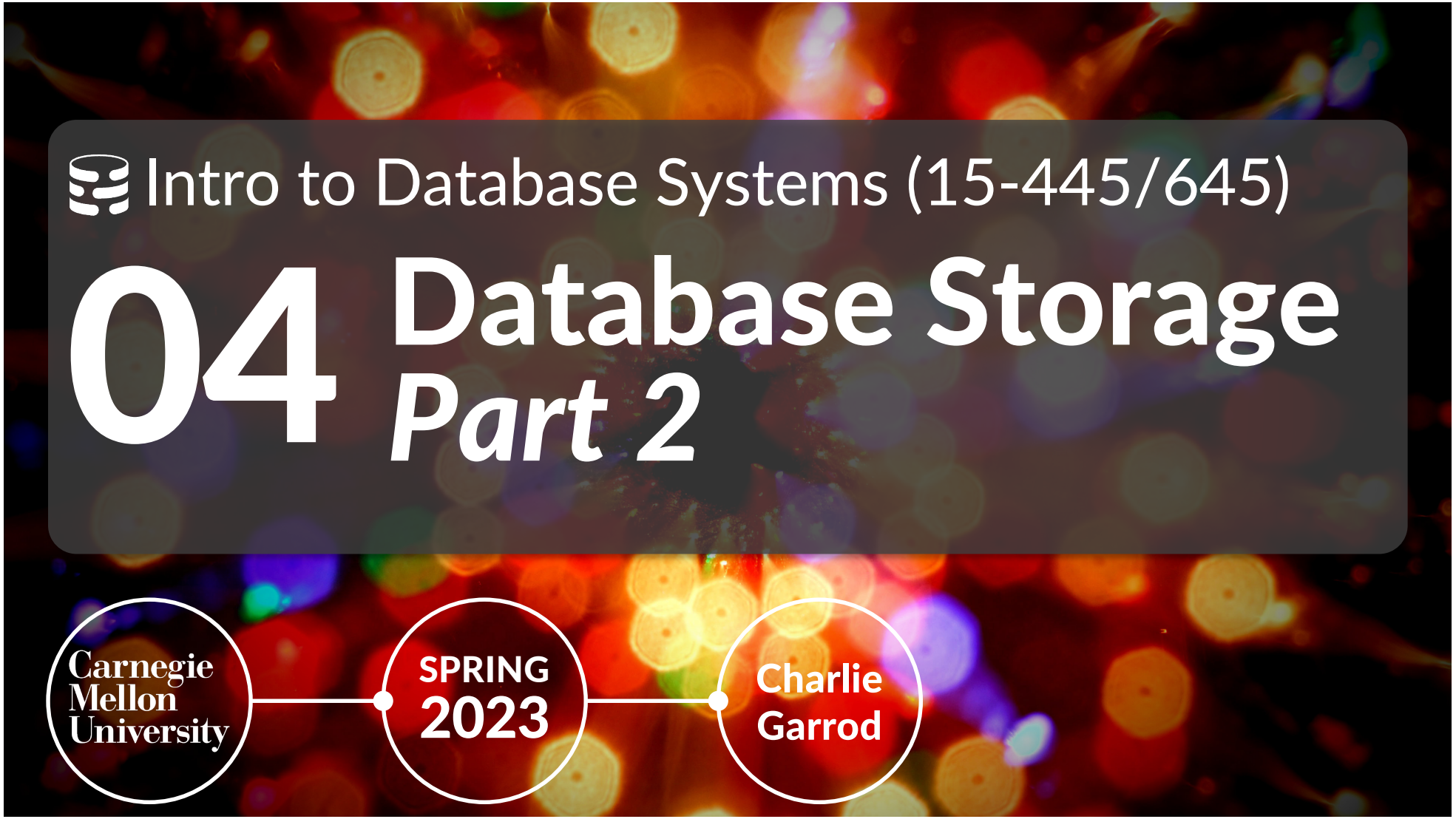

Intro to Database Systems (15-445/645)
04 Database Storage
Part 2
Carnegie Mellon University
SPRING 2023
Charlie Garrod

ADMINISTRIVIA

Project 0 due yesterday.

Homework 1 due this Friday, Feb 3rd.

Project 1 will be released later today.

DISK-ORIENTED ARCHITECTURE

The DBMS assumes that the primary storage location of the database is on non-volatile disk.

The DBMS's components manage the movement of data between non-volatile and volatile storage.

PAGE-ORIENTED ARCHITECTURE

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.
→ Overwrite existing data (if new data fits).

DISCUSSION

Problems with the slotted page design

→ Fragmentation
→ Useless Disk I/O
→ Random Disk I/O (e.g., update 20 tuples on 20 pages)

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

→ Examples: Some cloud storage, HDFS

TODAY'S AGENDA

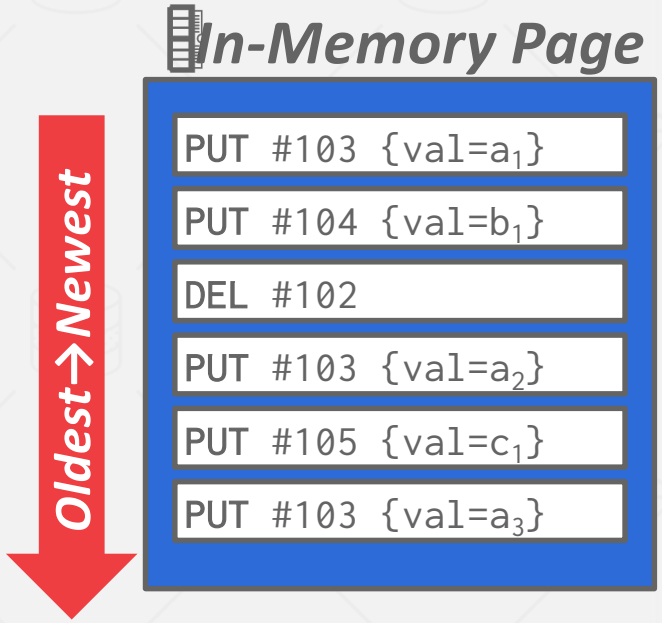
Log-Structured Storage
Data Representation
System Catalogs

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.

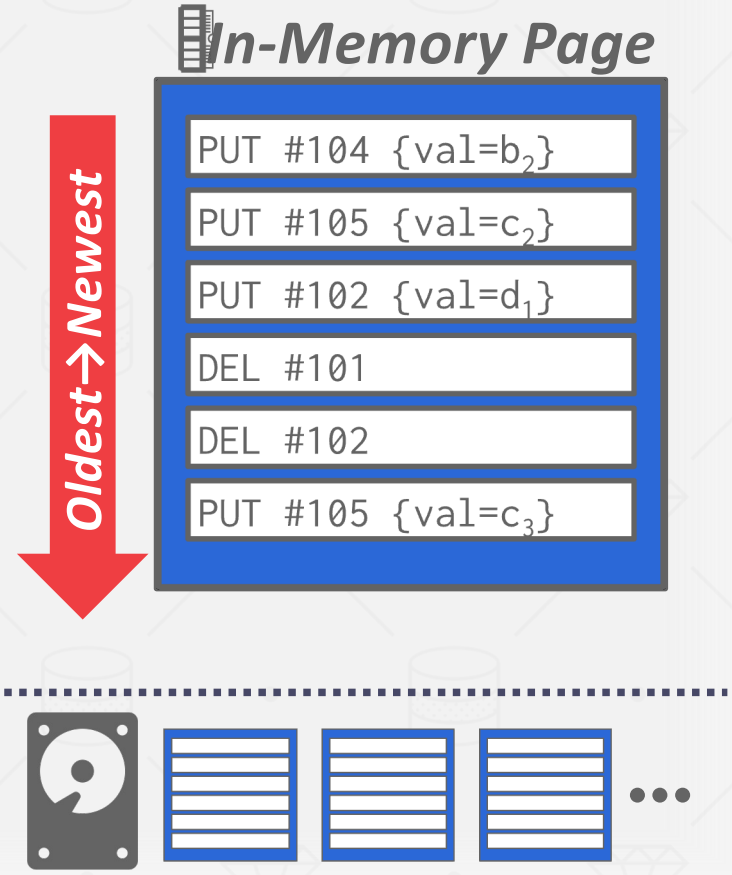


LOG-STRUCTURED STORAGE

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.



LOG-STRUCTURED STORAGE

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

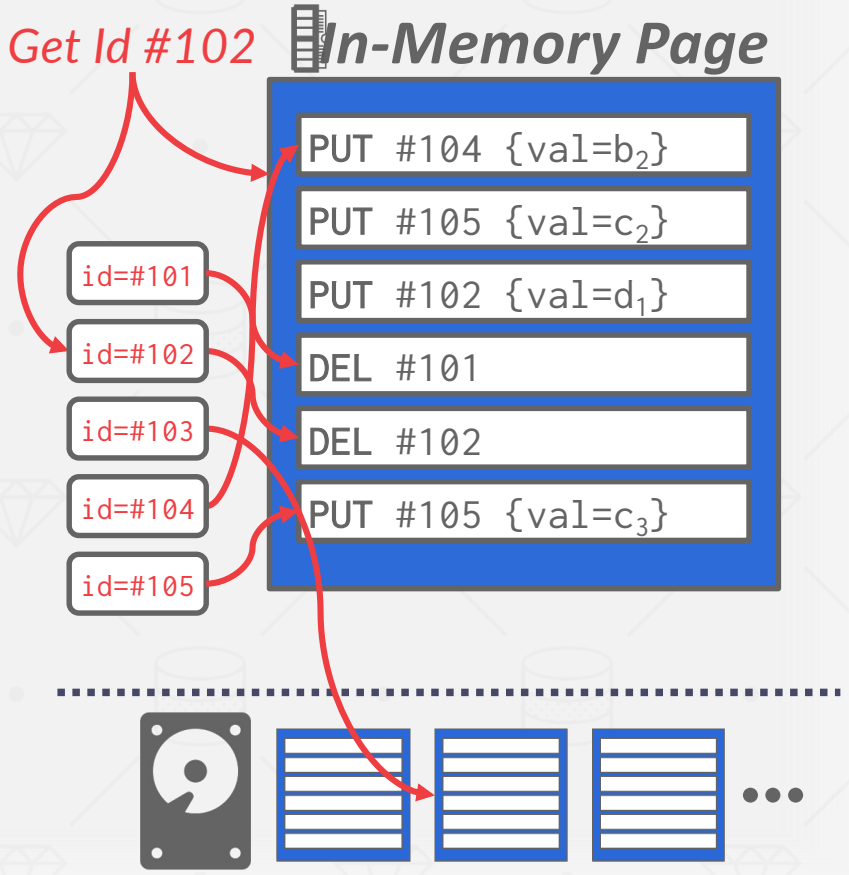
→ Scan log from newest to oldest.

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

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

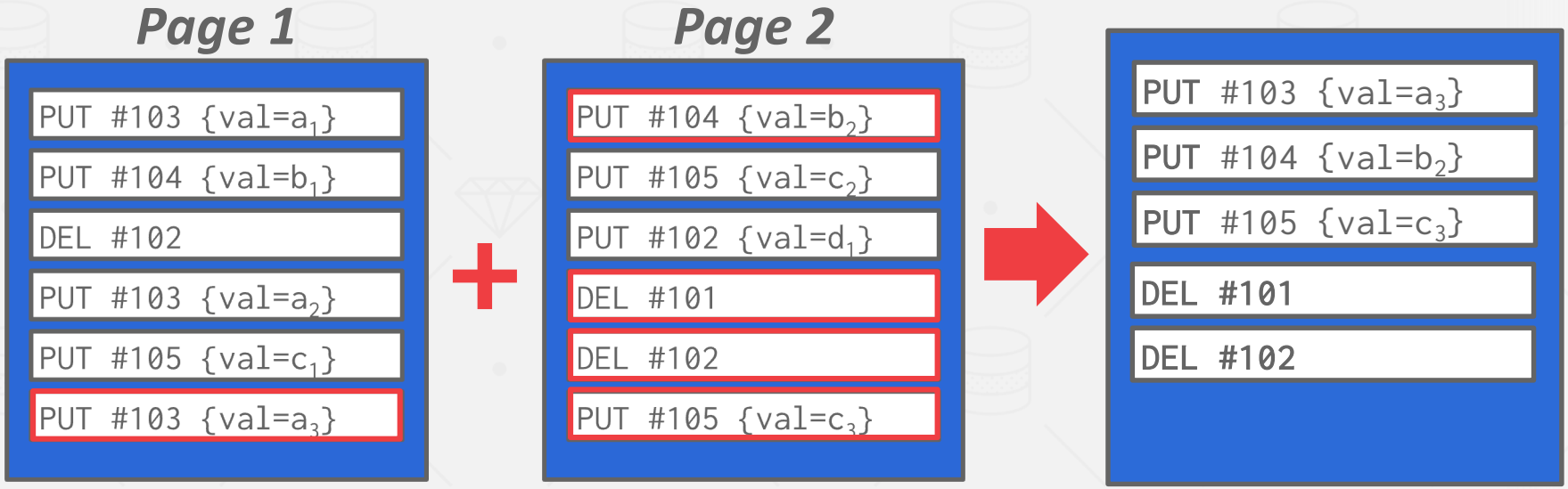
→ If log record is on a disk page, retrieve it.

→ We will discuss indexes in two weeks.



LOG-STRUCTURED COMPACTION

The log will grow forever. The DBMS needs to periodically compact pages to reduce wasted space.



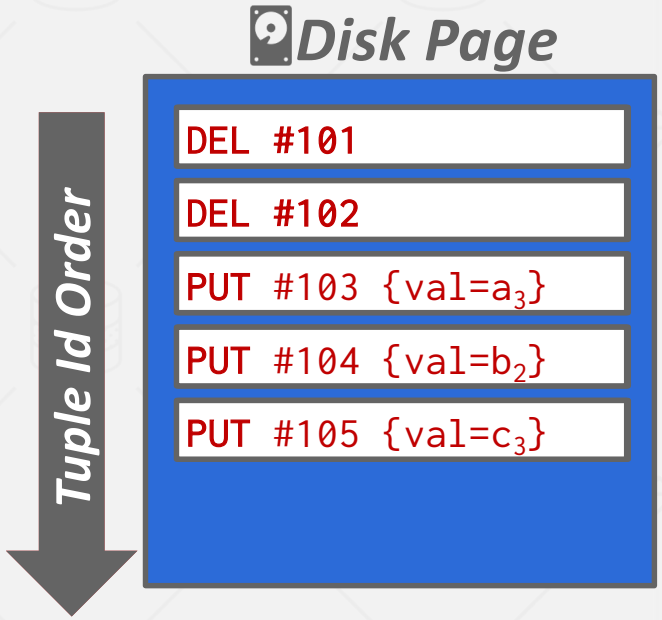
LOG-STRUCTURED COMPACTION

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.

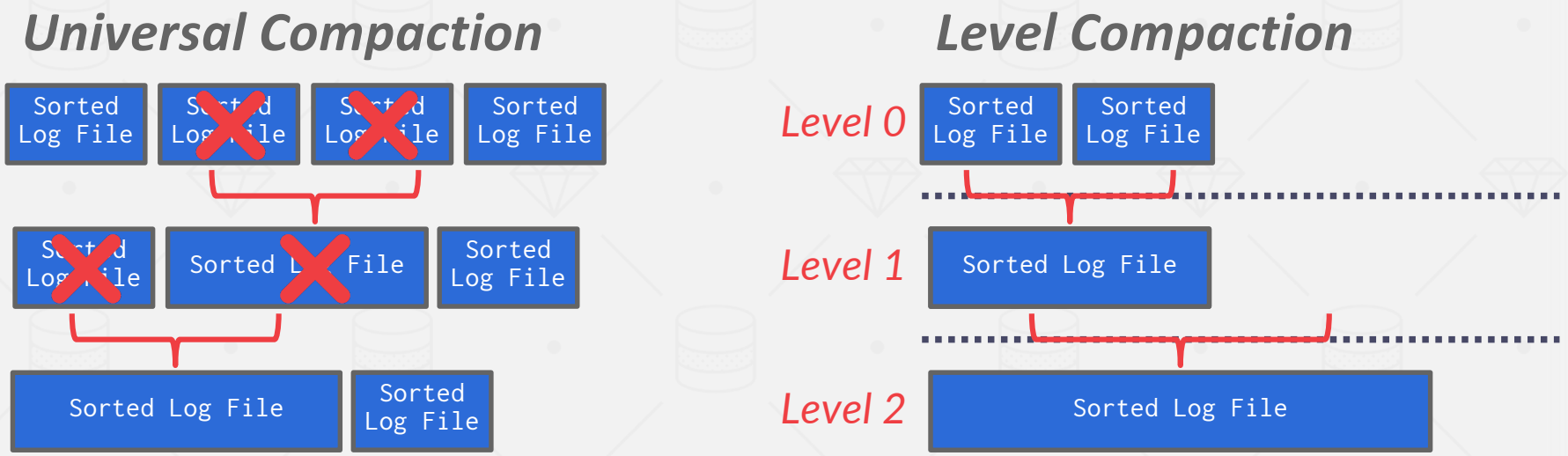
The DBMS can instead sort the page based on id order to improve efficiency of future look-ups.

→ Called Sorted String Tables (SSTables)



LOG-STRUCTURED COMPACTION

Compaction coalesces larger log files into smaller files by removing unnecessary records.



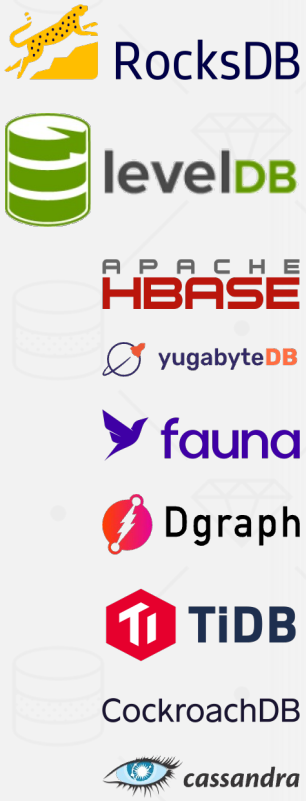
DISCUSSION

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



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 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.
→ Need to worry about collations / sorting.

TIME/DATE/TIMESTAMP
→ 32/64-bit integer of (micro)seconds since Unix epoch

VARIABLE PRECISION NUMBERS

Inexact, variable-precision numeric type that uses the "native" C/C++ types.
→ Examples: `FLOAT`, `REAL`/`DOUBLE`

Store directly as specified by **IEEE-754**.

Typically faster than arbitrary precision numbers but can have rounding errors…

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 you give up arbitrary precision.

(

Weight of

Sca

Positive/Negat

Digi

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

NumericDigit;

;

# of D

# of L

Po

```
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);
```

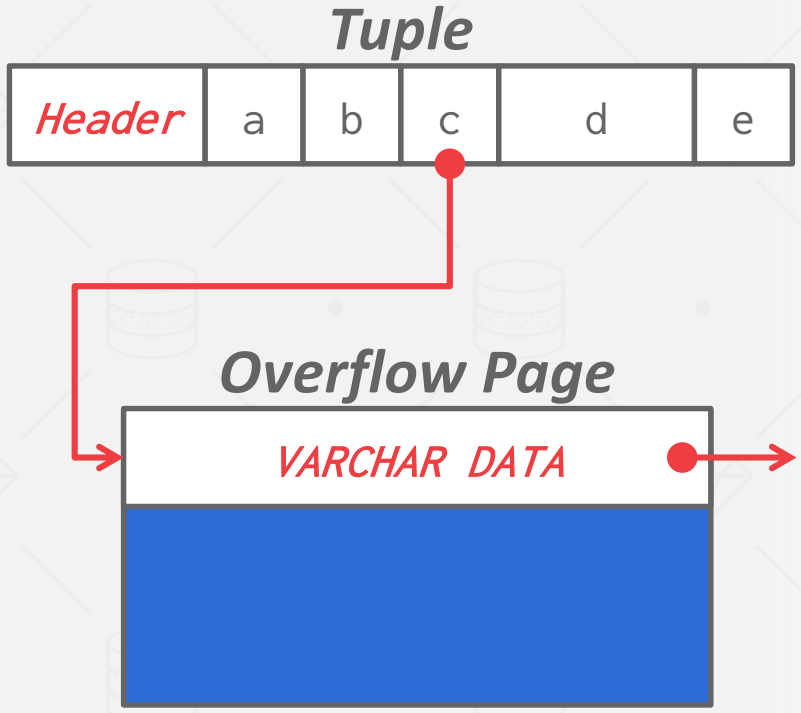
```
_digit_t;

;
```

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

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

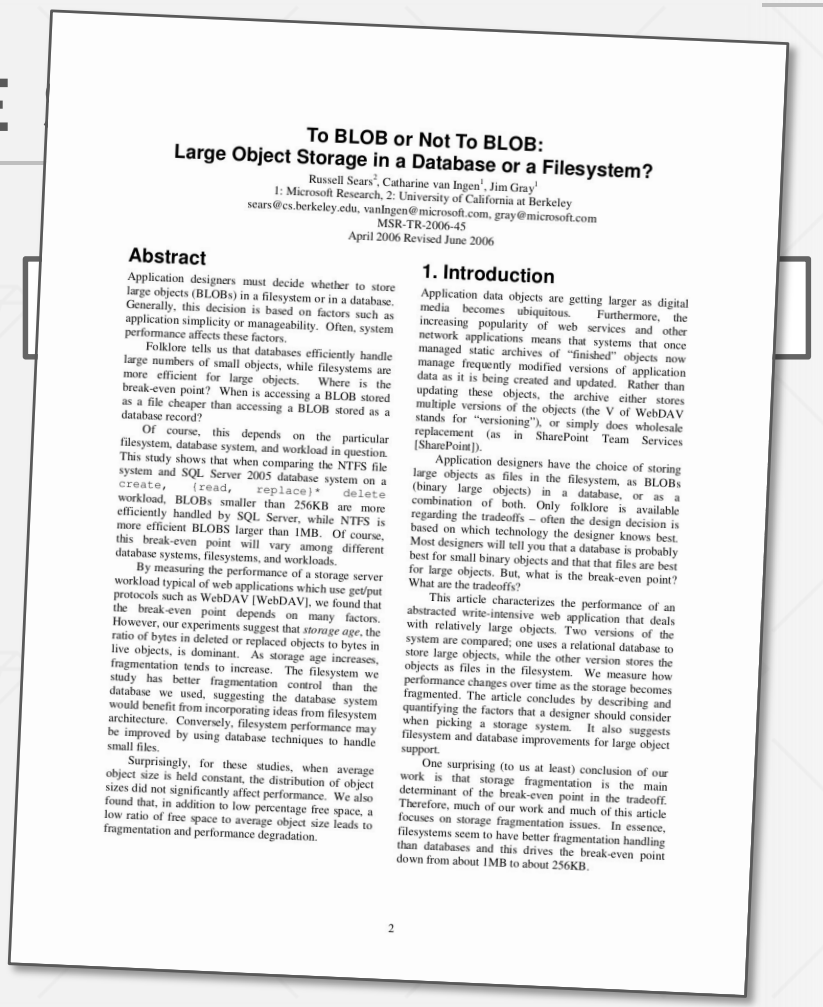
→ Oracle: **BFILE** data type

→ 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
```

CONCLUSION

Log-structured storage is an alternative approach to the page-oriented architecture we discussed last class.

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