15-721 DATABASE SYSTEMS

Lecture #10 – Storage Models & Data Layout

@Andy_Pavlo // Carnegie Mellon University // Spring 2017
TODAY’S AGENDA

Type Representation
In-Memory Data Layout
Storage Models
DATA ORGANIZATION

Index

Fixed-Length Data Blocks

Variable-Length Data Blocks

Memory Address
One can think of an in-memory database as just a large array of bytes. → The schema tells the DBMS how to convert the bytes into the appropriate type.

Each tuple is prefixed with a header that contains its meta-data.

Storing tuples with just their fixed-length data makes it easy to compute the starting point of any tuple.
DATA REPRESENTATION

**INTEGER/BIGINT/SMALLINT/TINYINT**
→ C/C++ Representation

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

**VARCHAR/VARBINARY/TEXT/BLOB**
→ Pointer to other location if type is ≥64-bits
→ Header with length and address to next location (if segmented), 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.

Store directly as specified by IEEE-754.

Typically faster than arbitrary precision numbers.
→ Example: FLOAT, REAL/DUAL
#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

\[ x+y = 0.300000001192092895508 \]
\[ 0.3 = 0.29999999999999998890 \]
FIXED PRECISION NUMBERS

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

Typically stored in a exact, variable-length binary representation with additional meta-data.
→ Like a **VARCHAR** but not stored as a string
typedef unsigned char NumericDigit;
typedef struct {
    int ndigits;
    int weight;
    int scale;
    int sign;
    NumericDigit *digits;
} numeric;
# CMU 15-721 (Spring 2017)

## POSTGRES: NUMERIC

### Typedef Definitions

- `typedef unsigned char NumericDigit;

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

### Attributes

- **# of Digits**
- **Weight of 1st Digit**
- **Scale Factor**
- **Positive/Negative/NaN**
- **Digit Storage**
/** * add_var() - * Full version of add functionality on variable level (handling signs). * result might point to one of the operands too without danger. */

int PGMPSnumeric_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 (add_abs(var1, var2, result) != 0)
                {
                    return -1;
                    result->sign = NUMERIC_POS;
                }
                break;
            }
        }
MSSQL: DECIMAL ENCODING

Values: $0.5, 10.77, 1.33$

Exponent: $3$ (i.e., $10^3$)

Initial Encoding: $0.5 \ 10^3\rightarrow 500$
$10.77 \ 10^3\rightarrow 10770$
$1.33 \ 10^3\rightarrow 1330$
MSSQL: DECIMAL ENCODING

Values: 0.5, 10.77, 1.33

Exponent: 3 (i.e., $10^3$)

Initial Encoding:  

- $0.5 \times 10^3 \rightarrow 500$
- $10.77 \times 10^3 \rightarrow 10770$
- $1.33 \times 10^3 \rightarrow 1330$

Base: 500
MSSQL: DECIMAL ENCODING

Values: 0.5, 10.77, 1.33
Exponent: 3 (i.e., $10^3$)
Initial Encoding:

- $0.5 \times 10^3 \rightarrow 500$
- $10.77 \times 10^3 \rightarrow 10770$
- $1.33 \times 10^3 \rightarrow 1330$

Base: 500
MSSQL: DECIMAL ENCODING

Values: 0.5, 10.77, 1.33
Exponent: 3 (i.e., $10^3$)
Initial Encoding:
- $0.5 \times 10^3 \rightarrow 500$
- $10.77 \times 10^3 \rightarrow 10770$
- $1.33 \times 10^3 \rightarrow 1330$

Base: 500

Final Encoding:
- $(0.5 \times 10^3) - 500 \rightarrow 0$
- $(10.77 \times 10^3) - 500 \rightarrow 10270$
- $(1.33 \times 10^3) - 500 \rightarrow 830$
CREATE TABLE AndySux ( id INT PRIMARY KEY, value BIGINT );
CREATE TABLE AndySux ( id INT PRIMARY KEY, value BIGINT );

```
char[]

header id value
```

reinterpret_cast<int32_t*>(address)
NULL DATA TYPES

Choice #1: Special Values
→ Designate a value to represent NULL for a particular data type (e.g., INT32_MIN).

Choice #2: Null Column Bitmap Header
→ Store a bitmap in the tuple header that specifies what attributes are null.

Choice #3: Per Attribute Null Flag
→ Store a flag that marks that a value is null.
→ Have to use more space than just a single bit because this messes up with word alignment.
## NULL DATA TYPES

### Integer Numbers

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Size</th>
<th>Size (Not Null)</th>
<th>Synonyms</th>
<th>Min Value</th>
<th>Max Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>2 bytes</td>
<td>1 byte</td>
<td>BOOLEAN</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>BIT</td>
<td>9 bytes</td>
<td>8 bytes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TINYINT</td>
<td>2 bytes</td>
<td>1 byte</td>
<td></td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>4 bytes</td>
<td>2 bytes</td>
<td></td>
<td>-32768</td>
<td>32767</td>
</tr>
<tr>
<td>MEDIUMINT</td>
<td>4 bytes</td>
<td>3 bytes</td>
<td></td>
<td>-8388608</td>
<td>8388607</td>
</tr>
<tr>
<td>INT</td>
<td>8 bytes</td>
<td>4 bytes</td>
<td>INTEGER</td>
<td>-2147483648</td>
<td>2147483647</td>
</tr>
<tr>
<td>BIGINT</td>
<td>12 bytes</td>
<td>8 bytes</td>
<td></td>
<td>-2 ** 63</td>
<td>(2 ** 63) - 1</td>
</tr>
</tbody>
</table>

Note: Having to use more space than just a single bit because this messes up with word alignment.
NULL DATA TYPES

Choice #1: Special Values
→ Designate a value to represent \texttt{NULL} for a particular data type (e.g., \texttt{INT32\_MIN}).

Choice #2: Null Column Bitmap Header
→ Store a bitmap in the tuple header that specifies what attributes are null.

Choice #3: Per Attribute Null Flag
→ Store a flag that marks that a value is null.
→ Have to use more space than just a single bit because this messes up with word alignment.
The truth is that you only need to worry about word-alignment for cache lines (e.g., 64 bytes).

I’m going to show you the basic idea using 64-bit words since it’s easier to see...
WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

CREATE TABLE AndySux (id INT PRIMARY KEY, cdate TIMESTAMP, color CHAR(2), zipcode INT);

char[]

64-bit Word 64-bit Word 64-bit Word 64-bit Word
All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

```sql
CREATE TABLE AndySux ( id INT PRIMARY KEY, cdate TIMESTAMP, color CHAR(2), zipcode INT );
```

- **id**: 32 bits
- **64-bit Words**
- **char[]**: 64 bits
WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

CREATE TABLE AndySux ( id INT PRIMARY KEY, cdate TIMESTAMP, color CHAR(2), zipcode INT );
WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

CREATE TABLE AndySux ( id INT PRIMARY KEY,
cdate TIMESTAMP,
color CHAR(2),
zipcode INT );
WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

```
CREATE TABLE AndySux (  
id INT PRIMARY KEY,  
cdate TIMESTAMP,  
color CHAR(2),  
zipcode INT  );
```

```
<table>
<thead>
<tr>
<th></th>
<th>32-bits</th>
<th>64-bits</th>
<th>16-bits</th>
<th>32-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>INT</td>
<td>PRIMARY</td>
<td>KEY</td>
<td>INT</td>
</tr>
<tr>
<td>cdate</td>
<td>TIMESTAMP</td>
<td></td>
<td>cdate</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>color</td>
<td>CHAR(2)</td>
<td></td>
<td>color</td>
<td>CHAR(2)</td>
</tr>
<tr>
<td>zipcode</td>
<td>INT</td>
<td></td>
<td>zipcode</td>
<td>INT</td>
</tr>
</tbody>
</table>
```

`char[]`

```
<table>
<thead>
<tr>
<th>id</th>
<th>cdate</th>
<th>c</th>
<th>zipcode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>c</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

64-bit Word 64-bit Word 64-bit Word 64-bit Word
WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

CREATE TABLE AndySux ( id INT PRIMARY KEY, cdate TIMESTAMP, color CHAR(2), zipcode INT );

<table>
<thead>
<tr>
<th>64-bit Word</th>
<th>64-bit Word</th>
<th>64-bit Word</th>
<th>64-bit Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>cdate</td>
<td>c</td>
<td>zipc</td>
</tr>
</tbody>
</table>

char[]
WORD-ALIGNED TUPLES

If the CPU fetches a 64-bit value that is not word-aligned, it has three choices:
→ Execute two reads to load the appropriate parts of the data word and reassemble them.
→ Read some unexpected combination of bytes assembled into a 64-bit word.
→ Throw an exception
WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

CREATE TABLE AndySux ( id INT PRIMARY KEY, cdate TIMESTAMP, color CHAR(2), zipcode INT );
STORAGE MODELS

N-ary Storage Model (NSM)
Decomposition Storage Model (DSM)
Hybrid Storage Model
N-ARY STORAGE MODEL (NSM)

The DBMS stores all of the attributes for a single tuple contiguously.

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

Use the tuple-at-a-time iterator model.
Choice #1: Heap-Organized Tables
→ Tuples are stored in blocks called a heap.
→ The heap does not necessarily define an order.

Choice #2: Index-Organized Tables
→ Tuples are stored in the index itself.
→ Not quite the same as a clustered index.
CLUSTERED INDEXES

The table is stored in the sort order specified by the primary key.
→ Can be either heap- or index-organized storage.

Some DBMSs always use a clustered index.
→ If a table doesn’t include a pkey, the DBMS will automatically make a hidden row id pkey.

Other DBMSs cannot use them at all.
→ A clustered index is non-practical in a MVCC DBMS using the Append Storage Method.
N-ARY STORAGE MODEL (NSM)

Advantages
→ Fast inserts, updates, and deletes.
→ Good for queries that need the entire tuple.
→ Can use index-oriented physical storage.

Disadvantages
→ Not good for scanning large portions of the table and/or a subset of the attributes.
DECOMPOSITION STORAGE MODEL (DSM)

The DBMS stores a single attribute for all tuples contiguously in a block of data.
→ Sometimes also called **vertical partitioning**.

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

Use the vector-at-a-time iterator model.
DECOMPOSITION STORAGE MODEL (DSM)

1970s: Cantor DBMS
1980s: DSM Proposal
1990s: SybaseIQ (in-memory only)
2000s: Vertica, Vectorwise, MonetDB
2010s: “The Big Three”
        Cloudera Impala, Amazon Redshift, SAP HANA, MemSQL
Some columnar DBMSs store data in sorted order to maximize compression.
→ Bitmap indexes with RLE from last class

Vertica does not even use indexes because all columns are sorted.
**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 work because the DBMS only reads the data that it needs.
→ Better compression.

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

Data is “hot” when first entered into database
→ A newly inserted tuple is more likely to be updated again the near future.

As a tuple ages, it is updated less frequently.
→ At some point, a tuple is only accessed in read-only queries along with other tuples.

What if we want to use this data to make decisions that affect new txns?
BIFURCATED ENVIRONMENT

OLTP Data Silos  ⏰ Extract ⏰ Transform ⏰ Load  ⏰ OLAP Data Warehouse
HYBRID STORAGE MODEL

Single logical database instance that uses different storage models for hot and cold data.

Store new data in NSM for fast OLTP
Migrate data to DSM for more efficient OLAP
HYBRID STORAGE MODEL

Choice #1: Separate Execution Engines
→ Use separate execution engines that are optimized for either NSM or DSM databases.

Choice #2: Single, Flexible Architecture
→ Use single execution engine that is able to efficiently operate on both NSM and DSM databases.
SEPARATE EXECUTION ENGINES

Run separate “internal” DBMSs that each only operate on DSM or NSM data.
→ Need to combine query results from both engines to appear as a single logical database to the application.
→ Have to use a synchronization method (e.g., 2PC) if a txn spans execution engines.

Two approaches to do this:
→ **Fractured Mirrors** (Oracle, IBM)
→ **Delta Store** (SAP HANA)
FRACTURED MIRRORS

Store a second copy of the database in a DSM layout that is automatically updated. → All updates are first entered in NSM then eventually copied into DSM mirror.
Store a second copy of the database in a DSM layout that is automatically updated.
→ All updates are first entered in NSM then eventually copied into DSM mirror.
Stage updates to the database in an NSM table. A background thread migrates updates from delta store and applies them to DSM data.
CATEGORIZING DATA

Choice #1: Manual Approach
→ DBA specifies what tables should be stored as DSM.

Choice #2: Off-line Approach
→ DBMS monitors access logs offline and then makes decision about what data to move to DSM.

Choice #3: On-line Approach
→ DBMS tracks access patterns at runtime and then makes decision about what data to move to DSM.
PELOTON ADAPTIVE STORAGE

Employ a single execution engine architecture that is able to operate on both NSM and DSM data.
→ Don’t need to store two copies of the database.
→ Don’t need to sync multiple database segments.

Note that a DBMS can still use the delta-store approach with this single-engine architecture.
**PELOTON ADAPTIVE STORAGE**

**Original Data**

**UPDATE**

**SET**

\[
\begin{align*}
A &= 123, \\
B &= 456, \\
C &= 789 \\
D &= "xxx"
\end{align*}
\]

**WHERE**

**SELECT**

\[
\begin{align*}
\text{AVG}(B)
\end{align*}
\]

**FROM**

AndySux

**WHERE**

\[
C = "yyy"
\]
PELOTON ADAPTIVE STORAGE

**Original Data**

```
UPDATE AndySux
SET A = 123,
    B = 456,
    C = 789
WHERE D = "xxx"

SELECT AVG(B)
FROM AndySux
WHERE C = "yyy"
```

**Adapted Data**
Introduce an indirection layer that abstracts the true layout of tuples from query operators.
TILE ARCHITECTURE

Introduce an indirection layer that abstracts the true layout of tuples from query operators.

<table>
<thead>
<tr>
<th>Tile Group Header</th>
<th>H</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</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>
</tbody>
</table>

Tile #1

Tile #2

Tile #3

Tile #4
TILE ARCHITECTURE

Introduce an indirection layer that abstracts the true layout of tuples from query operators.
TILE ARCHITECTURE

Introduce an indirection layer that abstracts the true layout of tuples from query operators.

```sql
SELECT AVG(B)
FROM AndySux
WHERE C = "yyy"
```
PELOTON ADAPTIVE STORAGE

Row Layout  Column Layout  Adaptive Layout
H₂O ADAPTIVE STORAGE

Examine the access patterns of queries and then dynamically reconfigure the database to optimize decomposition and layout.

Copies columns into a new layout that is optimized for each query.

→ Think of it like a mini fractured mirror.
→ Use query compilation to speed up operations.
H₂O ADAPTIVE STORAGE

Original Data

UPDATE AndySux
SET A = 123, B = 456, C = 789
WHERE D = “xxx”

SELECT AVG(B)
FROM AndySux
WHERE C = “yyy”
H₂O ADAPTIVE STORAGE

**Original Data**

UPDATE AndySux
SET A = 123,
B = 456,
C = 789
WHERE D = "xxx"

SELECT AVG(B)
FROM AndySux
WHERE C = "yyy"

**Adapted Data**
H₂O ADAPTIVE STORAGE

This approach is unable to handle updates to the database. It also unable to store tuples in the same table in a different layout.

This is because they are missing the ability to categorize whether data is hot or cold...
PARTING THOUGHTS

A flexible architecture that supports a hybrid storage model is the next major trend in DBMSs.

This will enable relational DBMSs to support all known database workloads except for matrices in machine learning.