Carnegie Mellon University

Intro to Database Systems (15-445/645)

Lecture #04

Database Storage *Part 2*

FALL 2023 » Prof. Andy Pavlo • Prof. Jignesh Patel

ADMINISTRIVIA

Homework #1 is due September 15th @ 11:59pm.

Project #1 is due October 1st @ 11:59pm.

UPCOMING DATABASE TALKS

Qdrant (ML⇄DB Seminar)
→ Monday Sept 11th @ 4:30pm

Databricks
→ Tuesday Sept 12th @ 6:00pm

OtterTune (ML⇄DB Seminar)
→ Monday Sept 18th @4:30pm

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.

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

TODAY'S AGENDA

Log-Structured Storage

Index-Organized Storage

Data Representation

LOG-STRUCTURED STORAGE

Instead of storing tuples in pages, 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.

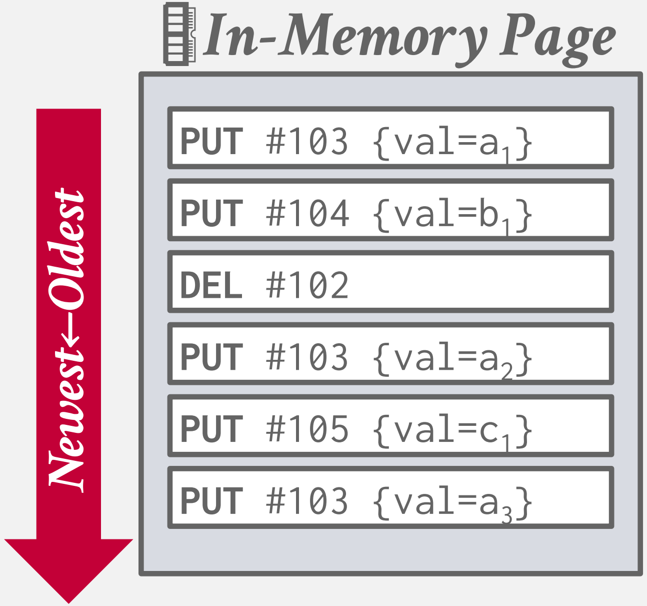
The DBMS appends new log entries to an in-memory buffer and then writes out the changes sequentially to disk.

LOG-STRUCTURED STORAGE

DBMS stores log records that contain 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.

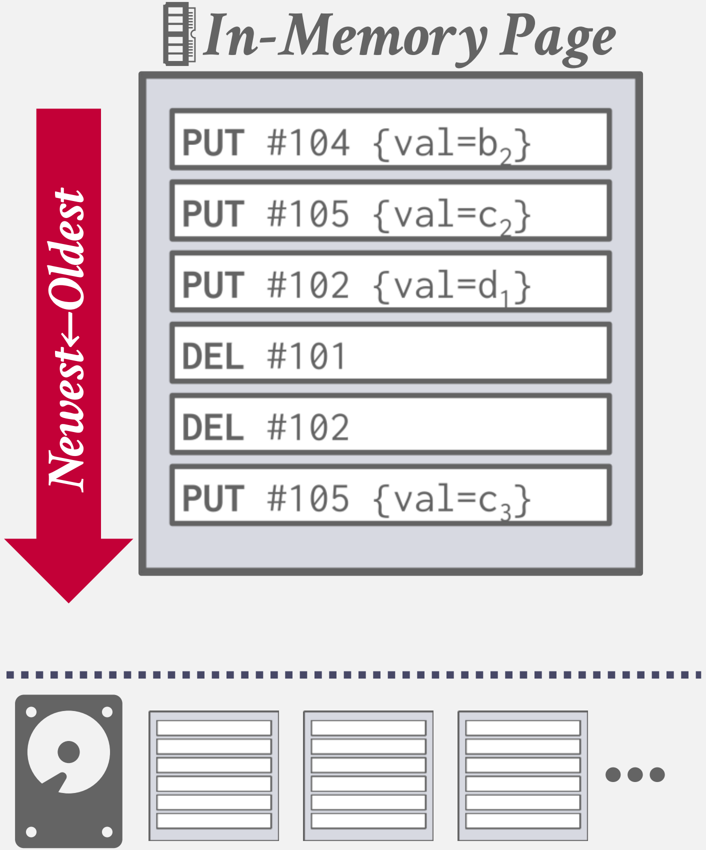
As the application makes changes to the database, the DBMS appends log records to the end of the file without checking previous log records.



LOG-STRUCTURED STORAGE

When the page gets full, the DBMS writes it out disk and starts filling up the next page with records.
→ All disk writes are sequential.
→ On-disk pages are immutable.

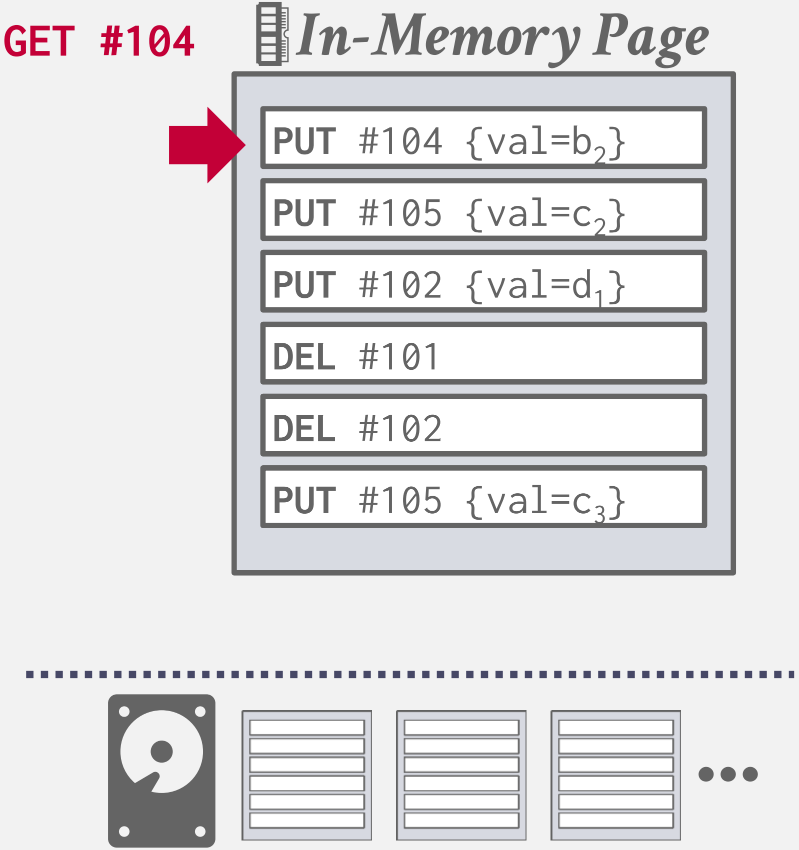
The DBMS may also flush partially full pages for transactions but we will ignore that for now…



LOG-STRUCTURED STORAGE

To read a tuple with a given id, the DBMS finds the newest log record corresponding to that id.

→ Scan log from newest to oldest.



LOG-STRUCTURED STORAGE

To read a tuple with a given id, the DBMS finds the newest log record corresponding to that id.

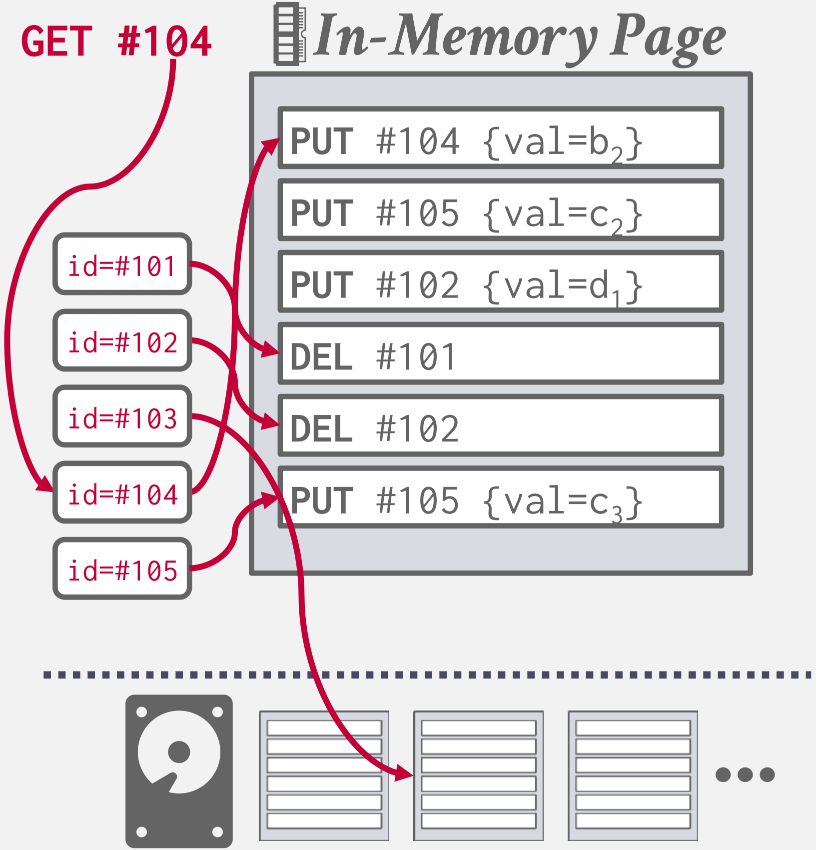
→ Scan log from newest to oldest.

Maintain an index that maps a tuple id to the newest log record.

→ If log record is in-memory, just read it.

→ If log record is on a disk page, retrieve it.

→ We will discuss indexes in two weeks.



LOG-STRUCTURED STORAGE

To read a tuple with a given id, the DBMS finds the newest log record corresponding to that id.

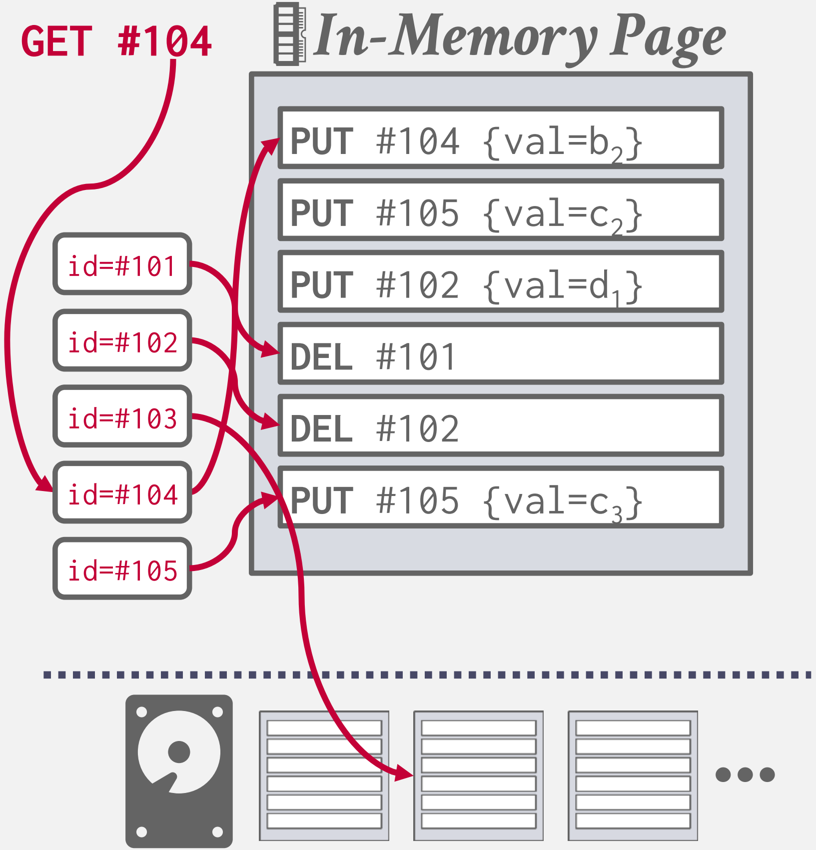
→ Scan log from newest to oldest.

Maintain an index that maps a tuple id to the newest log record.

→ If log record is in-memory, just read it.

→ If log record is on a disk page, retrieve it.

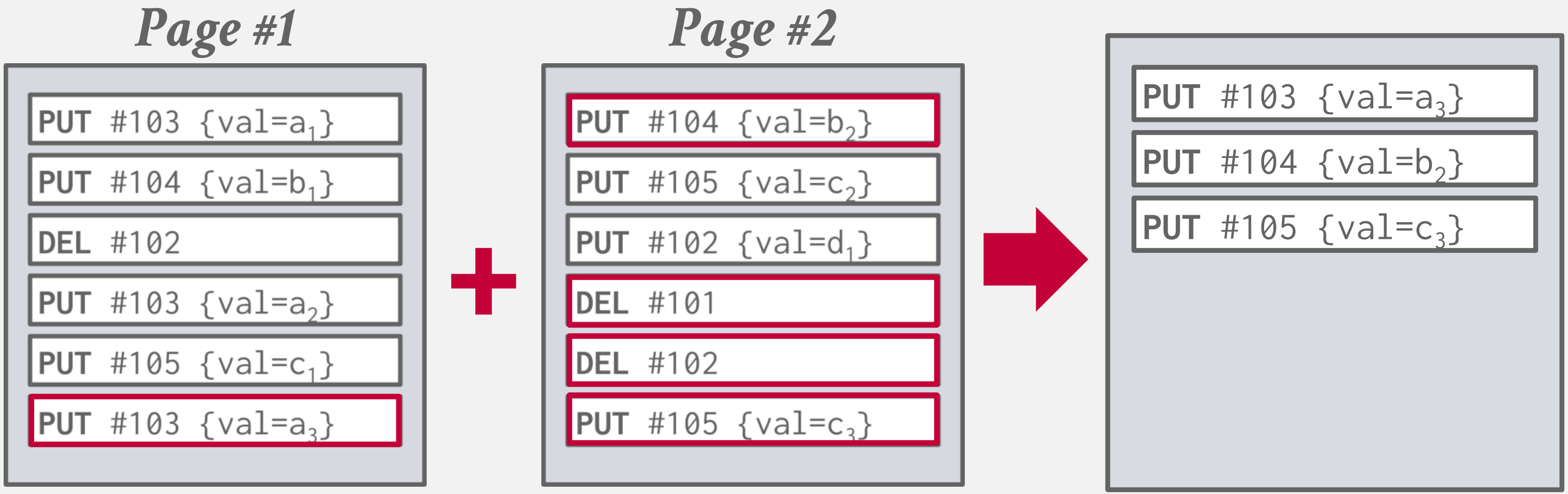
→ We will discuss indexes in two weeks.



LOG-STRUCTURED COMPACTION

DBMS (usually) does not need to maintain all older log entries for a tuple indefinitely.

- → Periodically compact pages to reduce wasted space.



Page #1

PUT #103 {val=a_1}
PUT #104 {val=b_1}
DEL #102
PUT #103 {val=a_2}
PUT #105 {val=c_1}
PUT #103 {val=a_3}

+

Page #2

PUT #104 {val=b_2}
PUT #105 {val=c_2}
PUT #102 {val=d_1}
DEL #101
DEL #102
PUT #105 {val=c_3}

PUT #103 {val=a_3}
PUT #104 {val=b_2}
PUT #105 {val=c_3}

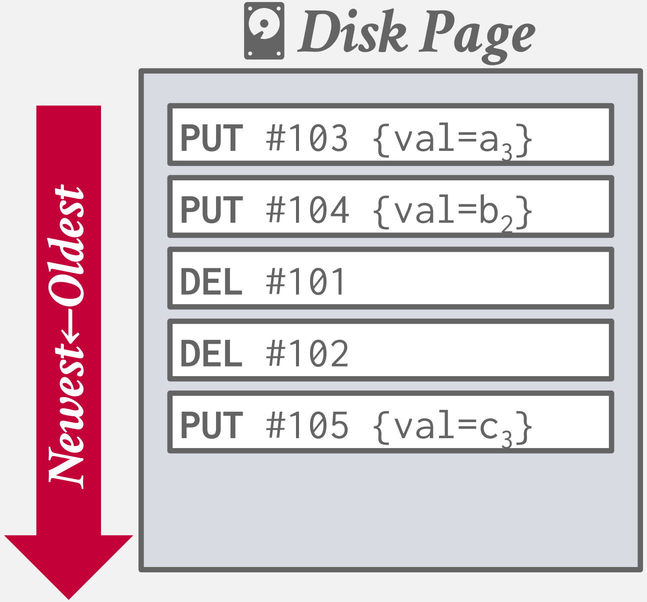
LOG-STRUCTURED COMPACTION

After a page is compacted, the DBMS does not need to maintain temporal ordering of records within the page.

- → Each tuple id is guaranteed to appear at most once in the page.

The DBMS can instead sort the page based on id order to improve efficiency of future look-ups.

- → Called Sorted String Tables (SSTables)
- → Embed indexes / filters in the header for reducing search times.



PUT #103 {val=a_3}
PUT #104 {val=b_2}
DEL #101
DEL #102
PUT #105 {val=c_3}

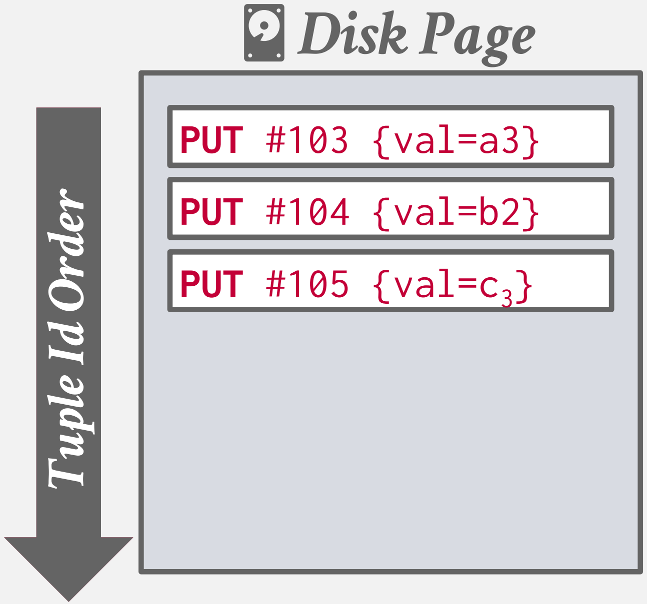
LOG-STRUCTURED COMPACTION

After a page is compacted, the DBMS does not need to maintain temporal ordering of records within the page.

- → Each tuple id is guaranteed to appear at most once in the page.

The DBMS can instead sort the page based on id order to improve efficiency of future look-ups.

- → Called Sorted String Tables (SSTables)
- → Embed indexes / filters in the header for reducing search times.



LOG-STRUCTURED COMPACTION

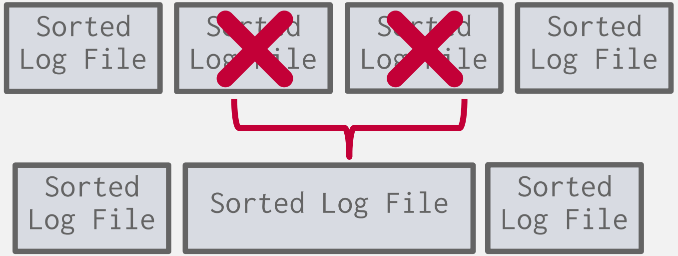
Coalesce larger disk-resident log files into smaller files by removing unnecessary records.

Universal Compaction

LOG-STRUCTURED COMPACTION

Coalesce larger disk-resident log files into smaller files by removing unnecessary records.

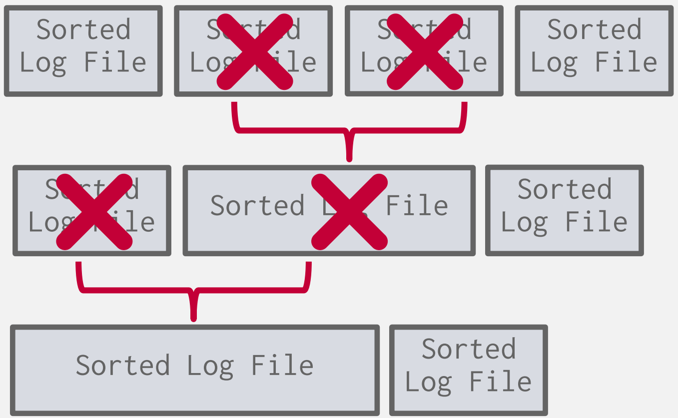
Universal Compaction



LOG-STRUCTURED COMPACTION

Coalesce larger disk-resident log files into smaller files by removing unnecessary records.

Universal Compaction



Sorted Log File
Sorted Log File
Sorted Log File
Sorted Log File

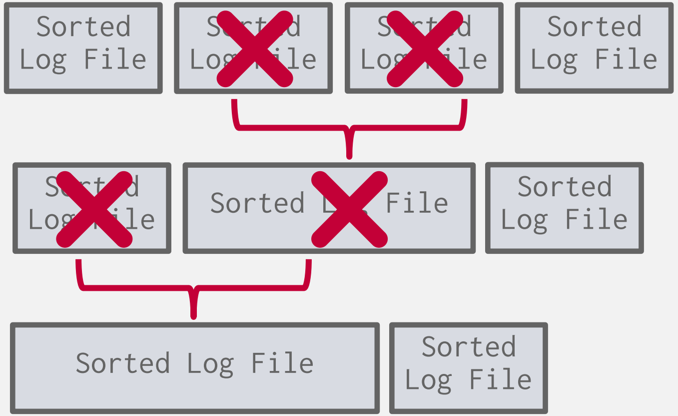
Sorted Log File
Sorted Log File
Sorted Log File

Sorted Log File
Sorted Log File

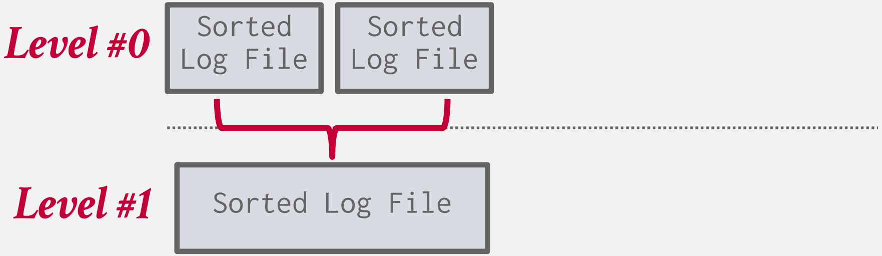
LOG-STRUCTURED COMPACTION

Coalesce larger disk-resident log files into smaller files by removing unnecessary records.

Universal Compaction



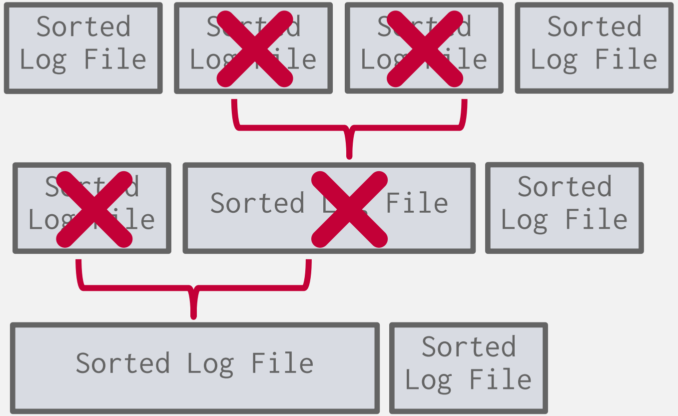
Level Compaction



LOG-STRUCTURED COMPACTION

Coalesce larger disk-resident log files into smaller files by removing unnecessary records.

Universal Compaction

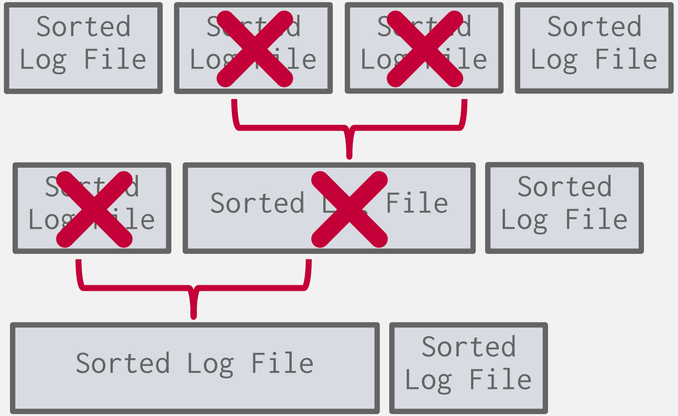


Level Compaction

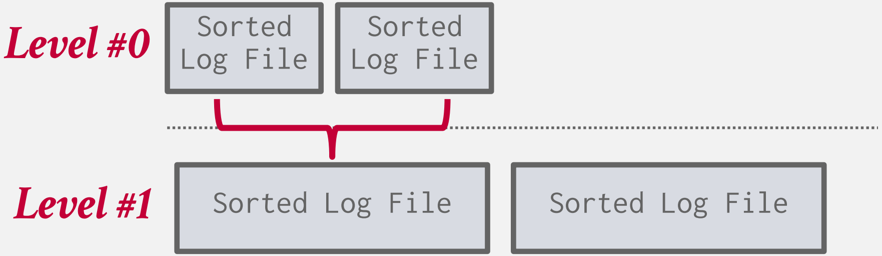
LOG-STRUCTURED COMPACTION

Coalesce larger disk-resident log files into smaller files by removing unnecessary records.

Universal Compaction



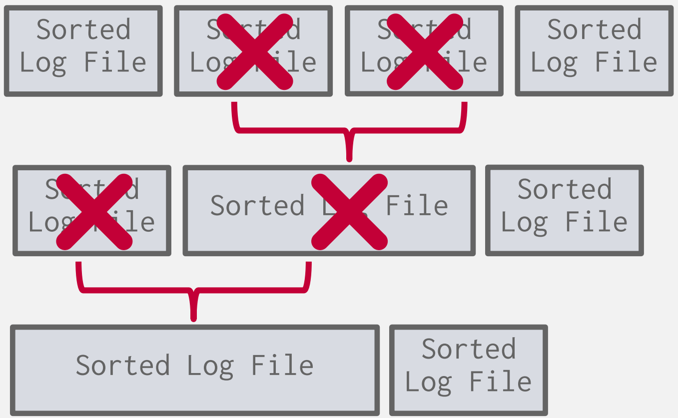
Level Compaction



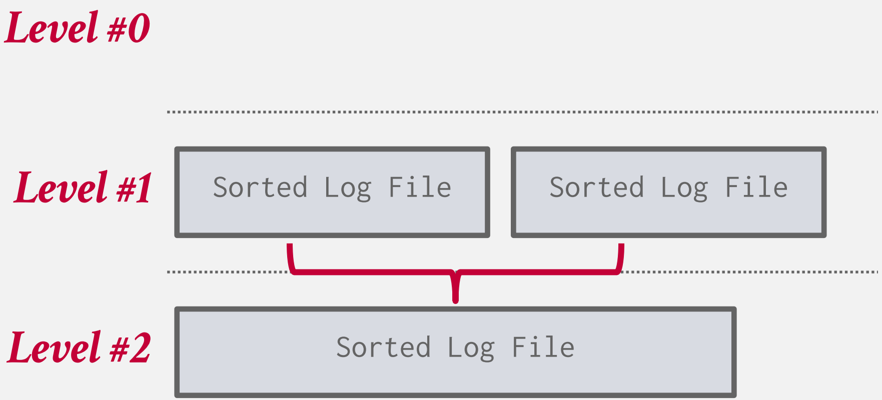
LOG-STRUCTURED COMPACTION

Coalesce larger disk-resident log files into smaller files by removing unnecessary records.

Universal Compaction



Level Compaction



DISCUSSION

Log-structured storage managers are more common today. This is partly due to the proliferation of RocksDB.

What are some downsides of this approach?

→ Write-Amplification
→ Compaction is Expensive

RocksDB
levelDB
APACHE HBASE
yugabyteDB
fauna
Dgraph
TiDB
CockroachDB
cassandra

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.

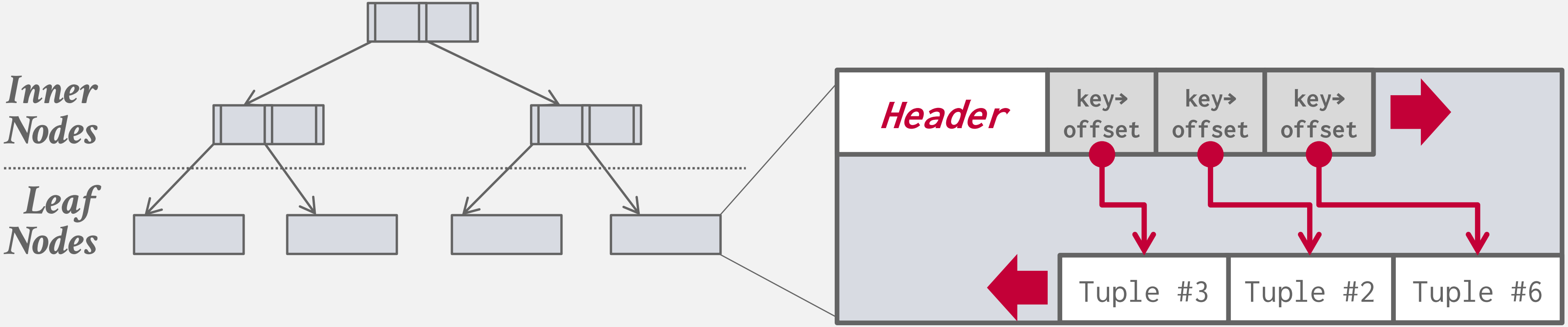
Tuples are typically sorted in page based on key.

SQLite

MySQL

ORACLE

Microsoft SQL Server



TUPLE STORAGE

A tuple is essentially a sequence of bytes.

It's 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

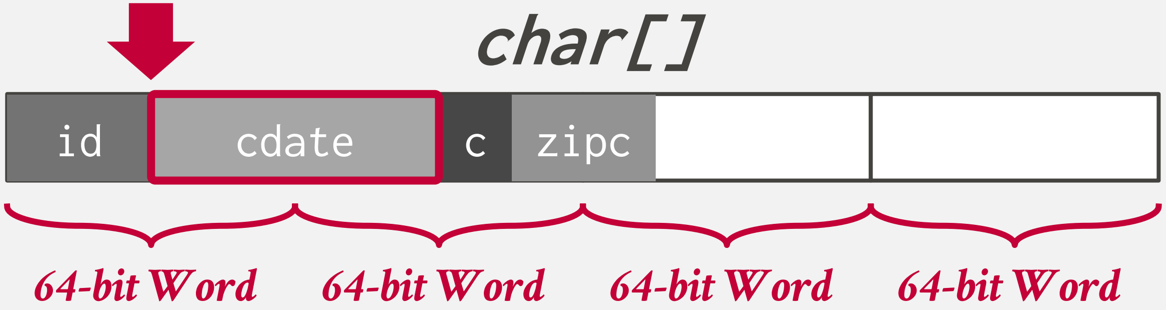
```
CREATE TABLE AndySux (
  id INT PRIMARY KEY,
  value BIGINT
);
```

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



WORD-ALIGNED TUPLES

Approach #1: Perform Extra Reads

→ Execute two reads to load the appropriate parts of the data word and reassemble them.

Approach #2: Random Reads

→ Read some unexpected combination of bytes assembled into a 64-bit word.

Approach #3: Reject

→ Throw an exception and hope app handles it.

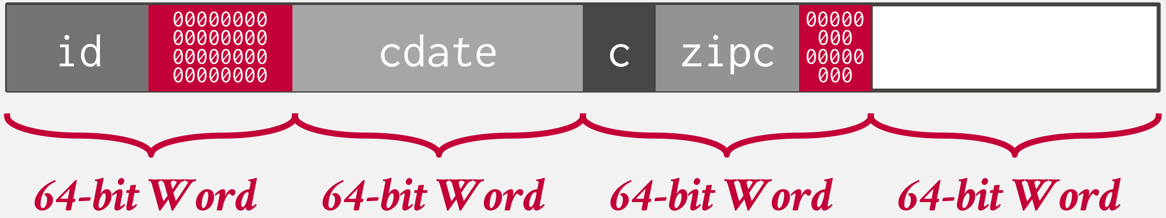
Source: Levente Kurusa

WORD-ALIGNMENT: PADDING

Add empty bits after attributes to ensure that tuple is word aligned.

```
CREATE TABLE dj2pl (
  id INT PRIMARY KEY,
  cdate TIMESTAMP,
  color CHAR(2),
  zipcode INT
);
```

32-bits
64-bits
16-bits
32-bits



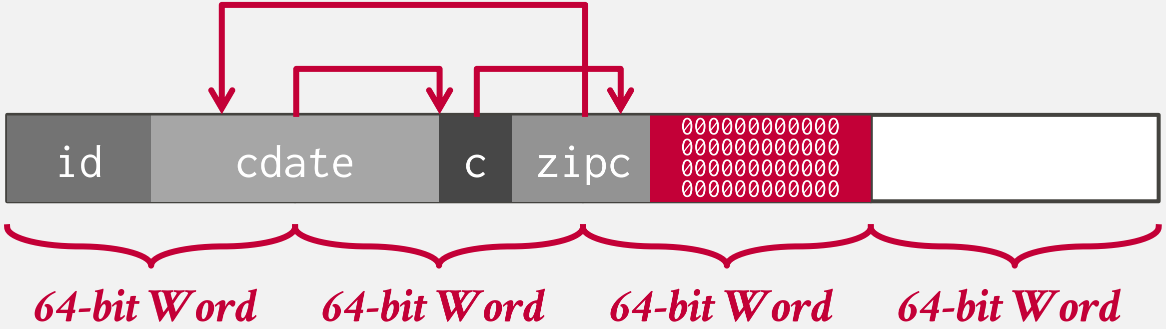
WORD-ALIGNMENT: REORDERING

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

→ May still have to use padding.

```
CREATE TABLE dj2pl (
  id INT PRIMARY KEY,
  cdate TIMESTAMP,
  color CHAR(2),
  zipcode INT
);
```

32-bits id
64-bits cdate
16-bits color
32-bits zipcode



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>

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.

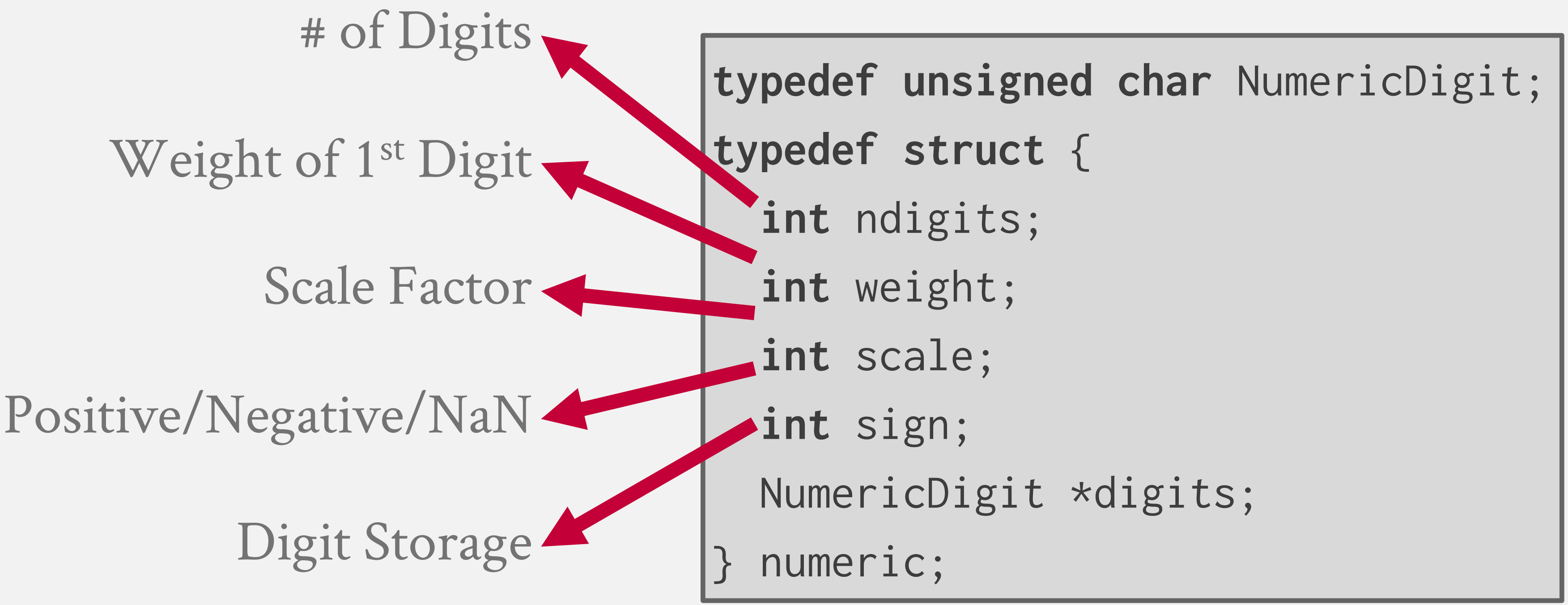
→ Example: **`NUMERIC`**, **`DECIMAL`**

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

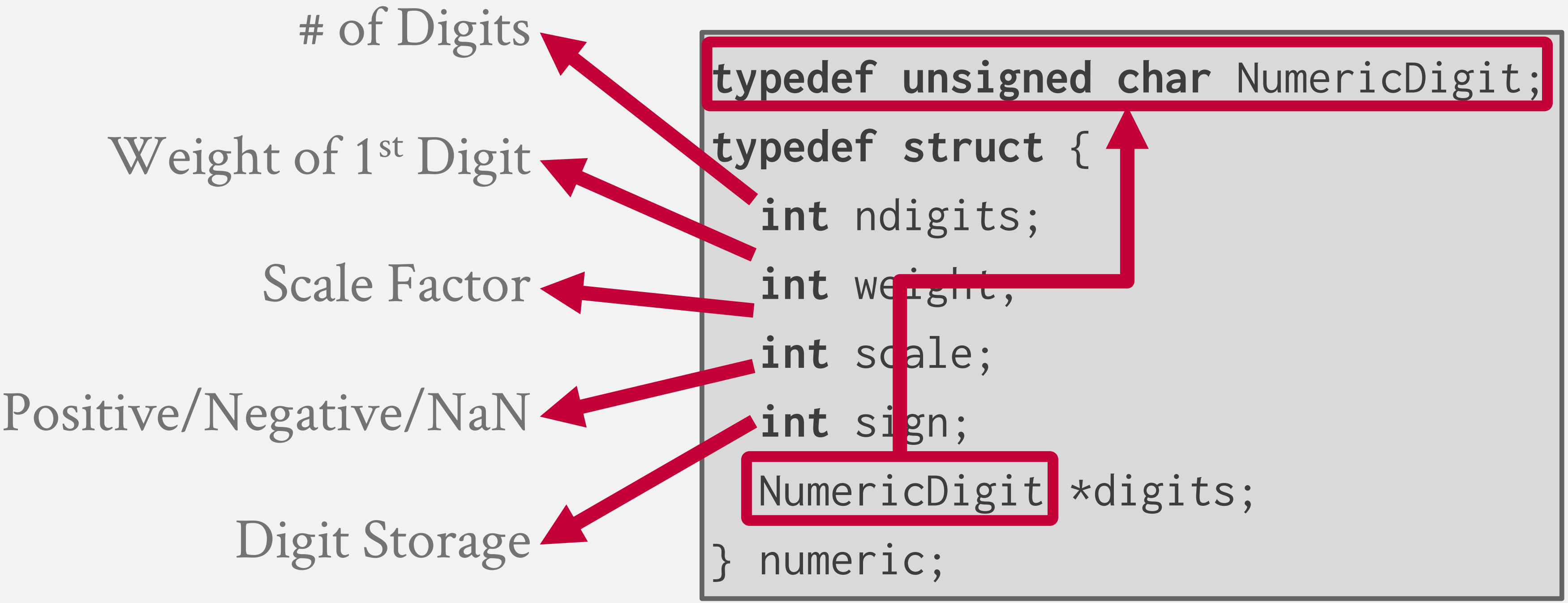
Scale Factor

Positive/Negative/NaN

Digit Storage

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

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;

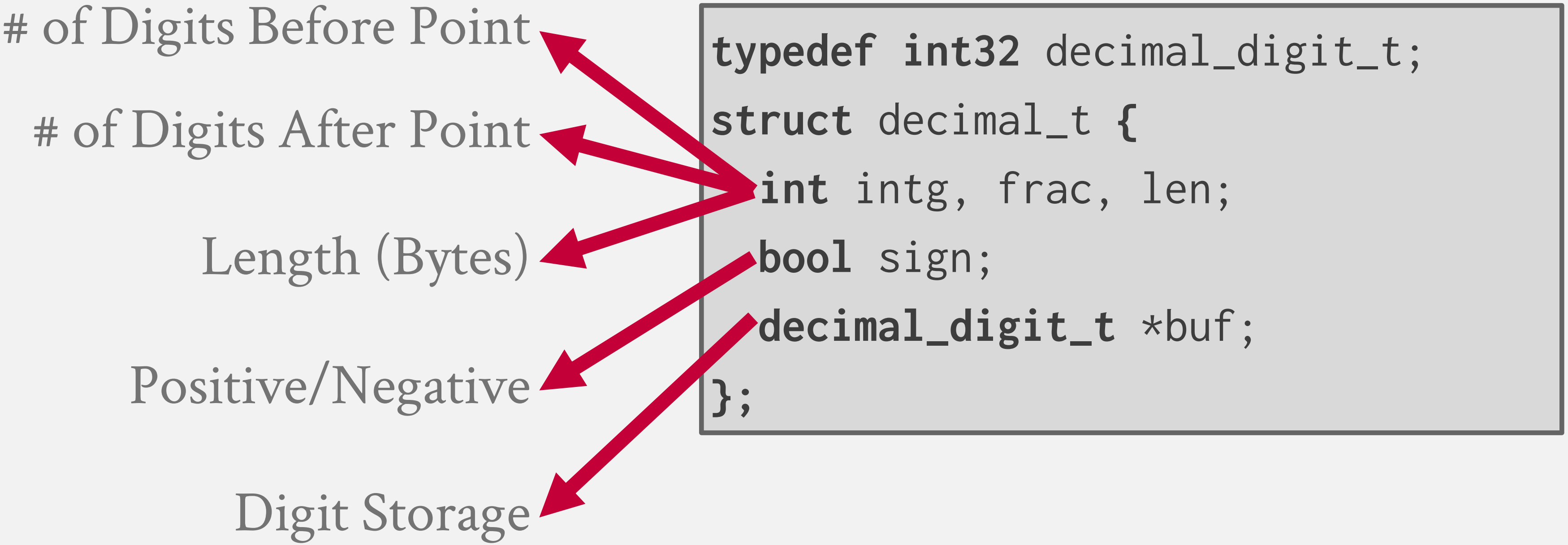
;

```
/* ----------
 * 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))
                     * ----------
```

MYSQL: NUMERIC



# of Digits Before Point

# of Digits After Point

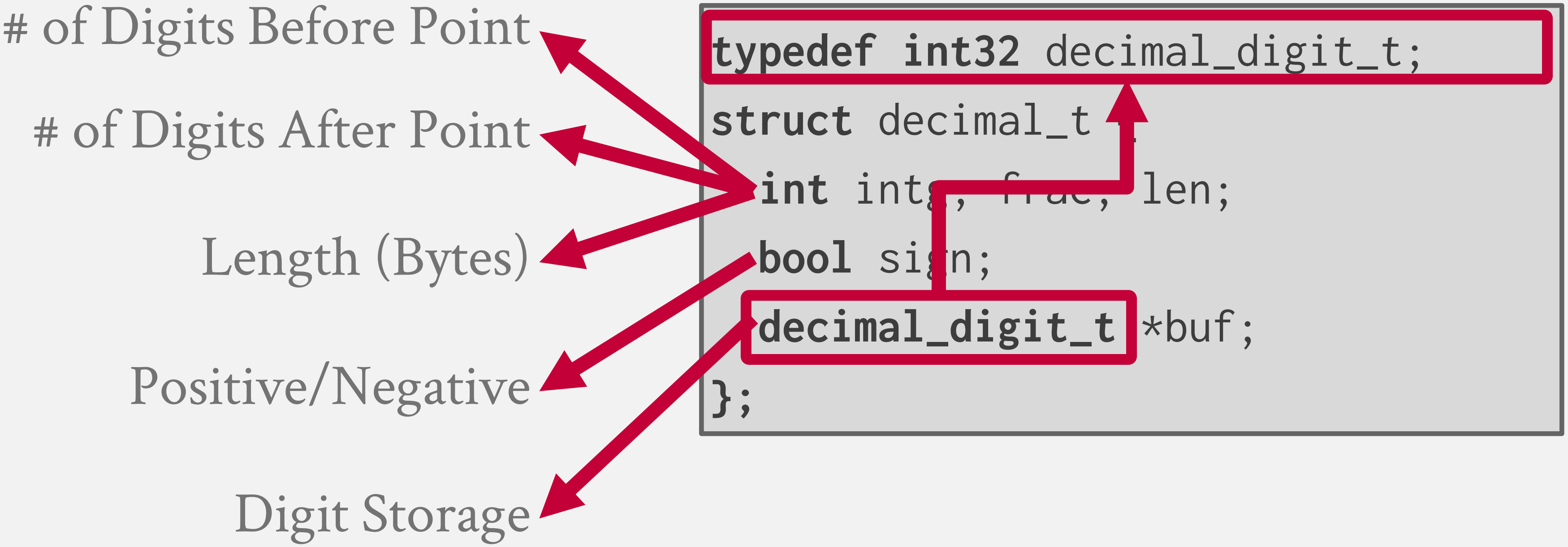
Length (Bytes)

Positive/Negative

Digit Storage

```
typedef int32 decimal_digit_t;
struct decimal_t {
  int intg, frac, len;
  bool sign;
  decimal_digit_t *buf;
};
```

MYSQL: NUMERIC



# of Digits Before Point

# of Digits After Point

Length (Bytes)

Positive/Negative

Digit Storage

```
typedef int32 decimal_digit_t;
struct decimal_t {
  int intg, frac, len;
  bool sign;
  decimal_digit_t *buf;
};
```

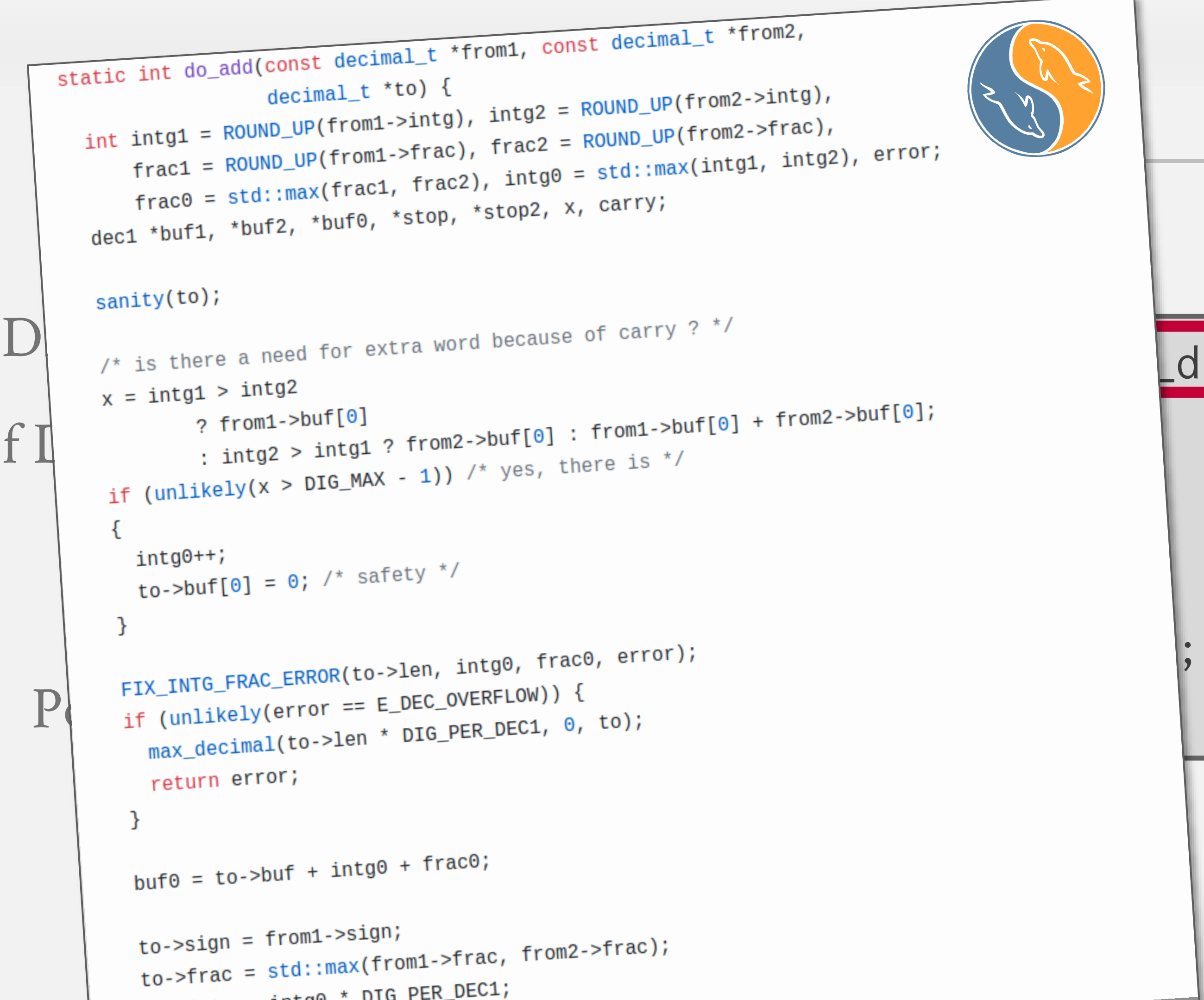
# of D

of

P

_digit_t;

;



```
static int do_add(const decimal_t *from1, const decimal_t *from2,
                  decimal_t *to) {
  int intg1 = ROUND_UP(from1->intg), intg2 = ROUND_UP(from2->intg),
      frac1 = ROUND_UP(from1->frac), frac2 = ROUND_UP(from2->frac),
      frac0 = std::max(frac1, frac2), intg0 = std::max(intg1, intg2), error;
  dec1 *buf1, *buf2, *buf0, *stop, *stop2, x, carry;

  sanity(to);

  /* is there a need for extra word because of carry ? */
  x = intg1 > intg2
          ? from1->buf[0]
          : intg2 > intg1 ? from2->buf[0] : from1->buf[0] + from2->buf[0];
  if (unlikely(x > DIG_MAX - 1)) /* yes, there is */
  {
    intg0++;
    to->buf[0] = 0; /* safety */
  }

  FIX_INTG_FRAC_ERROR(to->len, intg0, frac0, error);
  if (unlikely(error == E_DEC_OVERFLOW)) {
    max_decimal(to->len * DIG_PER_DEC1, 0, to);
    return error;
  }

  buf0 = to->buf + intg0 + frac0;

  to->sign = from1->sign;
  to->frac = std::max(from1->frac, from2->frac);
                intg0 * DIG_PER_DEC1;
```

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.

Choice #2: Special Values

→ Designate a value to represent `NULL` for a data type (e.g., `INT32_MIN`).

Choice #3: Per Attribute Null Flag

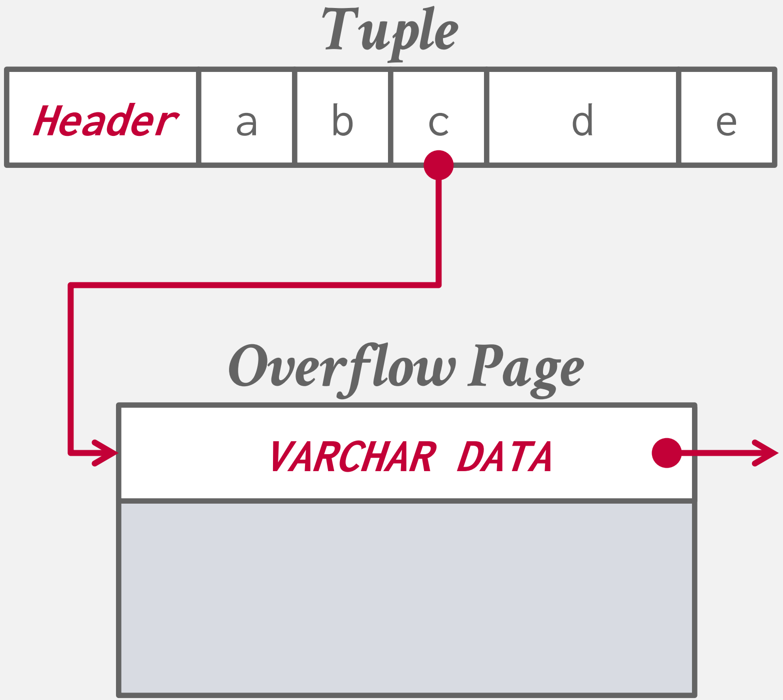
→ Store a flag that marks that a value is null.
→ Must use more space than just a single bit because this messes up with word alignment.

LARGE VALUES

Most DBMSs don't 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)



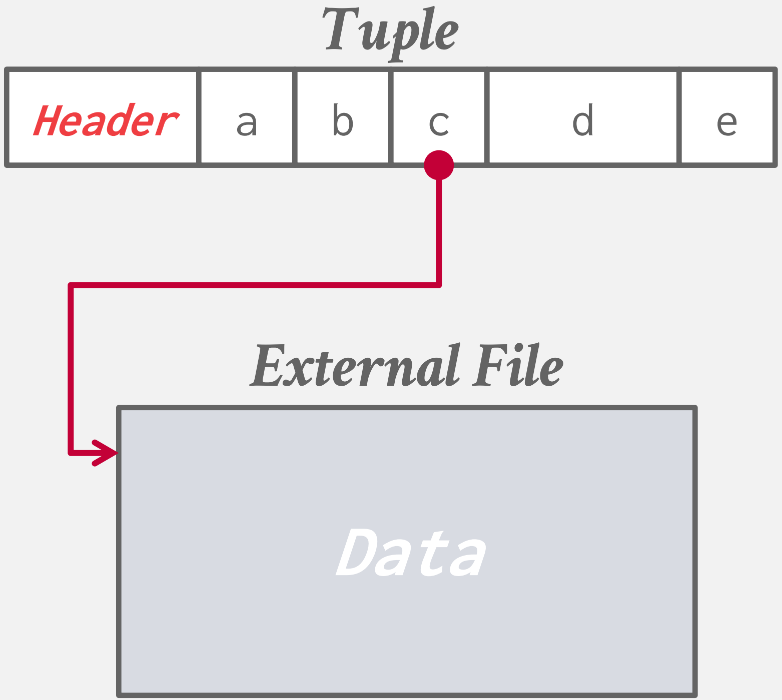
EXTERNAL VALUE STORAGE

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.



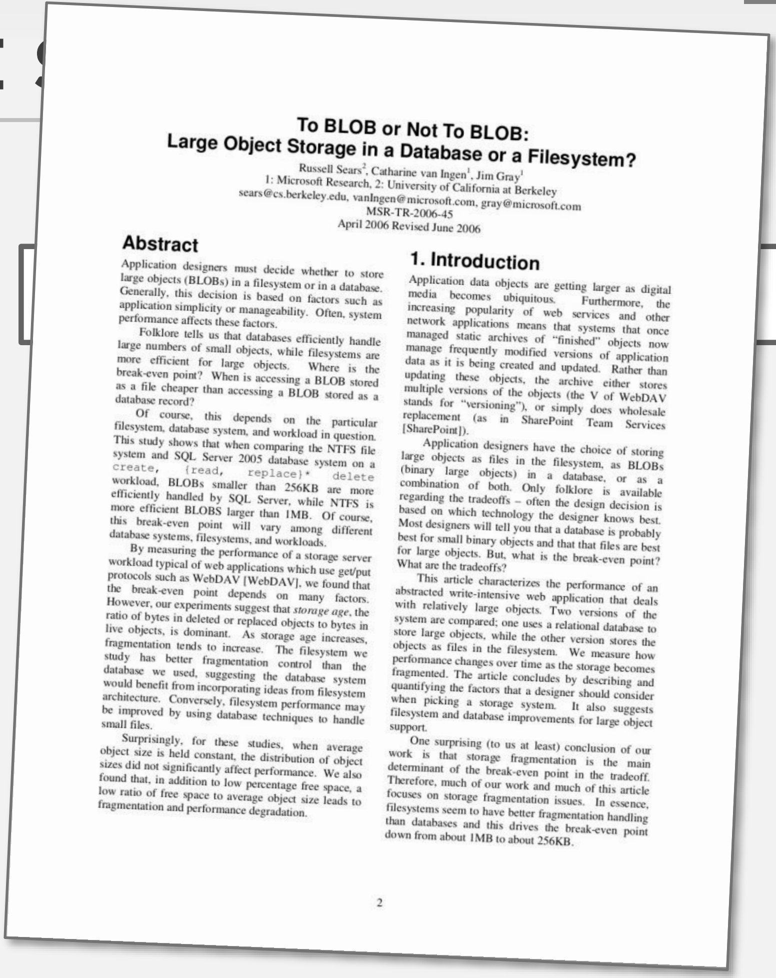
EXTERNAL VALUE

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

CONCLUSION

Log-structured storage is an alternative approach to the page-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.