Carnegie Mellon University

04 Database Storage – Part II

Intro to Database Systems
15-445/15-645
Fall 2020

AP
Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Project #1 will be released on September 14th

DISK-ORIENTED ARCHITECTURE

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

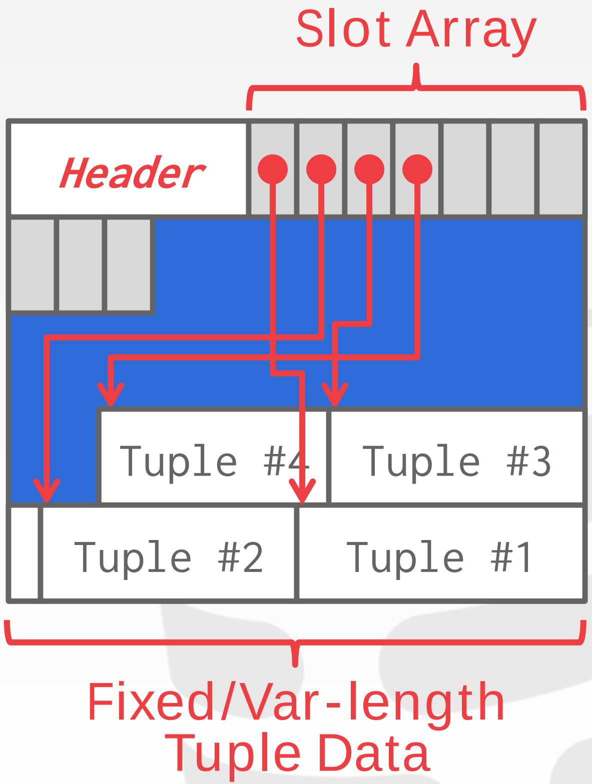
The DBMS's components manage the movement of data between non-volatile and volatile storage.

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.

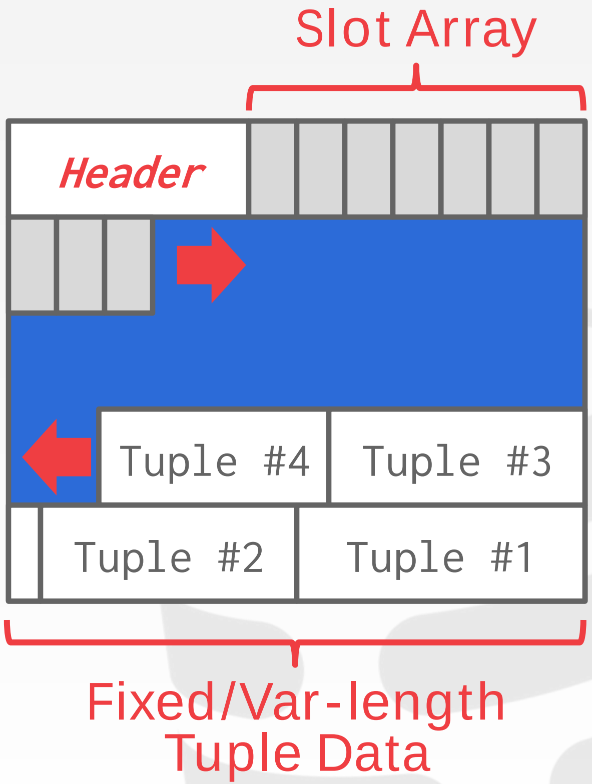


SLOTTED PAGES

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.



LOG-STRUCTURED FILE ORGANIZATION

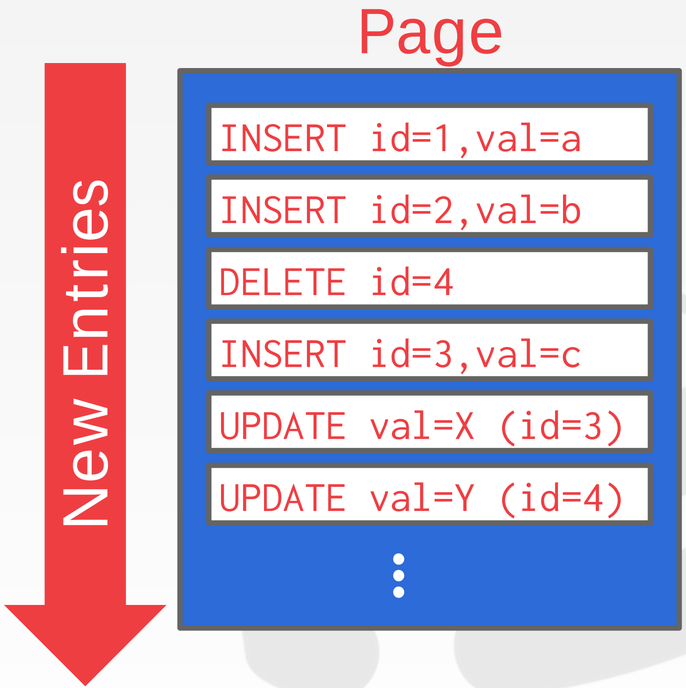
Instead of storing tuples in pages, the DBMS only stores log records.

The system appends log records to the file of how the database was modified:
→ Inserts store the entire tuple.
→ Deletes mark the tuple as deleted.
→ Updates contain the delta of just the attributes that were modified.

LOG-STRUCTURED FILE ORGANIZATION

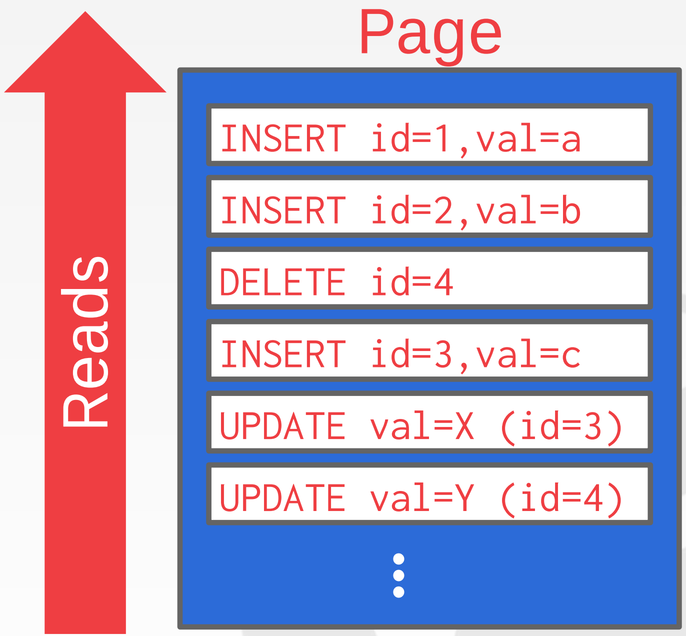
Instead of storing tuples in pages, the DBMS only stores log records.

The system appends log records to the file of how the database was modified:
- → Inserts store the entire tuple.
- → Deletes mark the tuple as deleted.
- → Updates contain the delta of just the attributes that were modified.



LOG-STRUCTURED FILE ORGANIZATION

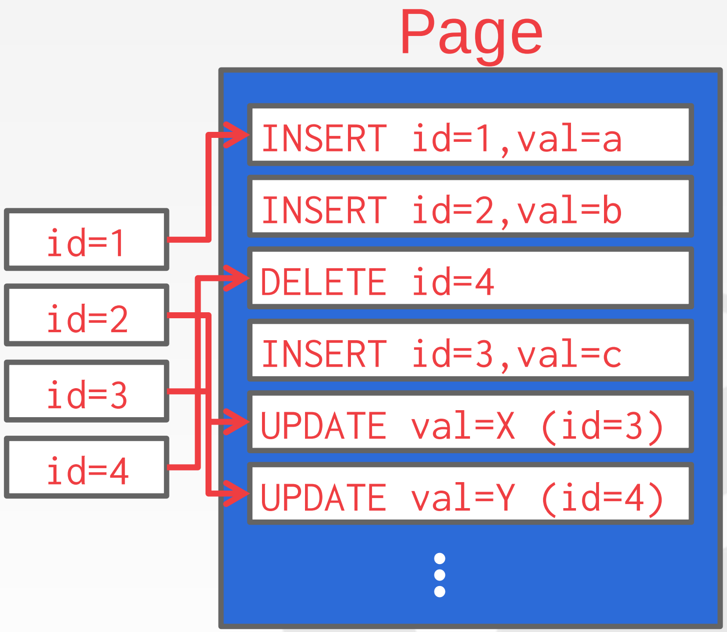
To read a record, the DBMS scans the log backwards and "recreates" the tuple to find what it needs.



LOG-STRUCTURED FILE ORGANIZATION

To read a record, the DBMS scans the log backwards and "recreates" the tuple to find what it needs.

Build indexes to allow it to jump to locations in the log.

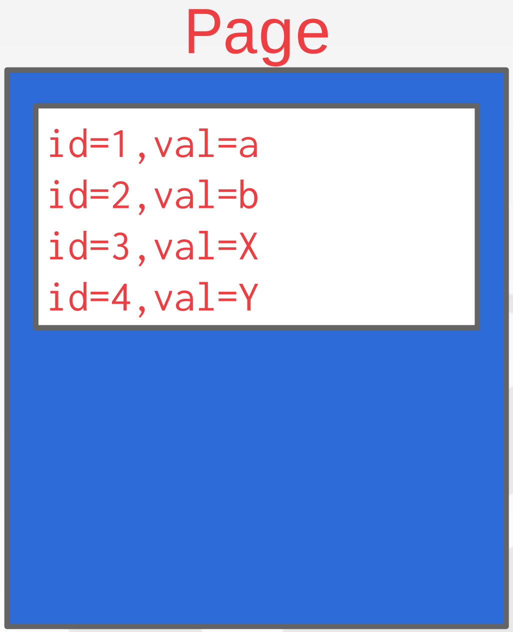


LOG-STRUCTURED FILE ORGANIZATION

To read a record, the DBMS scans the log backwards and "recreates" the tuple to find what it needs.

Build indexes to allow it to jump to locations in the log.

Periodically compact the log.

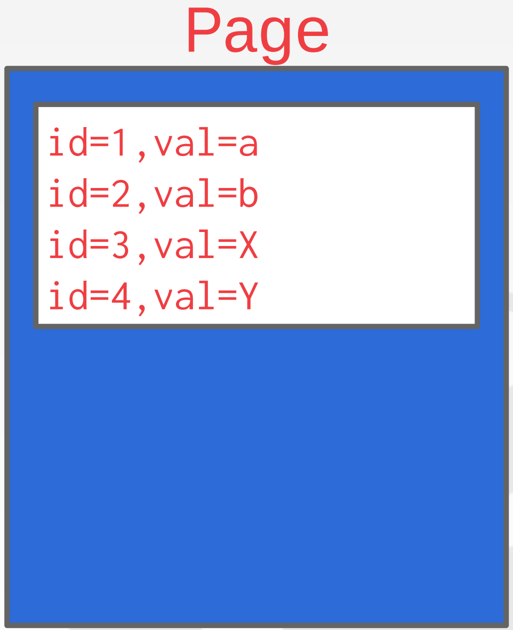


LOG-STRUCTURED FILE ORGANIZATION

To read a record, the DBMS scans the log backwards and "recreates" the tuple to find what it needs.

Build indexes to allow it to jump to locations in the log.

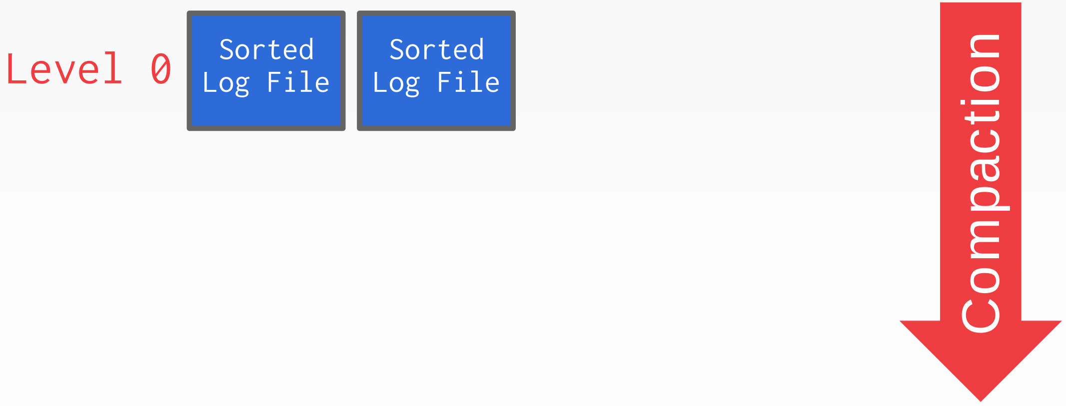
Periodically compact the log.



LOG-STRUCTURED COMPACTION

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

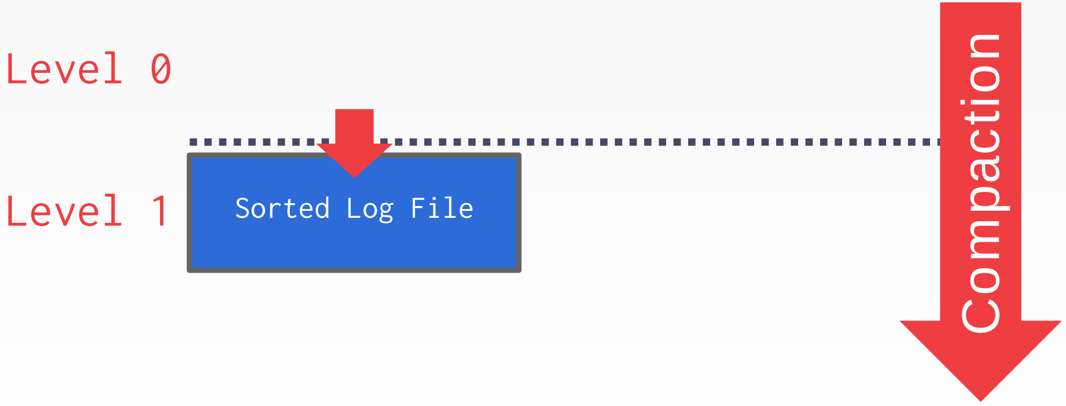
Level Compaction



LOG-STRUCTURED COMPACTION

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

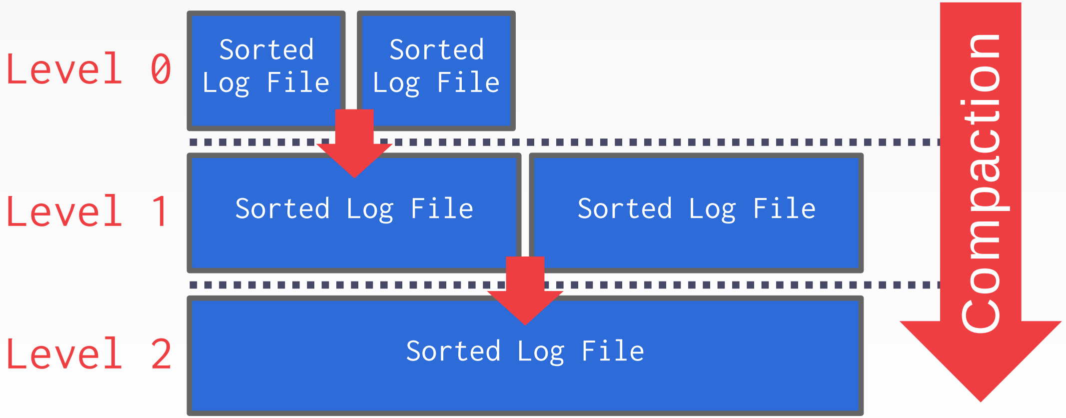
Level Compaction



LOG-STRUCTURED COMPACTION

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

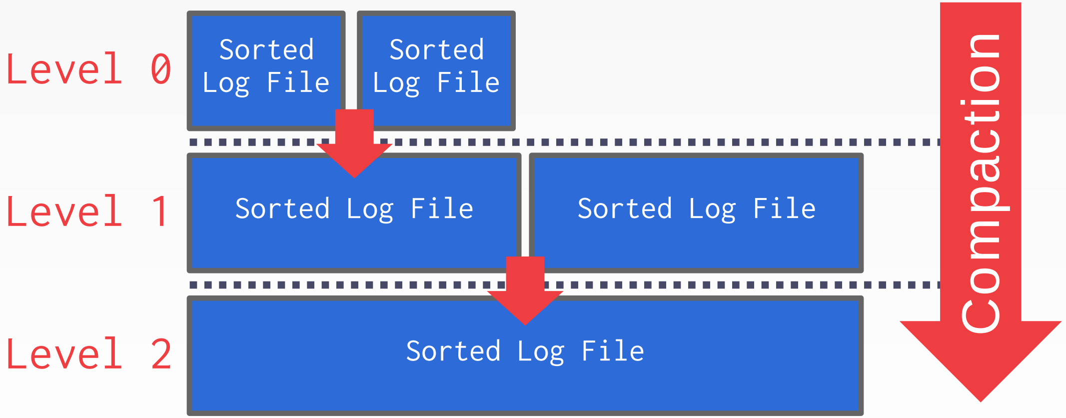
Level Compaction



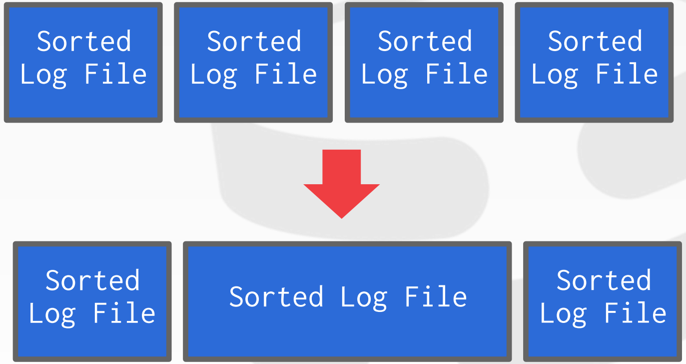
LOG-STRUCTURED COMPACTION

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

Level Compaction



Universal Compaction



TODAY'S AGENDA

Data Representation

System Catalogs

Storage Models

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
→ C/C++ Representation

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

VARCHAR/VARBINARY/TEXT/BLOB
→ Header with length, followed by data bytes.

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>

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

```c
#include <stdio.h>

in
}
```

```c
#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.

Demo: Postgres, MySQL, SQL Server, Oracle

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

#

Weight of

Sca

Positive/Negat

Digi

NumericDigit;

;

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 D

P

```
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);
  to->intg = intg0 * DIG_PER_DEC1;
```

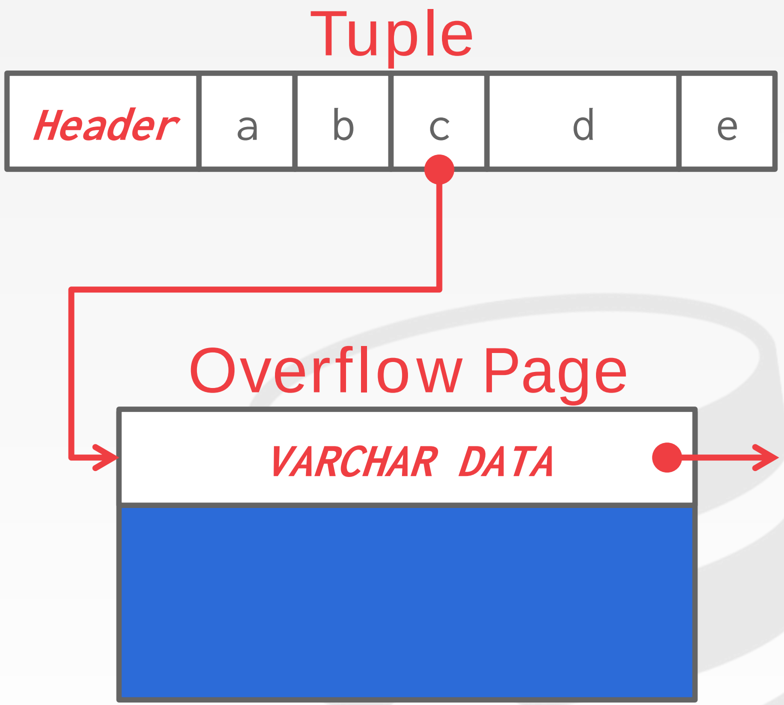
```
_digit_t;

;
```

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.

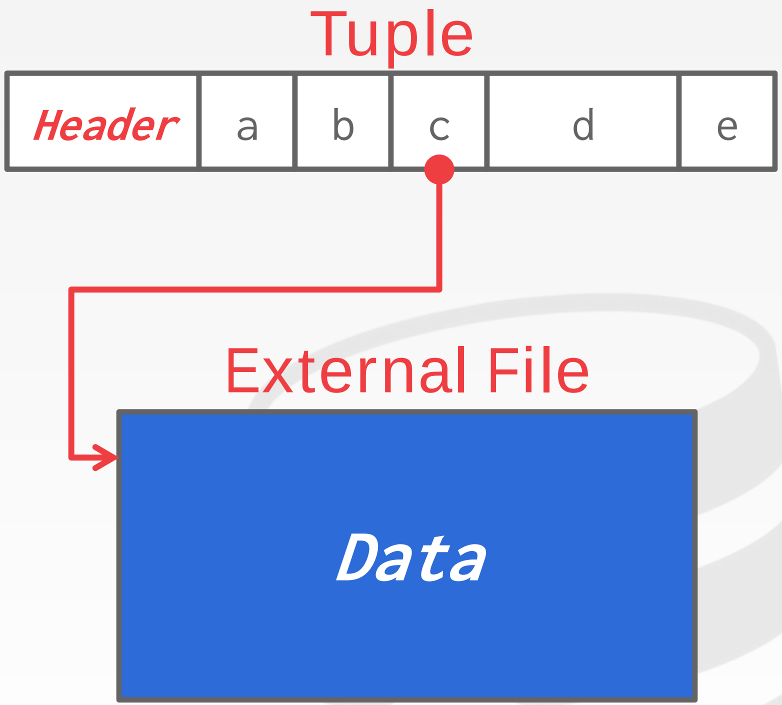
→ Postgres: TOAST (>2KB)
→ MySQL: Overflow (>½ size of page)
→ SQL Server: Overflow (>size of page)



EXTERNAL VALUE STORAGE

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.



EXTERNAL VALUE

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

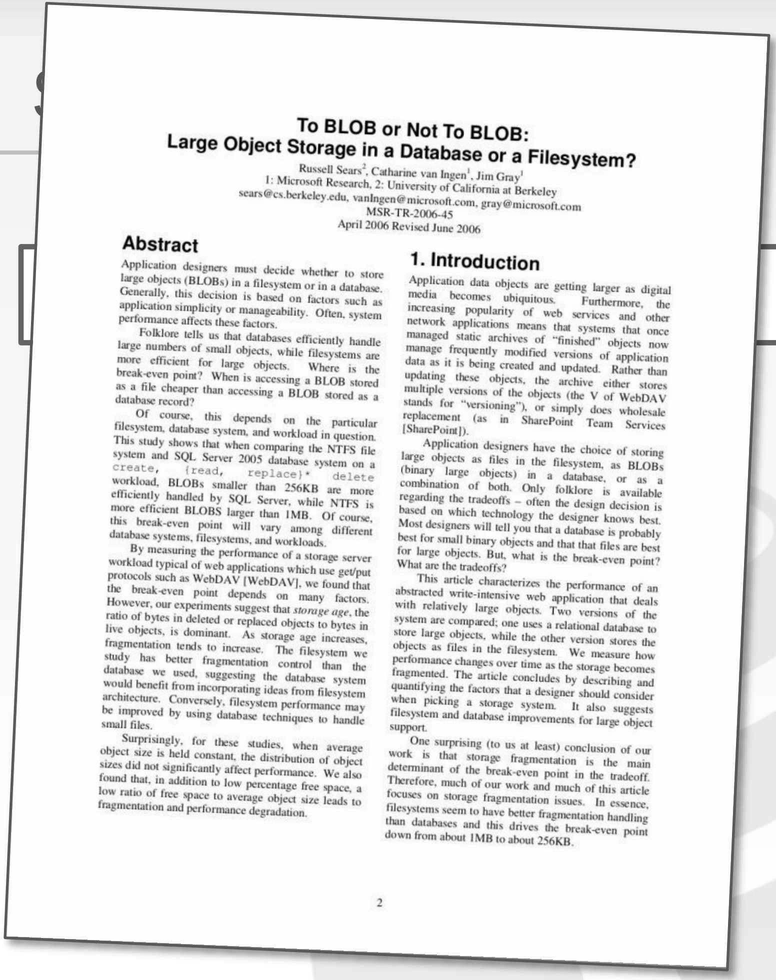
→ Oracle: **BFILE** data type

→ 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 databases' catalogs in another database.

→ 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
```

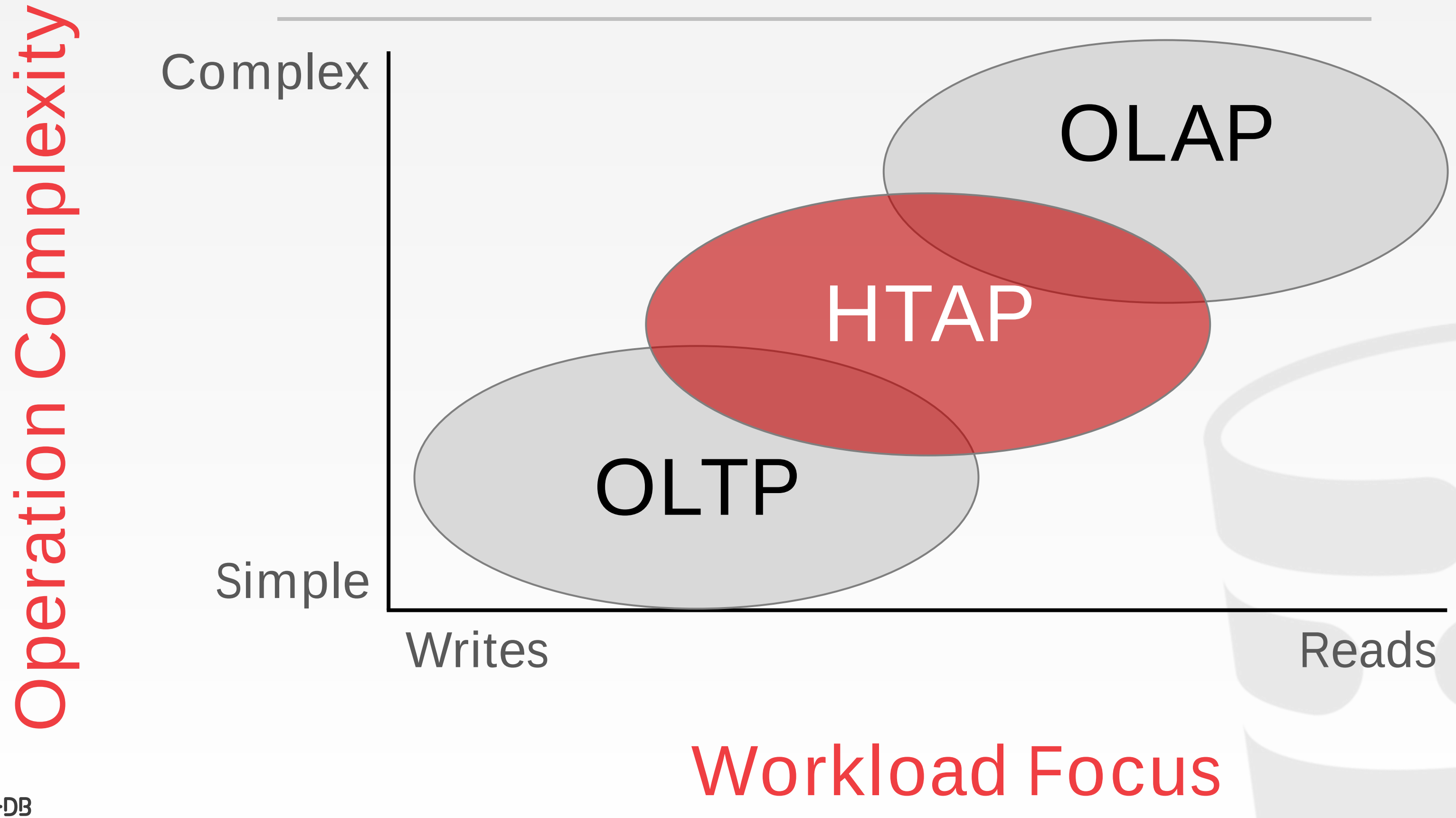
DATABASE WORKLOADS

On-Line Transaction Processing (OLTP)
→ Fast operations that only read/update a small amount of data each time.

On-Line Analytical Processing (OLAP)
→ Complex queries that read a lot of data to compute aggregates.

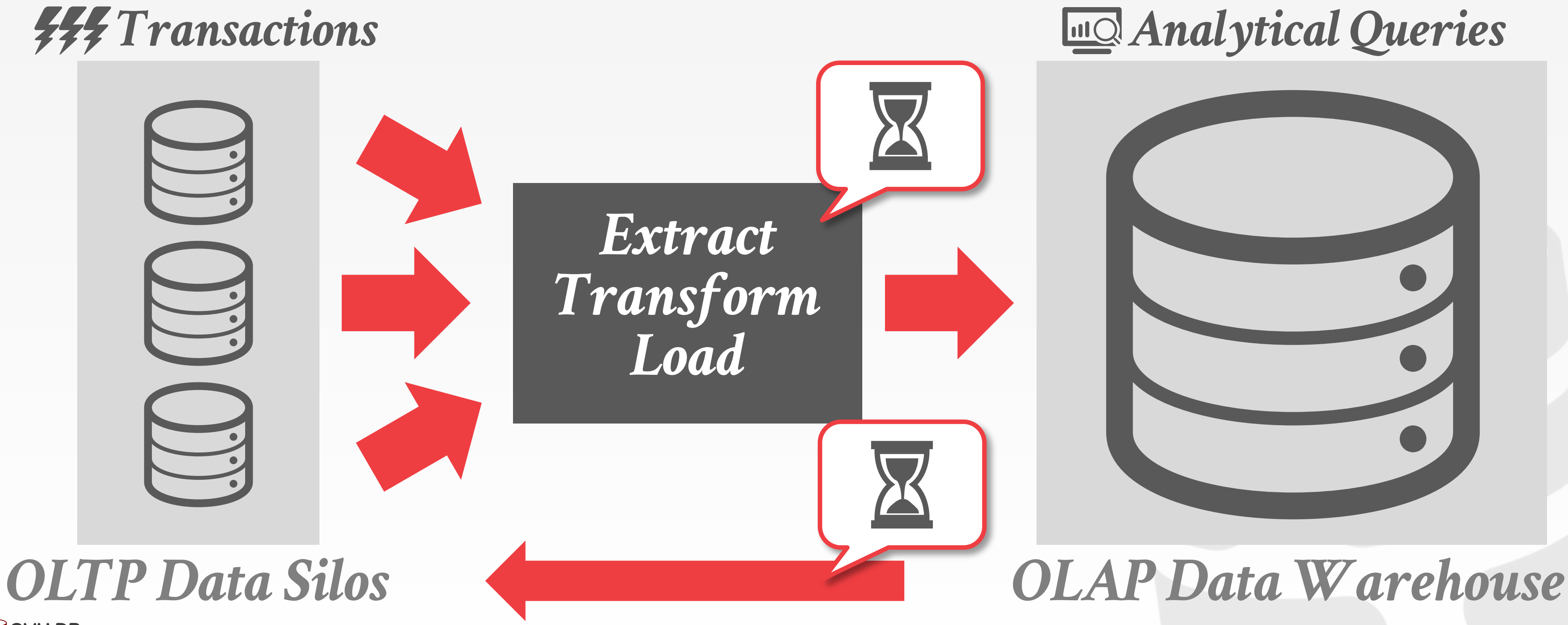
Hybrid Transaction + Analytical Processing
→ OLTP + OLAP together on the same database instance

DATABASE WORKLOADS



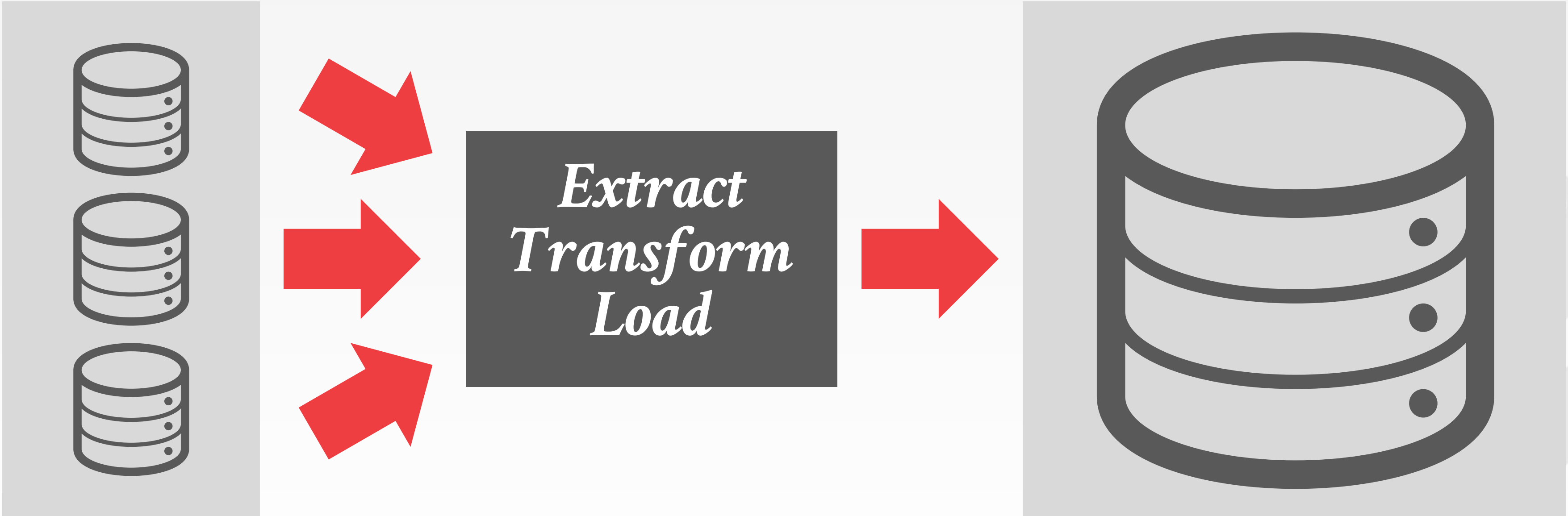
[SOURCE]

BIFURCATED ENVIRONMENT



BIFURCATED ENVIRONMENT

Transactions
Analytical Queries



HTAP Database

OLAP Data Warehouse

OBSERVATION

The relational model does **not** specify that we must store all of a tuple's attributes together in a single page.

This may **not** actually be the best layout for some workloads…

WIKIPEDIA EXAMPLE

```
CREATE TABLE useracct (
  userID INT PRIMARY KEY,
  userName VARCHAR UNIQUE,
  ⋮
);
```

```
CREATE TABLE pages (
  pageID INT PRIMARY KEY,
  title VARCHAR UNIQUE,
  latest INT
  ↳REFERENCES revisions (revID),
);
```

```
CREATE TABLE revisions (
  revID INT PRIMARY KEY,
  userID INT REFERENCES useracct (userID),
  pageID INT REFERENCES pages (pageID),
  content TEXT,
  updated DATETIME
);
```

OLTP

On-line Transaction Processing:

→ Simple queries that read/update a small amount of data that is related to a single entity in the database.

This is usually the kind of application that people build first.

```
SELECT P.*, R.*
  FROM pages AS P
 INNER JOIN revisions AS R
    ON P.latest = R.revID
 WHERE P.pageID = ?
```

```
UPDATE useracct
   SET lastLogin = NOW(),
       hostname = ?
 WHERE userID = ?
```

```
INSERT INTO revisions
VALUES (?,?…,?)
```

OLAP

On-line Analytical Processing:

→ Complex queries that read large portions of the database spanning multiple entities.

You execute these workloads on the data you have collected from your OLTP application(s).

```sql
SELECT COUNT(U.lastLogin),
       EXTRACT(month FROM
            U.lastLogin) AS month
  FROM useracct AS U
 WHERE U.hostname LIKE '%.gov'
 GROUP BY
  EXTRACT(month FROM U.lastLogin)
```

DATA STORAGE MODELS

The DBMS can store tuples in different ways that are better for either OLTP or OLAP workloads.

We have been assuming the **n-ary storage model** (aka "row storage") so far this semester.

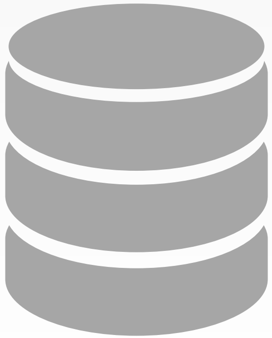
N-ARY STORAGE MODEL (NSM)

The DBMS stores all attributes for a single tuple contiguously in a page.

Ideal for OLTP workloads where queries tend to operate only on an individual entity and insert-heavy workloads.

N-ARY STORAGE MODEL (NSM)

The DBMS stores all attributes for a single tuple contiguously in a page.

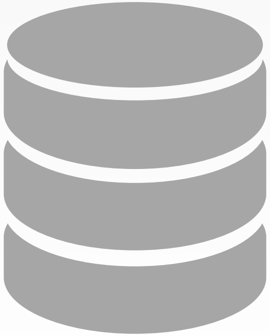


Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin
Header	-	-	-	-	-

←Tuple #1

N-ARY STORAGE MODEL (NSM)

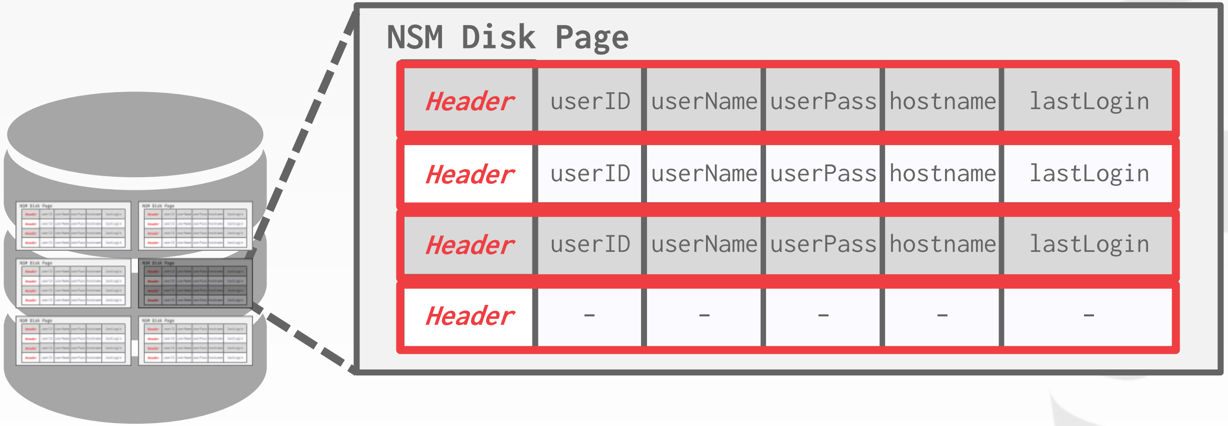
The DBMS stores all attributes for a single tuple contiguously in a page.



Header	userID	userName	userPass	hostname	lastLogin	←Tuple #1
Header	userID	userName	userPass	hostname	lastLogin	←Tuple #2
Header	userID	userName	userPass	hostname	lastLogin	←Tuple #3
Header	-	-	-	-	-	←Tuple #4

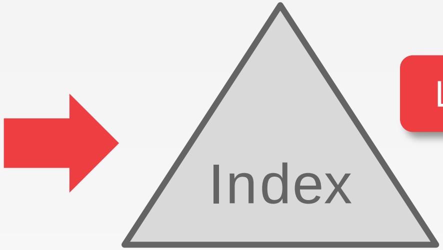
N-ARY STORAGE MODEL (NSM)

The DBMS stores all attributes for a single tuple contiguously in a page.



N-ARY STORAGE MODEL (NSM)

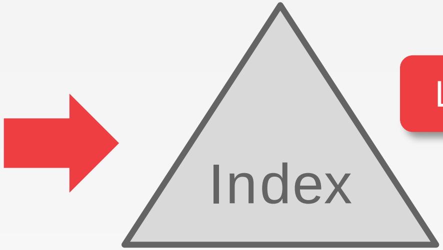
```
SELECT * FROM useracct
 WHERE userName = ?
   AND userPass = ?
```



Lecture 7

N-ARY STORAGE MODEL (NSM)

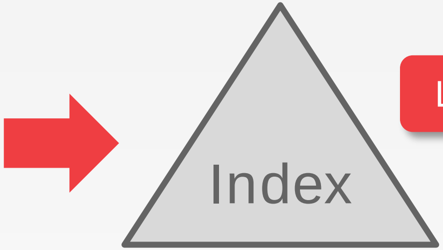
```
SELECT * FROM useracct
 WHERE userName = ?
   AND userPass = ?
```



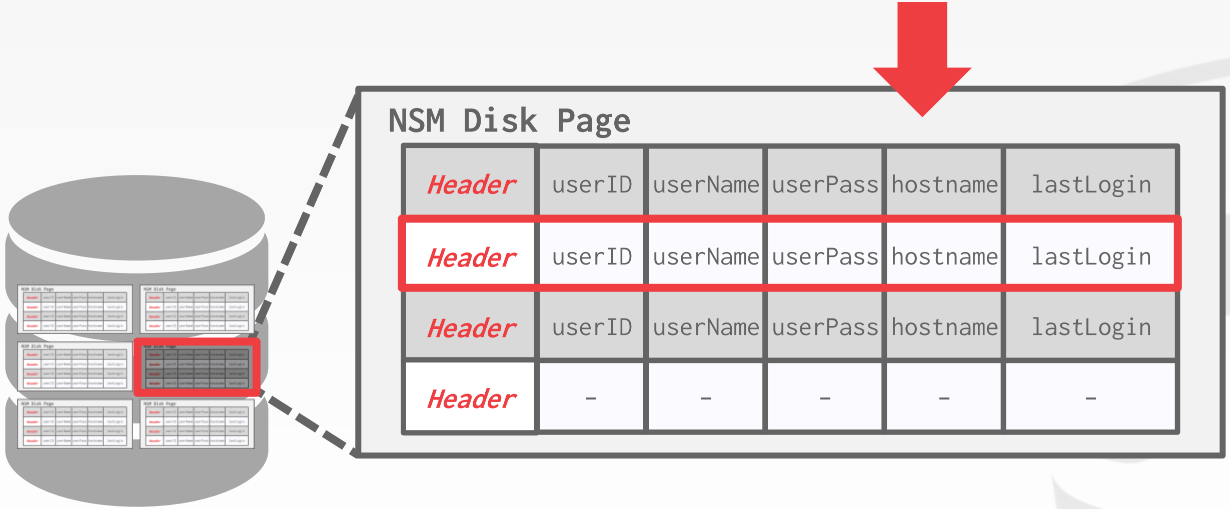
Lecture 7

N-ARY STORAGE MODEL (NSM)

```
SELECT * FROM useracct
 WHERE userName = ?
   AND userPass = ?
```



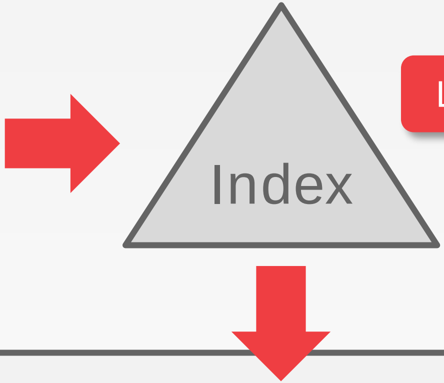
Lecture 7



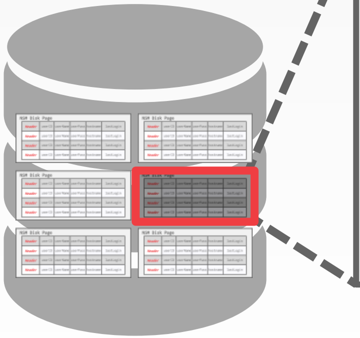
N-ARY STORAGE MODEL (NSM)

```
SELECT * FROM useracct
 WHERE userName = ?
   AND userPass = ?
```

```
INSERT INTO useracct
VALUES (?,?,…?)
```



Lecture 7



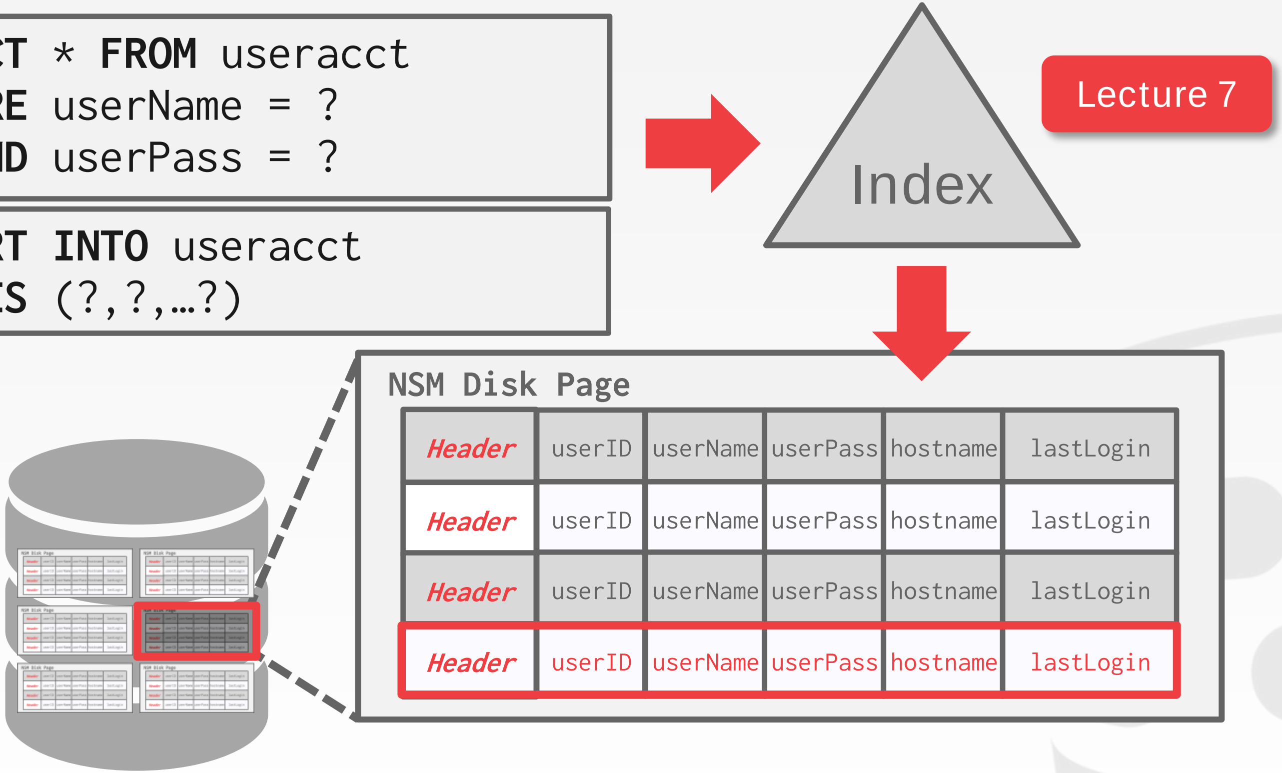
NSM Disk Page

Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin
Header	-	-	-	-	-

N-ARY STORAGE MODEL (NSM)

```
SELECT * FROM useracct
 WHERE userName = ?
   AND userPass = ?
```

```
INSERT INTO useracct
VALUES (?,?,…?)
```



Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin

N-ARY STORAGE MODEL (NSM)

```sql
SELECT COUNT(U.lastLogin),
       EXTRACT(month FROM U.lastLogin) AS month
  FROM useracct AS U
 WHERE U.hostname LIKE '%.gov'
 GROUP BY EXTRACT(month FROM U.lastLogin)
```

N-ARY STORAGE MODEL (NSM)

```
SELECT COUNT(U.lastLogin),
       EXTRACT(month FROM U.lastLogin) AS month
  FROM useracct AS U
 WHERE U.hostname LIKE '%.gov'
 GROUP BY EXTRACT(month FROM U.lastLogin)
```

NSM Disk Page

Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin

N-ARY STORAGE MODEL (NSM)

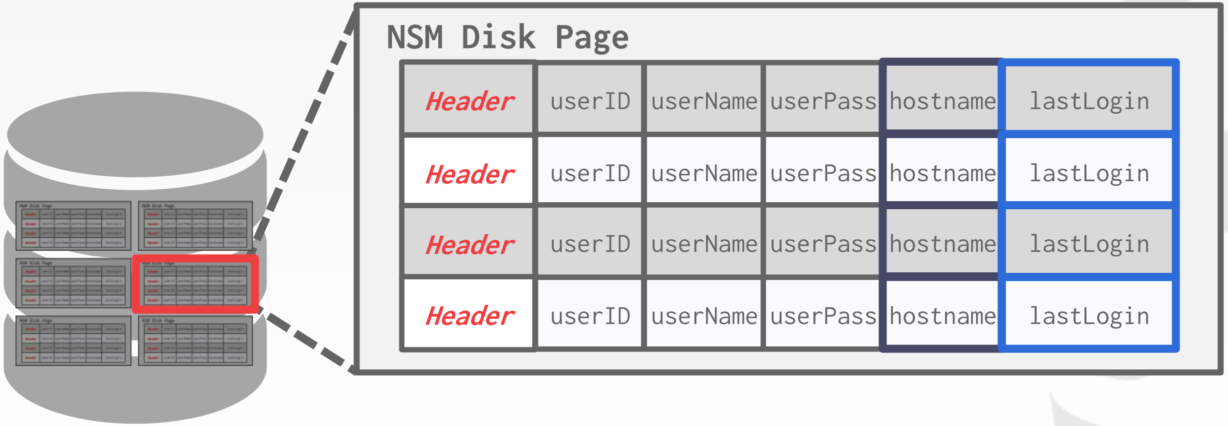
```sql
SELECT COUNT(U.lastLogin),
       EXTRACT(month FROM U.lastLogin) AS month
  FROM useracct AS U
 WHERE U.hostname LIKE '%.gov'
 GROUP BY EXTRACT(month FROM U.lastLogin)
```

NSM Disk Page

Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin

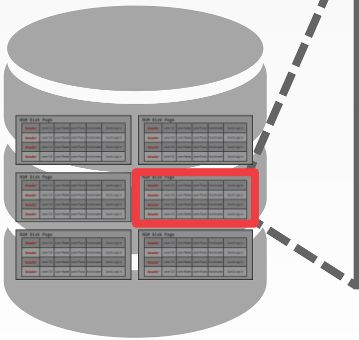
N-ARY STORAGE MODEL (NSM)

```sql
SELECT COUNT(U.lastLogin),
       EXTRACT(month FROM U.lastLogin) AS month
  FROM useracct AS U
 WHERE U.hostname LIKE '%.gov'
 GROUP BY EXTRACT(month FROM U.lastLogin)
```



N-ARY STORAGE MODEL (NSM)

```
SELECT COUNT(U.lastLogin),
       EXTRACT(month FROM U.lastLogin) AS month
  FROM useracct AS U
 WHERE U.hostname LIKE '%.gov'
 GROUP BY EXTRACT(month FROM U.lastLogin)
```



NSM Disk Page

Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin

Useless Data

N-ARY STORAGE MODEL

Advantages

→ Fast inserts, updates, and deletes.
→ Good for queries that need the entire tuple.

Disadvantages

→ Not good for scanning large portions of the table and/or a subset of the attributes.

DECOMPOSITION STORAGE MODEL (DSM)

The DBMS stores the values of a single attribute for all tuples contiguously in a page.

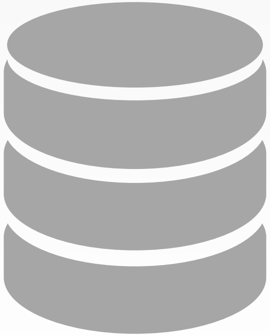
→ Also known as a "column store".

Ideal for OLAP workloads where read-only queries perform large scans over a subset of the table's attributes.

DECOMPOSITION STORAGE MODEL (DSM)

The DBMS stores the values of a single attribute for all tuples contiguously in a page.

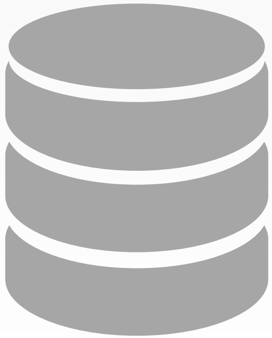
→ Also known as a "column store".



Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin

DECOMPOSITION STORAGE MODEL (DSM)

The DBMS stores the values of a single attribute for all tuples contiguously in a page.
→ Also known as a "column store".

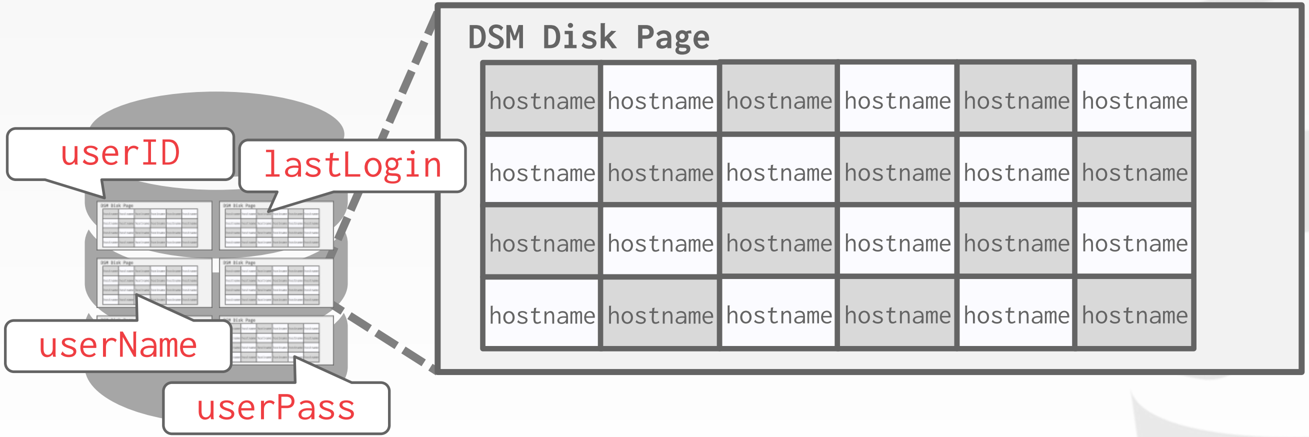


Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin

DECOMPOSITION STORAGE MODEL (DSM)

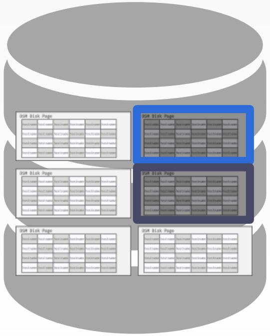
The DBMS stores the values of a single attribute for all tuples contiguously in a page.

→ Also known as a "column store".



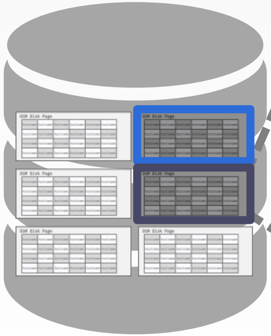
DECOMPOSITION STORAGE MODEL (DSM)

```
SELECT COUNT(U.lastLogin),
       EXTRACT(month FROM U.lastLogin) AS month
  FROM useracct AS U
 WHERE U.hostname LIKE '%.gov'
 GROUP BY EXTRACT(month FROM U.lastLogin)
```



DECOMPOSITION STORAGE MODEL (DSM)

```
SELECT COUNT(U.lastLogin),
       EXTRACT(month FROM U.lastLogin) AS month
  FROM useracct AS U
 WHERE U.hostname LIKE '%.gov'
 GROUP BY EXTRACT(month FROM U.lastLogin)
```



DSM Disk Page

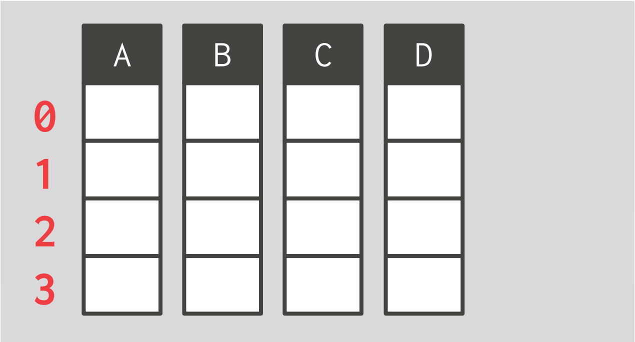
hostname	hostname	hostname	hostname	hostname	hostname
hostname	hostname	hostname	hostname	hostname	hostname
hostname	hostname	hostname	hostname	hostname	hostname
hostname	hostname	hostname	hostname	hostname	hostname

TUPLE IDENTIFICATION

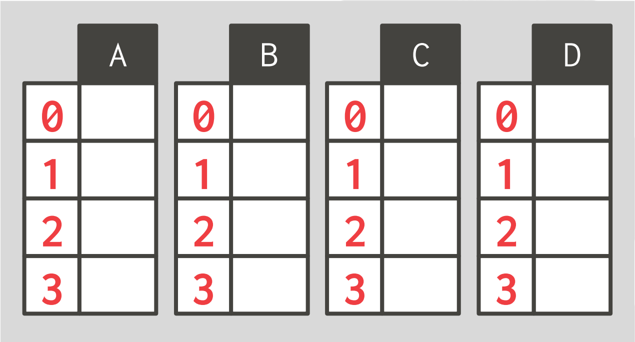
Choice #1: Fixed-length Offsets
→ Each value is the same length for an attribute.

Choice #2: Embedded Tuple Ids
→ Each value is stored with its tuple id in a column.

Offsets



Embedded Ids



DECOMPOSITION STORAGE MODEL (DSM)

Advantages

→ Reduces the amount wasted I/O because the DBMS only reads the data that it needs.

→ Better query processing and data compression (more on this later).

Disadvantages

→ Slow for point queries, inserts, updates, and deletes because of tuple splitting/stitching.

DSM SYSTEM HISTORY

1970s: Cantor DBMS

1980s: DSM Proposal

1990s: SybaseIQ (in-memory only)

2000s: Vertica, VectorWise, MonetDB

2010s: Everyone

CONCLUSION

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

It is important to choose the right storage model for the target workload:
→ OLTP = Row Store
→ OLAP = Column Store

DATABASE STORAGE

Problem #1: How the DBMS represents the database in files on disk.

Problem #2: How the DBMS manages its memory and move data back-and-forth from disk.

← Next