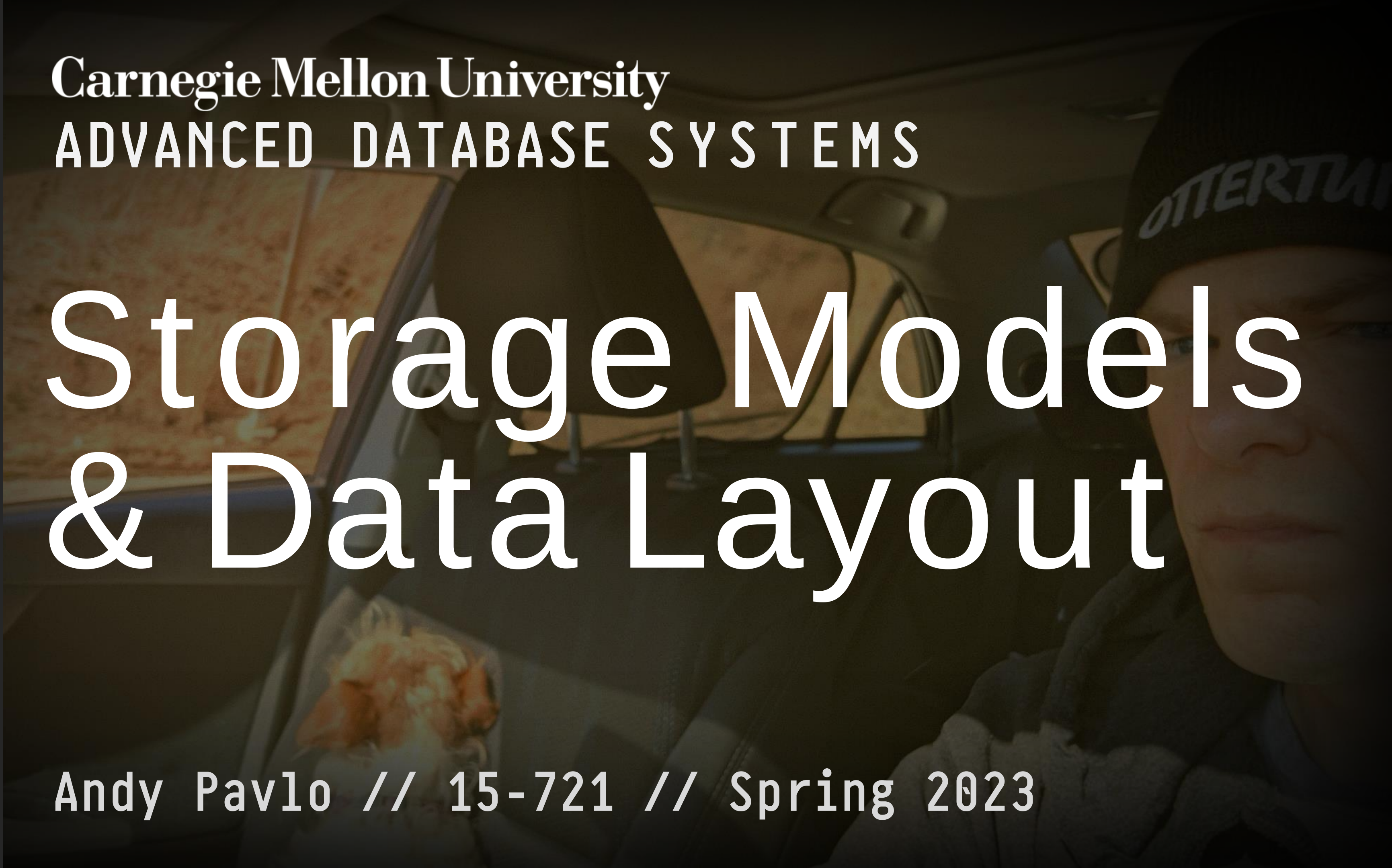
Lecture #03

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS



Storage Models & Data Layout

Andy Pavlo // 15-721 // Spring 2023

OBSERVATION

Today's lecture is about the lowest physical representation of data in a database.

What data "looks" like determines almost a DBMS's entire system architecture.

- → Processing Model
- → Tuple Materialization Strategy
- → Operator Algorithms
- → Data Ingestion / Updates
- → Concurrency Control (*we will ignore this*)
- → Query Optimization

TODAY'S AGENDA

Storage Models

Type Representation

Partitioning

STORAGE MODELS

A DBMS's **storage model** specifies how it physically organizes tuples on disk and in memory.

Choice #1: *N*-ary Storage Model (NSM)
Choice #2: Decomposition Storage Model (DSM)
Choice #3: Hybrid Storage Model (PAX)

COLUMN-STORES VS. ROW-STORES: HOW DIFFERENT ARE THEY REALLY?
SIGMOD 2008

N-ARY STORAGE MODEL (NSM)

The DBMS stores (almost) all the attributes for a single tuple contiguously in a single page.

Ideal for OLTP workloads where txns tend to access individual entities and insert-heavy workloads.

→ Use the tuple-at-a-time ***iterator processing model***.

NSM database page sizes are typically some constant multiple of **4 KB** hardware pages.

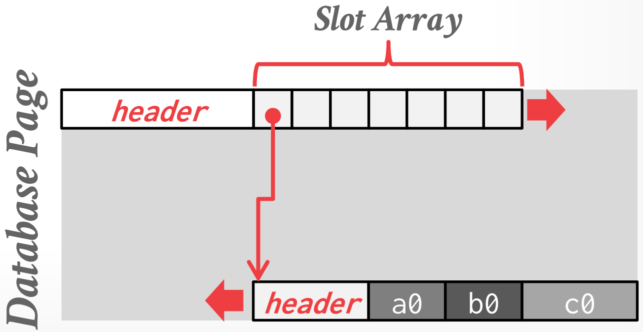
→ Example: Oracle (4 KB), Postgres (8 KB), MySQL (16 KB)

NSM: PHYSICAL ORGANIZATION

A disk-oriented NSM system stores a tuple's fixed-length and variable-length attributes contiguously in a single slotted page.

The tuple's **record id** (page#, slot#) is how the DBMS uniquely identifies a physical tuple.

	Col A	Col B	Col C
Row #0	a0	b0	c0
Row #1	a1	b1	c1
Row #2	a2	b2	c2
Row #3	a3	b3	c3
Row #4	a4	b4	c4
Row #5	a5	b5	c5

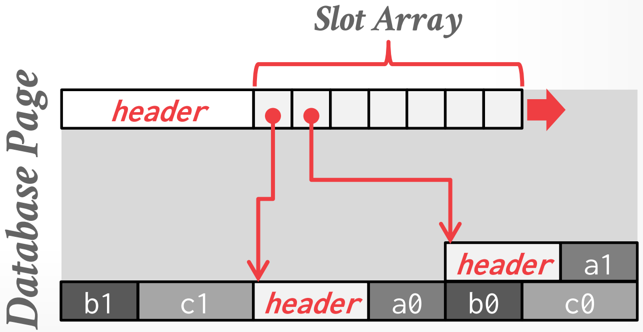


NSM: PHYSICAL ORGANIZATION

A disk-oriented NSM system stores a tuple's fixed-length and variable-length attributes contiguously in a single slotted page.

The tuple's **record id** (page#, slot#) is how the DBMS uniquely identifies a physical tuple.

	Col A	Col B	Col C
Row #0	a0	b0	c0
Row #1	a1	b1	c1
Row #2	a2	b2	c2
Row #3	a3	b3	c3
Row #4	a4	b4	c4
Row #5	a5	b5	c5

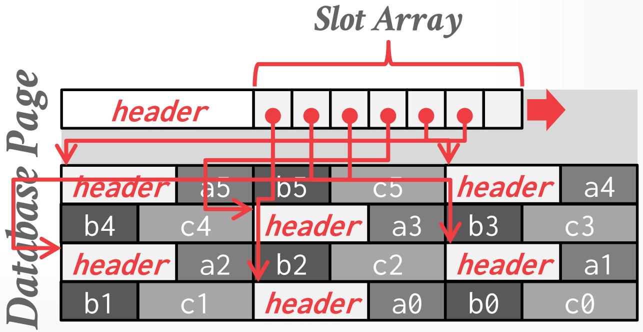


NSM: PHYSICAL ORGANIZATION

A disk-oriented NSM system stores a tuple's fixed-length and variable-length attributes contiguously in a single slotted page.

The tuple's **record id** (page#, slot#) is how the DBMS uniquely identifies a physical tuple.

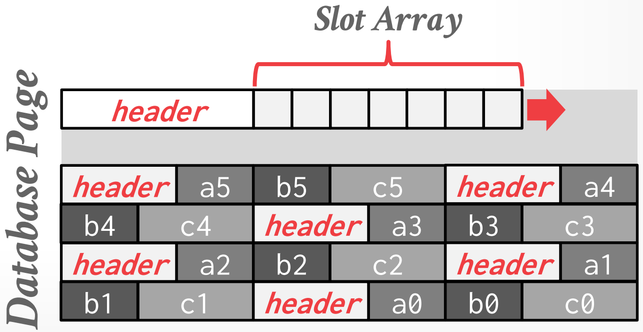
	Col A	Col B	Col C
Row #0	a0	b0	c0
Row #1	a1	b1	c1
Row #2	a2	b2	c2
Row #3	a3	b3	c3
Row #4	a4	b4	c4
Row #5	a5	b5	c5



NSM: PHYSICAL ORGANIZATION

```
SELECT SUM(colA), AVG(colC)
  FROM xxx
 WHERE colA > 1000
```

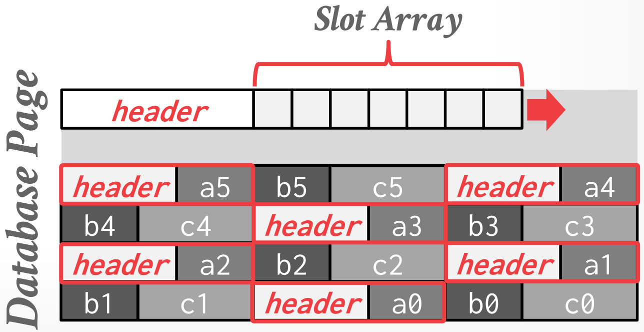
	Col A	Col B	Col C
Row #0	a0	b0	c0
Row #1	a1	b1	c1
Row #2	a2	b2	c2
Row #3	a3	b3	c3
Row #4	a4	b4	c4
Row #5	a5	b5	c5



NSM: PHYSICAL ORGANIZATION

```
SELECT SUM(colA), AVG(colC)
  FROM xxx
 WHERE colA > 1000
```

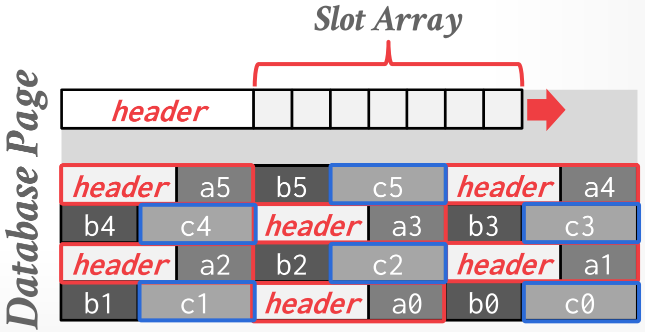
	Col A	Col B	Col C
Row #0	a0	b0	c0
Row #1	a1	b1	c1
Row #2	a2	b2	c2
Row #3	a3	b3	c3
Row #4	a4	b4	c4
Row #5	a5	b5	c5



NSM: PHYSICAL ORGANIZATION

```
SELECT SUM(colA), AVG(colC)
  FROM xxx
 WHERE colA > 1000
```

	Col A	Col B	Col C
Row #0	a0	b0	c0
Row #1	a1	b1	c1
Row #2	a2	b2	c2
Row #3	a3	b3	c3
Row #4	a4	b4	c4
Row #5	a5	b5	c5



N-ARY STORAGE MODEL (NSM)

Advantages

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

Disadvantages

→ Not good for scanning large portions of the table and/or a subset of the attributes.
→ Terrible memory locality in access patterns.
→ Not ideal for compression because of multiple value domains within a single page.

DECOMPOSITION STORAGE MODEL (DSM)

The DBMS stores a single attribute for all tuples contiguously in a block of data.

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

→ Use a batched ***vectorized processing model***.

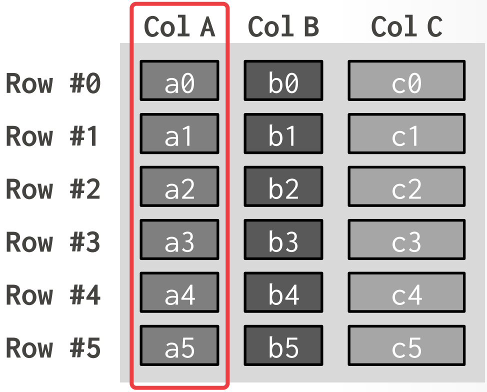
File sizes are larger (100s of MBs), but it may still organize tuples within the file into smaller groups.

DSM: PHYSICAL ORGANIZATION

Store attributes and meta-data (e.g., nulls) in separate arrays of **fixed-length** values.

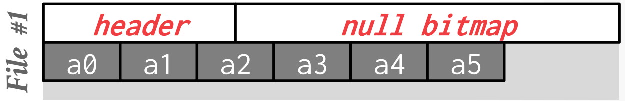
→ Most systems identify unique physical tuples using offsets into these arrays.

→ Need to handle variable-length values...



	Col A	Col B	Col C
Row #0	a0	b0	c0
Row #1	a1	b1	c1
Row #2	a2	b2	c2
Row #3	a3	b3	c3
Row #4	a4	b4	c4
Row #5	a5	b5	c5

Maintain a separate file per attribute with a dedicated header area for meta-data about entire column.

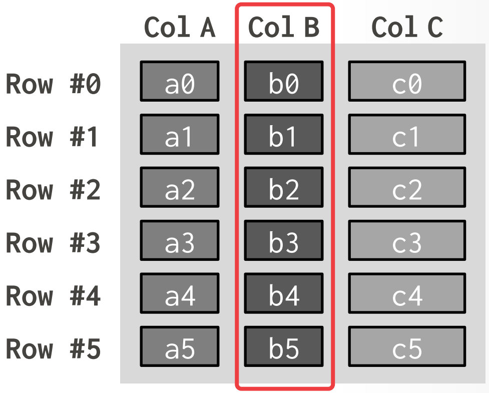


DSM: PHYSICAL ORGANIZATION

Store attributes and meta-data (e.g., nulls) in separate arrays of **fixed-length** values.

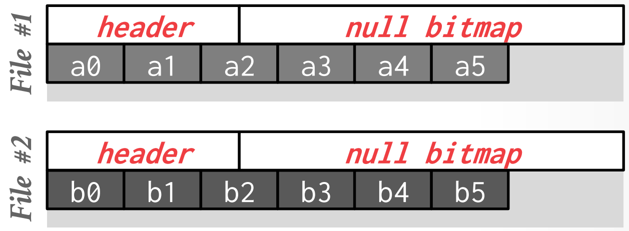
→ Most systems identify unique physical tuples using offsets into these arrays.

→ Need to handle variable-length values…



	Col A	Col B	Col C
Row #0	a0	b0	c0
Row #1	a1	b1	c1
Row #2	a2	b2	c2
Row #3	a3	b3	c3
Row #4	a4	b4	c4
Row #5	a5	b5	c5

Maintain a separate file per attribute with a dedicated header area for meta-data about entire column.



File #1

header | *null bitmap*

a0 | a1 | a2 | a3 | a4 | a5

File #2

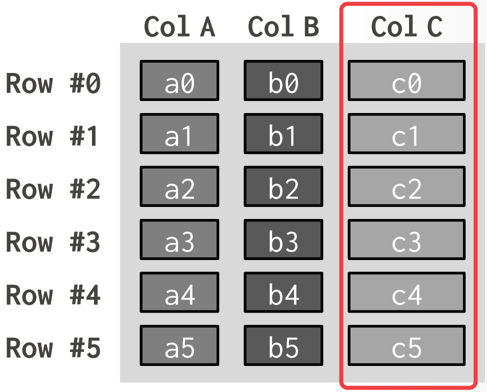
header | *null bitmap*

b0 | b1 | b2 | b3 | b4 | b5

DSM: PHYSICAL ORGANIZATION

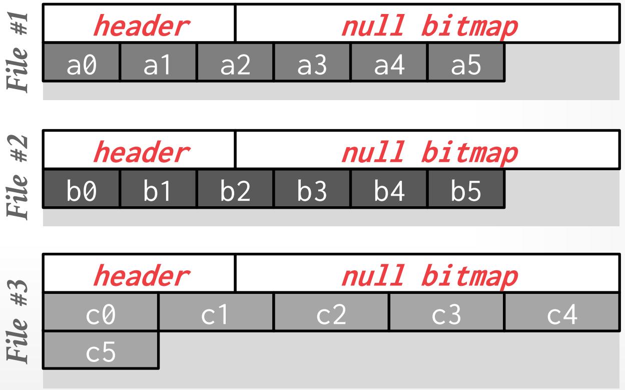
Store attributes and meta-data (e.g., nulls) in separate arrays of **fixed-length** values.

- → Most systems identify unique physical tuples using offsets into these arrays.
- → Need to handle variable-length values…



	Col A	Col B	Col C
Row #0	a0	b0	c0
Row #1	a1	b1	c1
Row #2	a2	b2	c2
Row #3	a3	b3	c3
Row #4	a4	b4	c4
Row #5	a5	b5	c5

Maintain a separate file per attribute with a dedicated header area for meta-data about entire column.



DSM: TUPLE IDENTIFICATION

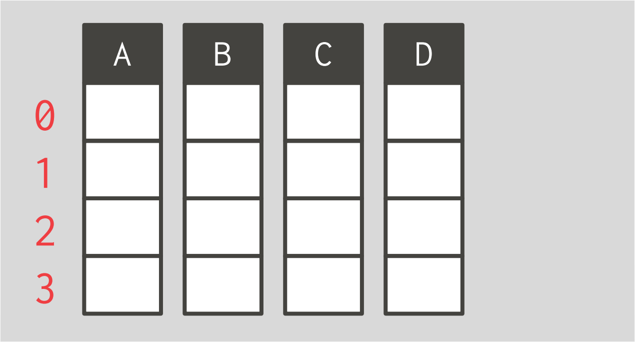
Choice #1: Fixed-length Offsets

→ Each value is the same length for an attribute.

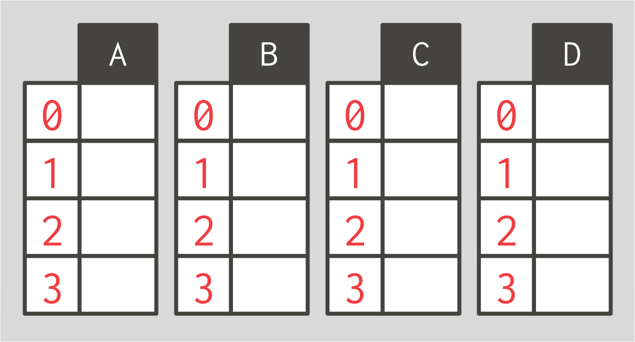
Choice #2: Embedded Tuple Ids

→ Each value is stored with its tuple id in a column.

Offsets



Embedded Ids



DSM: VARIABLE-LENGTH DATA

Padding variable-length fields to ensure they are fixed-length is wasteful, especially for large attributes.

A better approach is to use ***dictionary compression*** to convert repetitive variable-length data into fixed-length values (typically 32-bit integers).

→ More on this next week.

DSM: SYSTEM HISTORY

1970s: Cantor DBMS

1980s: DSM Proposal

1990s: SybaseIQ (in-memory only)

2000s: Vertica, Vectorwise, MonetDB

2010s: Everyone

DECOMPOSITION STORAGE MODEL (DSM)

Advantages

→ Reduces the amount wasted I/O per query because the DBMS only reads the data that it needs.
→ Faster query processing because of increased locality and cached data reuse.
→ Better data compression (more on this later)

Disadvantages

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

OBSERVATION

OLAP queries almost never access a single column in a table by itself.

→ At some point during query execution, the DBMS must get other columns and stitch the original tuple back together.

But we still need to store data in a columnar format to get the storage + execution benefits.

We need columnar scheme that still stores attributes separately but keeps the data for each tuple physically close to each other…

PAX STORAGE MODEL

Partition Attributes Across (PAX) is a hybrid storage model that vertically partitions attributes within a database page.

→ This is what Paraquet and Orc use.

The goal is to get the benefit of faster processing on columnar storage while retaining the spatial locality benefits of row storage.

DATA PAGE LAYOUTS FOR RELATIONAL DATABASES ON DEEP MEMORY HIERARCHIES
VLDB JOURNAL 2002

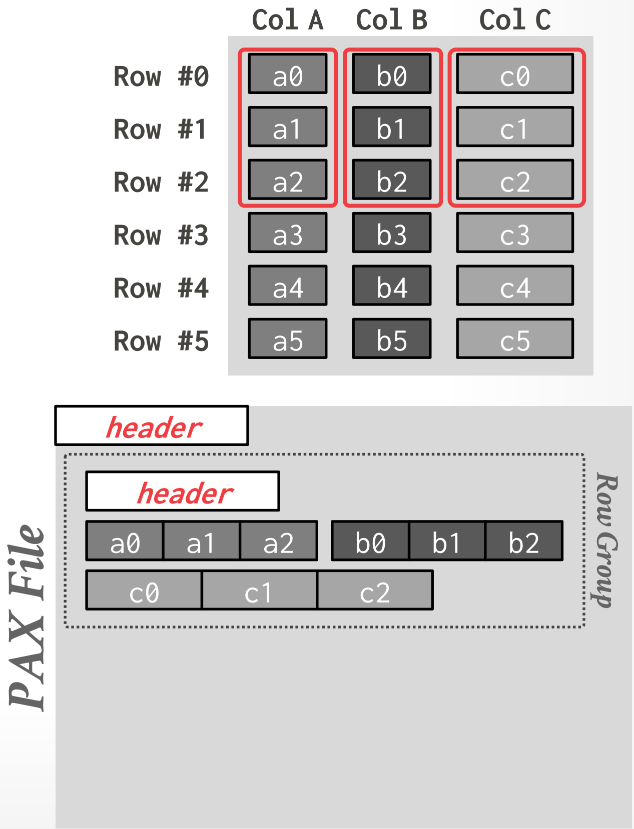
PAX: PHYSICAL ORGANIZATION

Horizontally partition rows into groups. Then vertically partition their attributes into columns.

Global header contains directory with the offsets to the file's row groups.

→ This is stored in the footer if the file is immutable (Parquet, Orc).

Each row group contains its own meta-data header about its contents.



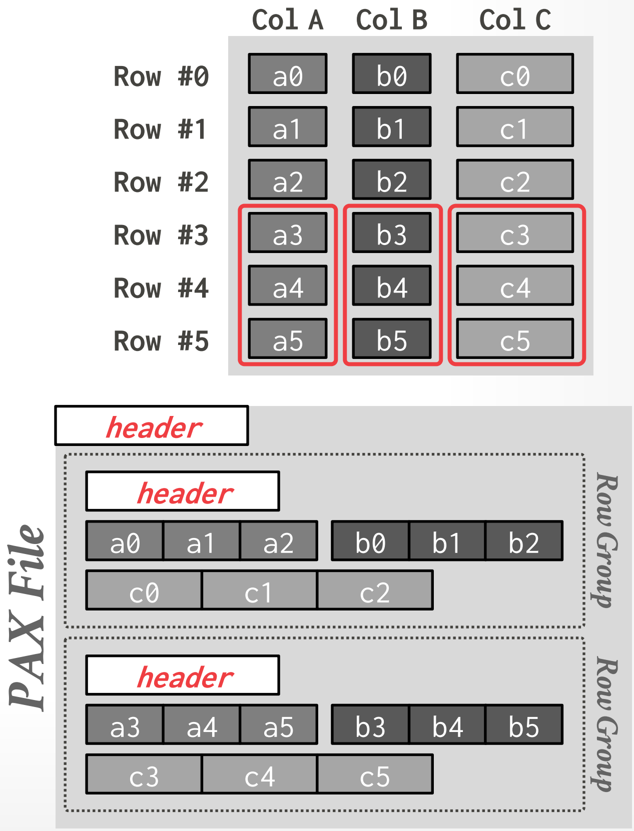
PAX: PHYSICAL ORGANIZATION

Horizontally partition rows into groups. Then vertically partition their attributes into columns.

Global header contains directory with the offsets to the file's row groups.

→ This is stored in the footer if the file is immutable (Parquet, Orc).

Each row group contains its own meta-data header about its contents.

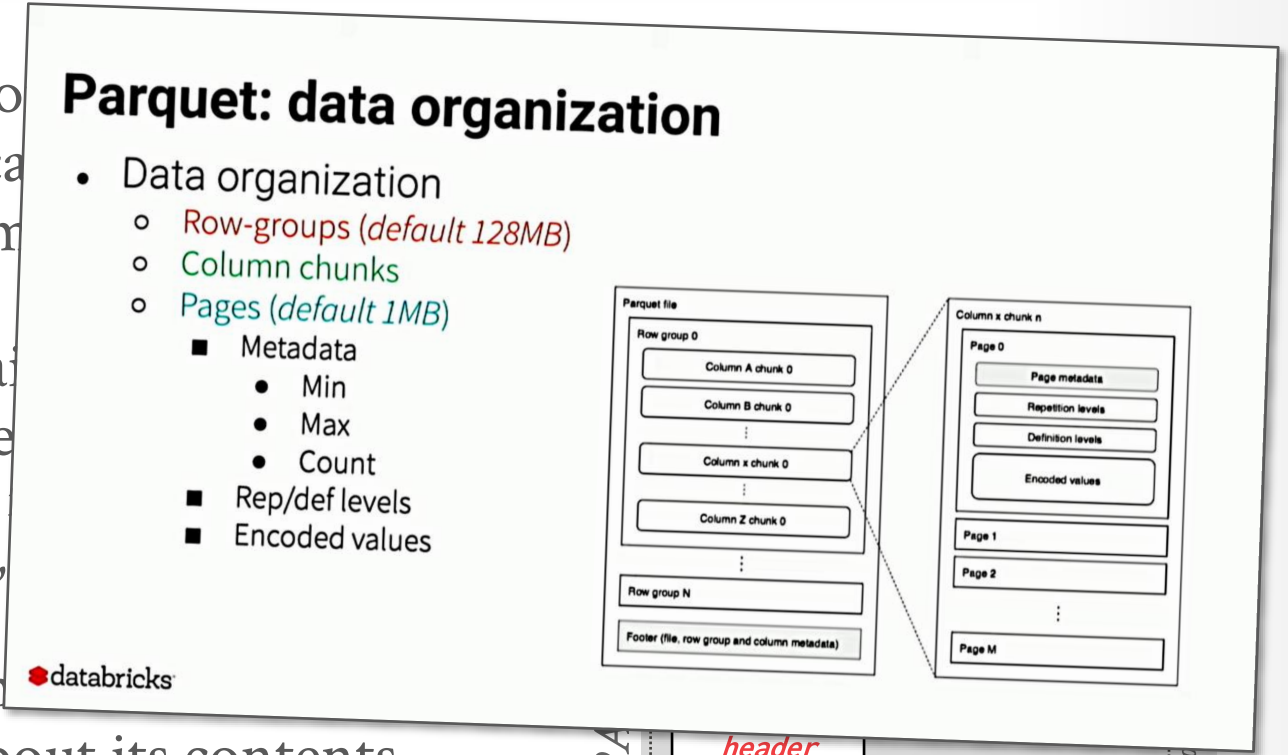


PAX: PHYSICAL ORGANIZATION

Horizontally partitio groups. Then vertica attributes into colum

Global header contai the offsets to the file

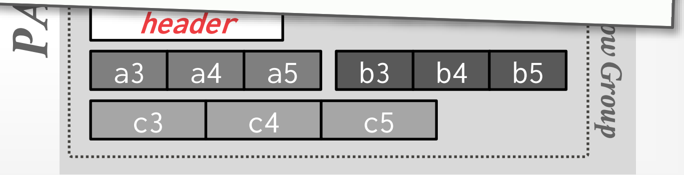
→ This is stored in the immutable (Parquet,



Parquet: data organization

- Data organization
  - Row-groups (*default 128MB*)
  - Column chunks
  - Pages (*default 1MB*)
    - Metadata
      - Min
      - Max
      - Count
    - Rep/def levels
    - Encoded values

Each row group con meta-data header about its contents.



MEMORY PAGES

An OLAP DBMS uses the buffer pool manager methods that we discussed in the intro course.

OS maps physical pages to virtual memory pages.

The CPU's MMU maintains a TLB that contains the physical address of a virtual memory page.

- → The TLB resides in the CPU caches.
- → It cannot obviously store every possible entry for a large memory machine.

When you allocate a block of memory, the allocator keeps that it aligned to page boundaries.

TRANSPARENT HUGE PAGES (THP)

Instead of always allocating memory in 4 KB pages, Linux supports creating larger pages (2MB to 1GB)

→ Each page must be a contiguous blocks of memory.

→ Greatly reduces the # of TLB entries

With THP, the OS reorganizes pages in the background to keep things compact.

→ Split larger pages into smaller pages.

→ Combine smaller pages into larger pages.

→ Can cause the DBMS process to stall on memory access.

Source: Alexandr Nikitin

TRANSPARENT HUGE PAGES (THP)

Historically, every DBMS advises you to disable this THP on Linux:

→ Oracle, SingleStore, NuoDB, MongoDB, Sybase, TiDB.
→ Vertica says to enable THP only for newer Linux distros.

Recent research from Google suggests that huge pages improved their data center workload by 7%.

→ 6.5% improvement in Spanner's throughput

Source: Evan Jones

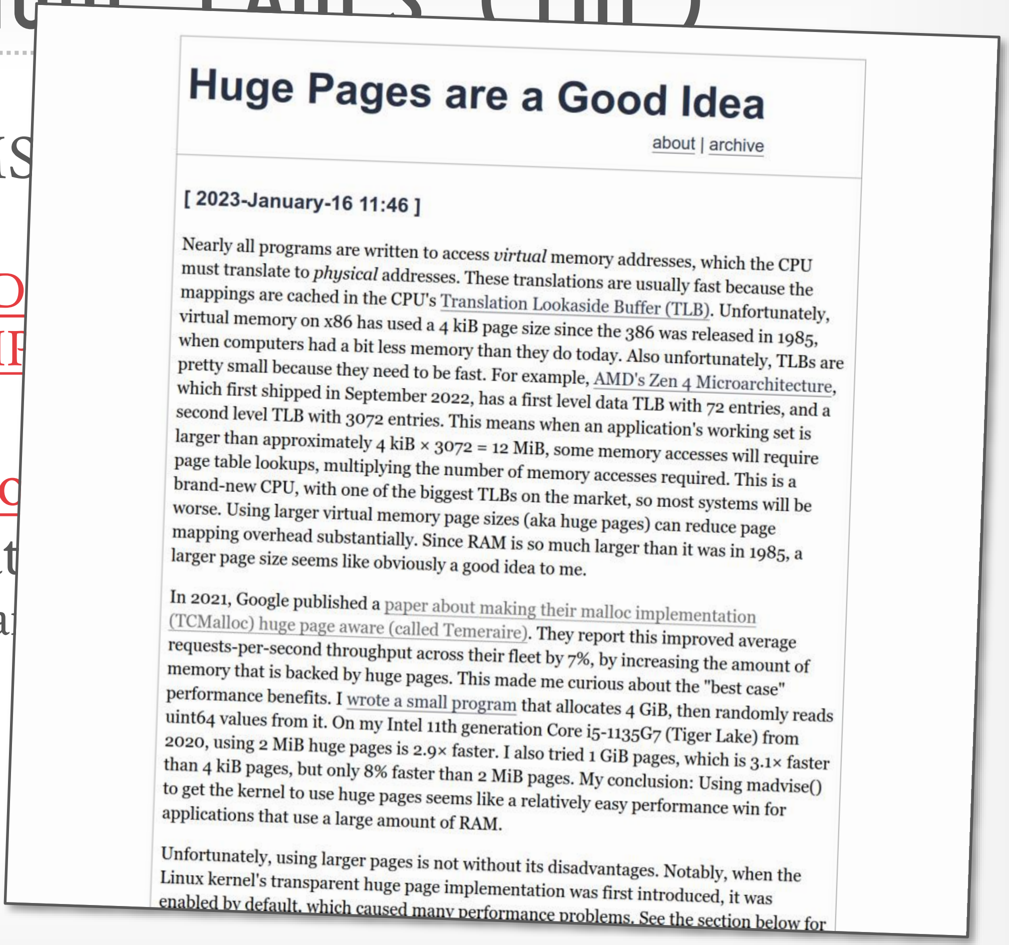
TRANSPARENT HUGE PAGES (THP)

Historically, every DBMS
THP on Linux:

→ Oracle, SingleStore, NuoD
→ Vertica says to enable THP

Recent research from Go
pages improved their dat

→ 6.5% improvement in Spa



Huge Pages are a Good Idea

about | archive

[2023-January-16 11:46]

Nearly all programs are written to access *virtual* memory addresses, which the CPU must translate to *physical* addresses. These translations are usually fast because the mappings are cached in the CPU's Translation Lookaside Buffer (TLB). Unfortunately, virtual memory on x86 has used a 4 kiB page size since the 386 was released in 1985, when computers had a bit less memory than they do today. Also unfortunately, TLBs are pretty small because they need to be fast. For example, AMD's Zen 4 Microarchitecture, which first shipped in September 2022, has a first level data TLB with 72 entries, and a second level TLB with 3072 entries. This means when an application's working set is larger than approximately 4 kiB × 3072 = 12 MiB, some memory accesses will require page table lookups, multiplying the number of memory accesses required. This is a brand-new CPU, with one of the biggest TLBs on the market, so most systems will be worse. Using larger virtual memory page sizes (aka huge pages) can reduce page mapping overhead substantially. Since RAM is so much larger than it was in 1985, a larger page size seems like obviously a good idea to me.

In 2021, Google published a paper about making their malloc implementation (TCMalloc) huge page aware (called Temeraire). They report this improved average requests-per-second throughput across their fleet by 7%, by increasing the amount of memory that is backed by huge pages. This made me curious about the "best case" performance benefits. I wrote a small program that allocates 4 GiB, then randomly reads uint64 values from it. On my Intel 11th generation Core i5-1135G7 (Tiger Lake) from 2020, using 2 MiB huge pages is 2.9× faster. I also tried 1 GiB pages, which is 3.1× faster than 4 kiB pages, but only 8% faster than 2 MiB pages. My conclusion: Using madvise() to get the kernel to use huge pages seems like a relatively easy performance win for applications that use a large amount of RAM.

Unfortunately, using larger pages is not without its disadvantages. Notably, when the Linux kernel's transparent huge page implementation was first introduced, it was enabled by default, which caused many performance problems. See the section below for

Source: Evan Jones

DATA REPRESENTATION

INTEGER/BIGINT/SMALLINT/TINYINT

→ C/C++ Representation

FLOAT/REAL vs. **NUMERIC/DECIMAL**

→ IEEE-754 Standard / Fixed-point Decimals

TIME/DATE/TIMESTAMP

→ 32/64-bit int of (micro/milli)seconds since Unix epoch

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.

→ Most DBMSs use dictionary compression for these.

VARIABLE PRECISION NUMBERS

Inexact, variable-precision numeric type that uses the "native" C/C++ types.

Store directly as specified by IEEE-754.

→ Example: **FLOAT**, **REAL**/**DOUBLE**

These types are typically faster than fixed precision numbers because CPU ISA's (Xeon, Arm) have instructions / registers to support them.

But they do not guarantee exact values...

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

```
#include <stdio.h>

in

}
```

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    float x = 0.1;
    float y = 0.2;
    printf("x+y = %.20f\n", x+y);
    printf("0.3 = %.20f\n", 0.3);
}
```

Output

```
x+y = 0.300000
0.3 = 0.300000
```

```
x+y = 0.30000001192092895508
0.3 = 0.29999999999999998890
```

FIXED PRECISION NUMBERS

Numeric data types with (potentially) arbitrary precision and scale. Used when rounding errors are unacceptable.

→ Example: NUMERIC, DECIMAL

Many different implementations.

→ Example: Store in an exact, variable-length binary representation with additional meta-data.

→ Can be less expensive if the DBMS does not provide arbitrary precision (e.g., decimal point can be in a different position per value).

FIXED PRECISION NUMBERS

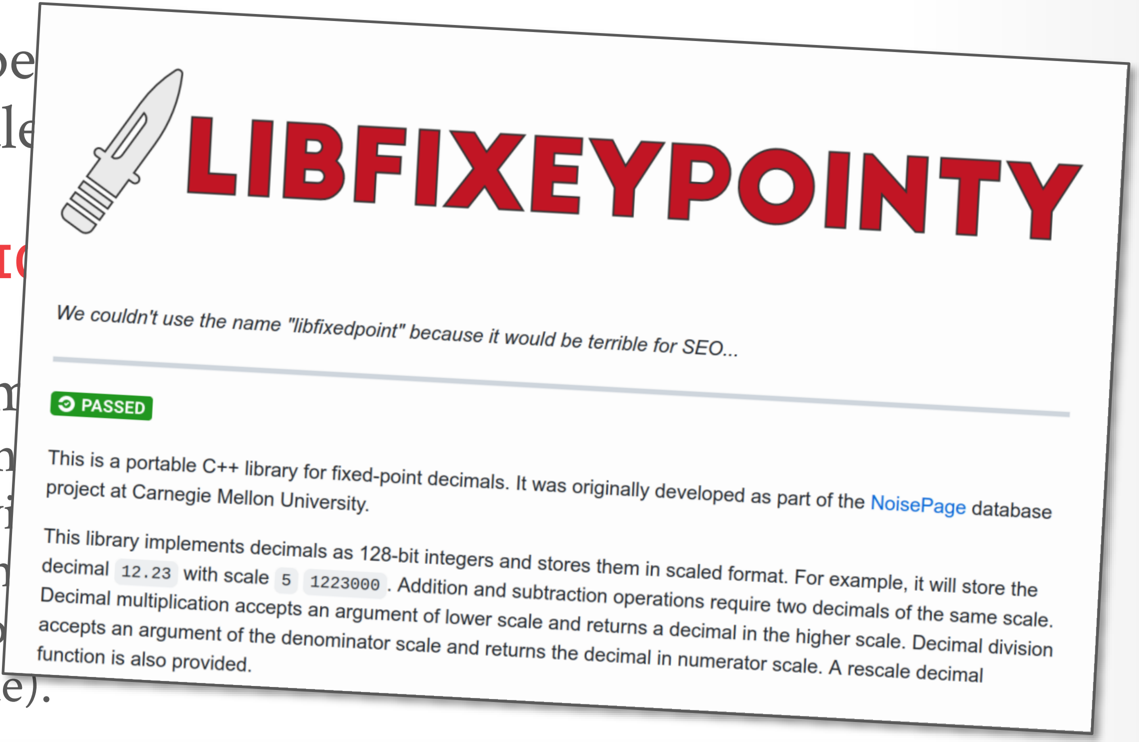
Numeric data type precision and scale unacceptable.

→ Example: **NUMERI**

Many different im

→ Example: Store in representation wi

→ Can be less expen arbitrary precisio position per value).



LIBFIXEYPOINTY

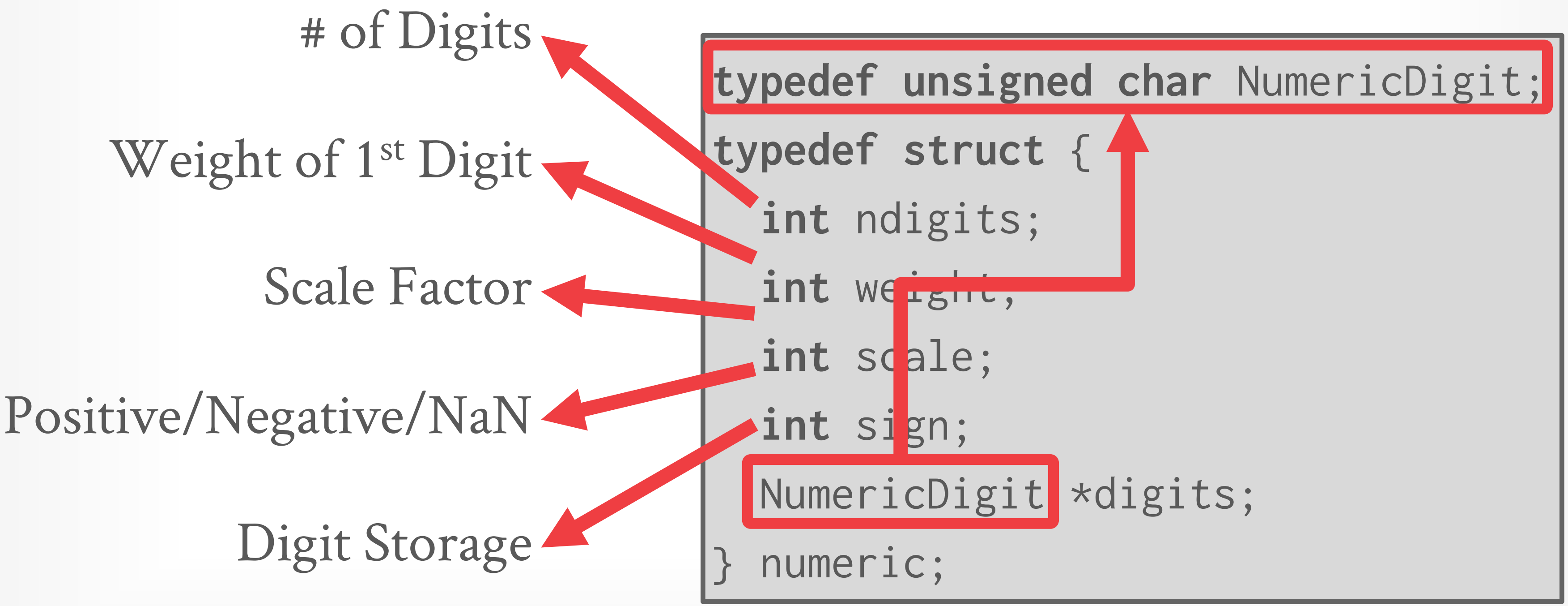
We couldn't use the name "libfixedpoint" because it would be terrible for SEO...

PASSED

This is a portable C++ library for fixed-point decimals. It was originally developed as part of the NoisePage database project at Carnegie Mellon University.

This library implements decimals as 128-bit integers and stores them in scaled format. For example, it will store the decimal `12.23` with scale `5` `1223000`. Addition and subtraction operations require two decimals of the same scale. Decimal multiplication accepts an argument of lower scale and returns a decimal in the higher scale. Decimal division accepts an argument of the denominator scale and returns the decimal in numerator scale. A rescale decimal function is also provided.

POSTGRES: NUMERIC



# of Digits

Weight of 1st Digit

Scale Factor

Positive/Negative/NaN

Digit Storage

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

c

Weight of

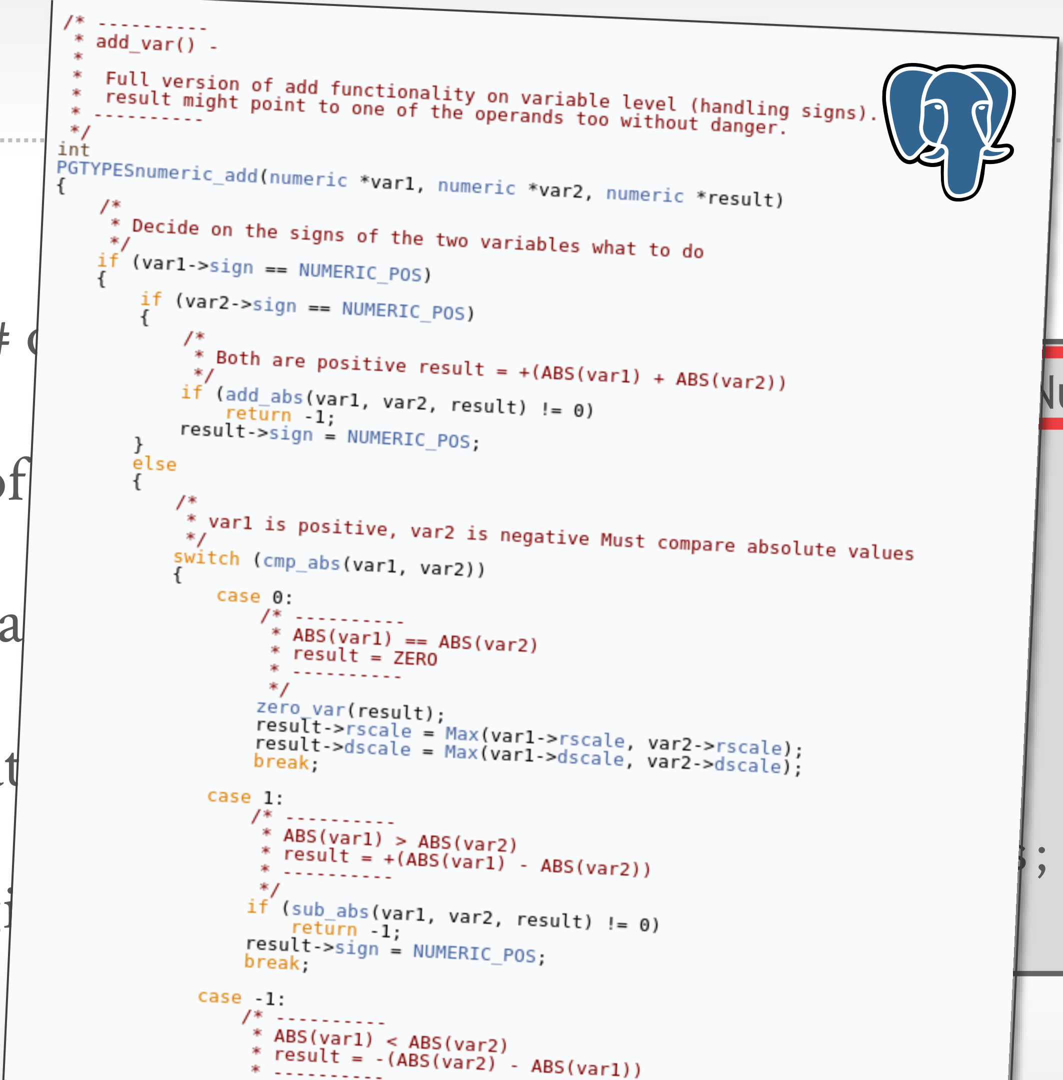
Sca

Positive/Negat

Digi

NumericDigit;

;

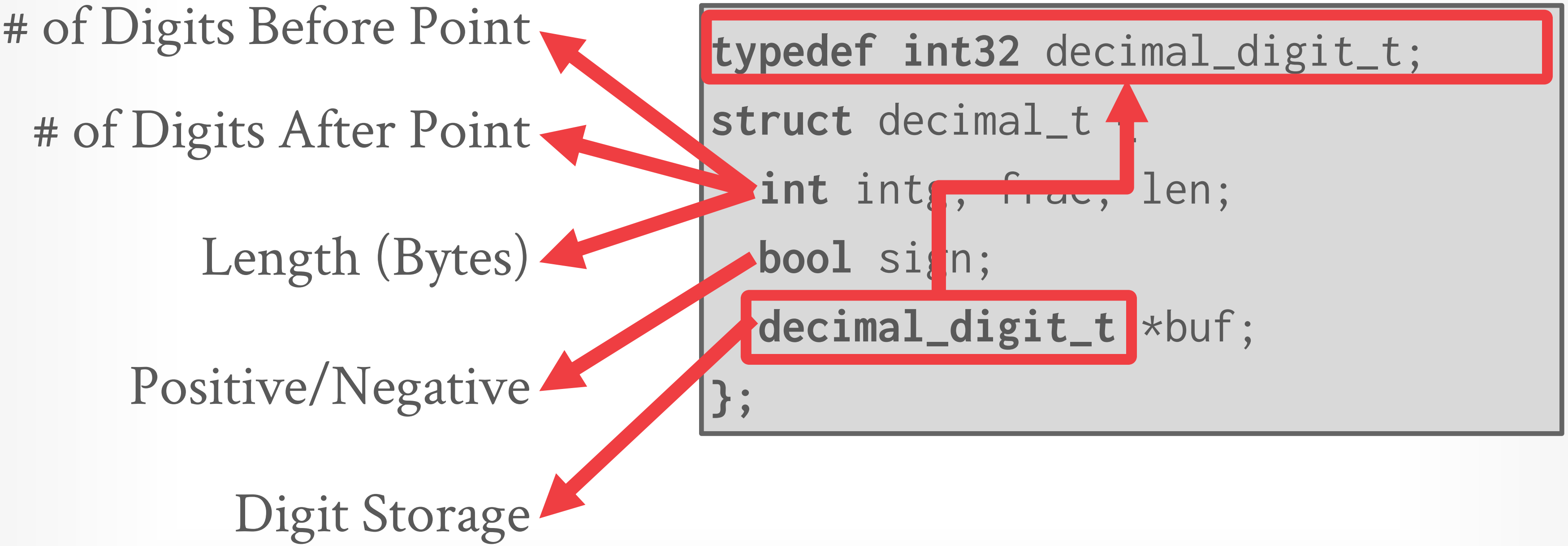


```
/* ----------
 * add_var() -
 *
 *  Full version of add functionality on variable level (handling signs).
 *  result might point to one of the operands too without danger.
 * ----------
 */
int
PGTYPESnumeric_add(numeric *var1, numeric *var2, numeric *result)
{
    /*
     * Decide on the signs of the two variables what to do
     */
    if (var1->sign == NUMERIC_POS)
    {
        if (var2->sign == NUMERIC_POS)
        {
            /*
             * Both are positive result = +(ABS(var1) + ABS(var2))
             */
            if (add_abs(var1, var2, result) != 0)
                return -1;
            result->sign = NUMERIC_POS;
        }
        else
        {
            /*
             * var1 is positive, var2 is negative Must compare absolute values
             */
            switch (cmp_abs(var1, var2))
            {
                case 0:
                    /* ----------
                     * ABS(var1) == ABS(var2)
                     * result = ZERO
                     * ----------
                     */
                    zero_var(result);
                    result->rscale = Max(var1->rscale, var2->rscale);
                    result->dscale = Max(var1->dscale, var2->dscale);
                    break;

                case 1:
                    /* ----------
                     * ABS(var1) > ABS(var2)
                     * result = +(ABS(var1) - ABS(var2))
                     * ----------
                     */
                    if (sub_abs(var1, var2, result) != 0)
                        return -1;
                    result->sign = NUMERIC_POS;
                    break;

                case -1:
                    /* ----------
                     * ABS(var1) < ABS(var2)
                     * result = -(ABS(var2) - ABS(var1))
                     * ----------
```

MYSQL: NUMERIC



# of Digits Before Point

# of Digits After Point

Length (Bytes)

Positive/Negative

Digit Storage

```
typedef int32 decimal_digit_t;
struct decimal_t {
  int intg, frac, len;
  bool sign;
  decimal_digit_t *buf;
};
```

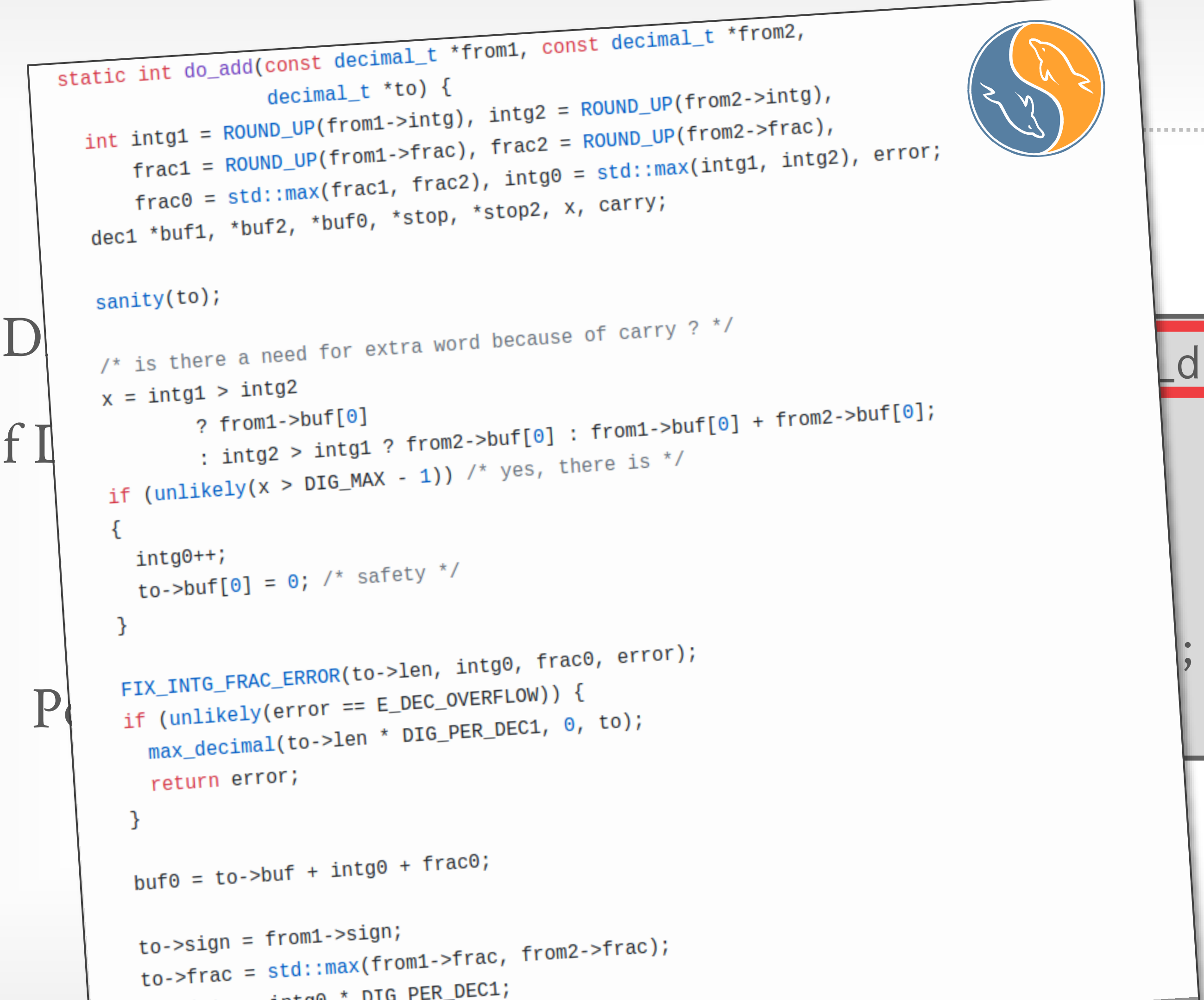
# of D

# of I

P

_digit_t;

;



```
static int do_add(const decimal_t *from1, const decimal_t *from2,
                  decimal_t *to) {
  int intg1 = ROUND_UP(from1->intg), intg2 = ROUND_UP(from2->intg),
      frac1 = ROUND_UP(from1->frac), frac2 = ROUND_UP(from2->frac),
      frac0 = std::max(frac1, frac2), intg0 = std::max(intg1, intg2), error;
  dec1 *buf1, *buf2, *buf0, *stop, *stop2, x, carry;

  sanity(to);

  /* is there a need for extra word because of carry ? */
  x = intg1 > intg2
          ? from1->buf[0]
          : intg2 > intg1 ? from2->buf[0] : from1->buf[0] + from2->buf[0];
  if (unlikely(x > DIG_MAX - 1)) /* yes, there is */
  {
    intg0++;
    to->buf[0] = 0; /* safety */
  }

  FIX_INTG_FRAC_ERROR(to->len, intg0, frac0, error);
  if (unlikely(error == E_DEC_OVERFLOW)) {
    max_decimal(to->len * DIG_PER_DEC1, 0, to);
    return error;
  }

  buf0 = to->buf + intg0 + frac0;

  to->sign = from1->sign;
  to->frac = std::max(from1->frac, from2->frac);
           intg0 * DIG_PER_DEC1;
```

NULL DATA TYPES

Choice #1: Special Values

→ Designate a value to represent **NULL** for a data type (e.g., `INT32_MIN`).

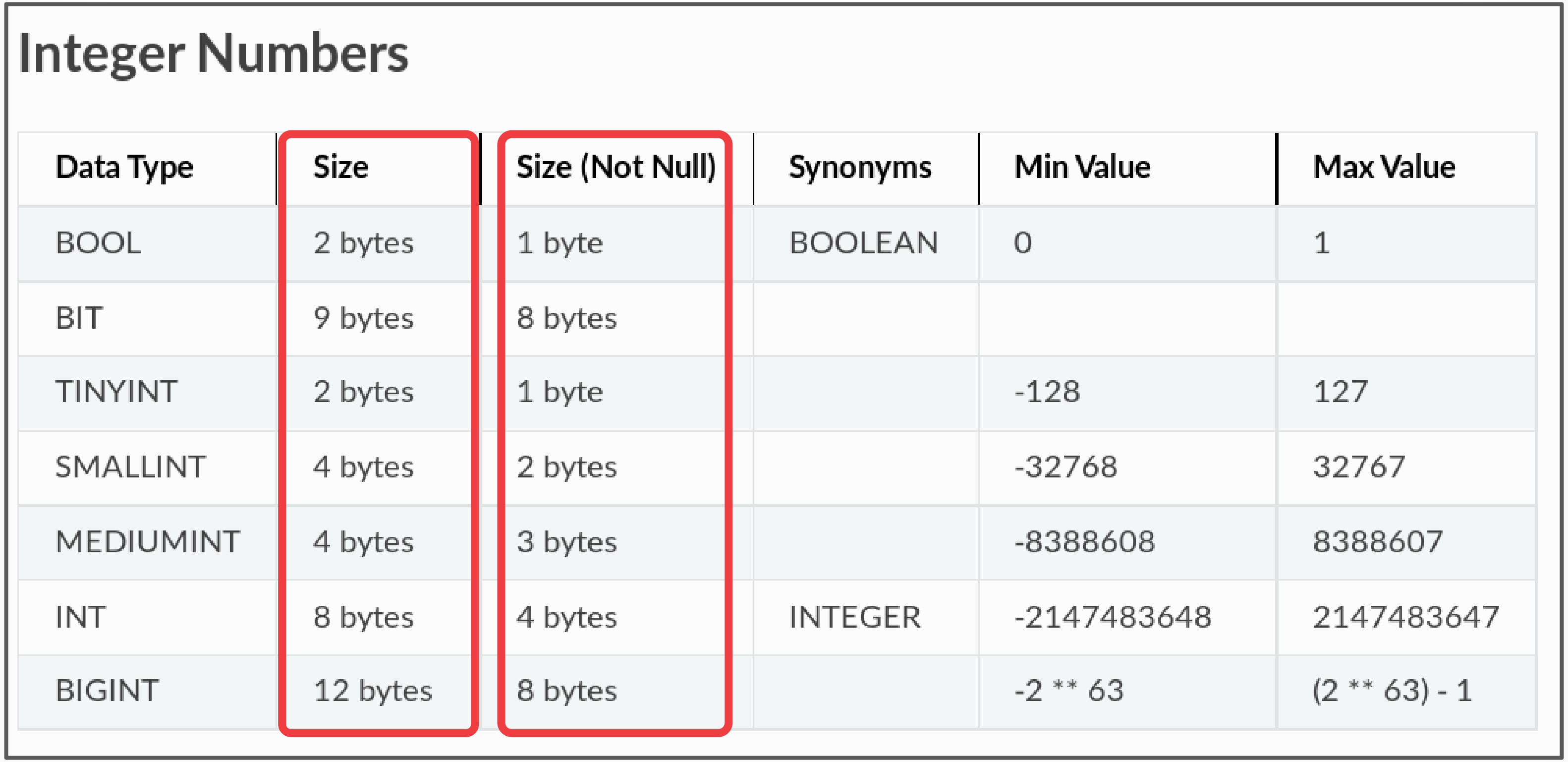
Choice #2: Null Column Bitmap Header

→ Store a bitmap in a centralized header that specifies what attributes are null.

Choice #3: Per Attribute Null Flag

→ Store a flag that marks that a value is null.
→ Must use more space than just a single bit because this messes up with word alignment.

NULL DATA TYPES



Integer Numbers

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

OBSERVATION

Data is "hot" when it enters the 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.

HYBRID STORAGE MODEL

Use separate execution engines that are optimized for either NSM or DSM databases.

→ Store new data in NSM for fast OLTP.
→ Migrate data to DSM for more efficient OLAP.
→ Combine query results from both engines to appear as a single logical database to the application.

Choice #1: Fractured Mirrors

→ Examples: Oracle, IBM DB2 Blu, Microsoft SQL Server

Choice #2: Delta Store

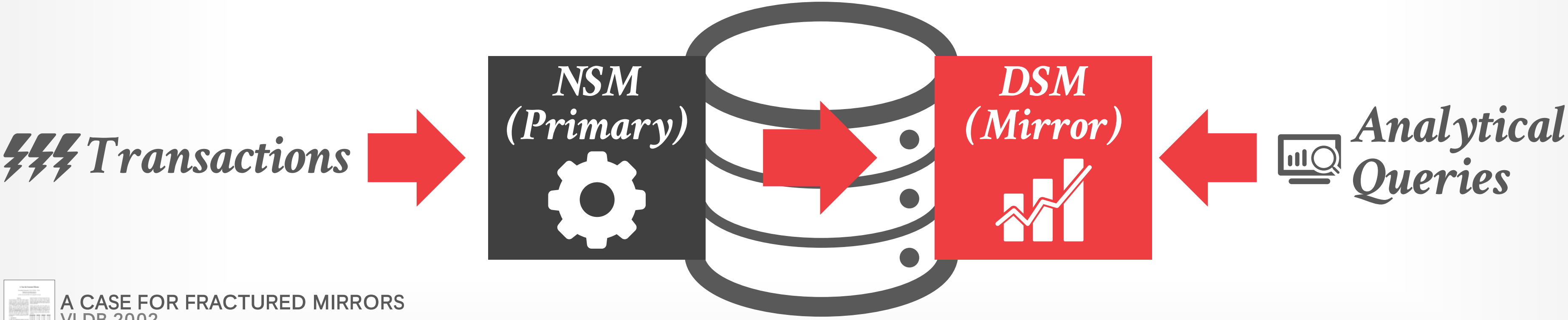
→ Examples: SAP HANA, Vertica, SingleStore, Databricks, Google Napa

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.

→ If the DBMS supports updates, it must invalidate tuples in the DSM mirror.



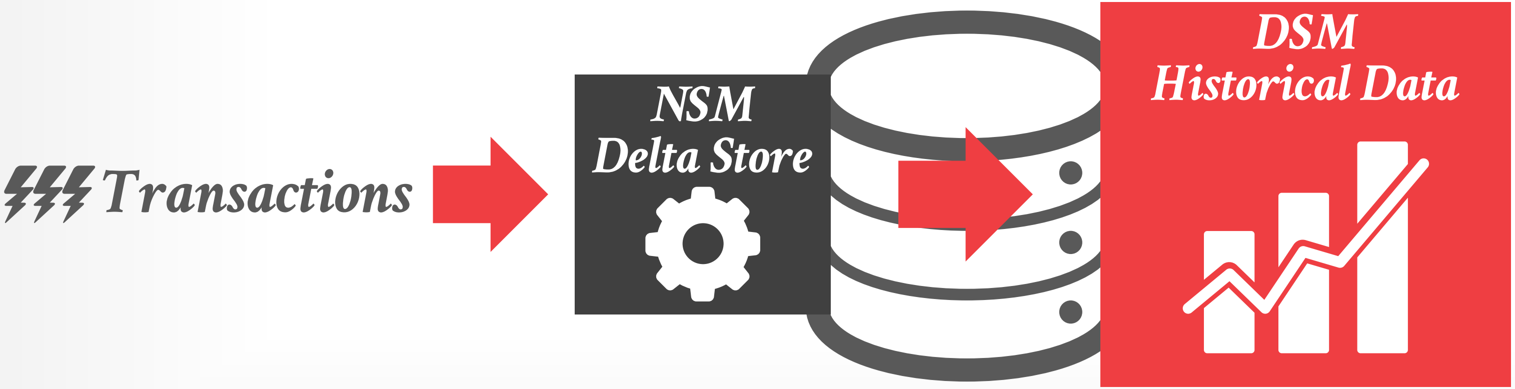
A CASE FOR FRACTURED MIRRORS
VLDB 2002

DELTA STORE

Stage updates to the database in an NSM table.

A background thread migrates updates from delta store and applies them to DSM data.

→ Batch large chunks and then write them out as a PAX file.



DATABASE PARTITIONING

Split database across multiple resources:

- → Disks, nodes, processors.
- → Often called "sharding" in NoSQL systems.

The DBMS executes query fragments on each partition and then combines the results to produce a single answer.

The DBMS can partition a database **physically** (shared nothing) or **logically** (shared disk).

HORIZONTAL PARTITIONING

Split a table's tuples into disjoint subsets based on some partitioning key and scheme.

→ Choose column(s) that divides the database equally in terms of size, load, or usage.

Partitioning Schemes:

→ Hashing
→ Ranges
→ Predicates

HORIZONTAL PARTITIONING

Partitioning Key

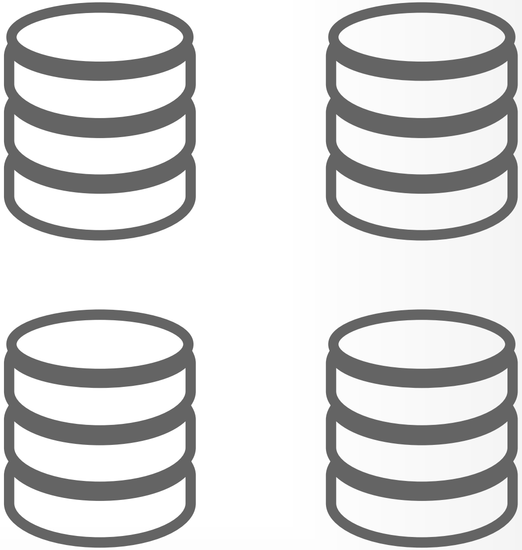
Table1

101	a	XXX	2022-11-29	*hash(a)%4 = P2*
102	b	XXY	2022-11-28	*hash(b)%4 = P4*
103	c	XYZ	2022-11-29	*hash(c)%4 = P3*
104	d	XYX	2022-11-27	*hash(d)%4 = P2*
105	e	XYY	2022-11-29	*hash(e)%4 = P1*

Ideal Query:

```
SELECT * FROM table
 WHERE partitionKey = ?
```

Partitions



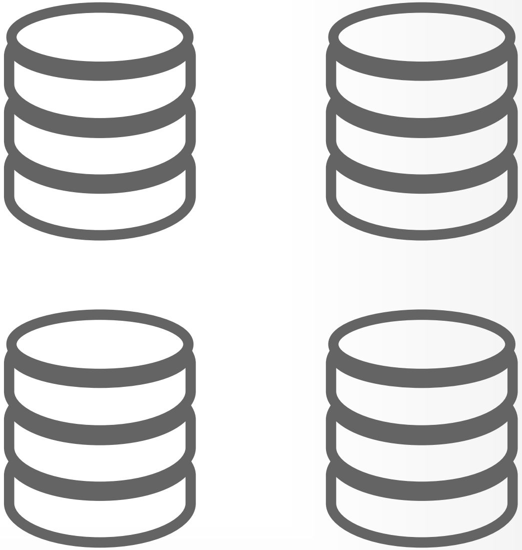
HORIZONTAL PARTITIONING

Partitioning Key

Table1

101	a	XXX	2022-11-29	*hash(a)%4 = P2*
102	b	XXY	2022-11-28	*hash(b)%4 = P4*
103	c	XYZ	2022-11-29	*hash(c)%4 = P3*
104	d	XYX	2022-11-27	*hash(d)%4 = P2*
105	e	XYY	2022-11-29	*hash(e)%4 = P1*

Partitions



Ideal Query:

```
SELECT * FROM table
 WHERE partitionKey = ?
```

HORIZONTAL PARTITIONING

Partitioning Key

Table1

101	a	XXX	2022-11-29
102	b	XXY	2022-11-28
103	c	XYZ	2022-11-29
104	d	XYX	2022-11-27
105	e	XYY	2022-11-29

hash(a)%4 = P2

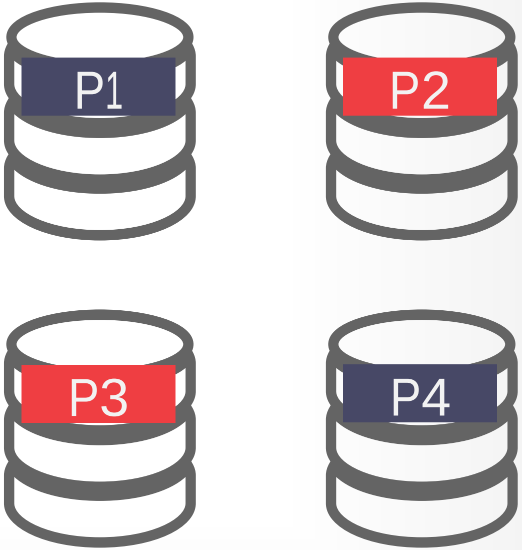
hash(b)%4 = P4

hash(c)%4 = P3

hash(d)%4 = P2

hash(e)%4 = P1

Partitions



Ideal Query:

```
SELECT * FROM table
 WHERE partitionKey = ?
```

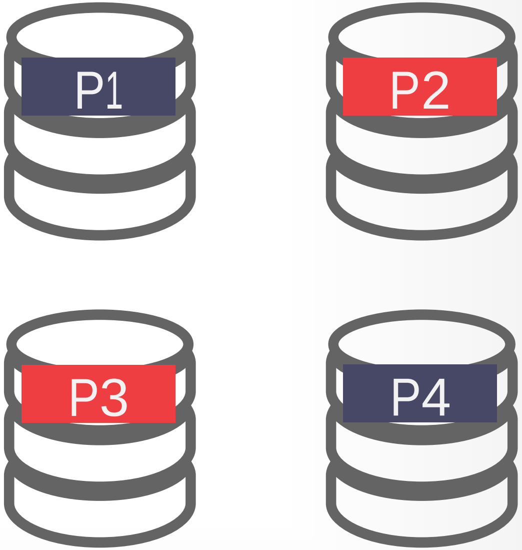
HORIZONTAL PARTITIONING

Partitioning Key

Table1

101	a	XXX	2022-11-29	*hash(a)%4 = P2*
102	b	XXY	2022-11-28	*hash(b)%4 = P4*
103	c	XYZ	2022-11-29	*hash(c)%4 = P3*
104	d	XYX	2022-11-27	*hash(d)%4 = P2*
105	e	XYY	2022-11-29	*hash(e)%4 = P1*

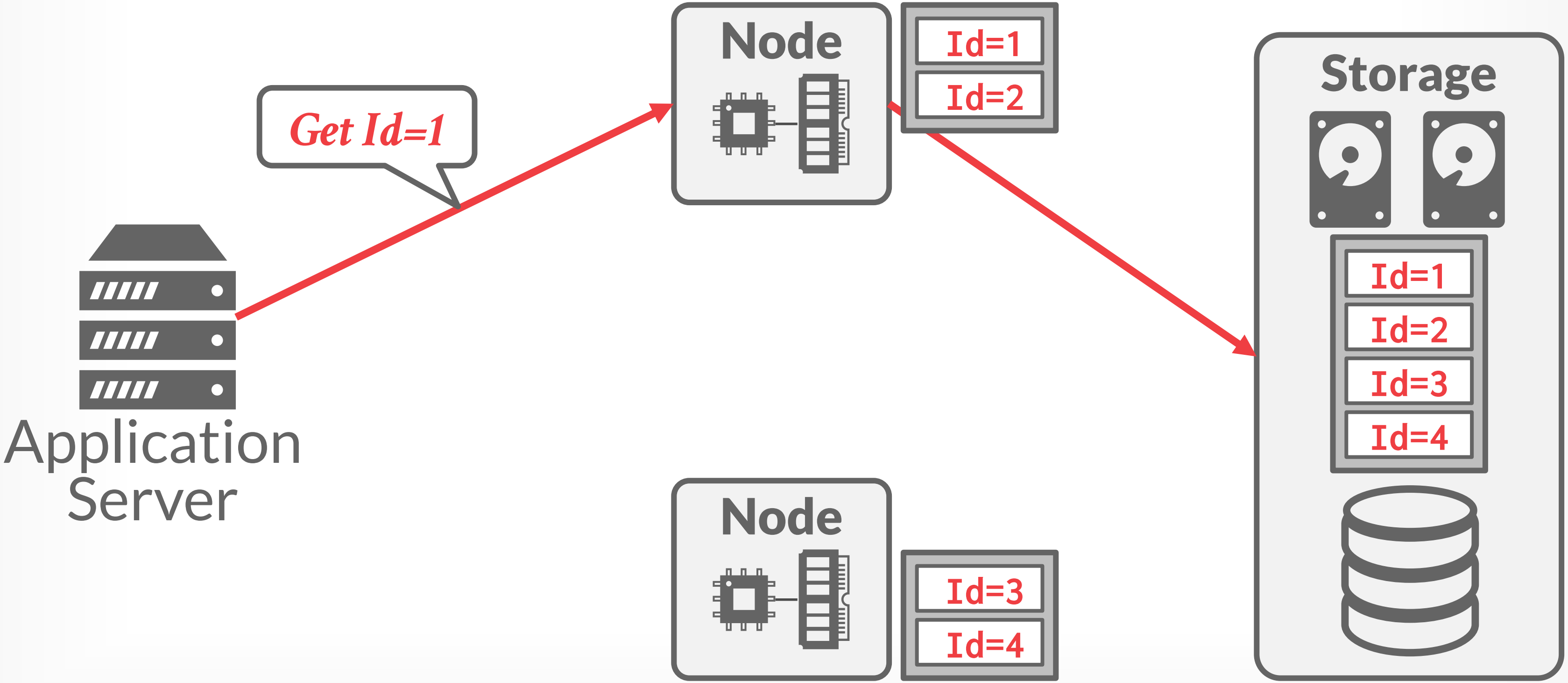
Partitions



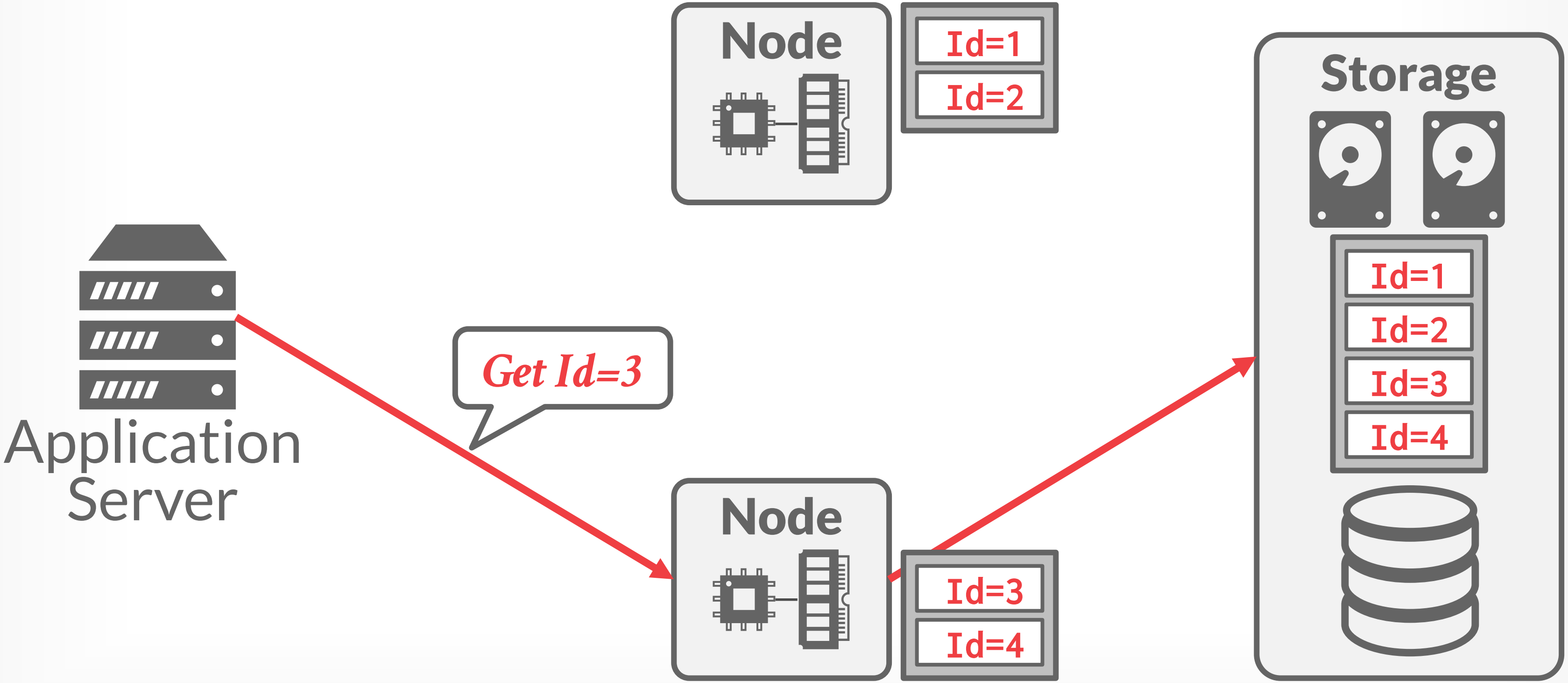
Ideal Query:

```
SELECT * FROM table
 WHERE partitionKey = ?
```

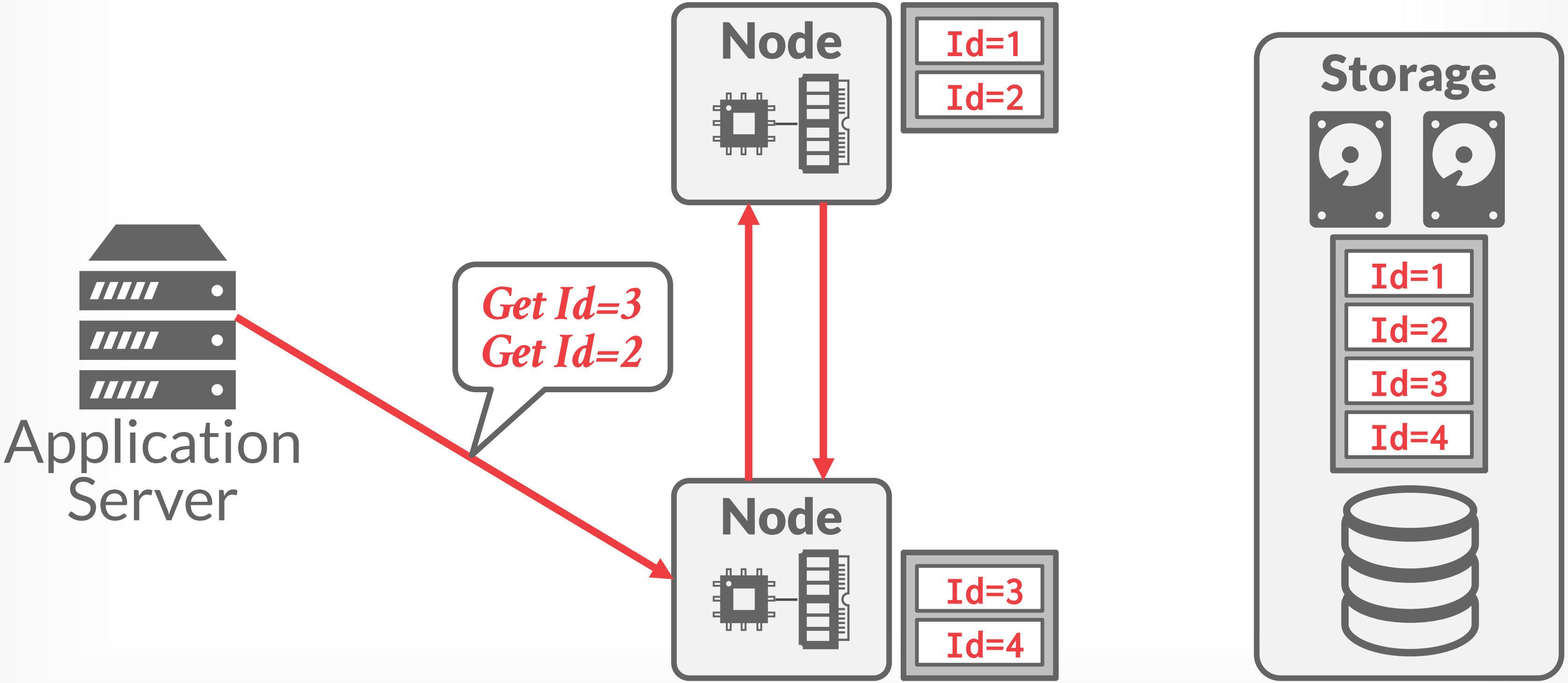
LOGICAL PARTITIONING



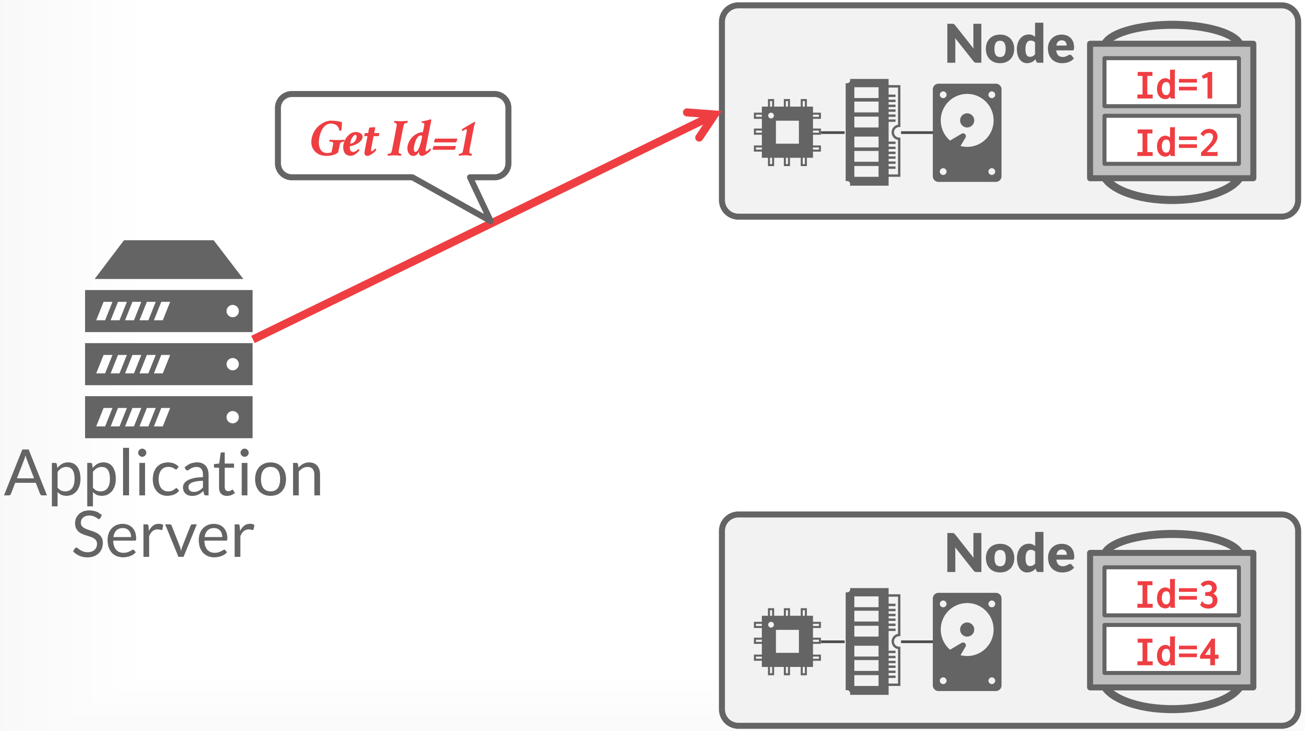
LOGICAL PARTITIONING



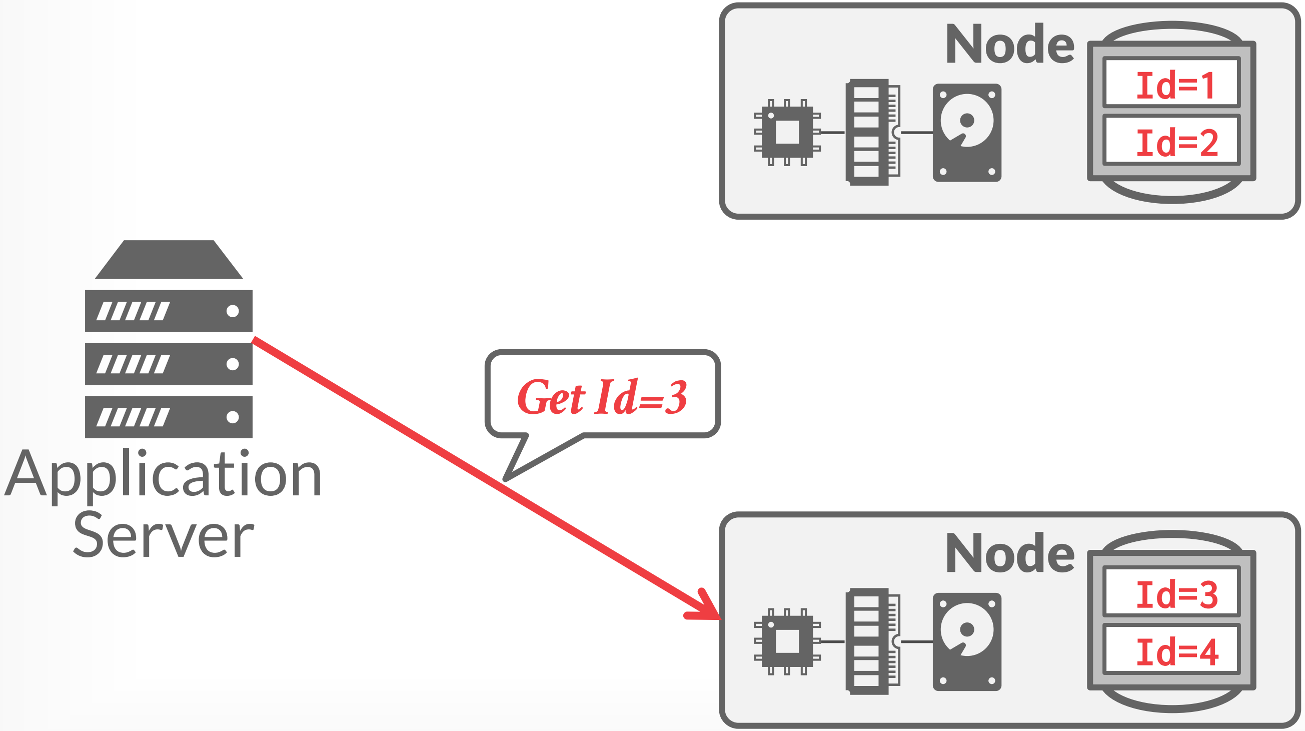
LOGICAL PARTITIONING



PHYSICAL PARTITIONING



PHYSICAL PARTITIONING



PARTING THOUGHTS

Every modern OLAP system is using some variant of PAX storage. The key idea is that all data must be **fixed-length**.

Real-world tables contain mostly numeric attributes (int/float), but their occupied storage is mostly comprised of string data.

Modern columnar systems are so fast that most people do not denormalize data warehouse schemas.

NEXT CLASS

How to accelerate OLAP queries on columnar data with auxiliary data structures.

→ Zone Maps
→ Bitmap Indexes
→ Sketches

We will also discuss Project #1.