02: Data Storage

Andrew Crotty // CS497 // Fall 2023

Observation

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

What data "looks" like determines almost the entire system architecture of a DBMS.
→ 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 attributes for a single tuple contiguously in a page.

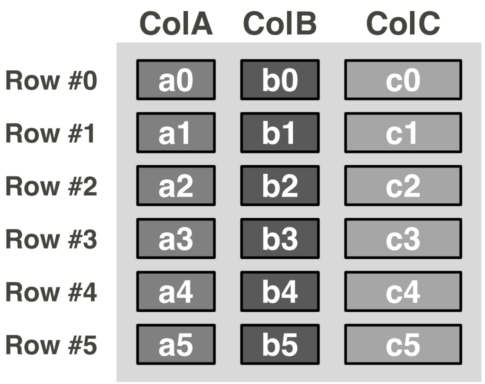
Ideal for insert-heavy and OLTP workloads, where txns tend to access individual entities.
→ Use the tuple-at-a-time ***iterator processing model***.

NSM DBMS page sizes are typically some constant multiple of **4 KB** hardware pages.
→ Examples: Oracle (4 KB), Postgres (8 KB), MySQL (16 KB)

NSM: Physical Organization

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

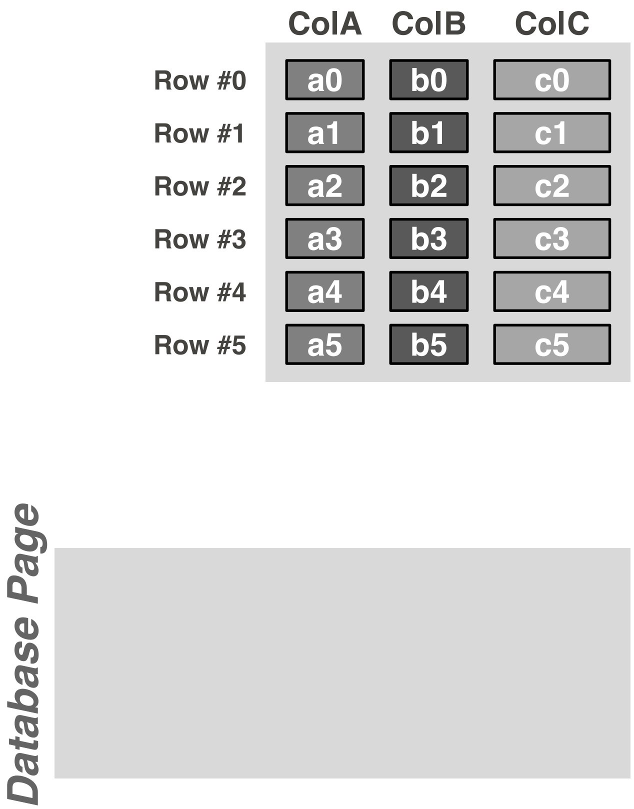
The tuple's **record id** (page#,slot#) is how the DBMS uniquely identifies physical tuples.



NSM: Physical Organization

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

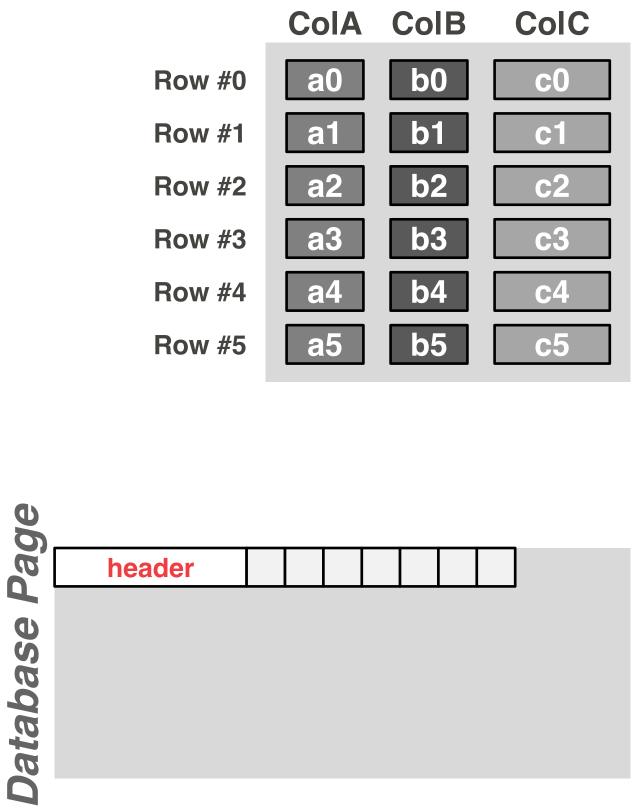
The tuple's **record id** (page#,slot#) is how the DBMS uniquely identifies physical tuples.



NSM: Physical Organization

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

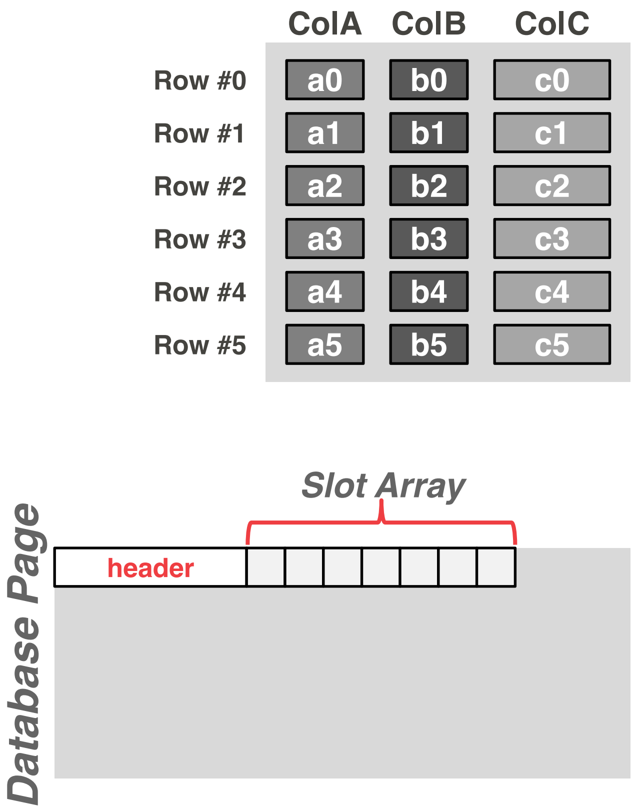
The tuple's **record id** (page#,slot#) is how the DBMS uniquely identifies physical tuples.



NSM: Physical Organization

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

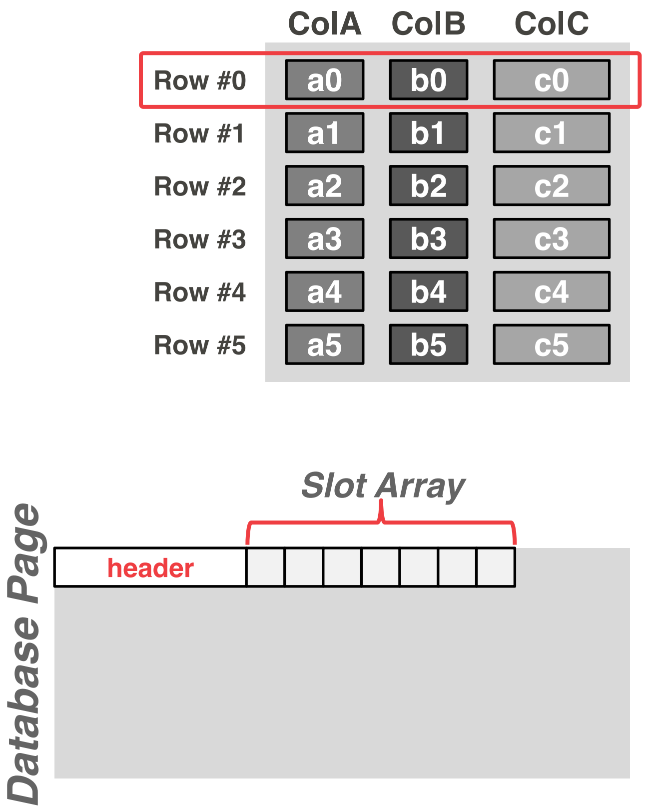
The tuple's **record id** (page#,slot#) is how the DBMS uniquely identifies physical tuples.



NSM: Physical Organization

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

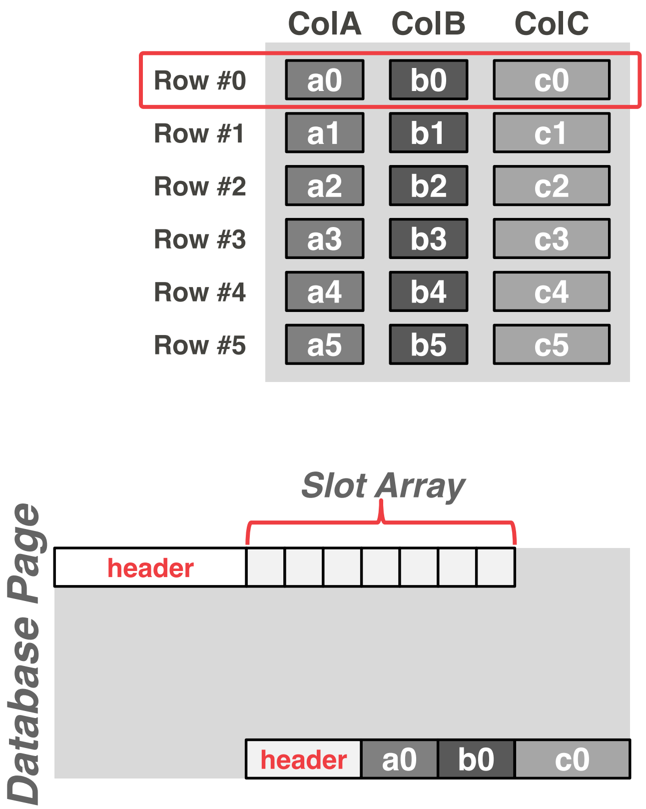
The tuple's **record id** (page#,slot#) is how the DBMS uniquely identifies physical tuples.



NSM: Physical Organization

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

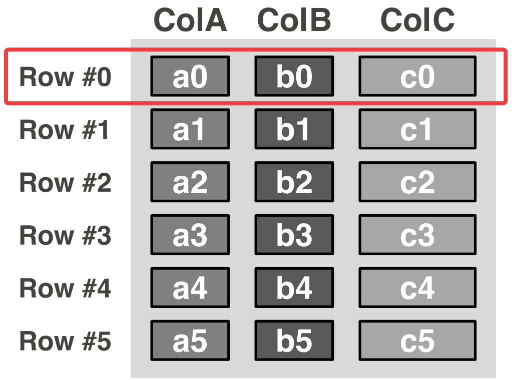
The tuple's **record id** (page#,slot#) is how the DBMS uniquely identifies physical tuples.

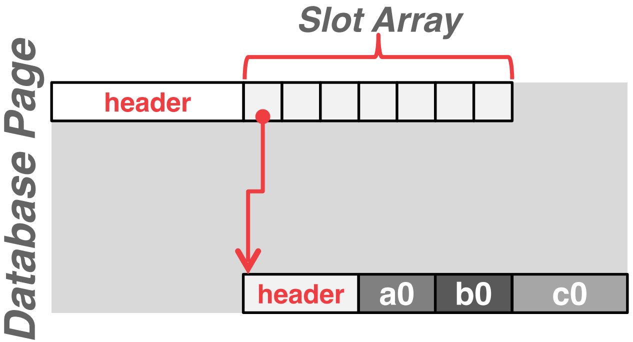


NSM: Physical Organization

A disk-based NSM DBMS 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 physical tuples.

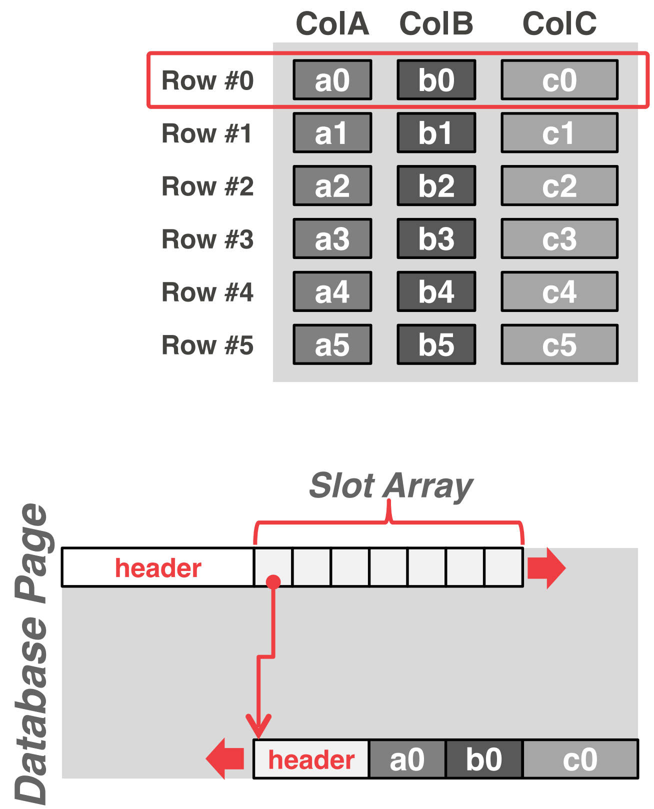




NSM: Physical Organization

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

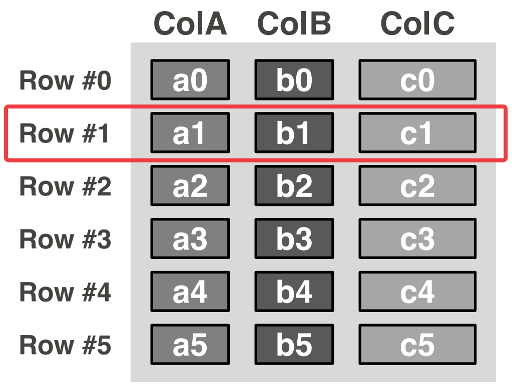
The tuple's **record id** (page#,slot#) is how the DBMS uniquely identifies physical tuples.

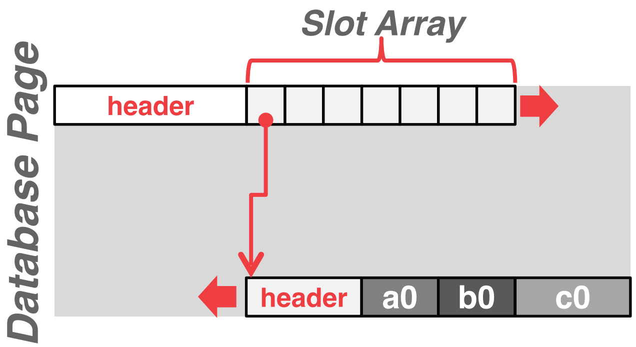


NSM: Physical Organization

A disk-based NSM DBMS 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 physical tuples.

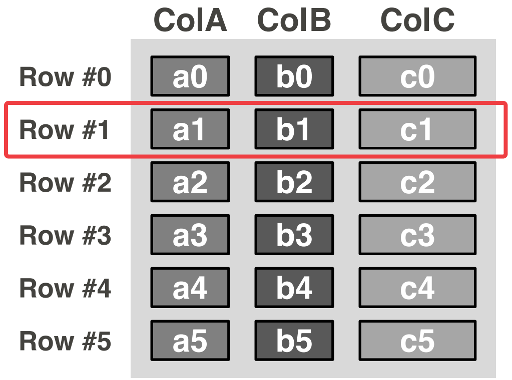


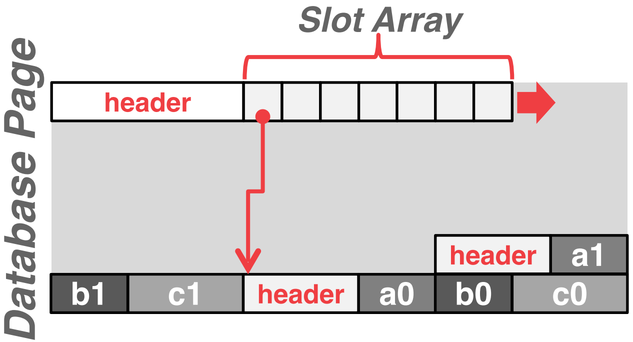


NSM: Physical Organization

A disk-based NSM DBMS 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 physical tuples.

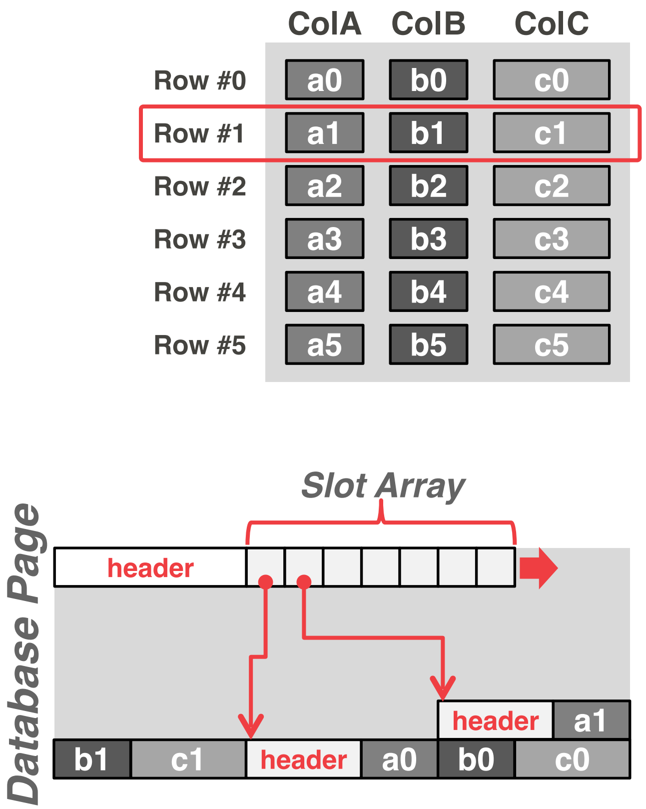




NSM: Physical Organization

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

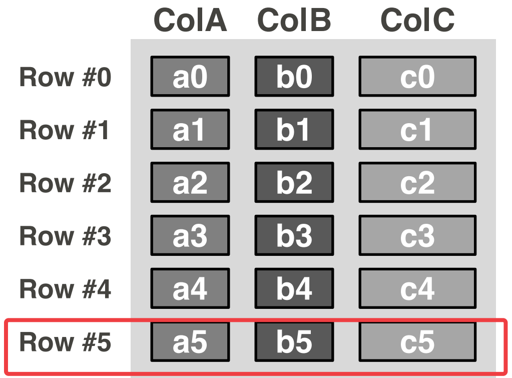
The tuple's **record id** (page#,slot#) is how the DBMS uniquely identifies physical tuples.

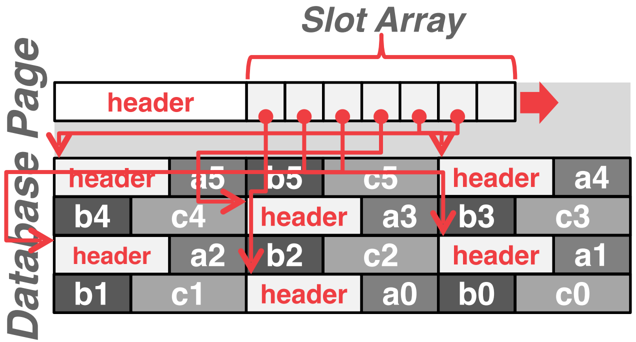


NSM: Physical Organization

A disk-based NSM DBMS 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 physical tuples.





NSM: Physical Organization

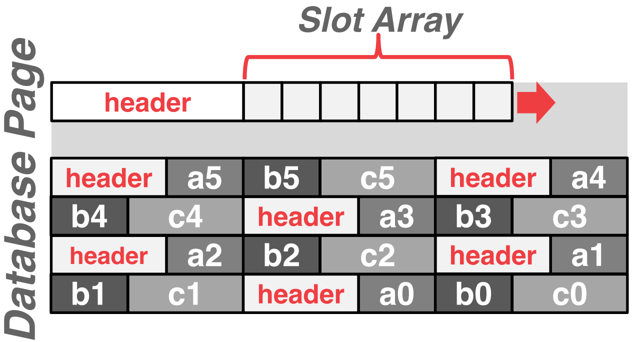
A disk-based NSM DBMS stores a tuple's fixed-length and variable-length attributes contiguously in a si

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

Th

(page#,slot#) is how the DBMS uniquely identifies physical tuples.

	ColA	ColB	ColC
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



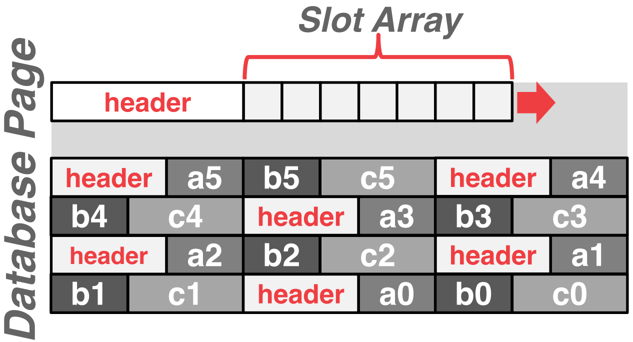
NSM: Physical Organization

A disk-based NSM DBMS stores a tuple's fixed-length and variable-length attributes contiguously in a si

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

Th **record id**
(page#,slot#) is how the DBMS uniquely identifies physical tuples.

	ColA	ColB	ColC
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



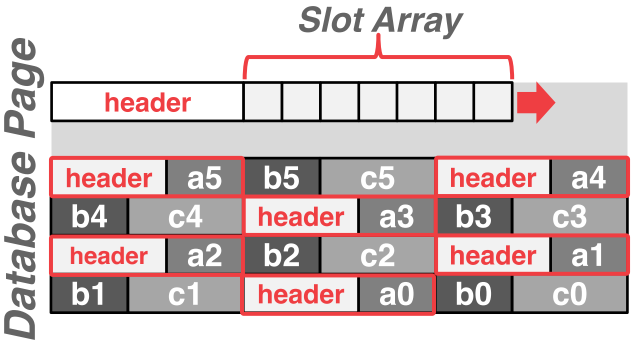
NSM: Physical Organization

A disk-based NSM DBMS stores a tuple's fixed-length and variable-length attributes contiguously in a si

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

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

	ColA	ColB	ColC
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

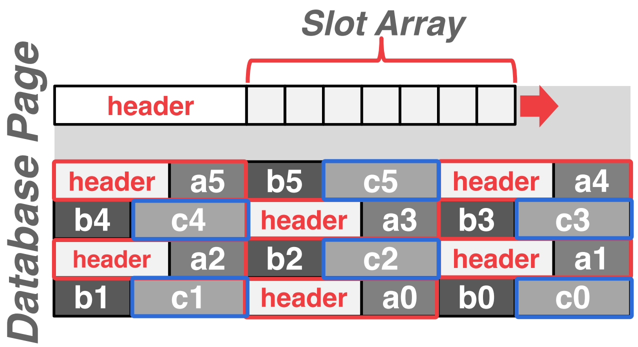
A disk-based NSM DBMS stores a tuple's fixed-length and variable-length attributes contiguously in a si

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

Th

(page#,slot#) is how the DBMS uniquely identifies physical tuples.

	ColA	ColB	ColC
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 indexes 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.

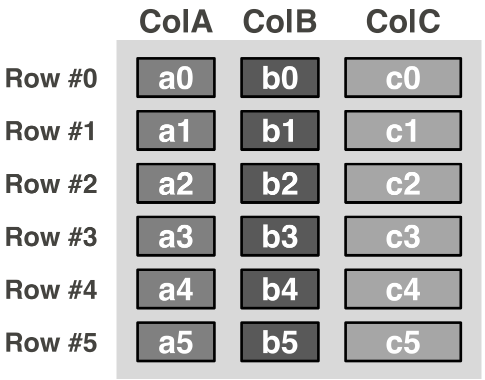
Ideal for OLAP workloads where read-only queries perform large scans over a subset of the table's attributes.
→ Use a batched ***vectorized processing model***.

File sizes are larger (100s of MBs), but tuples might still be organized 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…

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

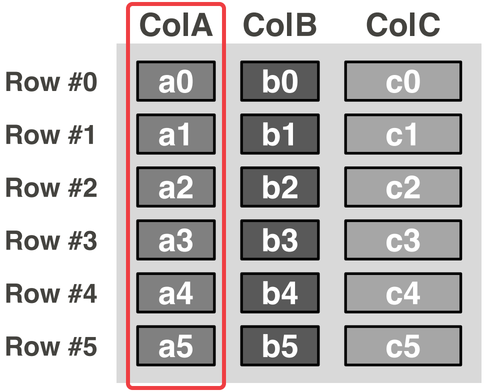


DSM: Physical Organization

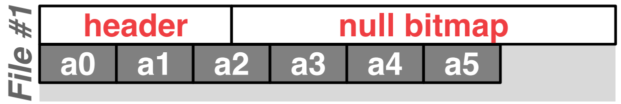
Store attributes and meta-data (e.g., nulls) in separate arrays of **fixed-length** values.

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

→ Need to handle variable-length values…



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

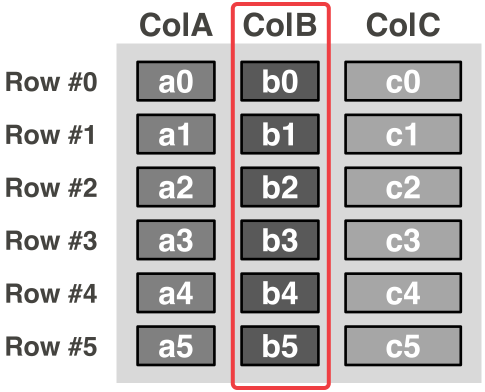


DSM: Physical Organization

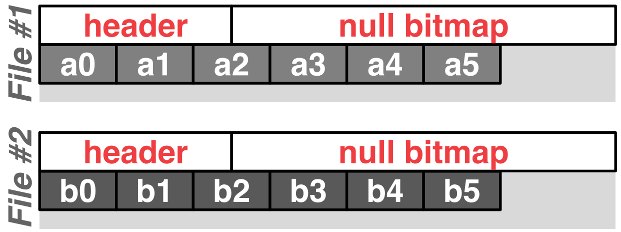
Store attributes and meta-data (e.g., nulls) in separate arrays of **fixed-length** values.

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

→ Need to handle variable-length values…



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

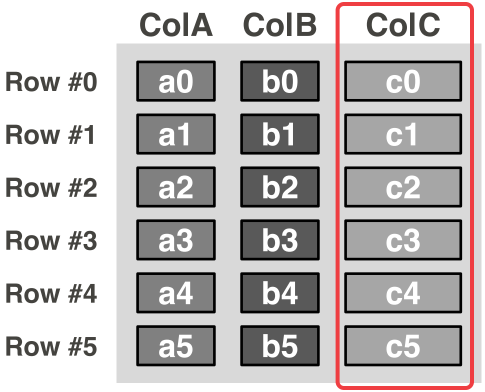


DSM: Physical Organization

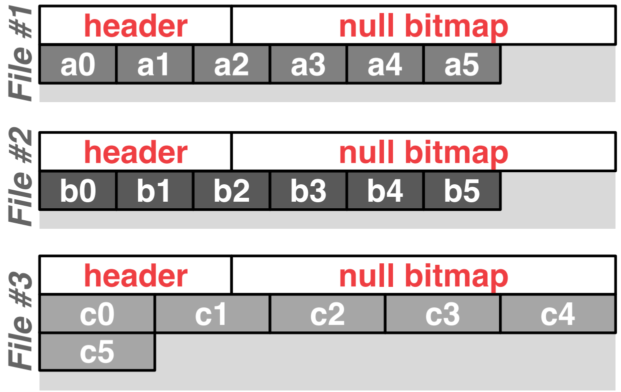
Store attributes and meta-data (e.g., nulls) in separate arrays of **fixed-length** values.

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

→ Need to handle variable-length values…



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



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 class.

DSM: System History

1970s: Cantor DBMS
1980s: DSM Proposal
1990s: SybaseIQ (in-memory only)
2000s: Vertica, Vectorwise, MonetDB
2010s: Everyone

DSM: System History

1970s: Cantor DBMS

1980s: DSM Proposal

1990s: SybaseIQ (in-memory only)

2000s: Vertica, Vectorwise, MonetDB

2010s: Everyone

DSM: System History

1970s: Cantor DBMS

1980s: DSM Proposal

1990s: SybaseIQ (in-memory only)

2000s: Vertica, Vectorwise, MonetDB

2010s: Everyone

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 of 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 next class).

Disadvantages

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

Observation

Observation

OLAP queries almost never access a single column in a table by itself.
→ At some point, the query 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 a columnar scheme that still stores attributes separately but keeps the data for each tuple physically close together…

PAX Storage Model

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

→ This is what Parquet and ORC use.

The goal is to get the benefit of <u>faster processing</u> on columns while retaining the <u>spatial locality</u> benefits of rows.

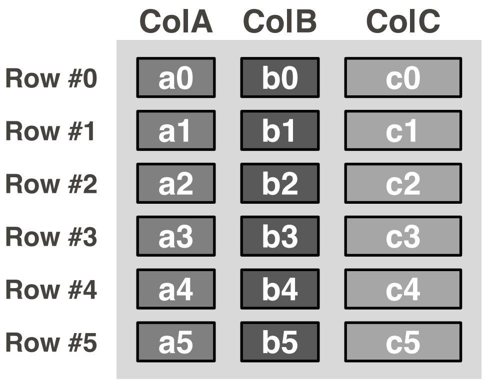
DATA PAGE LAYOUTS FOR RELATIONAL DATABASES ON DEEP MEMORY HIERARCHIES
VLDB JOURNAL 2002

PAX: Physical Organization

Horizontally partition rows into groups, then vertically partition their attributes into columns.

Global header contains directory with 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.

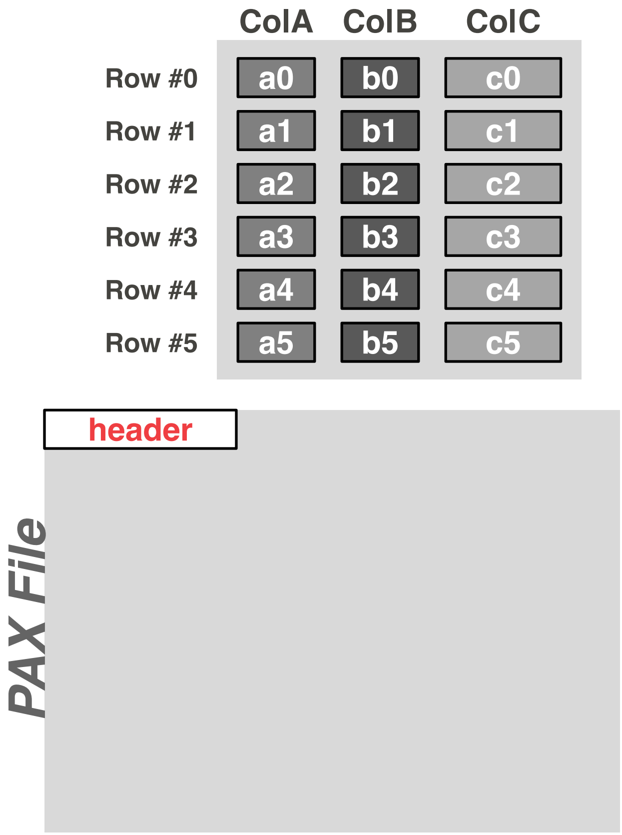


PAX: Physical Organization

Horizontally partition rows into groups, then vertically partition their attributes into columns.

Global header contains directory with 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.

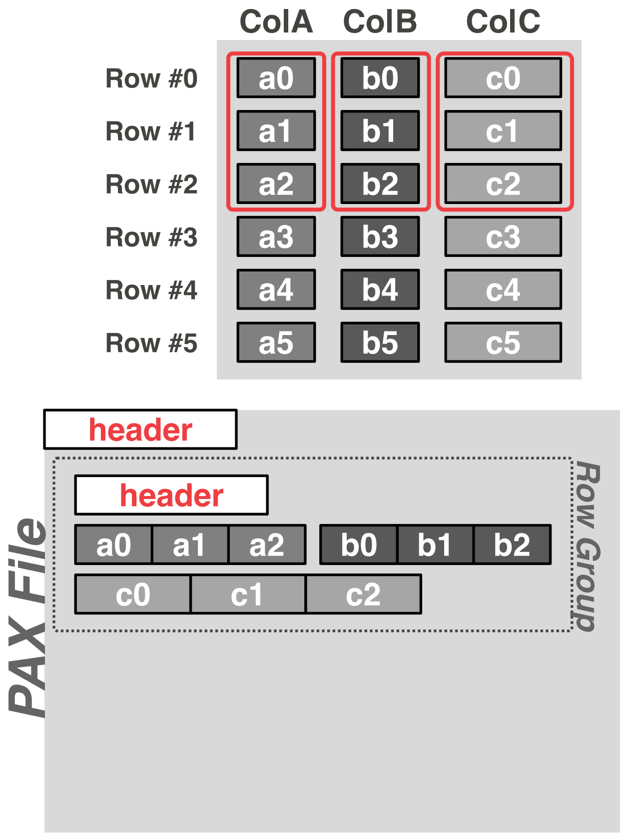


PAX: Physical Organization

Horizontally partition rows into groups, then vertically partition their attributes into columns.

Global header contains directory with 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.

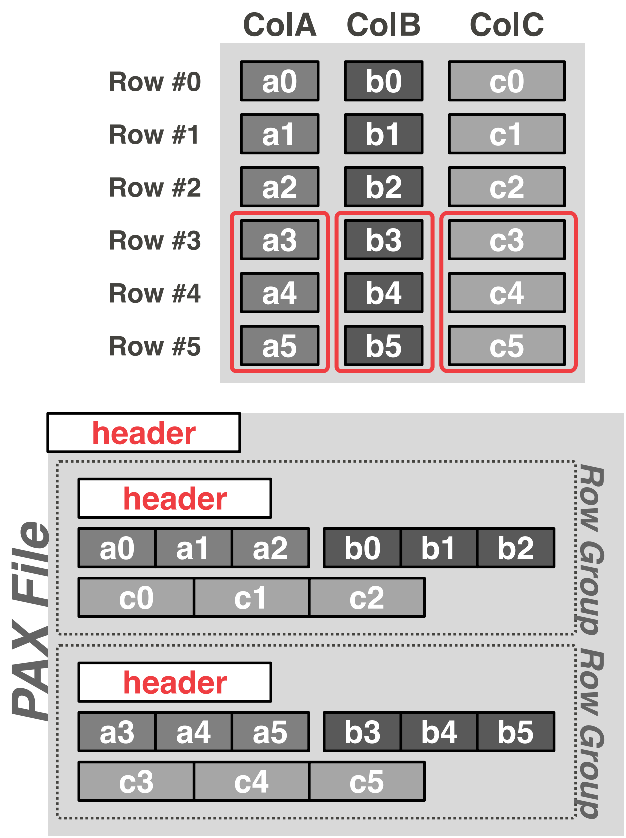


PAX: Physical Organization

Horizontally partition rows into groups, then vertically partition their attributes into columns.

Global header contains directory with 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.

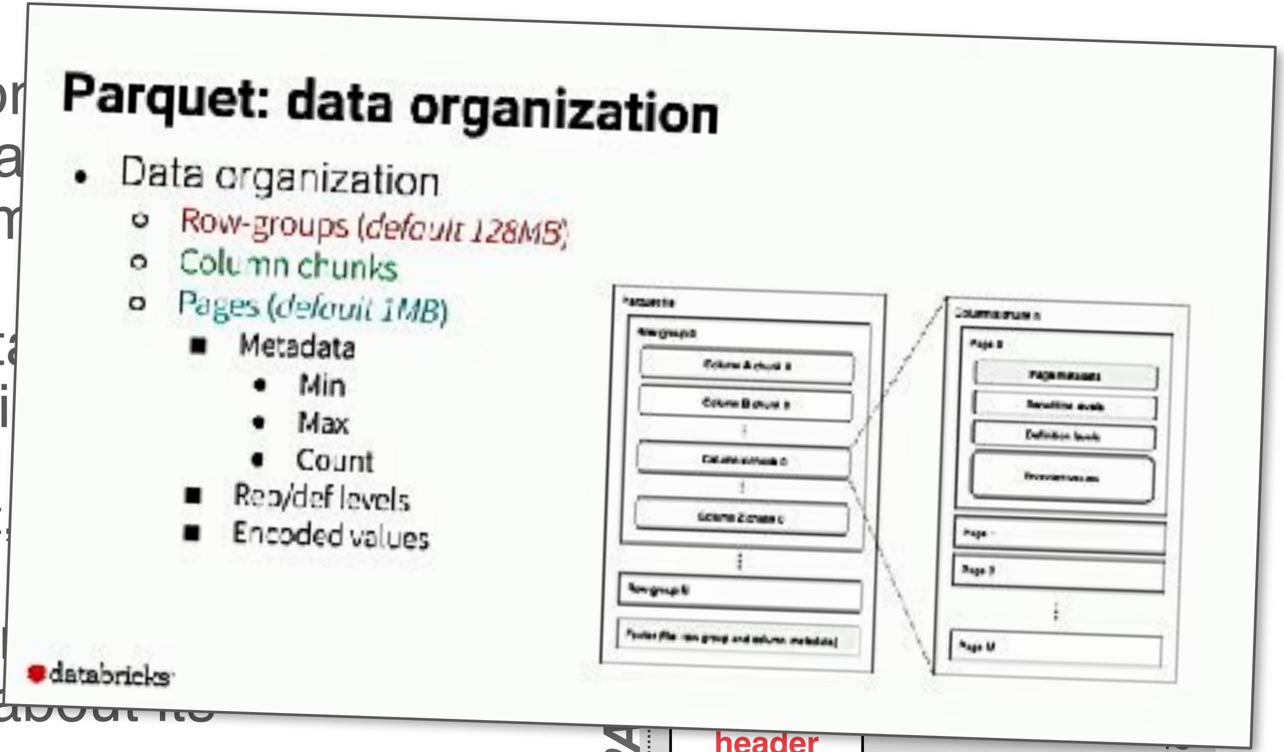


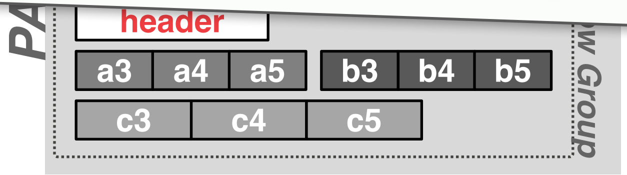
PAX: Physical Organization

Horizontally partition
groups, then vertica
attributes into colum

Global header conta
with offsets to the fi
→ This is stored in the
immutable (Parquet

Each row group co
meta-data header a
contents.





Paging

A DBMS uses a buffer pool, but why not use the OS instead?

With memory-mapped files, the OS is responsible for transparent paging of the file so the DBMS doesn't need to worry about it.

Is this a good idea?

Paging

A DBMS uses a buffer pool, but why not use the OS instead?

With memory-mapped files, the OS is responsible for transparent paging of the file so the DBMS doesn't need to worry about it.

Is this a good idea?

page1	page2	page3	page4

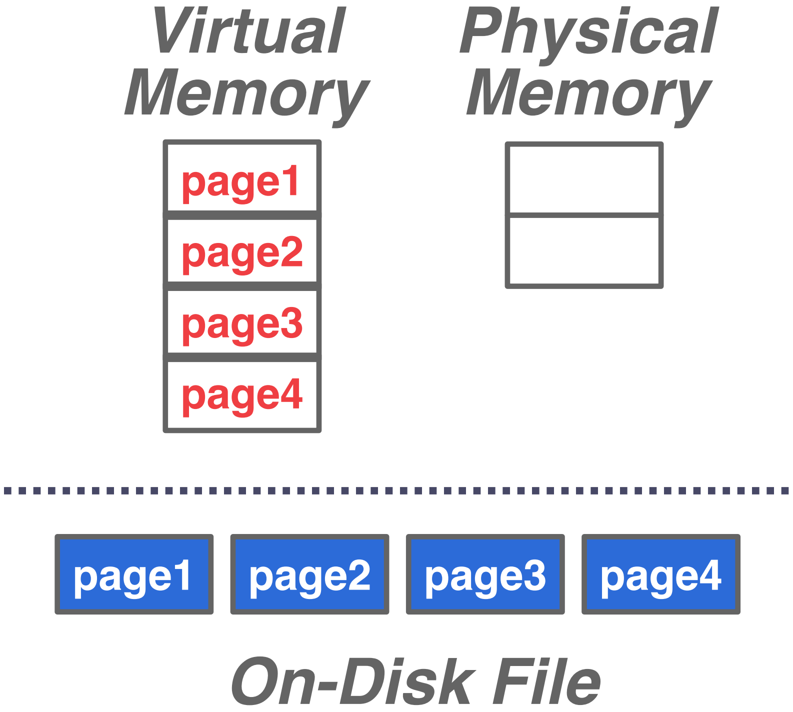
On-Disk File

Paging

A DBMS uses a buffer pool, but why not use the OS instead?

With memory-mapped files, the OS is responsible for transparent paging of the file so the DBMS doesn't need to worry about it.

Is this a good idea?

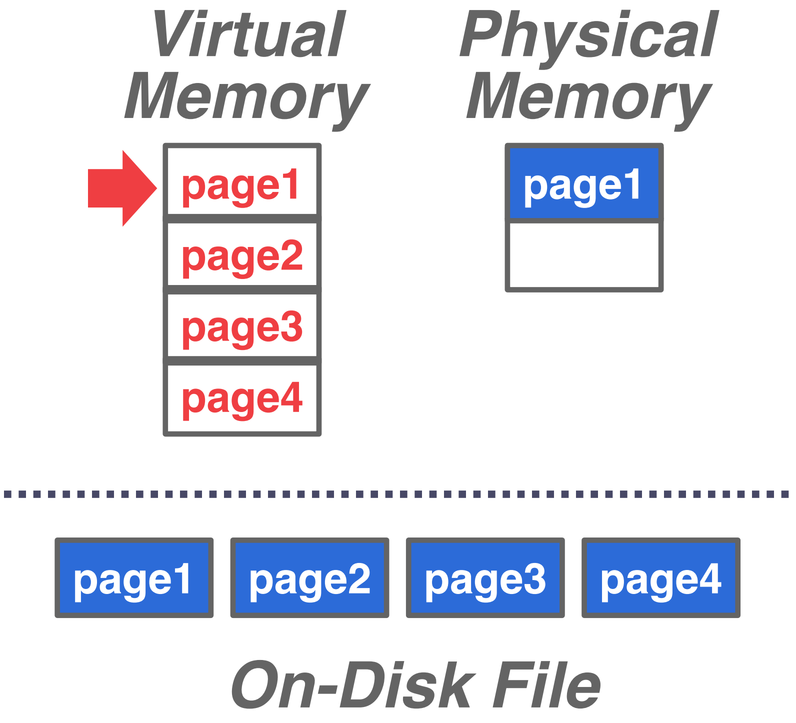


Paging

A DBMS uses a buffer pool, but why not use the OS instead?

With memory-mapped files, the OS is responsible for transparent paging of the file so the DBMS doesn't need to worry about it.

Is this a good idea?

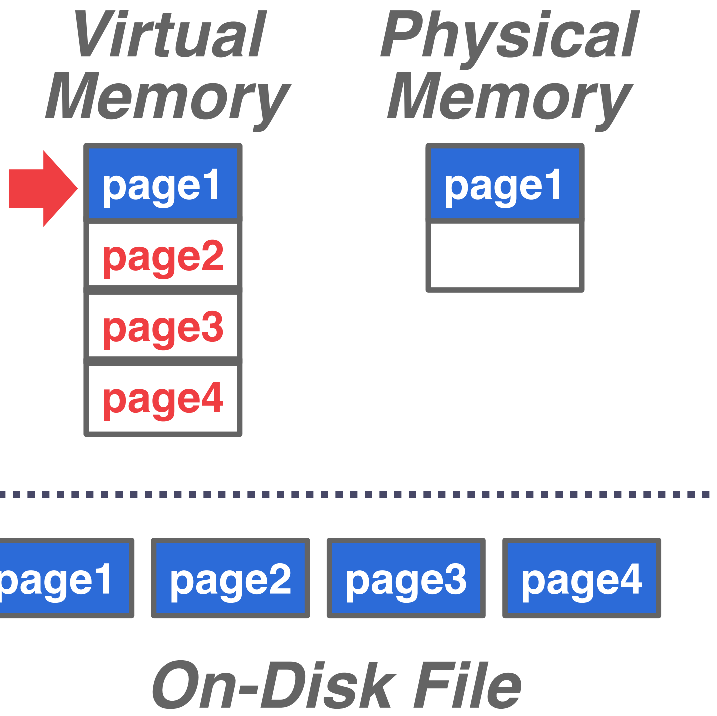


Paging

A DBMS uses a buffer pool, but why not use the OS instead?

With memory-mapped files, the OS is responsible for transparent paging of the file so the DBMS doesn't need to worry about it.

Is this a good idea?

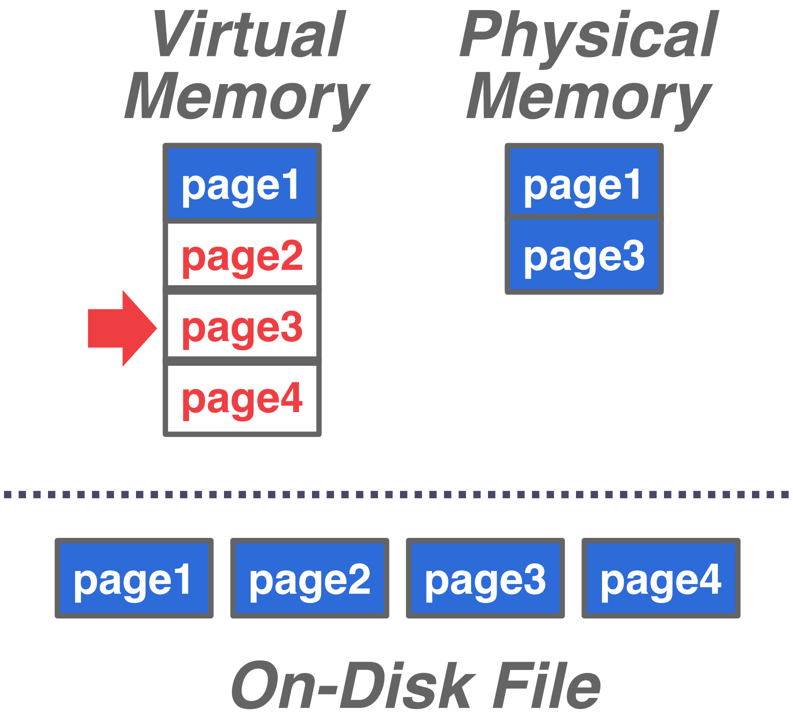


Paging

A DBMS uses a buffer pool, but why not use the OS instead?

With memory-mapped files, the OS is responsible for transparent paging of the file so the DBMS doesn't need to worry about it.

Is this a good idea?

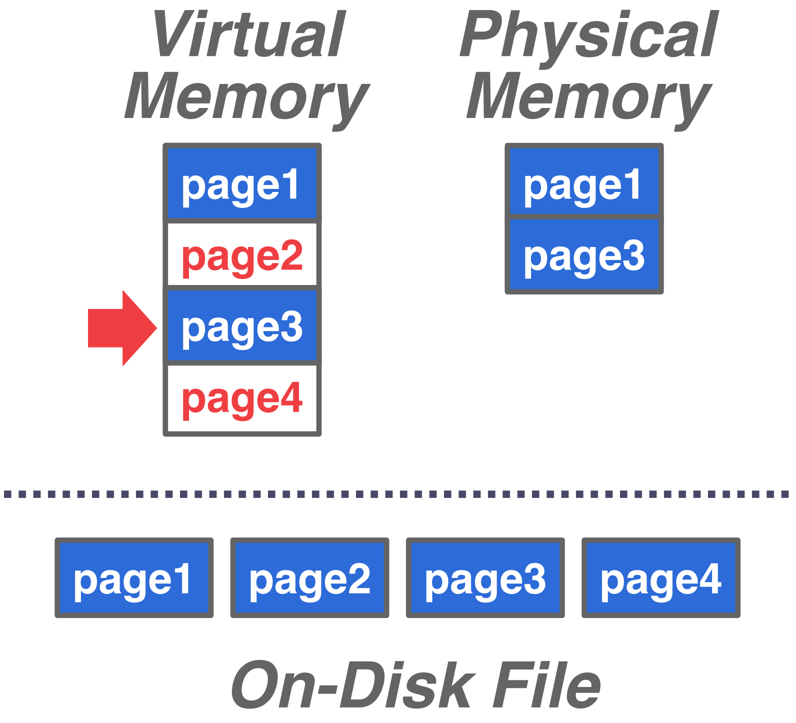


Paging

A DBMS uses a buffer pool, but why not use the OS instead?

With memory-mapped files, the OS is responsible for transparent paging of the file so the DBMS doesn't need to worry about it.

Is this a good idea?

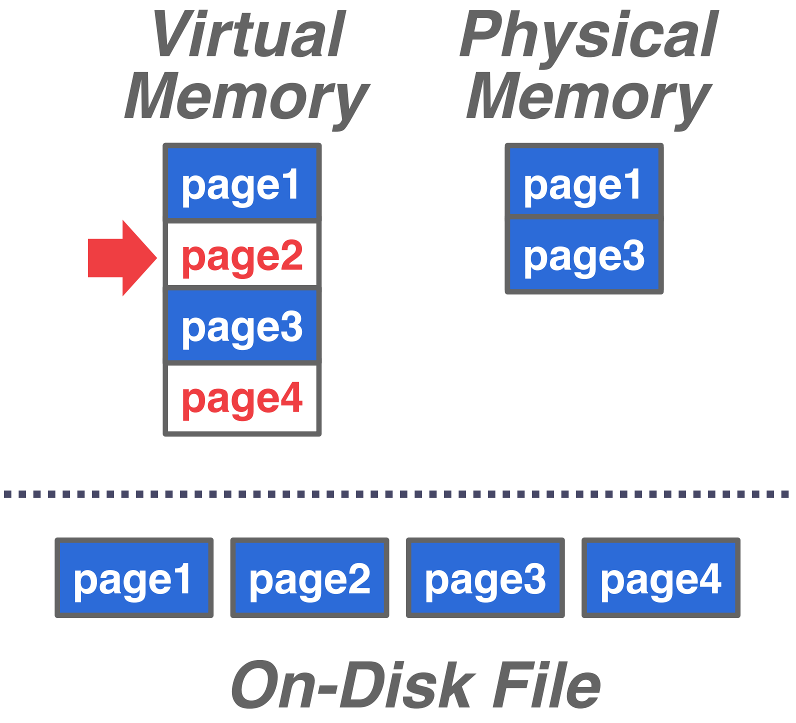


Paging

A DBMS uses a buffer pool, but why not use the OS instead?

With memory-mapped files, the OS is responsible for transparent paging of the file so the DBMS doesn't need to worry about it.

Is this a good idea?

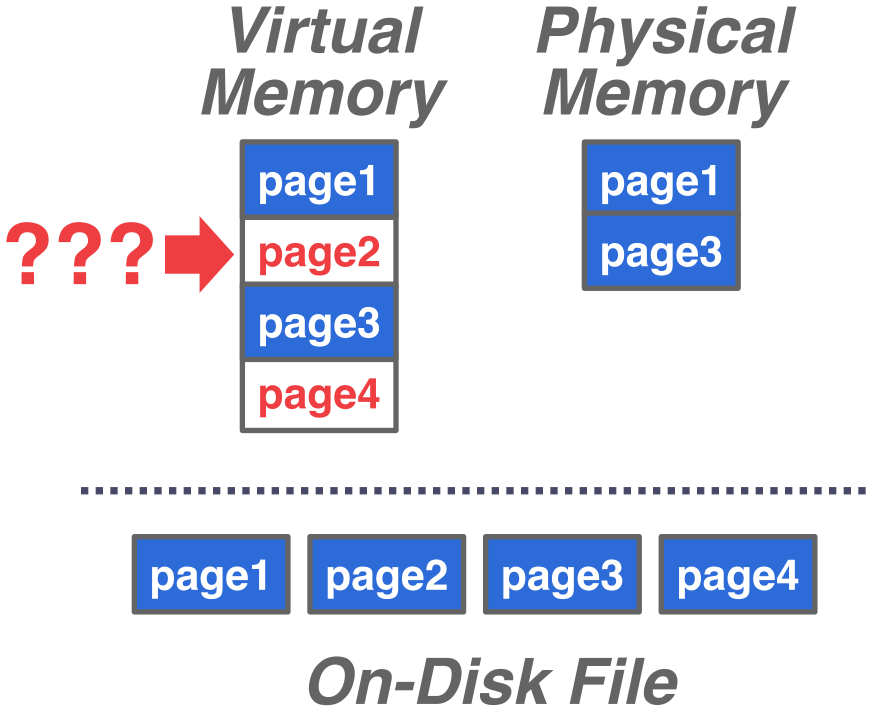


Paging

A DBMS uses a buffer pool, but why not use the OS instead?

With memory-mapped files, the OS is responsible for transparent paging of the file so the DBMS doesn't need to worry about it.

Is this a good idea?

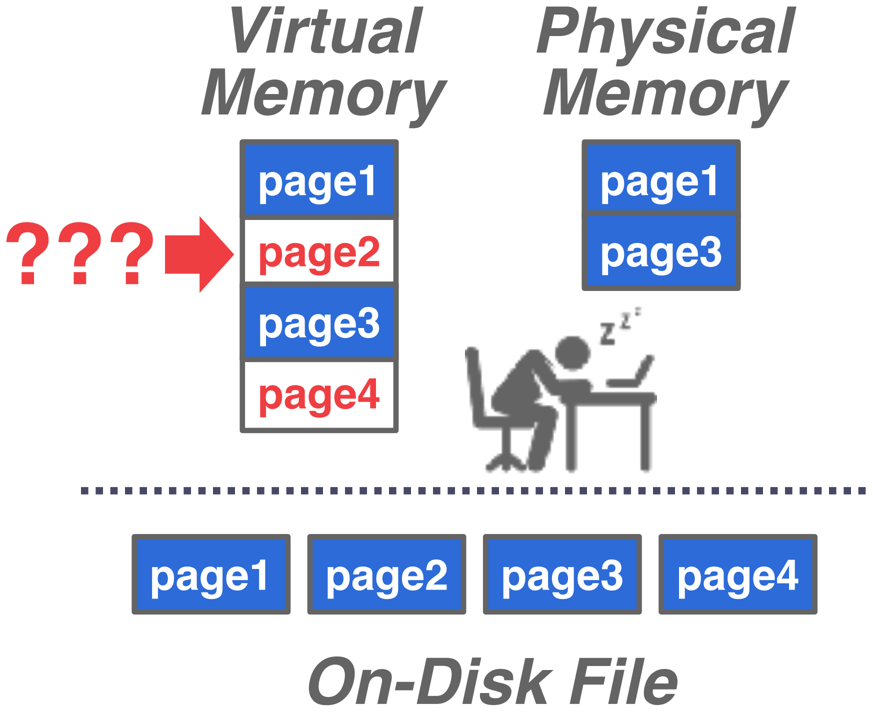


Paging

A DBMS uses a buffer pool, but why not use the OS instead?

With memory-mapped files, the OS is responsible for transparent paging of the file so the DBMS doesn't need to worry about it.

Is this a good idea?



Paging

What if we allow multiple threads to access memory-mapped files to hide page fault stalls?

This works (well enough) for read-only access but becomes complicated with multiple writers.

Memory-Mapped Files

Problem #1: Transaction Safety
→ OS can flush dirty pages at any time.

Problem #2: I/O Stalls
→ DBMS doesn’t know which pages are in memory. OS will stall a thread on page fault.

Problem #3: Error Handling
→ Difficult to validate pages. Any access can cause a SIGBUS that DBMS must handle.

Problem #4: Performance Issues
→ OS data structure contention. TLB shootdowns.

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 have instructions / registers to support them.

But they do not guarantee exact values…

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);
}
```

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>

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
```

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 typ
precision and sc
errors are unacc
→ Example: **NUME**

Many different i
→ Example: Store
representation w
→ Can be less exp
arbitrary precisi
different position per value).

Postgres: NUMERIC

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

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;
```

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;
```

#

Weight o

Sca

Positive/Nega

Dig

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

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

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;
};
```

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 l

o

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

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

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.

Observation

Data is "hot" when it enters the database.
→ A newly inserted tuple is more likely to be updated again in 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.

HTAP 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.

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.

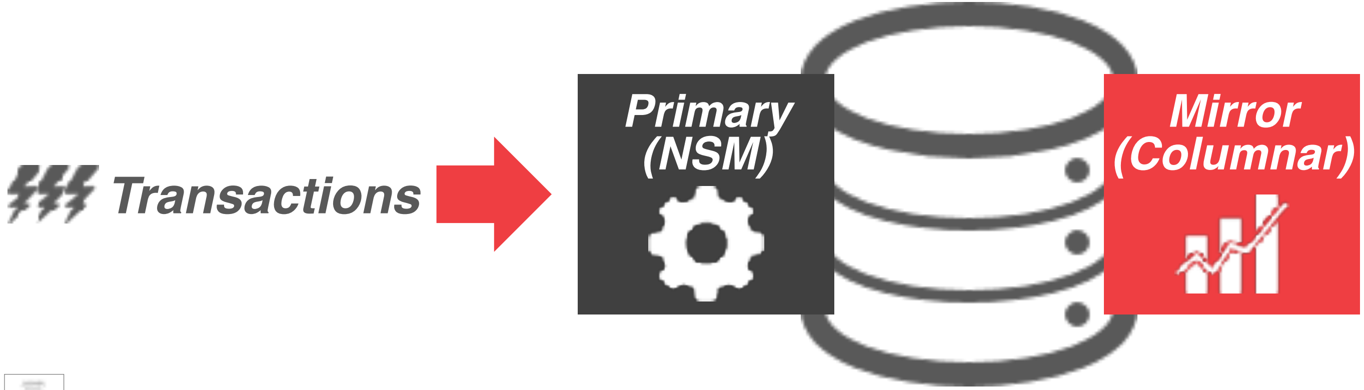
→ If the DBMS supports updates, it must invalidate tuples in the DSM mirror.

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.

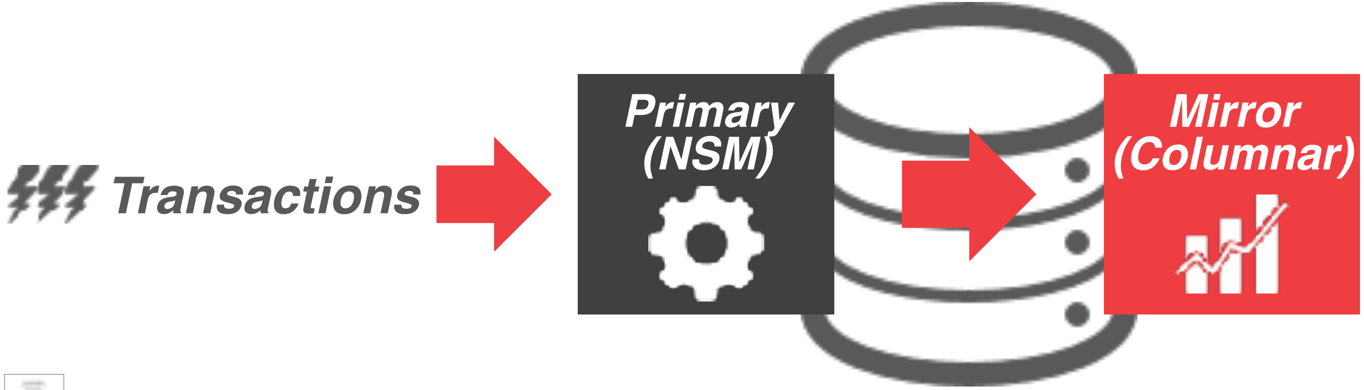


Fractured Mirrors

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.

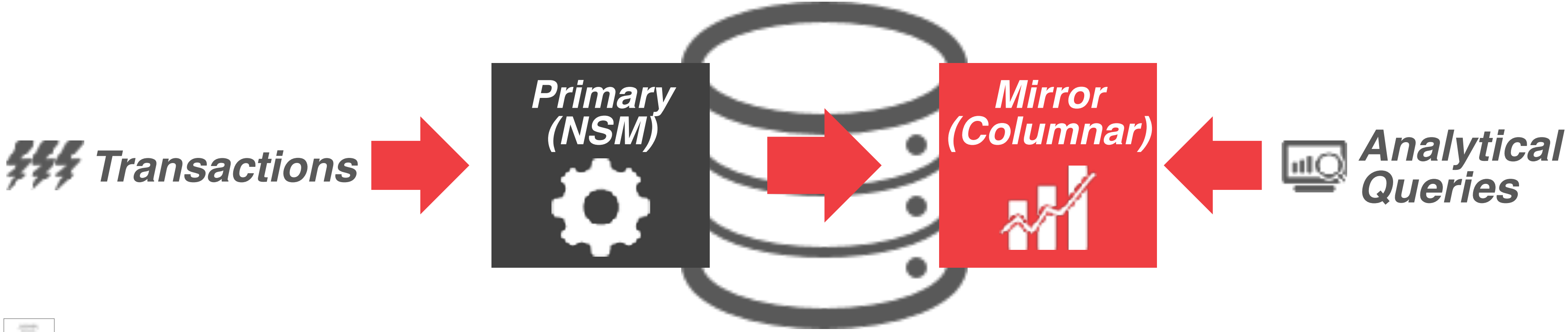


Fractured Mirrors

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.



Delta Store

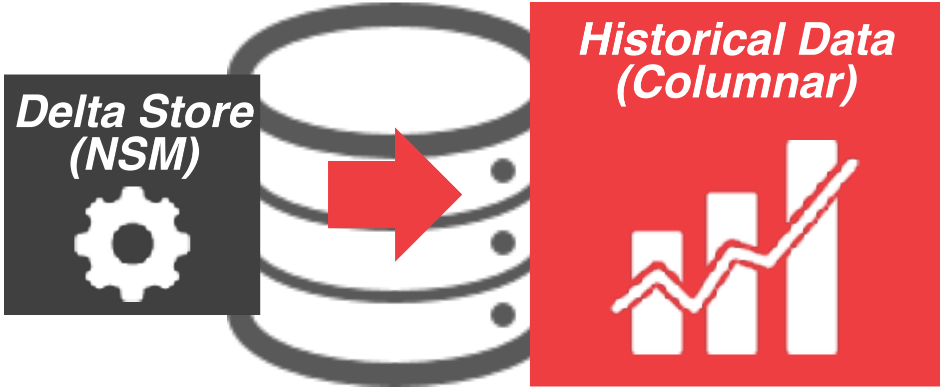
Stage updates in an NSM table.

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.

Delta Store

Stage updates 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.



Delta Store

Stage updates 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

Parting Thoughts

Every modern OLAP system is using some variant of PAX storage. Ideally, all data should 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

Compression