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.
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.
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.
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.

**New Entries**

- INSERT id=1, val=a
- INSERT id=2, val=b
- DELETE id=4
- INSERT id=3, val=c
- UPDATE val=X (id=3)
- UPDATE val=Y (id=4)
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.

**Page**

id=1, val=a
id=2, val=b
id=3, val=X
id=4, val=Y
LOG-STRUCTURED COMPACTION

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

Level Compaction

Sorted Log File  Sorted Log File

Compaction

Level 0
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

<table>
<thead>
<tr>
<th>Level</th>
<th>Sorted Log File</th>
<th>Sorted Log File</th>
<th>Compaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level 2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Compaction coalesces larger log files into smaller files by removing unnecessary records.

**Level Compaction**

- **Level 0**: Sorted Log File, Sorted Log File
- **Level 1**: Sorted Log File, Sorted Log File
- **Level 2**: Sorted Log File

**Universal Compaction**

- Sorted Log File, Sorted Log File, Sorted Log File, Sorted Log File
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/DUOUBLE

Store directly as specified by IEEE-754.

Typically faster than arbitrary precision numbers but can have rounding errors...
#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);
}

Rounding Example  

Output

x+y = 0.300000
0.3 = 0.300000
#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);
}

Rounding Example

Output

```
x+y = 0.30000000000000000000
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**: `typedef unsigned char NumericDigit;`
- **Weight of 1st Digit**: `typedef struct {
    int ndigits;
    int weight;
    int scale;
    int sign;
    NumericDigit *digits;
} numeric;`
- **Scale Factor**
- **Positive/Negative/NaN**
- **Digit Storage**
```c
typedef unsigned char NumericDigit;

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

int numeric_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 */
            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 */
                /* */
                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)) */
                /* */
                return -1;
            }
        }
    } else {
        /* var2 is positive */
        if (add_abs(var2, var1, result) != 0)
            return -1;
        result->sign = NUMERIC_POS;
    }
    return 0;
}
```
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;
};
static int do_add(const decimal_t *from1, const decimal_t *from2,
                   decimal_t *to) {
  int int1 = ROUND_UP(from1->intg), int2 = ROUND_UP(from2->intg),
  frac1 = ROUND_UP(from1->frac), frac2 = ROUND_UP(from2->frac),
  frac0 = std::max(frac1, frac2), intg0 = std::max(int1, intg2), error;
  decl *buf1, *buf2, *buf0, *stop1, *stop2, x, carry;

  sanity(to);

  /* is there a need for extra word because of carry? */
  x = int1 > 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->len = intg1 + intg2 + frac0;
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.
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.
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 \texttt{INFORMATION\_SCHEMA} catalog to get info about the database.

$\rightarrow$ 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.
List all the tables in the current database:

**SQL-92**

```sql
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:

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

- SQL-92

MySQL

```
DESCRIBE student;
```

Postgres

```
\d 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

<table>
<thead>
<tr>
<th>Operation Complexity</th>
<th>Writes</th>
<th>Reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>OLTP</td>
<td></td>
</tr>
<tr>
<td>Complex</td>
<td>OLAP</td>
<td>HTAP</td>
</tr>
</tbody>
</table>

**Workload Focus**
BIFURCATED ENVIRONMENT

Transactions

Extract
Transform
Load

OLTP Data Silos

Analytical Queries

OLAP Data Warehouse
BIFURCATED ENVIRONMENT

Transactions
Analytical Queries

HTAP Database

Extract Transform Load

OLAP Data Warehouse
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...
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 
);
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.
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.
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.

<table>
<thead>
<tr>
<th>Header</th>
<th>userID</th>
<th>userName</th>
<th>userPass</th>
<th>hostname</th>
<th>lastLogin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuple #1</td>
<td>userID</td>
<td>userName</td>
<td>userPass</td>
<td>hostname</td>
<td>lastLogin</td>
</tr>
<tr>
<td>Header</td>
<td>userID</td>
<td>userName</td>
<td>userPass</td>
<td>hostname</td>
<td>lastLogin</td>
</tr>
<tr>
<td>Header</td>
<td>userID</td>
<td>userName</td>
<td>userPass</td>
<td>hostname</td>
<td>lastLogin</td>
</tr>
<tr>
<td>Header</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
**N-ARY STORAGE MODEL (NSM)**

The DBMS stores all attributes for a single tuple contiguously in a page.
**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 = ?
```
N-ARY STORAGE MODEL (NSM)

SELECT * FROM useracct
WHERE userName = ?
AND userPass = ?
**N-ARY STORAGE MODEL (NSM)**

```
SELECT * FROM useracct
WHERE userName = ?
AND userPass = ?
```
**N-ARY STORAGE MODEL (NSM)**

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

INSERT INTO useracct
VALUES (?, ?, ..., ?)
```
**N-ARY STORAGE MODEL (NSM)**

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

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

**NSM Disk Page**

<table>
<thead>
<tr>
<th>Header</th>
<th>userID</th>
<th>userName</th>
<th>userPass</th>
<th>hostname</th>
<th>lastLogin</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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)**

```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)
```
**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)
```
**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)
```
$N$-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".
**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".

<table>
<thead>
<tr>
<th>Header</th>
<th>userID</th>
<th>userName</th>
<th>userPass</th>
<th>hostname</th>
<th>lastLogin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>userID</td>
<td>userName</td>
<td>userPass</td>
<td>hostname</td>
<td>lastLogin</td>
</tr>
<tr>
<td>Header</td>
<td>userID</td>
<td>userName</td>
<td>userPass</td>
<td>hostname</td>
<td>lastLogin</td>
</tr>
<tr>
<td>Header</td>
<td>userID</td>
<td>userName</td>
<td>userPass</td>
<td>hostname</td>
<td>lastLogin</td>
</tr>
</tbody>
</table>
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".
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)
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)
**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**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Embedded Ids**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
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
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.