CS525: Advanced Database Organization

Notes 3: Database Storage
Part II

Yousef M. Elmehdwi

Department of Computer Science

Illinois Institute of Technology

yelmehdwi@iit.edu

January 24th 2024

Slides: adapted from courses taught by Andy Pavlo, Carnegie Mellon University, Hector Garcia-Molina, Stanford, & Shun Yan Cheung, Emory University

Reading

- Database Systems: The Complete Book, 2nd Edition,
  - *Chapter 2: Data Storage*
- Database System Concepts (6th/7th Edition)
  - *Chapter 10 (6th)/ Chapter 13 (7th)*

Database Storage

- `Problem#1`: How the DBMS represents the database in files on disk
    - i.e., how to lay out data on disk.
- `Problem#2`: How the DBMS manages its memory and move data back-and-forth from disk.

Today's Agenda

- Data Representation
  - *How the system stores the actual binary data for individual attributes (columns) within the database.*
- System Catalogs
  - *Internal metadata is maintained by the database to understand both the data that is actually stored and how to interpret the bytes within the tuples*
- Storage Models
  - *How data is organized and stored within the database system.*
- Modification of Tuples

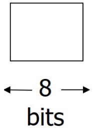
Tuple Storage

- A tuple is essentially a sequence of bytes (byte arrays).
- It is up to the DBMS to know how to interpret those bytes to derive the values for attributes.
- The DBMS's catalogs contain the schema information about tables that the system uses to figure out the tuple's layout.

What are the data items we want to store?

- a salary
- a name
- a date
- a picture

⇒ What we have available: Bytes



Data Representation

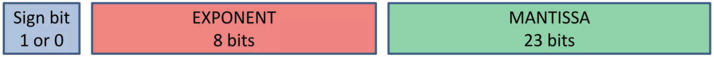
- *How a DBMS stores the bytes for a value?*
    - *How the DBMS stores the binary data for different types of values or attributes?*
- There are five high level data types that can be stored in tuples:
    - integers,
    - variable precision numbers,
    - fixed point precision numbers,
    - variable length values, and
    - dates/times.

Floating-point numbers

- Floating-point numbers are a way for computers to represent real numbers (numbers with decimals) that aren't necessarily whole integers.
- They use a combination of three components to represent a real number:
  - Sign bit: Indicates whether the number is positive or negative (0 for positive, 1 for negative).
  - Exponent: An integer that controls the magnitude of the number. Think of it as a scaling factor, similar to scientific notation.
  - Mantissa (Significand): The significant digits of the number, representing the actual value without the decimal point.

IEEE-754 Standard[1]

- This standard establishes a common way for computers to represent and work with floating-point numbers, ensuring consistency across different systems and languages.
- Defines the binary representation of floating-point numbers (like float and double) and their arithmetic operations in computing.
- Ensures consistency in how floating-point numbers are stored and manipulated across different computer architectures and programming languages.
- It specifies the format for representing real numbers, including the sign bit, exponent, and mantissa (fractional part).
- Example: 32 bit in Standard IEEE 754:



To be represented in this format, a number should be in the following normalized form.
(+ or -) 1.(mantissa) x 2^(exponent)

[1] *https://en.wikipedia.org/wiki/IEEE_754*

Data Representation

How a DBMS stores the bytes for a value

- `INTEGER/BIGINT/SMALLINT/TINYINT`
  - `C/C++` Representation
    - All integers are stored in their “native” `C/C++` types[1] within the database.
- `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

[1] *refer to the fundamental data types that are supported directly by the C/C++ programming languages without any additional libraries or custom data types. These native data types are typically used for storing and manipulating data efficiently in these languages.*

Data Representation: Integers

- `C/C++` Representation
  - Most DBMSs store integers using their "native" `C/C++` types as specified by the IEEE-754 standard.
- These values are fixed length.
- Examples: `INTEGER/BIGINT/SMALLINT/TINYINT`

- *Most DBMSs use native C/C++ integer types (as specified by the IEEE-754 standard) for fixed-length storage.*

Variable Precision Numbers

- These are inexact[1], variable-precision numeric types that uses the "native" `C/C++` types.
- Store directly as specified by IEEE-754 standard.
- These values are also fixed length.
- Typically faster than arbitrary precision numbers because the CPU can execute instructions on them directly.
  - Example: `FLOAT, REAL/DOUBLE`
- But can have rounding errors[2] when performing computations due to the fact that some numbers cannot be represented precisely in binary floating-point format.
- As a result, calculations may yield slightly inaccurate results.
  - To avoid this issue, we use Fixed-Point Precision Numbers.
- *Storage: Use native C/C++ floating-point types (IEEE-754) for fixed-length storage.*

[1] *Inexact means that some values cannot be converted exactly to the internal format and are stored as approximations, so that storing and retrieving a value might show slight discrepancies.*

[2] *rounding error, is the difference between the result produced by a given algorithm using exact arithmetic and the result produced by the same algorithm using finite-precision, rounded arithmetic.*

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

```
#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 = 0.29999999999999998890
```

Variable Precision Numbers

- Rounding Example

```java
public class RoundingError {
 public static void main(String[] args) {
   float x = 1.0f;
   for (int i = 0; i < 10; i++) {
         x -= 0.1f;
   }
   System.out.printf("Result w precision: %.1f\n", x);
   System.out.printf("Result w precision 10: %.10f\n",x);
 }
}
```

```
Output
Result w precision 1 :  -0.0
Result w precision 10:  -0.0000000745
```

Data Representation

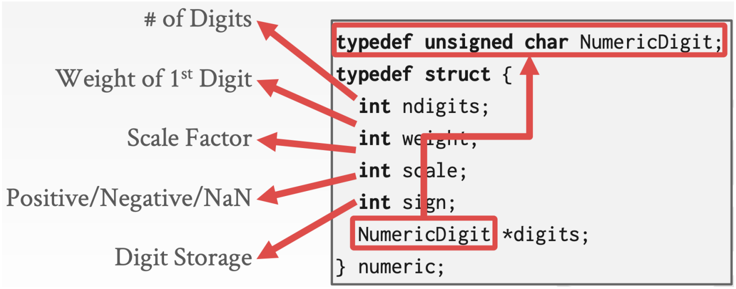
- *Precise data types (fixed-point numbers) offer accuracy but may impact performance. Inexact types (variable precision numbers) are faster but introduce rounding errors.*

Data Representation: Fixed Point Precision Numbers

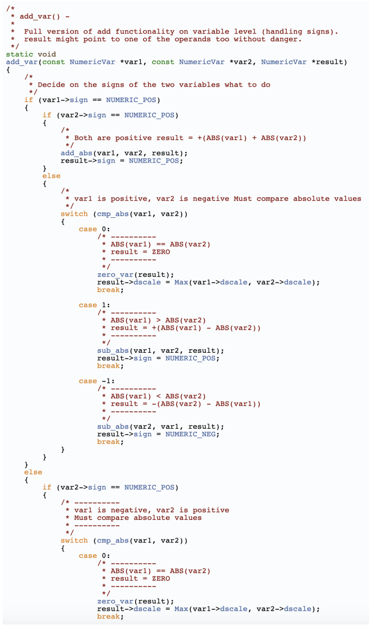
- Numeric data types with arbitrary precision and scale.
- Used when round errors are unacceptable.
    - Example: `NUMERIC, DECIMAL`
- Typically stored in an exact, variable-length binary representation with additional meta-data that specifies the length of the data and the position of the decimal point.
    - Like a `VARCHAR` but not stored as a string
- But the DBMS pays a performance penalty to get this accuracy.
    - *Calculations involving fixed-point precision numbers may be slower compared to operations with native numeric types like FLOAT or DOUBLE, which use hardware-based floating-point arithmetic.*

- *Storage: Exact, variable-length binary representation (like a string) with metadata for length and decimal point placement.*

PostgreSQL: NUMERIC



```
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.
 */
static void
add_var(const NumericVar *var1, const NumericVar *var2, NumericVar *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))
             */
            add_abs(var1, var2, result);
            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->dscale = Max(var1->dscale, var2->dscale);
                    break;

                case 1:
                    /* ----------
                     * ABS(var1) > ABS(var2)
                     * result = +(ABS(var1) - ABS(var2))
                     * ----------
                     */
                    sub_abs(var1, var2, result);
                    result->sign = NUMERIC_POS;
                    break;

                case -1:
                    /* ----------
                     * ABS(var1) < ABS(var2)
                     * result = -(ABS(var2) - ABS(var1))
                     * ----------
                     */
                    sub_abs(var2, var1, result);
                    result->sign = NUMERIC_NEG;
                    break;
            }
        }
    }
    else
    {
        if (var2->sign == NUMERIC_POS)
        {
            /* ----------
             * var1 is negative, var2 is positive
             * Must compare absolute values
             * ----------
             */
            switch (cmp_abs(var1, var2))
            {
                case 0:
                    /* ----------
                     * ABS(var1) == ABS(var2)
                     * result = ZERO
                     * ----------
                     */
                    zero_var(result);
                    result->dscale = Max(var1->dscale, var2->dscale);
                    break;
```

PostgreSQL Source Code, numeric.c

Data Representation: Variable Length Data

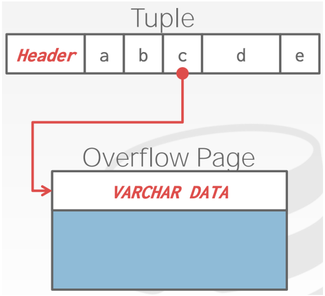
- These represent data types of arbitrary length.
    - An array of bytes of arbitrary length.
- Has a header that keeps track of the length of the string to make it easy to jump to the next value. It may also contain a checksum for the data.
- Example: `VARCHAR, VARBINARY, TEXT, BLOB`.
- *Storage: Header with length information, followed by the actual data. May include checksum for integrity.*

Large Values

- Most DBMSs don't allow a tuple to exceed the size of a single page.
- Handling tuples that exceed the size of a single page in a DBMS is a common challenge.
- DBMSs typically have strategies to deal with this situation to ensure data integrity and efficient storage.
- Two common approaches to handle such cases are:
  - overflow page
  - external storage

Large Values: Overflow Page

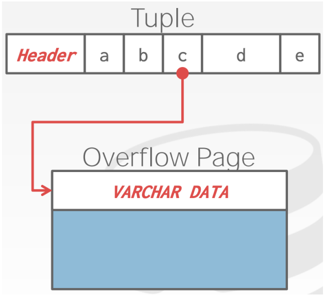
- To store values that are larger than a page, the DBMS uses separate overflow storage pages and have the tuple contain a reference to that page.
    - The main part of the tuple, which can fit within a single page, is stored in the primary page, while the overflowed part is stored in one or more additional overflow pages.
    - Overflow pages are linked to the primary page, forming a chain of pages that together represent the complete tuple.
    - When querying the data, the DBMS follows these chains of pages to reconstruct the complete tuple.



- If necessary, overflow pages can contain pointers to additional overflow pages, creating a longer chain until all data is accommodated.

Large Values: Overflow Page

- Different DBMSs have their own terminology and thresholds for employing overflow pages:
  - `Postgres: TOAST (The Oversized-Attribute Storage Technique) (>2KB)`
  - `MySQL: Overflow (>` $\frac{1}{2}$ `size of page)`
  - `SQL Server: Overflow (> size of page)`



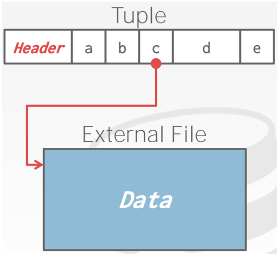
TOAST (The Oversized-Attribute Storage Technique) https://www.postgresql.org/docs/current/storage-toast.html

External Value Storage

- Some systems allow to store large data values, such as files or binary objects, in an external file rather than directly within the database, and then the tuple will contain a pointer or locator referencing that external file.
- Example:
    - if the database is storing photo information, the DBMS can store the photos in the external files rather than having them take up large amounts of space in the DBMS.

External Value Storage: Common Implementations

- Treated as a `BLOB` type
  - `Oracle: BFILE data type`
    - Stores a locator pointing to a large binary file external to the database.
  - `Microsoft: FILESTREAM data type`
- The DBMS **cannot** manipulate the contents of an external file.



- `BLOB (Binary Large Object)` data types are designed to store large, unstructured data within the database itself.
- Reading: A paper explains the trade-offs between these two options:
  - To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem

- *The optimal choice between external value storage and BLOBs depends on specific requirements, data characteristics, and performance needs. Carefully evaluate factors like data size, access patterns, recovery needs, and security constraints when making a decision.*

Data Representation: Dates and Times

- Varies widely across different database systems?
- However, a common approach is to represent dates and times as the number of (micro/milli) seconds since the Unix epoch[1].
  - Formatting and time zones are handled at the application level.
  - Storage details vary between DBMSs.
- Example: `TIME, DATE, TIMESTAMP`.

- *Storage: Usually as a fixed-length integer representing the number of seconds (or milliseconds) since a specific epoch (e.g., January 1, 1970).*

[1] *The Unix epoch is a reference point in time, representing January 1, 1970, at 00:00:00 UTC (Coordinated Universal Time). It is widely used as a starting point for measuring time intervals.*

System Catalog

- In order for the DBMS to be able to decipher the contents of tuples, it maintains an internal catalog to tell it meta-data about the databases
    - A DBMS stores meta-data about databases in its internal catalogs.
- The meta-data will contain what tables and columns the databases have along with their types and the orderings of the values.
    - Tables, columns, indexes, views
    - Users, permissions
    - Internal statistics
- Almost every DBMS stores their a database's catalog in itself in the format that they use for their tables
- Bootstrapping[1]
    - The DBMS employs specialized code to access the catalog during startup, wrapping low-level access methods to ensure it can read and interpret the metadata before fully accessing user-defined tables and data.

[1] *Bootstrapping is the process of initializing a DBMS's catalog during system setup or database creation. During this phase, the DBMS uses low-level access methods or internal mechanisms to create the catalog tables and populate them with initial data*

System Catalog

- `INFORMATION_SCHEMA`: Provides a standardized, portable way to query and retrieve metadata about various database objects, promoting consistency across different DBMSs.
- You can query the DBMS's internal `INFORMATION_SCHEMA` catalog to get info about the database.
  - ANSI standard[1] 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.

[1] *ANSI Standard: Adheres to the ANSI SQL standard, ensuring compatibility across DBMSs that support it.*

Accessing Table Schema

- List all of the tables in the current database:

```
-- SQL-92
SELECT *
FROM INFORMATION_SCHEMA.TABLES
WHERE table_catalog = '<db name>';
```

```
\d;           -- Postgres
SHOW TABLES;  -- MySQL
.tables;      -- SQLite
```

Accessing Table Schema

- List all of the columns in the *student* table:

```
-- SQL-92
SELECT *
FROM INFORMATION_SCHEMA.TABLES
WHERE table_name = 'student'
```

```
\d student;          -- Postgres
DESCRIBE student;    -- MySQL
.schema student;     -- SQLite
```

Today's Agenda

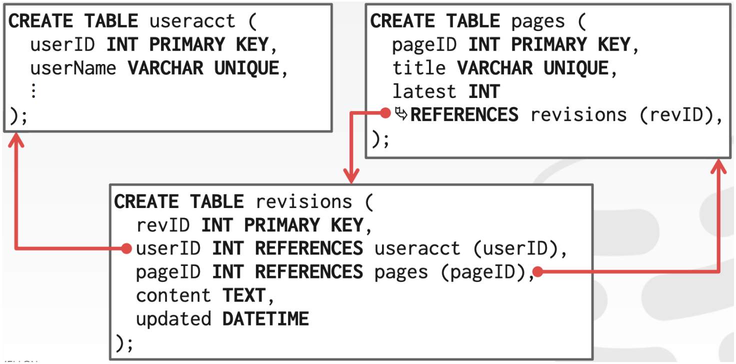
- Data Representation
- System Catalogs
- Storage Models
    - Ways to store tuples in pages

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
- There are many different workloads for database systems.
- By workload[1], we are referring to the general nature of requests a system will have to handle.
- Different workloads have different requirements for data storage and access patterns.

[1] *refers to the types of queries, transactions, and operations the system is expected to handle.*

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
);
```

- User Account: Store information about users who create or make changes to articles.
- Pages: Contain the latest revision for a particular page.
- Revision Table: Store all new updates for every page (single article). It should include a foreign key reference to the user who created or made changes to the article.

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 (?,?...,?)
```

- *OLTP systems are designed to handle large volumes of short, simple transactions that typically involve reading and updating small amounts of data. These transactions are often associated with individual events or operations within a business or application.*

OLTP: On-line Transaction Processing

- Fast, short running operations
- Simple queries that operate on single entity at a time
- Typically handle more writes than reads
- Repetitive operations
- Usually the kind of application that people build first
- Example
  - User invocations of Amazon (Amazon storefront).
    - Users can add things to their cart,
    - they can make purchases,
    - but the actions only affect their accounts.

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)
```

- *OLAP systems offer powerful tools for analyzing large datasets, extracting valuable insights, and informing business decisions. They complement OLTP applications by providing deeper understanding and strategic direction based on historical data trends.*

OLAP: On-line Analytical Processing

- Long running, more complex queries
- Reads large portions of the database
- Analyzing and deriving new data from existing data collected on the OLTP side
- Example
  - Amazon computing the five most bought items over a one month period for these geographical locations.

HTAP: Hybrid Transaction + Analytical Processing

- A new type of workload which has become popular recently is HTAP, which is like a combination which tries to do `OLTP` and `OLAP` together on the same database.

- Watch HTAP Databases: What is New and What is Next - SIGMOD22-HTAP-Tutorial- June 2022

Data Storage Model

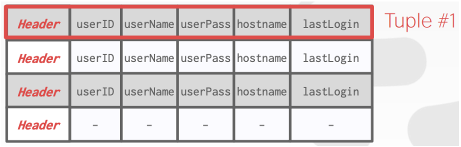
- There are different ways to store tuples in pages.
- The DBMS can store tuples in different ways that are better for either OLTP or OLAP workloads
- We have been assuming the n`-ary storage model` (aka “row storage”) so far this semester.

N-ARY Storage Model (NSM)

- The DBMS stores all attributes for a single tuple contiguously in a single page.
- Ideal for OLTP workloads where requests are insert-heavy and transactions tend to operate only an individual entity
    - it takes only one fetch to be able to get all of the attributes for a single tuple.

N-ARY Storage Model (NSM)

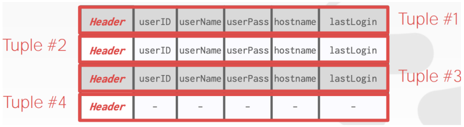
- The DBMS stores all attributes for a single tuple contiguously in a single page.



Header	userID	userName	userPass	hostname	lastLogin	Tuple #1
Header	userID	userName	userPass	hostname	lastLogin	
Header	userID	userName	userPass	hostname	lastLogin	
Header	-	-	-	-	-	

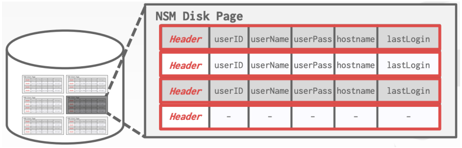
N-ARY Storage Model (NSM)

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



N-ARY Storage Model (NSM)

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

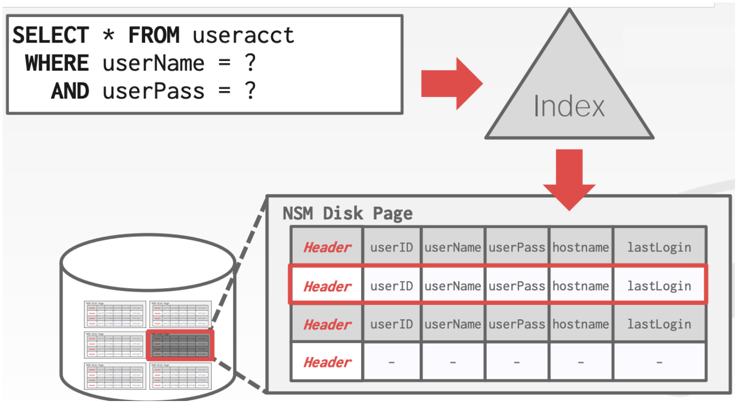

NSM Disk Page
Header | userID | userName | userPass | hostname | lastLogin
Header | userID | userName | userPass | hostname | lastLogin
Header | userID | userName | userPass | hostname | lastLogin
Header | - | - | - | - | -

N-ARY Storage Model (NSM)

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

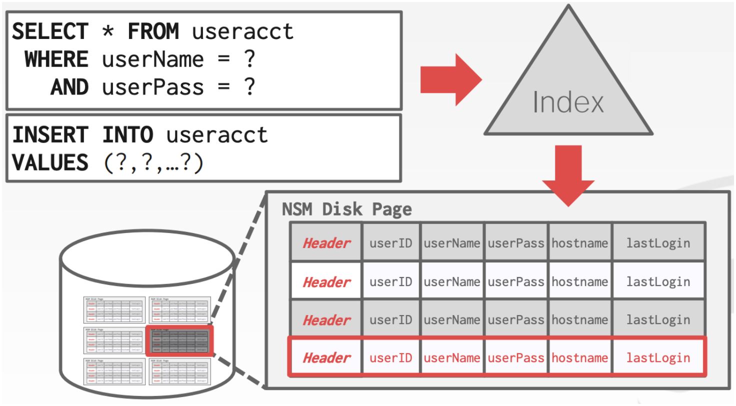
```
INSERT INTO useracct
VALUES (?,?,..,?)
```

N-ARY Storage Model (NSM)


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

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 = ?
```

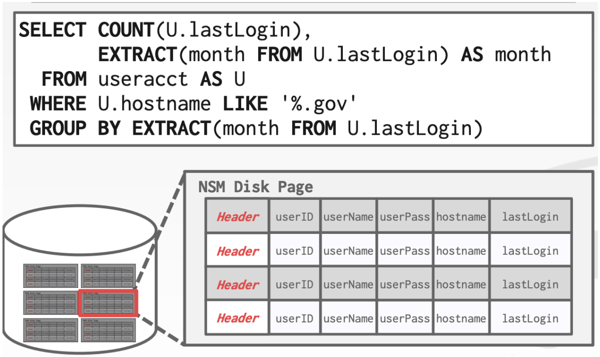
```
INSERT INTO useracct
VALUES (?,?,…?)
```

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)

```
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)



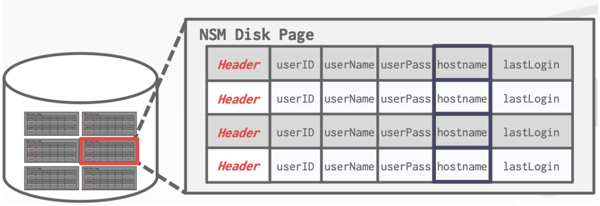
```
SELECT COUNT(U.lastLogin),
       EXTRACT(month FROM U.lastLogin) AS month
  FROM useracct AS U
 WHERE U.hostname LIKE '%.gov'
 GROUP BY EXTRACT(month FROM U.lastLogin)
```

NSM Disk Page

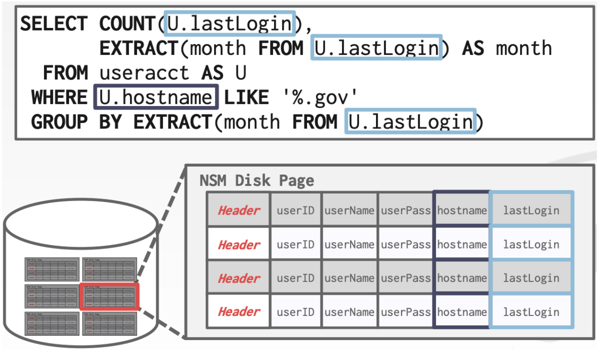
Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin

N-ARY Storage Model (NSM)

```
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)
```

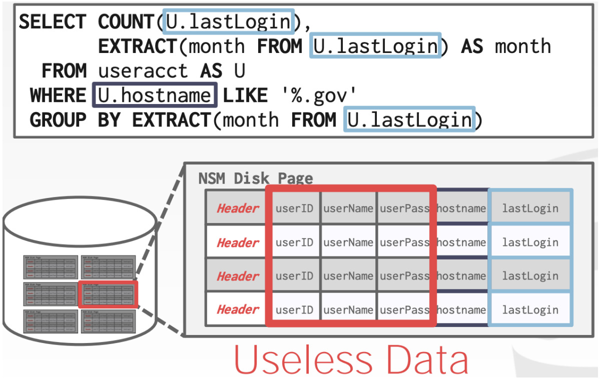

NSM Disk Page
Header userID userName userPass hostname lastLogin
Header userID userName userPass hostname lastLogin
Header userID userName userPass hostname lastLogin
Header userID userName userPass hostname lastLogin

N-ARY Storage Model (NSM)



```
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)

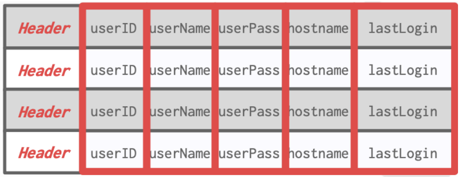
- Advantages
  - Fast inserts, updates, and deletes.
  - Good for queries that need the entire tuple.
- Disadvantages
  - Not good for scanning large portions of the table and/or a subset of the attributes.
  - This is because it pollutes the buffer pool by fetching data that is not needed for processing the query.

Decomposition Storage Model (DSM)

- The DBMS stores the values of a single attribute (column) for all tuples contiguously in a block of data.
    - *Vertically partition a database into a collection of individual columns that are stored separately*
    - 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.
- Common DSM Databases:
    - `Vertica, Amazon Redshift, Snowflake, Google BigQuery`

Decomposition Storage Model (DSM)

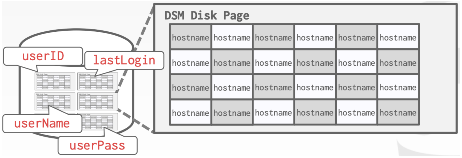
- The DBMS stores the values of a single attribute for all tuples contiguously in a page.
    - Also known as a “column store”.



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)

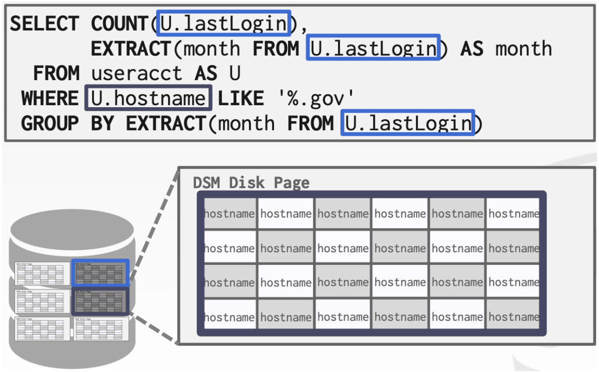
- The DBMS stores the values of a single attribute for all tuples contiguously in a page.
    - Also known as a “column store”.



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)
```

- *If there is a match on one page, how can we figure out a match on another page?*

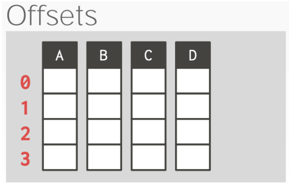
Decomposition Storage Model: Tuple Identification

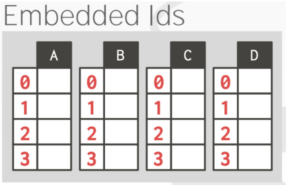
- To put the tuples back together when we are using a column store, we can use:
  - Choice #1: Fixed-length Offsets (most commonly used approach)
  - Choice #2: Embedded Tuple Ids (less common approach)

- *When decomposing a relational database into a column store model, it's essential to have a mechanism to identify and reconstruct the original tuples when needed.*

Choice #1: Fixed-length Offsets

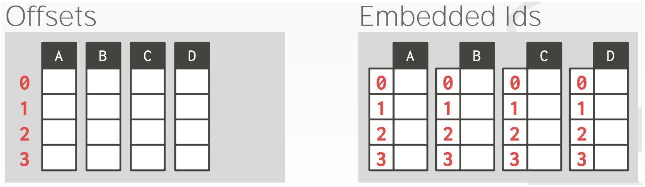
- Each column of the table is stored in its own independent block of data.
- Within each column, values are stored with fixed lengths, ensuring they occupy the same amount of space.
- To reconstruct a tuple, the system retrieves values at the same offset from each column, knowing that those values belong to the same tuple.
- Variable-length fields can be padded with extra spaces or characters to create fixed-length representations, but this can lead to storage inefficiencies.
- A more efficient approach involves using dictionaries:
  - Assign each unique value a fixed-size integer.
  - Store the dictionary mapping integers to values separately.
  - In the column, store only the integer references, ensuring fixed-length offsets.





Choice #2: Embedded Tuple Ids

- A less common approach
- Each column is still stored separately, but instead of using fixed-length offsets, each column contains embedded tuple IDs or pointers that indicate the position or identity of the tuple to which each value belongs.
- These embedded IDs link the values across columns, allowing the DBMS to reconstruct tuples by following the tuple IDs across columns.
- Note that this method has a large storage overhead because it needs to store a tuple id for every attribute entry.



Decomposition Storage Model (DSM)

- Advantages
  - Reduces the amount wasted I/O because the DBMS only reads the data that it needs.
  - Enable better query processing and data compression.
    - because all of the values for the same attribute are stored contiguously
- Disadvantages
  - Slow for point queries, inserts, updates, and deletes because of tuple splitting/stitching.

Modification of Tuples

How to handle the following operations on the tuple level?

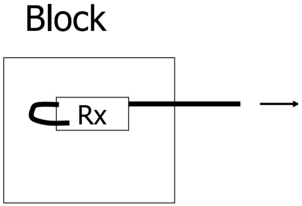
1. Insertion
2. Deletion
3. Update

1) Insertion

- **Easy case** Tuples with Fixed Length and not in Sequence (Unordered)
  - Insert new tuple at end of file
  - or, in deleted slot if one is available.
  - Deleted slots can be used to reuse space efficiently.
- **A little harder**: Tuples with Variable Size
  - If records are variable size, insertion becomes more complex.
  - Reusing space may not always be possible, leading to fragmentation where empty spaces are scattered throughout the storage.
  - This can impact storage efficiency over time.
- **A Difficult case**: Tuples in Sequence (Ordered)
  - Inserting tuples into a sequence can be challenging
  - Find position and slide following tuples to make room
  - If tuples are sequenced by linking, insert overflow blocks (there isn't enough space in the current block or node)

- *Overflow blocks are essentially additional blocks or nodes that are linked to the main sequence. They hold excess data when the main sequence's blocks are full.*

2) Deletion



Options

(a) Deleted and immediately reclaim space by shifting other tuples or removing overflows

(b) Mark deleted and list as free for re-use
  - May need chain of deleted tuples (for re-use)
  - Need a way to mark

  Trade-offs
  - How expensive is immediate reclaim?
    - How expensive is to move valid tuple to free space for immediate reclaim?
  - How much space is wasted?
    - e.g., deleted tuples, delete fields, ...

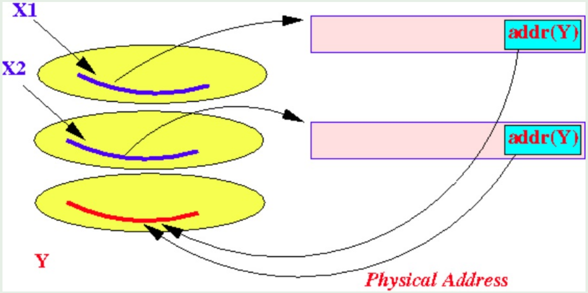
- *Space Efficiency vs. Performance: Immediate reclamation is more space-efficient but can be performance-intensive. Marking as deleted is less performance-intensive but can lead to space wastage.*

Concern with deletions

A caveat when using physical addresses to reference a block/record

Example

- Record Y can be referenced by other tuples (e.g., tuples $X1$ & $X2$)

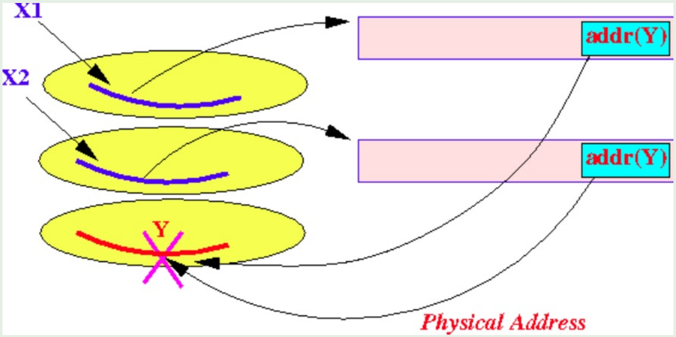


Concern with deletions

A caveat when using physical addresses to reference a block/record

Example

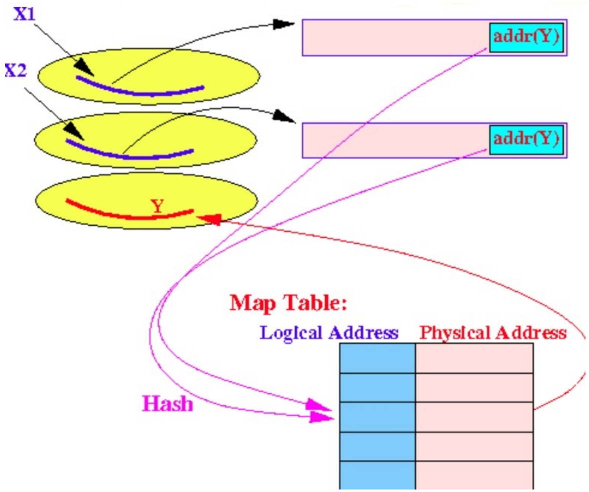
- When the tuple Y is deleted



- the physical addresses will reference an incorrect tuple

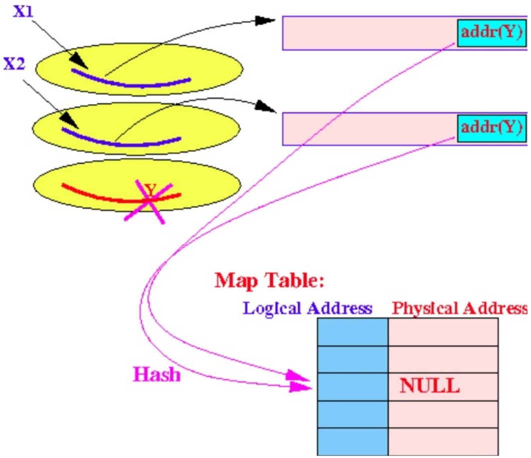
Techniques to handle tuple deletion

- Using logical addresses is easy
- Before deleting tuple Y that is referenced by tuples $X1$ and $X2$



Techniques to handle tuple deletion

- Using logical addresses is easy
- After deleting tuple Y



- Deleted tuple is identified by a NULL physical address in the Map table

Very important

- The logical address used by tuple Y must remain in the map table
- Furthermore:
  - The logical address used by tuple Y cannot be re-used

Techniques to handle tuple deletion

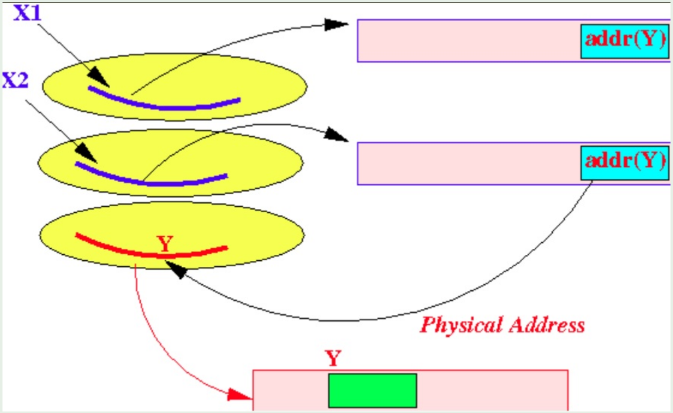
- Deleting a tuple using physical address: use a tombstone record
- Tombstone record
  - Used to mark tuples as deleted without physically removing them from the database storage.
  - Small tuples that typically contain minimal information, such as a tuple identifier and a deletion timestamp.
- When a tuple is deleted, it is replaced by the tombstone record
  - Instead of physically removing the tuple from the database or shifting other records, you replace the target tuple with a tombstone record
  - This tombstone record essentially **marks** the tuple as deleted without removing it entirely.
- This tombstone is permanent, it must exist until the entire database is reconstructed

- *A map table or mapping structure is used to keep track of the relationships between logical or physical addresses and the actual data. When a tuple is deleted, the tombstone record can be represented in the map table, often as a null pointer or a specific value indicating deletion, in place of the physical address.*

Tombstones

Example

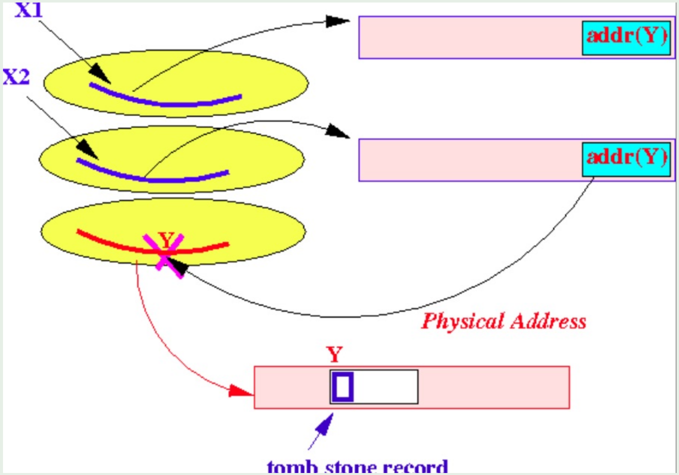
- Before deleting tuple Y



Tombstones

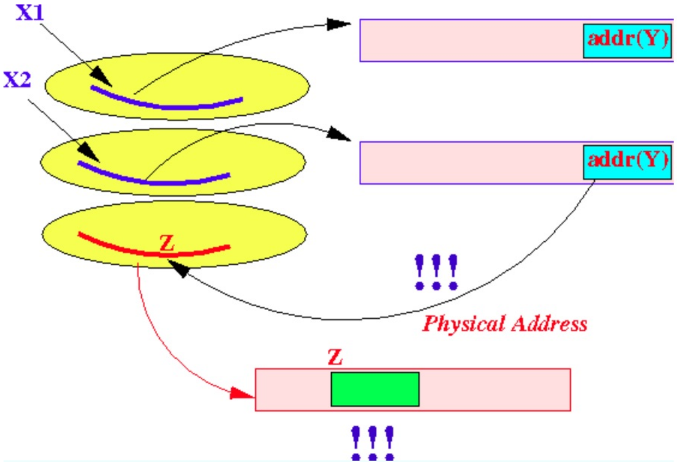
Example

- After deleting tuple Y



Tombstones

- When you insert a new tuple, you cannot use the space of a tombstone tuple (tombstone tuple must be preserved)
- Because: Existing tuple references to the deleted tuple will then references to the newly inserted tuple:



Update

- If new tuple is shorter than previous, easy
- If it is longer, need to shift tuples, create overflow blocks
- Note: We will never create a tombstone tuple in an update operation

Conclusion

- The storage manager is not entirely independent from the rest of the DBMS.
- A DBMS encodes and decodes the tuple's bytes into a set of attributes based on its schema.
- It is important to choose the right storage model for the target workload:
  - OLTP = Row Store
  - OLAP = Column Store

Database Storage: Next

- `Problem#1`: How the DBMS represents the database in files on disk
    - i.e., how to lay out data on disk.
- `Problem#2`: How the DBMS manages its memory and move data back-and-forth from disk.