oracle: **BFILE** data type
→ Microsoft: **FILESTREAM** data type

The DBMS **cannot** manipulate the contents of an external file.

→ No durability protections.
→ No transaction protections.



To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem?

Russell Sears[2], Catharine van Ingen[1], Jim Gray[1]
1: Microsoft Research, 2: University of California at Berkeley
sears@cs.berkeley.edu, vanIngen@microsoft.com, gray@microsoft.com

MSR-TR-2006-45
April 2006 Revised June 2006

Abstract

Application designers must decide whether to store large objects (BLOBs) in a filesystem or in a database. Generally, this decision is based on factors such as application simplicity or manageability. Often, system performance affects these factors.

Folklore tells us that databases efficiently handle large numbers of small objects, while filesystems are more efficient for large objects. Where is the break-even point? When is accessing a BLOB stored as a file cheaper than accessing a BLOB stored as a database record?

Of course, this depends on the particular filesystem, database system, and workload in question. This study shows that when comparing the NTFS file system and SQL Server 2005 database system on a `create, {read, replace}* delete` workload, BLOBs smaller than 256KB are more efficiently handled by SQL Server, while NTFS is more efficient BLOBS larger than 1MB. Of course, this break-even point will vary among different database systems, filesystems, and workloads.

By measuring the performance of a storage server workload typical of web applications which use get/put protocols such as WebDAV [WebDAV], we found that the break-even point depends on many factors. However, our experiments suggest that *storage age*, the ratio of bytes in deleted or replaced objects to bytes in live objects, is dominant. As storage age increases, fragmentation tends to increase. The filesystem we study has better fragmentation control than the database we used, suggesting the database system would benefit from incorporating ideas from filesystem architecture. Conversely, filesystem performance may be improved by using database techniques to handle small files.

Surprisingly, for these studies, when average object size is held constant, the distribution of object sizes did not significantly affect performance. We also found that, in addition to low percentage free space, a low ratio of free space to average object size leads to fragmentation and performance degradation.

1. Introduction

Application data objects are getting larger as digital media becomes ubiquitous. Furthermore, the increasing popularity of web services and other network applications means that systems that once managed static archives of "finished" objects now manage frequently modified versions of application data as it is being created and updated. Rather than updating these objects, the archive either stores multiple versions of the objects (the V of WebDAV stands for "versioning"), or simply does wholesale replacement (as in SharePoint Team Services [SharePoint]).

Application designers have the choice of storing large objects as files in the filesystem, as BLOBs (binary large objects) in a database, or as a combination of both. Only folklore is available regarding the tradeoffs – often the design decision is based on which technology the designer knows best. Most designers will tell you that a database is probably best for small binary objects and that that files are best for large objects. But, what is the break-even point? What are the tradeoffs?

This article characterizes the performance of an abstracted write-intensive web application that deals with relatively large objects. Two versions of the system are compared; one uses a relational database to store large objects, while the other version stores the objects as files in the filesystem. We measure how performance changes over time as the storage becomes fragmented. The article concludes by describing and quantifying the factors that a designer should consider when picking a storage system. It also suggests filesystem and database improvements for large object support.

One surprising (to us at least) conclusion of our work is that storage fragmentation is the main determinant of the break-even point in the tradeoff. Therefore, much of our work and much of this article focuses on storage fragmentation issues. In essence, filesystems seem to have better fragmentation handling than databases and this drives the break-even point down from about 1MB to about 256KB.

2

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:
- → **NSM**: Copy tuples into new region in memory.
- → **DSM**: Just create the new column segment on disk.

DROP COLUMN:
- → **NSM #1**: Copy tuples into new region of memory.
- → **NSM #2**: Mark column as "deprecated", clean up later.
- → **DSM**: Just drop the column and free memory.

CHANGE COLUMN:
- → Check whether the conversion is allowed to happen. Depends on default values.

INDEXES

CREATE INDEX:

→ Scan the entire table and populate the index.
→ Have to 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:

→ Just 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…