04 Database Storage – Part II

Intro to Database Systems
15-445/15-645
Fall 2019

Andy Pavlo
Computer Science
Carnegie Mellon University
Homework #1 is due September 11th @ 11:59pm

Project #1 will be released on September 11th
UPCOMING DATABASE EVENTS

SalesForce Talk
→ Friday Sep 13th @ 12:00pm
→ CIC 4th Floor

Impira Talk
→ Monday Sep 16th @ 4:30pm
→ GHC 8102

Vertica Talk
→ Monday Sep 23rd @ 4:30pm
→ GHC 8102
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.
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)
LOG-STRUCTURED FILE ORGANIZATION

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

To illustrate:

- 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)

...
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.
TODAY'S AGENDA

Data Representation
System Catalogs
Storage Models
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

```c
#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>

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 = 0.30000001192092895508
0.3 = 0.2999999999999998890
```
**FIXED PRECISION NUMBERS**

Numeric data types with arbitrary precision and scale. Used when round errors are unacceptable. → Example: **NUMERIC, DECIMAL**

Typically stored in an exact, variable-length binary representation with additional meta-data. → Like a **VARCHAR** but not stored as a string

**Demo: Postgres, SQL Server, Oracle**
POSTGRES: NUMERIC

- **typedef unsigned char** NumericDigit;
- **typedef struct** {
  
  int ndigits;
  int weight;
  int scale;
  int sign;
  NumericDigit *digits;
} numeric;
/* ------------------------
 * add_var() -
 * Full version of add functionality on variable level (handling signs).
 * result might point to one of the operands too without danger.
 * ------------------------
 */

int PGTYPEnumeric_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))
                            * ------------------------
                            */
                    if (sub_abs(var2, var1, result) != 0)
                        return -1;
                    result->sign = -NUMERIC_POS;
                    break;

                default:
                    /* Handle special cases */
                    break;
            }
        }
    }
    else
    {
        /* var2 is positive, var1 is negative Must compare absolute values */
        switch (cmp_abs(var1, var2))
        {
            case 0:
            case 1:
            case -1:
                /* Handle special cases */
                break;
            default:
                /* Handle special cases */
                break;
        }
    }
}

typedef unsigned char NumericDigit;

typedef struct {
    int ndigits;
    int weight;
    int scale;
    int sign;
    NumericDigit *digits;
} numeric;
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 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.
A DBMS stores meta-data about databases in its internal catalogs.
→ Tables, columns, indexes, views
→ Users, permissions
→ Internal statistics

Almost every DBMS stores their a database's catalog in itself.
→ 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.

→ ANSI standard set of read-only views that provide info about all of 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:

```
SELECT * FROM INFORMATION_SCHEMA.TABLES WHERE table_catalog = '<db name>';  \\d;  Postgres
SHOW TABLES;  MySQL
.tables;  SQLite
```
ACCESSING TABLE SCHEMA

List all the tables in the student table:

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

- **Postgres**
  
  `\d student;`

- **MySQL**
  
  `DESCRIBE student;`

- **SQLite**
  
  `./schema student;`
OBSERVATION

The relational model does **not** specify that we have to 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).

```
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)
```
WORKLOAD CHARACTERIZATION

Complex

Simple

Writes

Reads

Workload Focus

OLAP

HTAP

OLTP

Operation Complexity
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.
**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)**

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

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

**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)
```
**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)**

```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

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