Lecture #08

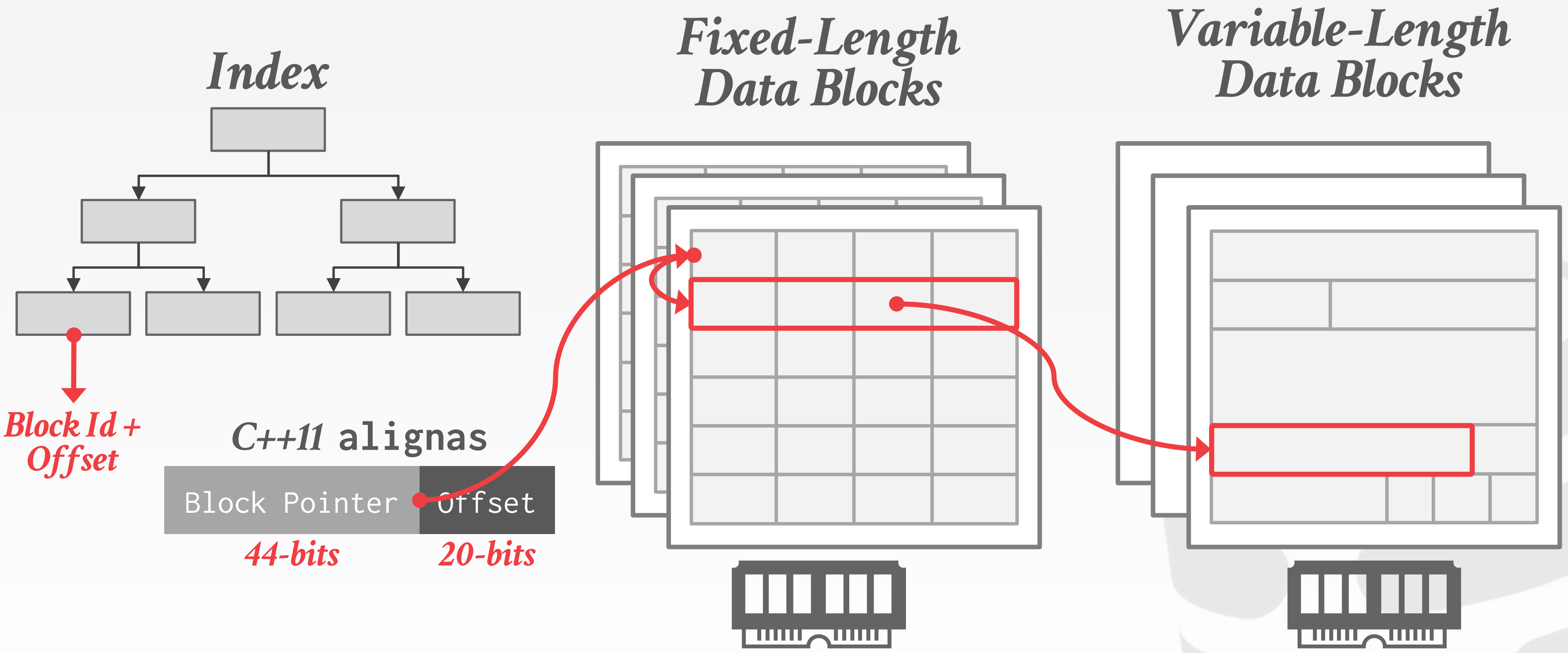
Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Storage Models & Data Layout

@Andy_Pavlo // 15-721 // Spring 2020

DATA ORGANIZATION



DATA ORGANIZATION

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 as fixed-length data makes it easy to compute the starting point of any tuple.

TODAY'S AGENDA

Type Representation
Data Layout / Alignment
Storage Models
System Catalogs

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.

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**/**DOUBLE**

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
```

FIXED PRECISION NUMBERS

Numeric data types with arbitrary precision and scale. Used when round errors are unacceptable.

→ Example: **NUMERIC**, **DECIMAL**

Typically stored in an exact, variable-length binary representation with additional meta-data.

→ Like a **VARCHAR** but not stored as a string

DATA LAYOUT

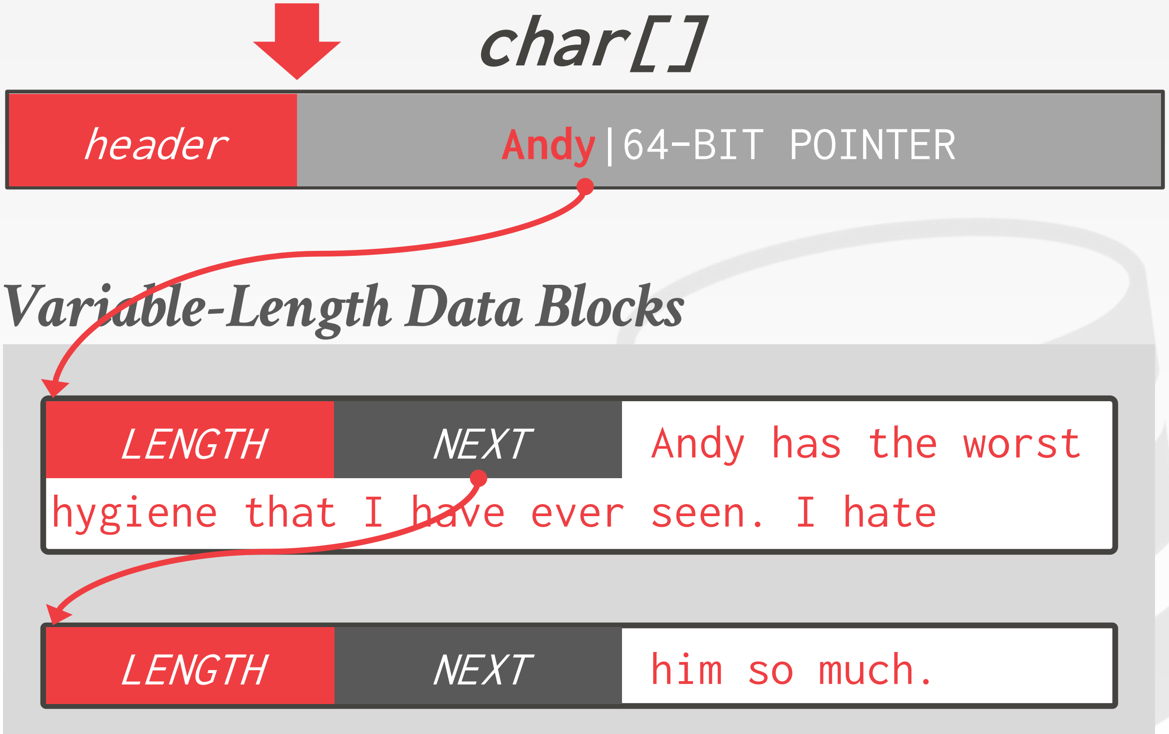
```
CREATE TABLE AndySux (
  id INT PRIMARY KEY,
  value BIGINT
);
```


```
reinterpret_cast<int32_t*>(address)
```

VARIABLE-LENGTH FIELDS

```
CREATE TABLE AndySux (
  value VARCHAR(1024)
);
```

```
INSERT INTO AndySux
    VALUES ("Andy has the worst
hygiene that I have ever seen. I
hate him so much.");
```



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

messes up with word alignment.

DISCLAIMER

The truth is that you only need to worry about word-alignment for cache lines (e.g., 64 bytes).

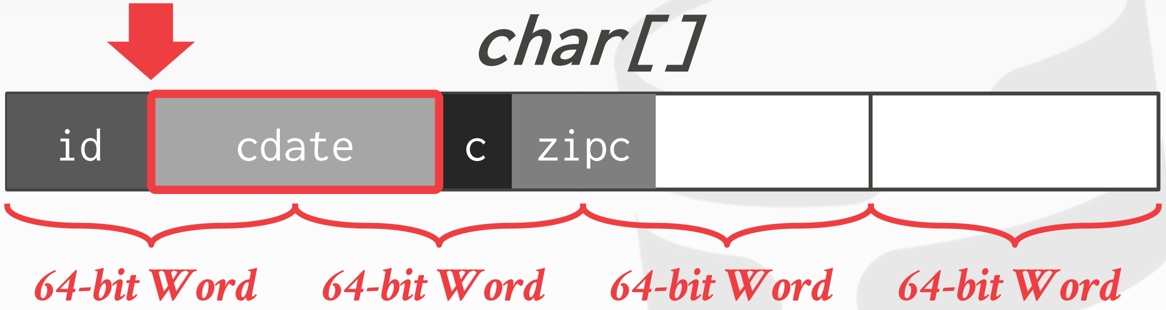
I'm going to show you the basic idea using 64-bit words since it's easier to see…

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
);
```

32-bits id, *64-bits* cdate, *16-bits* color, *32-bits* zipcode



WORD-ALIGNED TUPLES

Approach #1: Perform Extra Reads

→ Execute two reads to load the appropriate parts of the data word and reassemble them.

Approach #2: Random Reads

→ Read some unexpected combination of bytes assembled into a 64-bit word.

Approach #3: Reject

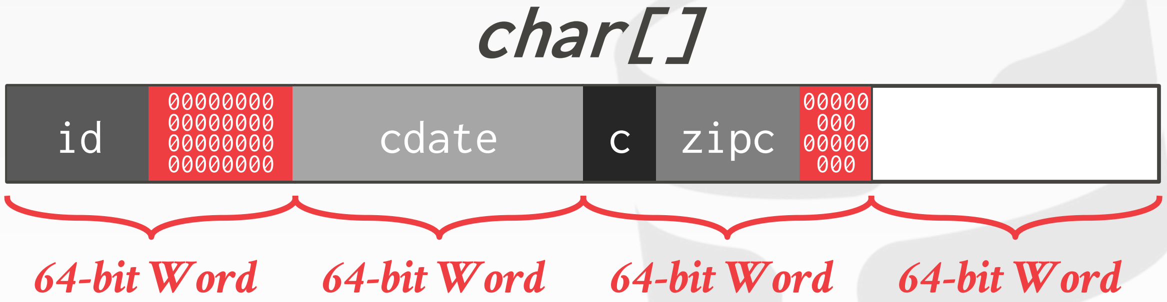
→ Throw an exception and hope app handles it.

Source: Levente Kurusa

WORD-ALIGNMENT: PADDING

Add empty bits after attributes to ensure that tuple is word aligned.

```
CREATE TABLE AndySux (
32-bits  id INT PRIMARY KEY,
64-bits  cdate TIMESTAMP,
16-bits  color CHAR(2),
32-bits  zipcode INT
);
```

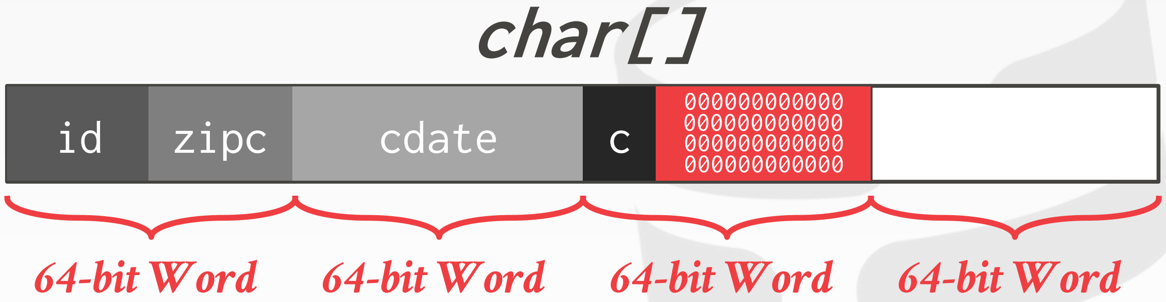


WORD-ALIGNMENT: REORDERING

Switch the order of attributes in the tuples' physical layout to make sure they are aligned.

→ May still have to use padding.

```
        CREATE TABLE AndySux (
32-bits id INT PRIMARY KEY,
64-bits cdate TIMESTAMP,
16-bits color CHAR(2),
32-bits zipcode INT
        );
```



CMU-DB ALIGNMENT EXPERIMENT

Processor: 1 socket, 4 cores w/ 2×HT
Workload: Insert Microbenchmark

	Avg. Throughput
No Alignment	0.523 MB/sec
Padding	11.7 MB/sec
Padding + Sorting	814.8 MB/sec

Source: Tianyu Li

STORAGE MODELS

N-ary Storage Model (NSM)

Decomposition Storage Model (DSM)

Hybrid Storage Model

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

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.

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.

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

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.

DSM SYSTEM HISTORY

1970s: Cantor DBMS

1980s: DSM Proposal

1990s: SybaseIQ (in-memory only)

2000s: Vertica, VectorWise, MonetDB

2010s: Everyone

DSM: DESIGN DECISIONS

Tuple Identification

Data Organization

Update Policy

Buffering Location

OPTIMAL COLUMN LAYOUT FOR HYBRID WORKLOADS
VLDB 2019

DSM: TUPLE IDENTIFICATION

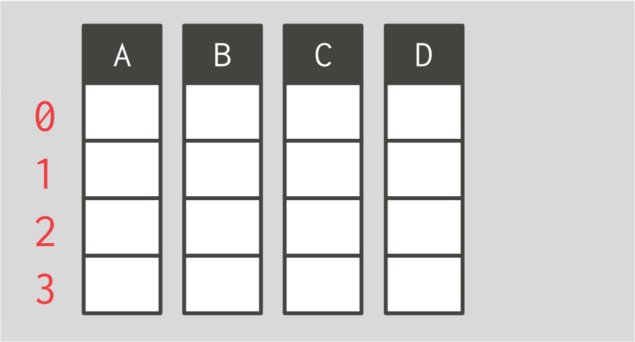
Choice #1: Fixed-length Offsets

→ Each value is the same length for an attribute.

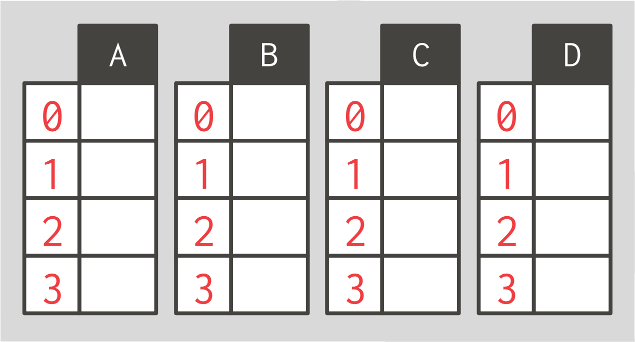
Choice #2: Embedded Tuple Ids

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

Offsets



Embedded Ids



DSM: DATA ORGANIZATION

Choice #1: Insertion Order

→ Tuples are inserted into any free slot that is available in existing blocks.

Choice #2: Sorted Order

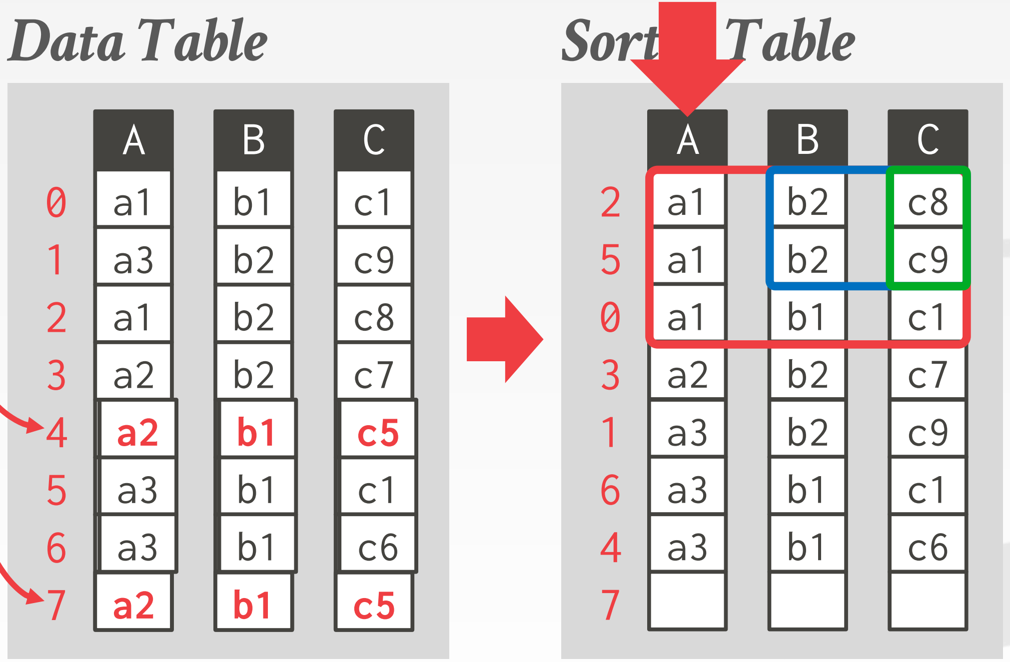
→ Tuples are inserted based into a slot according to some ordering scheme.

Choice #3: Partitioned

→ Assign tuples to blocks according to their attribute values and some partitioning scheme (e.g., hashing, range).

DSM: DATA ORGANIZATION

```
INSERT INTO xxx
VALUES (a2, b1, c5);
```



Sort Order: (A↑, B↓, C↑)

CASPER DELTA STORE

Range-partitioned column store with a "shallow" order-preserving index above it.

→ Shallow index maps value ranges to partitions.

→ Index keys are sorted but the individual columns are not.

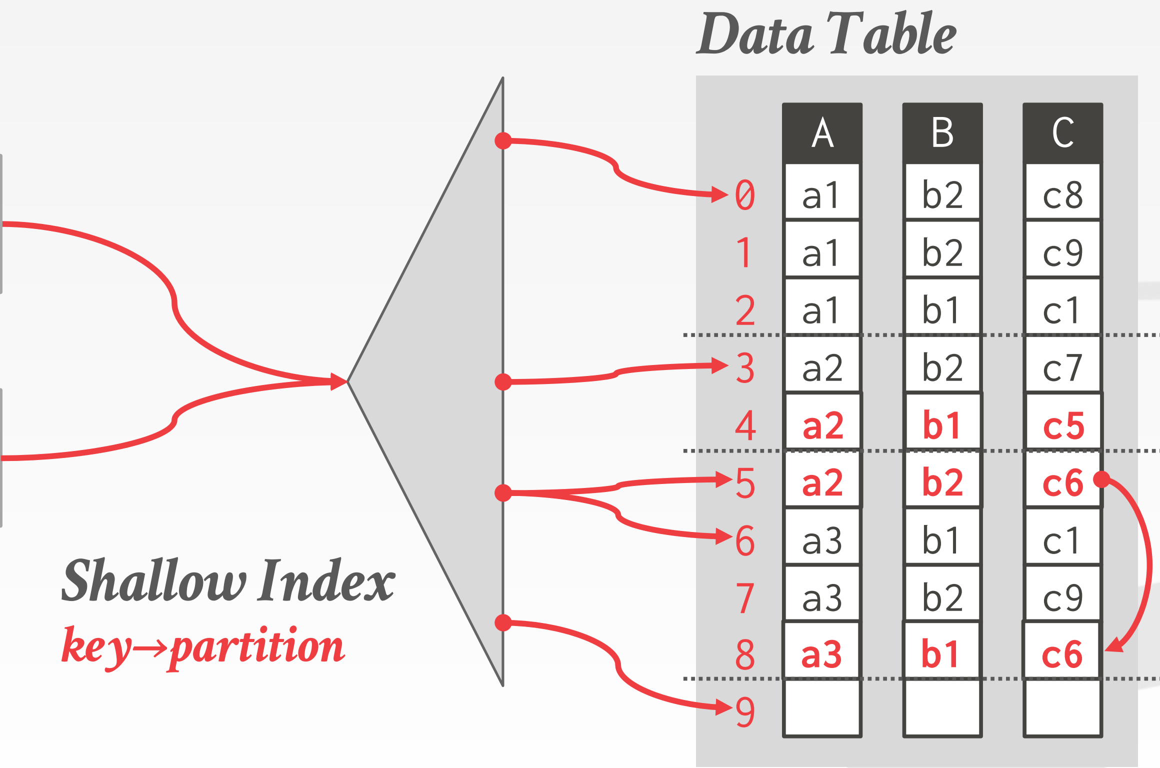
DBMS runs an offline optimization algorithm to determine the optimal partitioning of data.

OPTIMAL COLUMN LAYOUT FOR HYBRID WORKLOADS
VLDB 2019

CASPER DELTA STORE

```
INSERT INTO xxx
VALUES (a2, b1, c5);
```

```
INSERT INTO xxx
VALUES (a2, b2, c6);
```



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

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 can 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.
→ Must use a synchronization method (e.g., 2PC) if a txn spans execution engines.

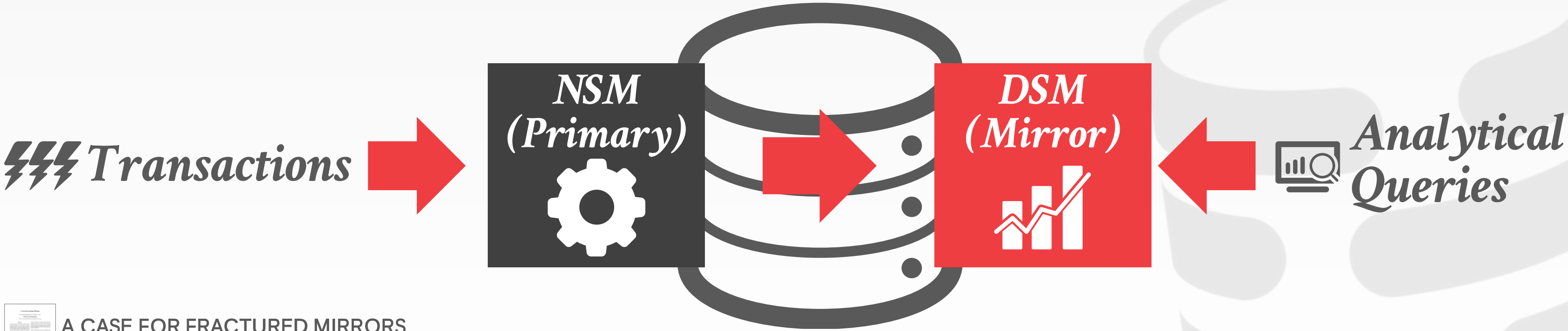
Two approaches to do this:

→ **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.



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.

PELOTON ADAPTIVE STORAGE

Employ a single execution engine architecture that can 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.

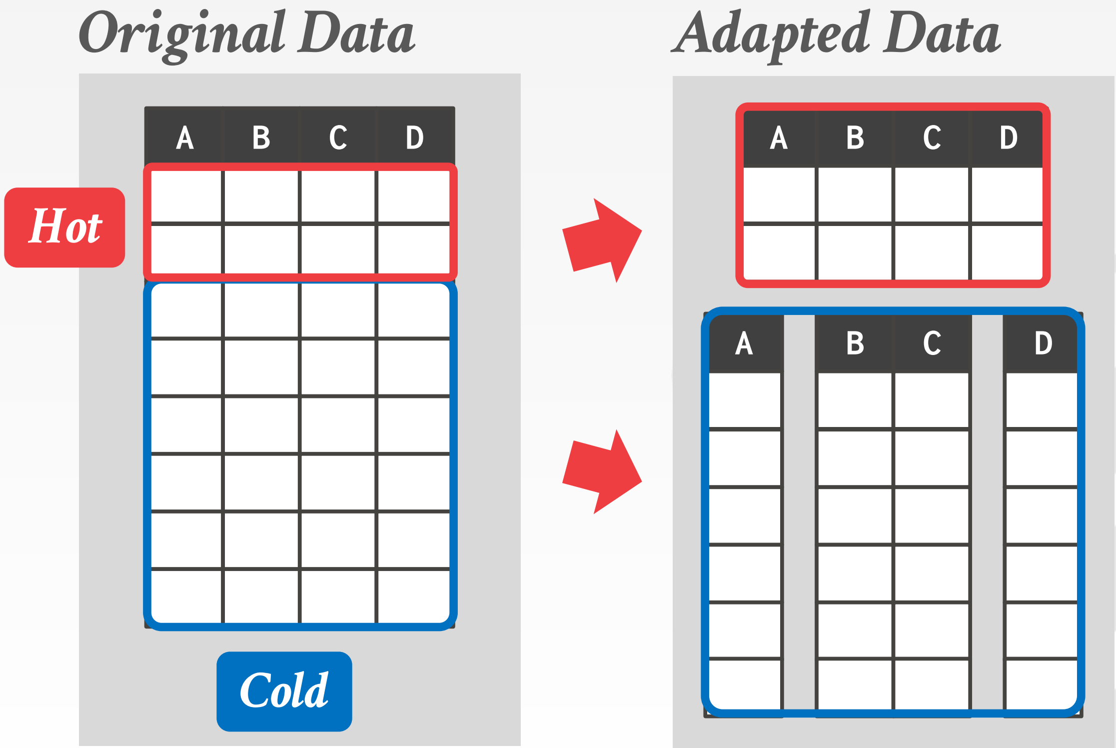
Note that a DBMS can still use the delta-store approach with this single-engine architecture.

BRIDGING THE ARCHIPELAGO BETWEEN ROW-STORES AND COLUMN-STORES FOR HYBRID WORKLOADS
SIGMOD 2016

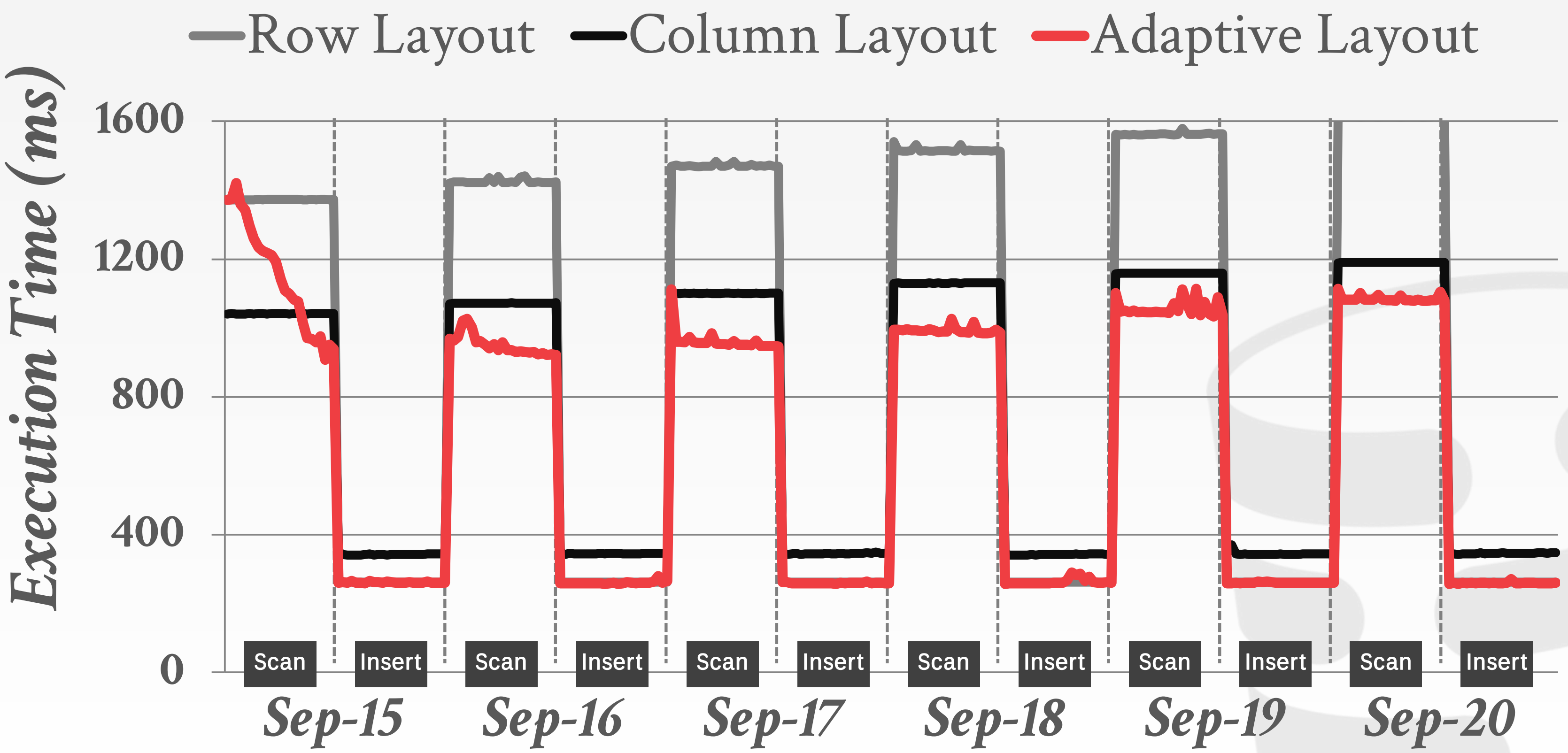
PELOTON ADAPTIVE STORAGE

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

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



PELOTON ADAPTIVE STORAGE



SYSTEM CATALOGS

Almost every DBMS stores their database's catalogs the same way that they store regular data.

→ Wrap object abstraction around tuples.

→ Specialized code for "bootstrapping" catalog tables.

The entire DBMS should be aware of transactions in order to automatically provide ACID guarantees for DDL commands and concurrent transactions.

SCHEMA CHANGES

ADD COLUMN:

→ **NSM**: Copy tuples into new region in memory.

→ **DSM**: Just create the new column segment

DROP COLUMN:

→ **NSM #1**: Copy tuples into new region of memory.

→ **NSM #2**: Mark column as "deprecated", clean up later.

→ **DSM**: Just drop the column and free memory.

CHANGE COLUMN:

→ Check whether the conversion can happen. Depends on default values.

INDEXES

CREATE INDEX:

→ Scan the entire table and populate the index.
→ Must record changes made by txns that modified the table while another txn was building the index.
→ When the scan completes, lock the table and resolve changes that were missed after the scan started.

DROP INDEX:

→ Just drop the index logically from the catalog.
→ It only becomes "invisible" when the txn that dropped it commits. All existing txns will still have to update it.

SEQUENCES

Typically stored in the catalog. Used for maintaining a global counter

→ Also called "auto-increment" or "serial" keys

Sequences are not maintained with the same isolation protection as regular catalog entries.

→ Rolling back a txn that incremented a sequence does not rollback the change to that sequence.

→ All **INSERT** queries would incur write-write conflicts.

PARTING THOUGHTS

We abandoned the hybrid storage model
→ Significant engineering overhead.
→ Delta version storage + column store is almost equivalent.

Catalogs are hard.