Lecture #09

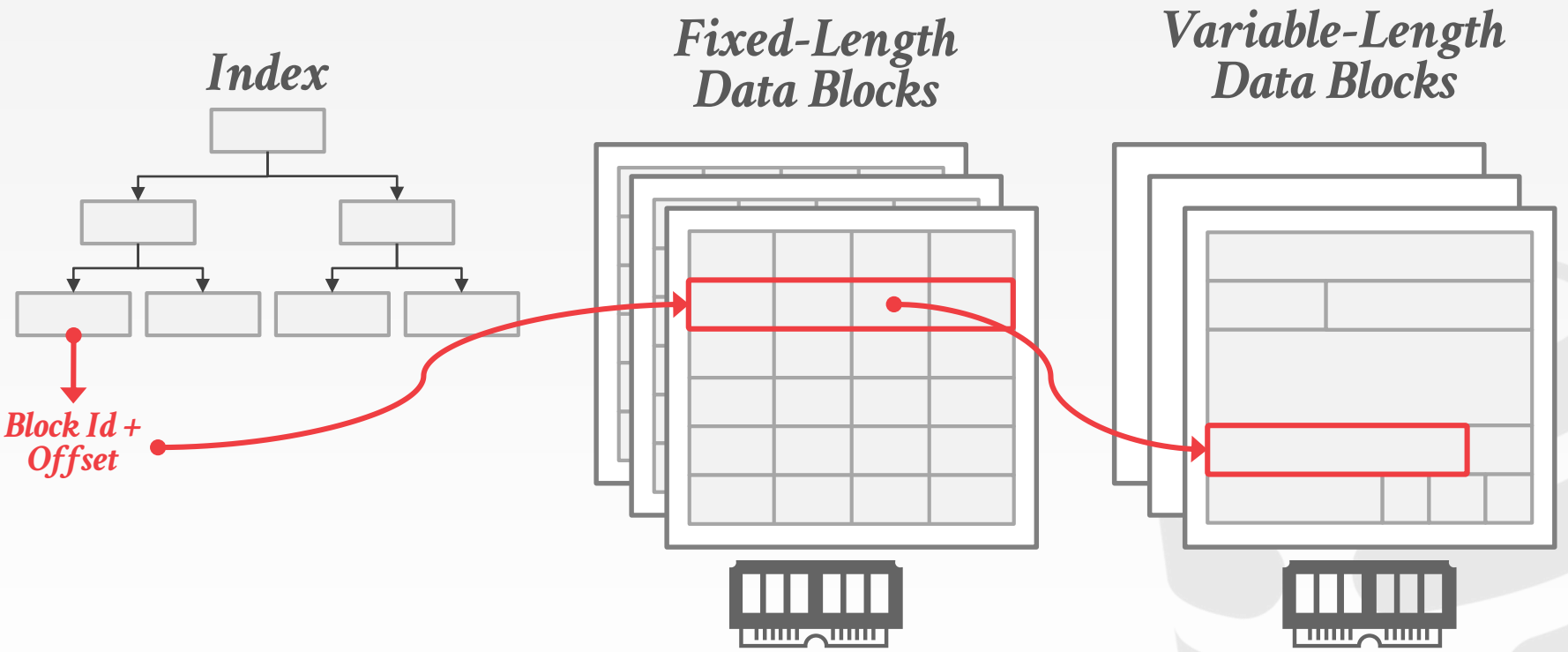
Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Storage Models & Data Layout

@Andy_Pavlo // 15-721 // Spring 2019

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

Output

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

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);
}
```

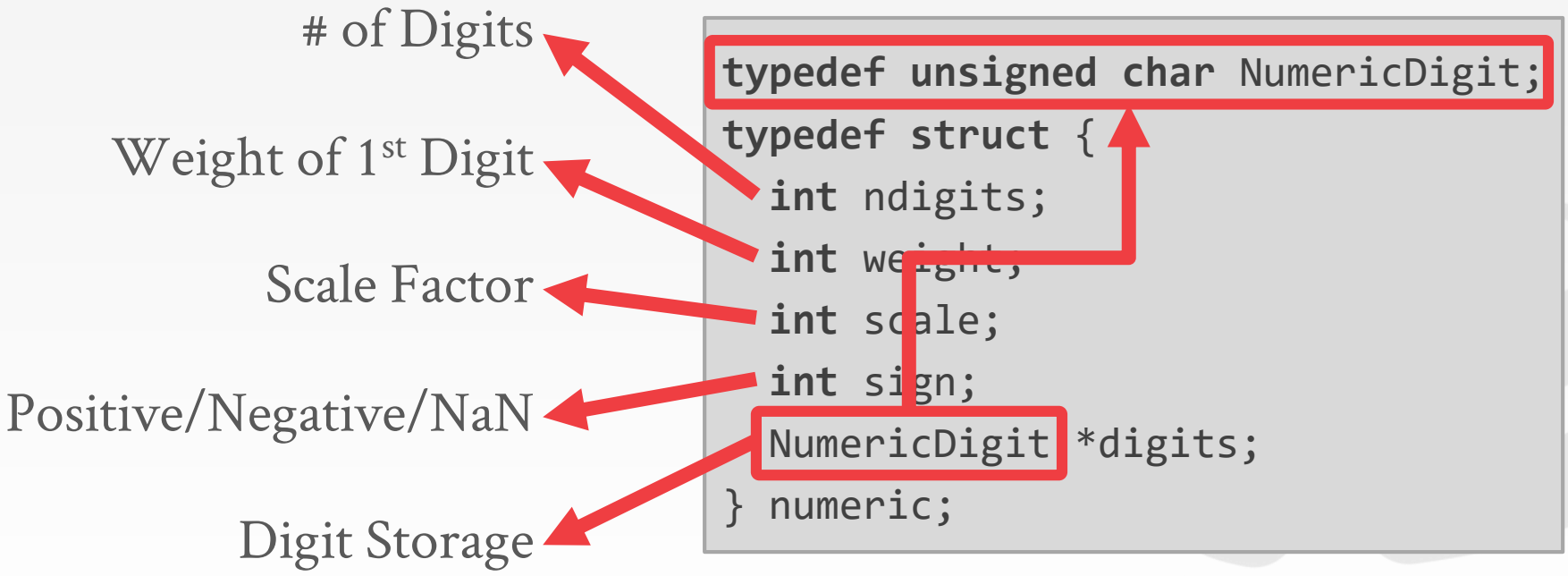
FIXED PRECISION NUMBERS

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

Postgres Demo

POSTGRES: NUMERIC



#

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))
                     *
```

DATA LAYOUT

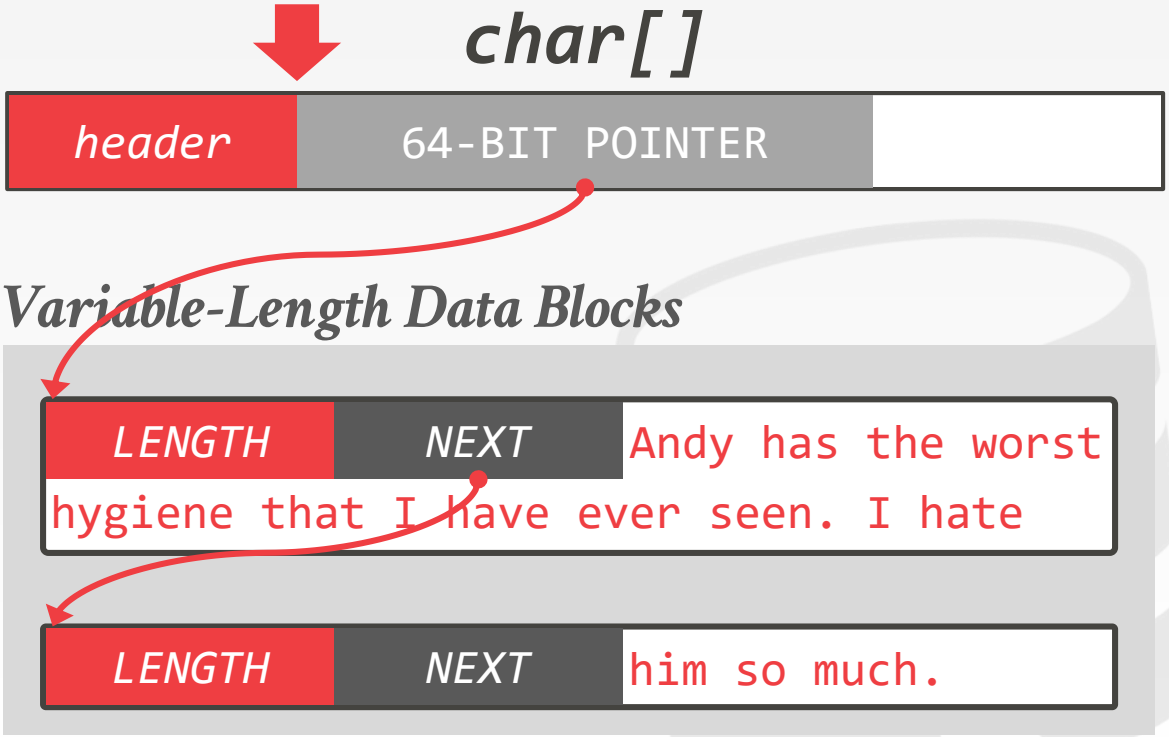
```
CREATE TABLE AndySux (
  id INT PRIMARY KEY,
  value BIGINT
);
```


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

VARIABLE-LENGTH FIELDS

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

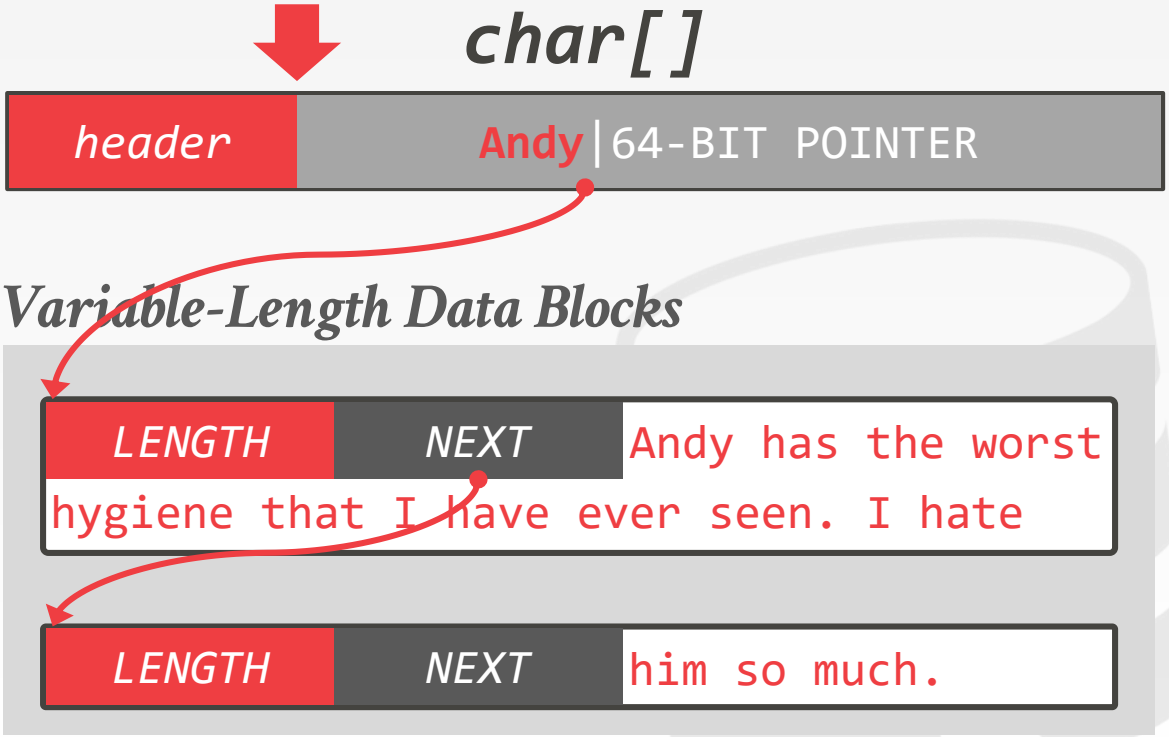
```
INSERT INTO AndySux
VALUES ("Andy has the worst
hygiene that I have ever
seen. I hate him so much.");
```



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

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

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

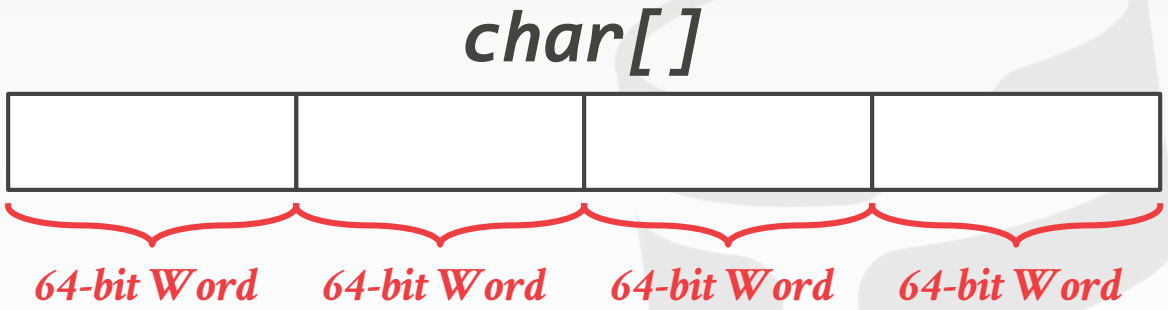
The truth is that you only need to worry about word-alignment for cache lines (e.g., 64 bytes).

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
);
```

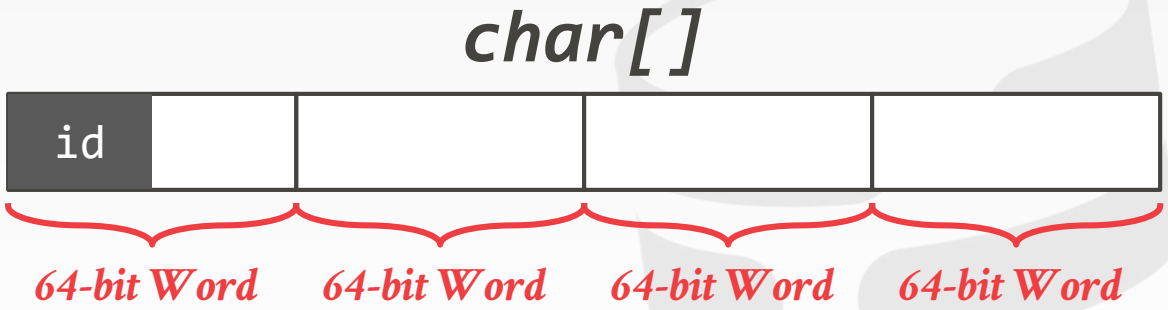


WORD-ALIGNED TUPLES

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

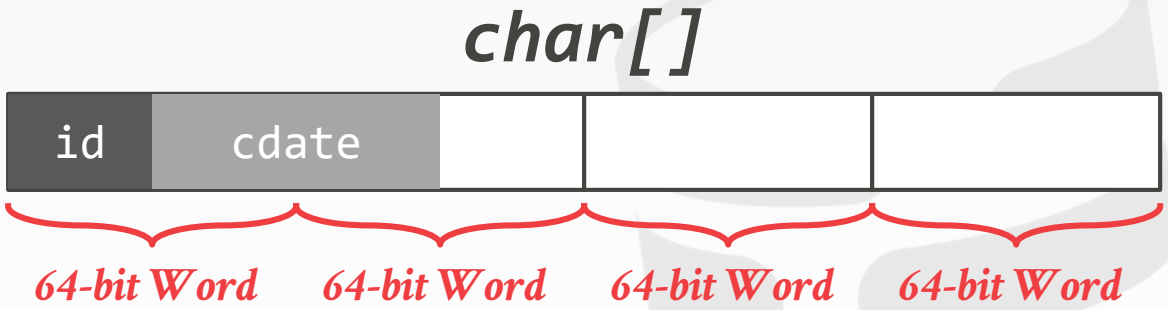


WORD-ALIGNED TUPLES

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
64-bits

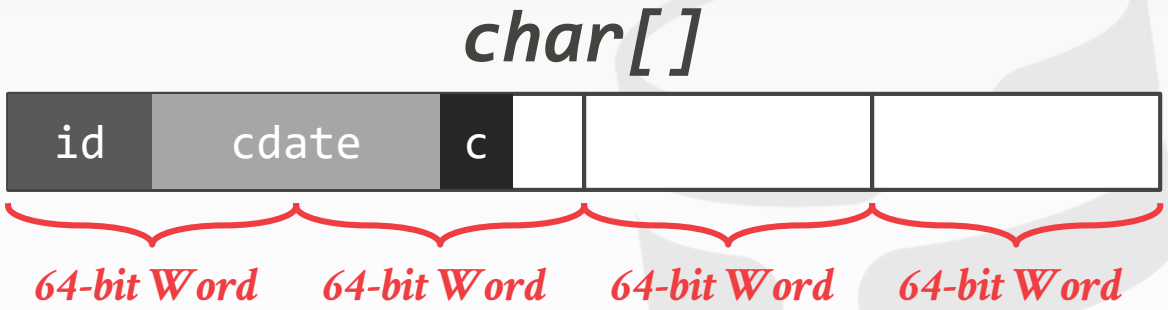


WORD-ALIGNED TUPLES

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
);
```

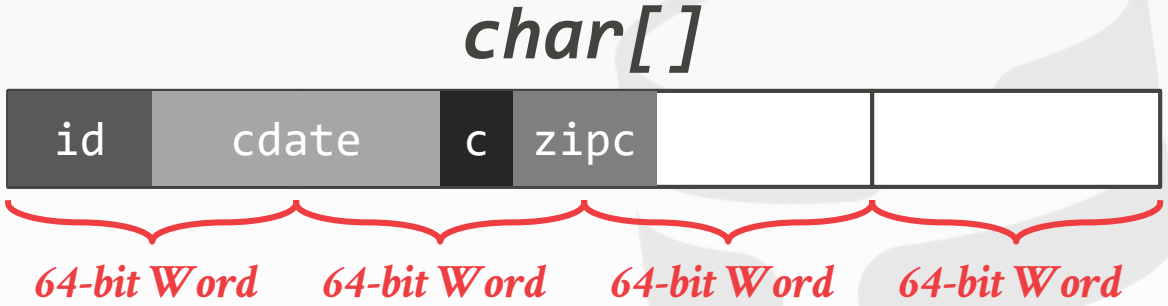
32-bits
64-bits
16-bits



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.

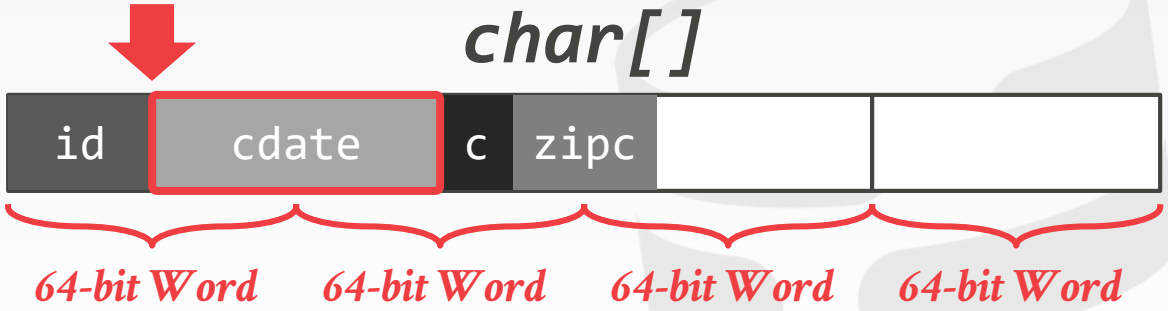
```
CREATE TABLE AndySux (
  id INT PRIMARY KEY,      -- 32-bits
  cdate TIMESTAMP,         -- 64-bits
  color CHAR(2),           -- 16-bits
  zipcode INT              -- 32-bits
);
```



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,      -- 32-bits
  cdate TIMESTAMP,         -- 64-bits
  color CHAR(2),           -- 16-bits
  zipcode INT              -- 32-bits
);
```



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

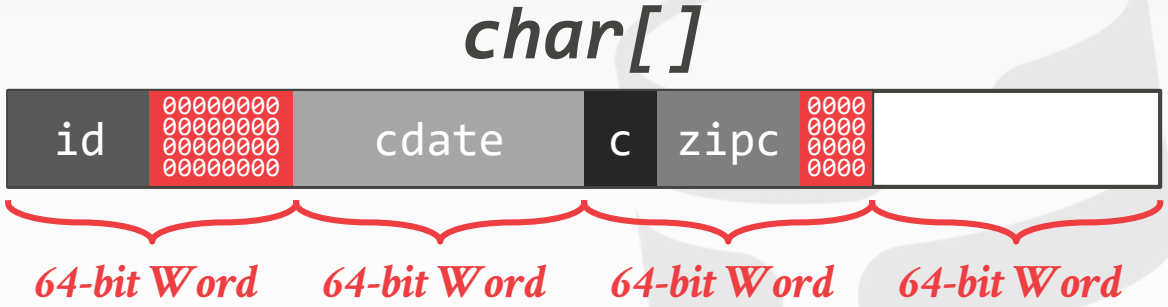
→ Throw an exception

Source: Levente Kurusa

WORD-ALIGNMENT: PADDING

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

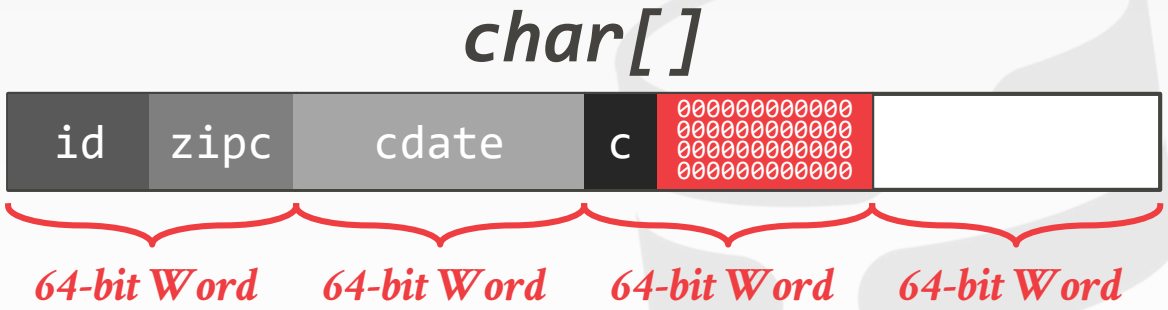
```
CREATE TABLE AndySux (
32-bits id INT PRIMARY KEY,
64-bits cdate TIMESTAMP,
16-bits color CHAR(2),
32-bits zipcode INT
);
```



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
Optimization #1	11.7 MB/sec
Optimization #2	814.8 MB/sec

Source: Tianyu Li

STORAGE MODELS

N-ary Storage Model (NSM)

Decomposition Storage Model (DSM)

Hybrid Storage Model

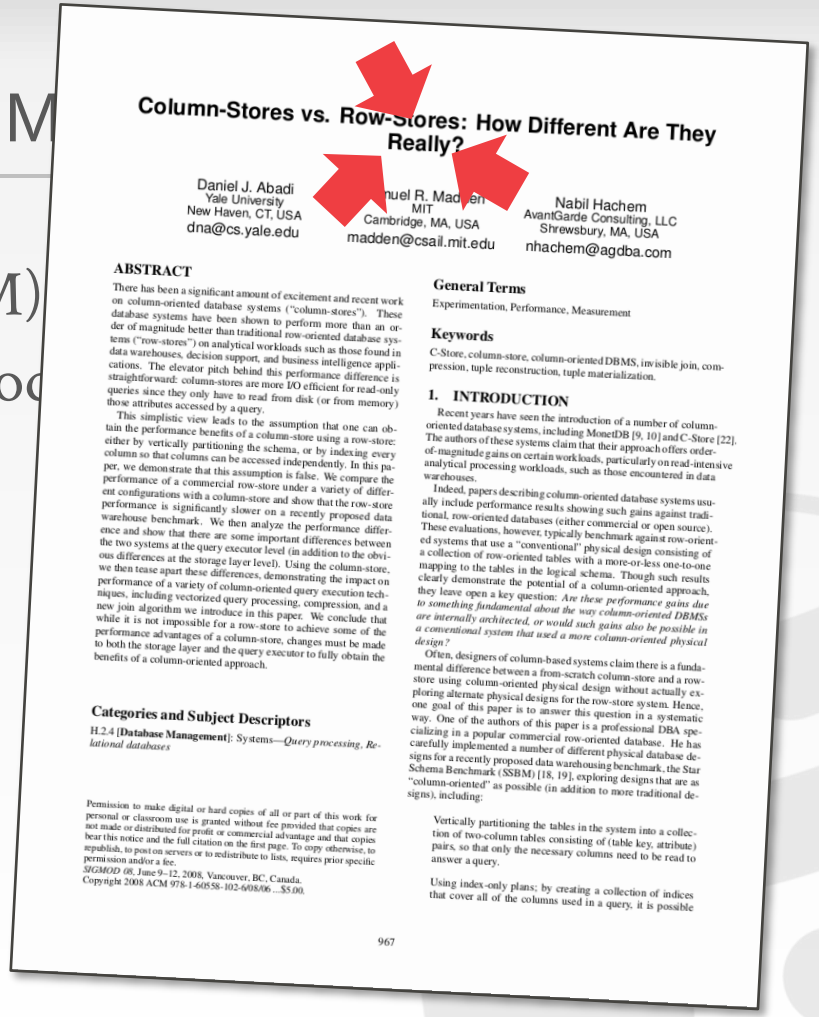
COLUMN-STORES VS. ROW-STORES: HOW DIFFERENT ARE THEY REALLY?
SIGMOD 2008

STORAGE M

N-ary Storage Model (NSM)

Decomposition Storage Mod

Hybrid Storage Model



Column-Stores vs. Row-Stores: How Different Are They Really?

Daniel J. Abadi
Yale University
New Haven, CT, USA
dna@cs.yale.edu

[illegible]uel R. Madden
MIT
Cambridge, MA, USA
madden@csail.mit.edu

Nabil Hachem
AvantGarde Consulting, LLC
Shrewsbury, MA, USA
nhachem@agdba.com

ABSTRACT

There has been a significant amount of excitement and recent work on column-oriented database systems ("column-stores"). These database systems have been shown to perform more than an order of magnitude better than traditional row-oriented database systems ("row-stores") on analytical workloads such as those found in data warehouses, decision support, and business intelligence applications. The elevator pitch behind this performance difference is straightforward: column-stores are more I/O efficient for read-only queries since they only have to read from disk (or from memory) those attributes accessed by a query.

This simplistic view leads to the assumption that one can obtain the performance benefits of a column-store using a row-store: either by vertically partitioning the schema, or by indexing every column so that columns can be accessed independently. In this paper, we demonstrate that this assumption is false. We compare the performance of a commercial row-store under a variety of different configurations with a column-store and show that the row-store performance is significantly slower on a recently proposed data warehouse benchmark. We then analyze the performance difference and show that there are some important differences between the two systems at the query executor level (in addition to the obvious differences at the storage layer level). Using the column-store, we then tease apart these differences, demonstrating the impact on performance of a variety of column-oriented query execution techniques, including vectorized query processing, compression, and a new join algorithm we introduce in this paper. We conclude that while it is not impossible for a row-store to achieve some of the performance advantages of a column-store, changes must be made to both the storage layer and the query executor to fully obtain the benefits of a column-oriented approach.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing, Relational databases*

General Terms

Experimentation, Performance, Measurement

Keywords

C-Store, column-store, column-oriented DBMS, invisible join, compression, tuple reconstruction, tuple materialization.

1. INTRODUCTION

Recent years have seen the introduction of a number of column-oriented database systems, including MonetDB [9, 10] and C-Store [22]. The authors of these systems claim that their approach offers order-of-magnitude gains on certain workloads, particularly on read-intensive analytical processing workloads, such as those encountered in data warehouses.

Indeed, papers describing column-oriented database systems usually include performance results showing such gains against traditional, row-oriented databases (either commercial or open source). These evaluations, however, typically benchmark against row-oriented systems that use a "conventional" physical design consisting of a collection of row-oriented tables with a more-or-less one-to-one mapping to the tables in the logical schema. Though such results clearly demonstrate the potential of a column-oriented approach, they leave open a key question: *Are these performance gains due to something fundamental about the way column-oriented DBMSs are internally architected, or would such gains also be possible in a conventional system that used a more column-oriented physical design?*

Often, designers of column-based systems claim there is a fundamental difference between a from-scratch column-store and a row-store using column-oriented physical design without actually exploring alternate physical designs for the row-store system. Hence, one goal of this paper is to answer this question in a systematic way. One of the authors of this paper is a professional DBA specializing in a popular commercial row-oriented database. He has carefully implemented a number of different physical database designs for a recently proposed data warehousing benchmark, the Star Schema Benchmark (SSBM) [18, 19], exploring designs that are as "column-oriented" as possible (in addition to more traditional designs), including:

- Vertically partitioning the tables in the system into a collection of two-column tables consisting of (table key, attribute) pairs, so that only the necessary columns need to be read to answer a query.
- Using index-only plans; by creating a collection of indices that cover all of the columns used in a query, it is possible

SIGMOD '08, June 9–12, 2008, Vancouver, BC, Canada.

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.

NSM: PHYSICAL STORAGE

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 primary key index itself.
→ Not quite the same as a clustered index.

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.

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, Clickhouse,
LinkedIn Pinot, and most OLAP systems

DSM: TUPLE IDENTIFICATION

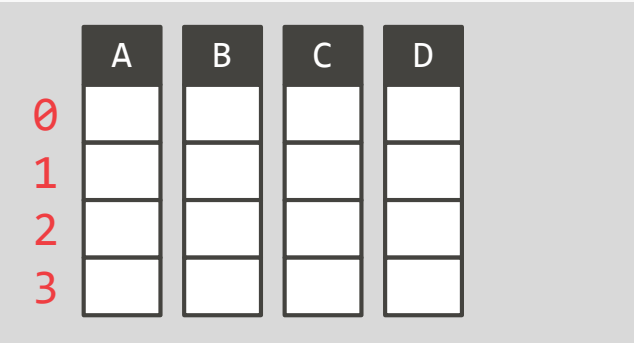
Choice #1: Fixed-length Offsets

→ Each value is the same length for an attribute.

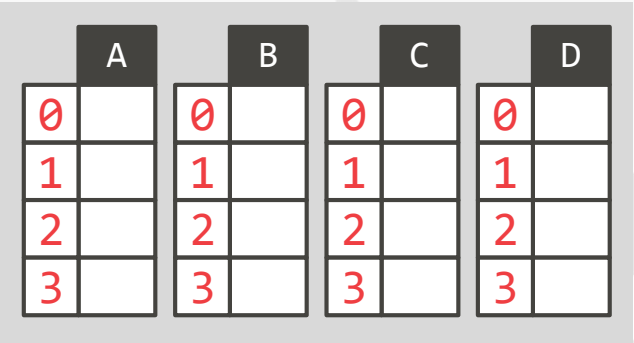
Choice #2: Embedded Tuple Ids

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

Offsets



Embedded Ids



DSM: QUERY PROCESSING

Late Materialization

Columnar Compression

Block/Vectorized Processing Model

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?

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.

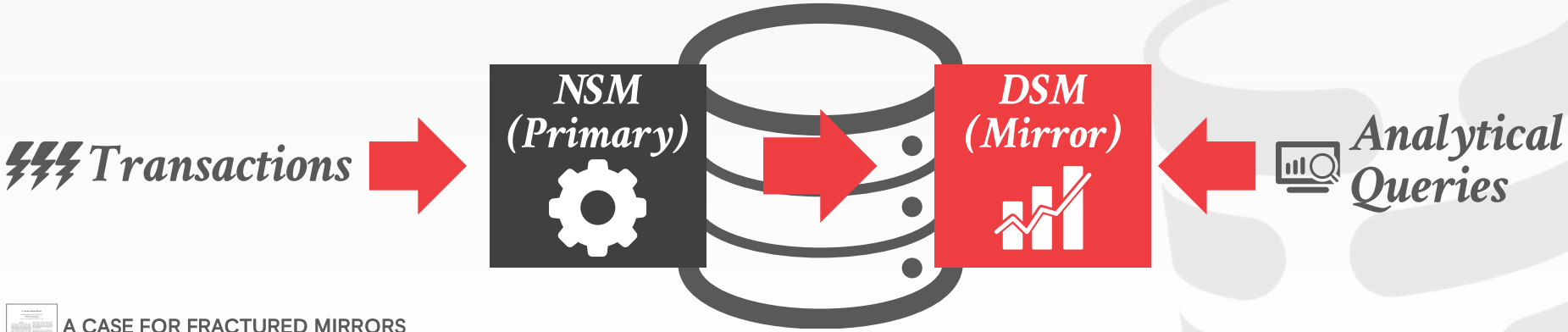
Two approaches to do this:

→ **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.



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.

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.

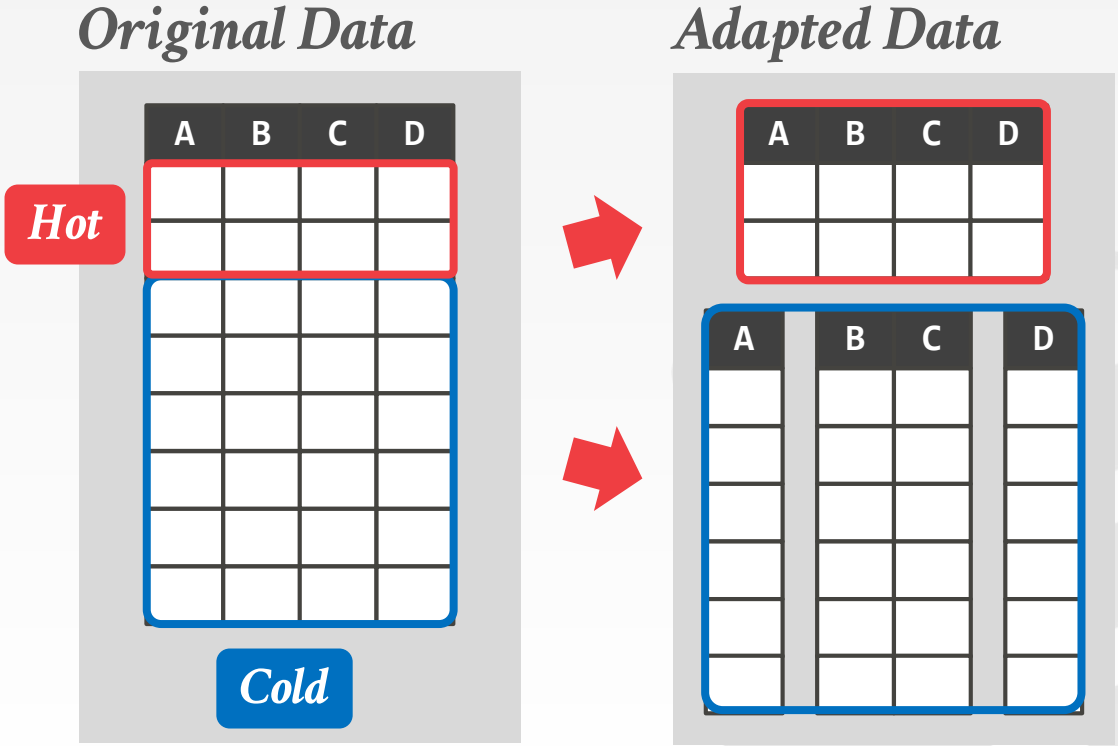
Note that a DBMS can still use the delta-store approach with this single-engine architecture.

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

PELOTON ADAPTIVE STORAGE

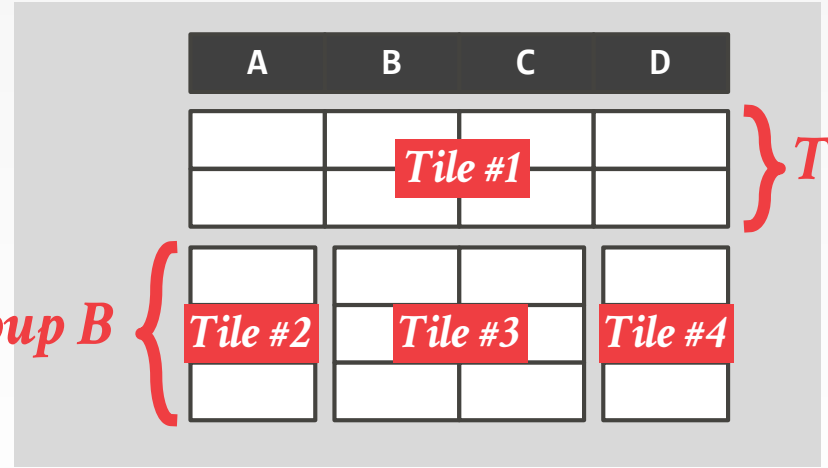
```
UPDATE AndySux
   SET A = 123,
       B = 456,
       C = 789
 WHERE D = "xxx"
```

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



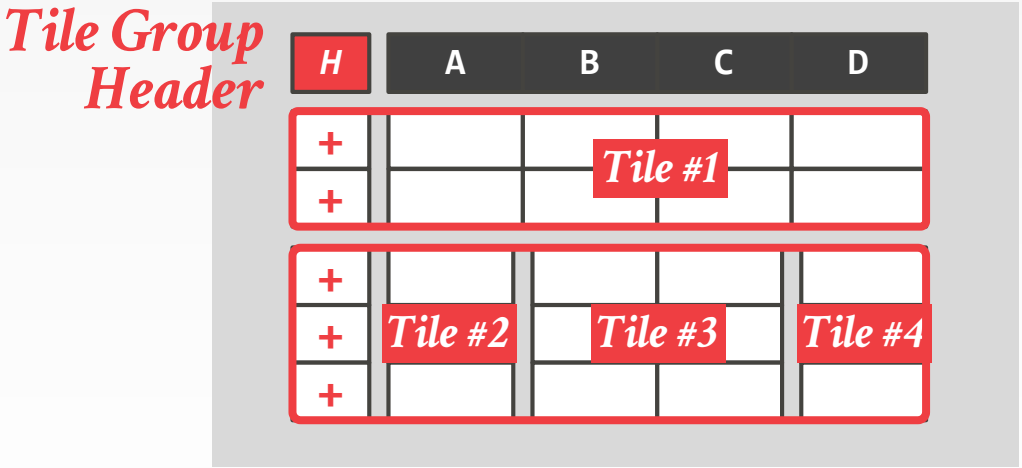
TILE ARCHITECTURE

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



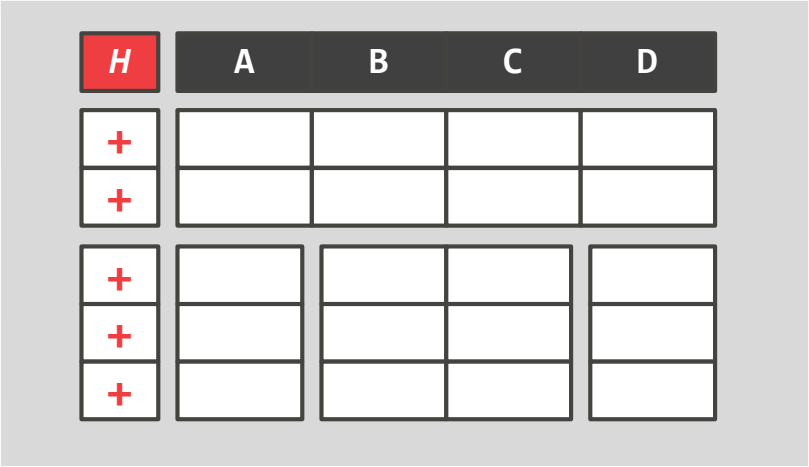
TILE ARCHITECTURE

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

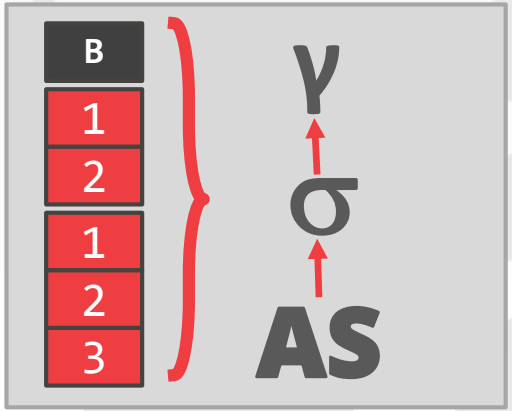


TILE ARCHITECTURE

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

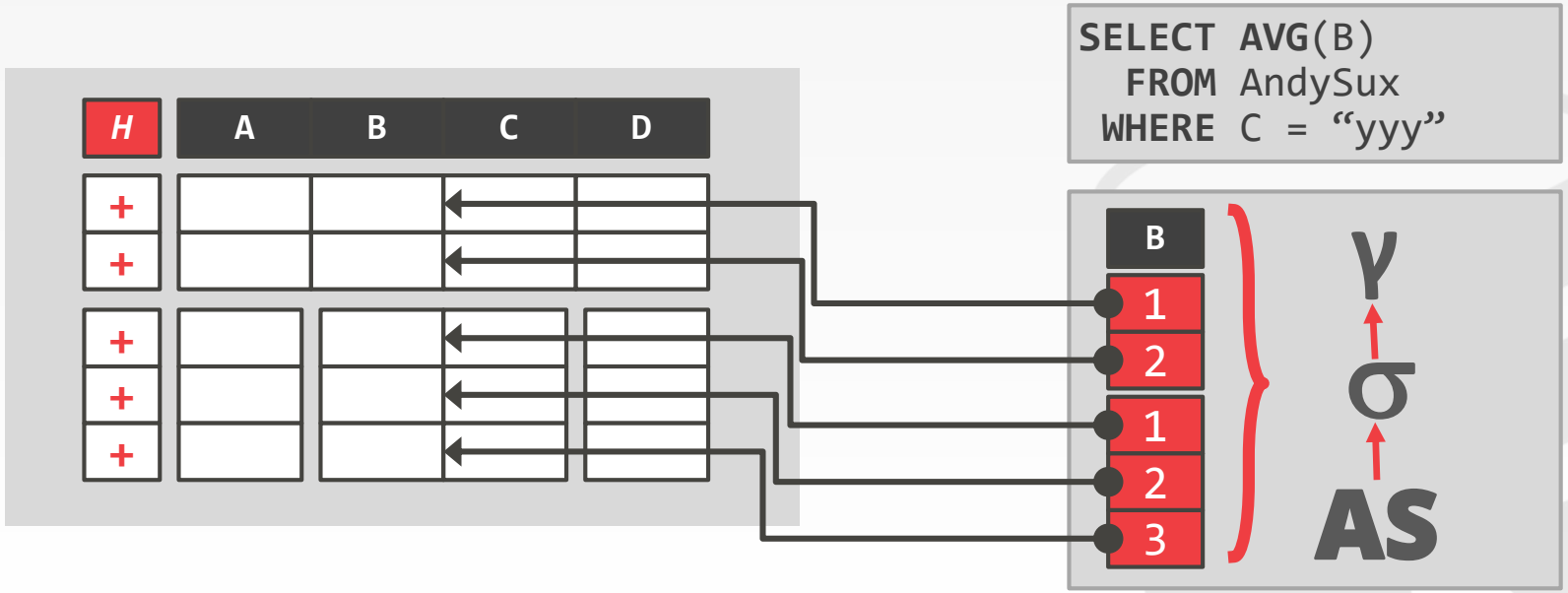



```
SELECT AVG(B)
  FROM AndySux
 WHERE C = "yyy"
```

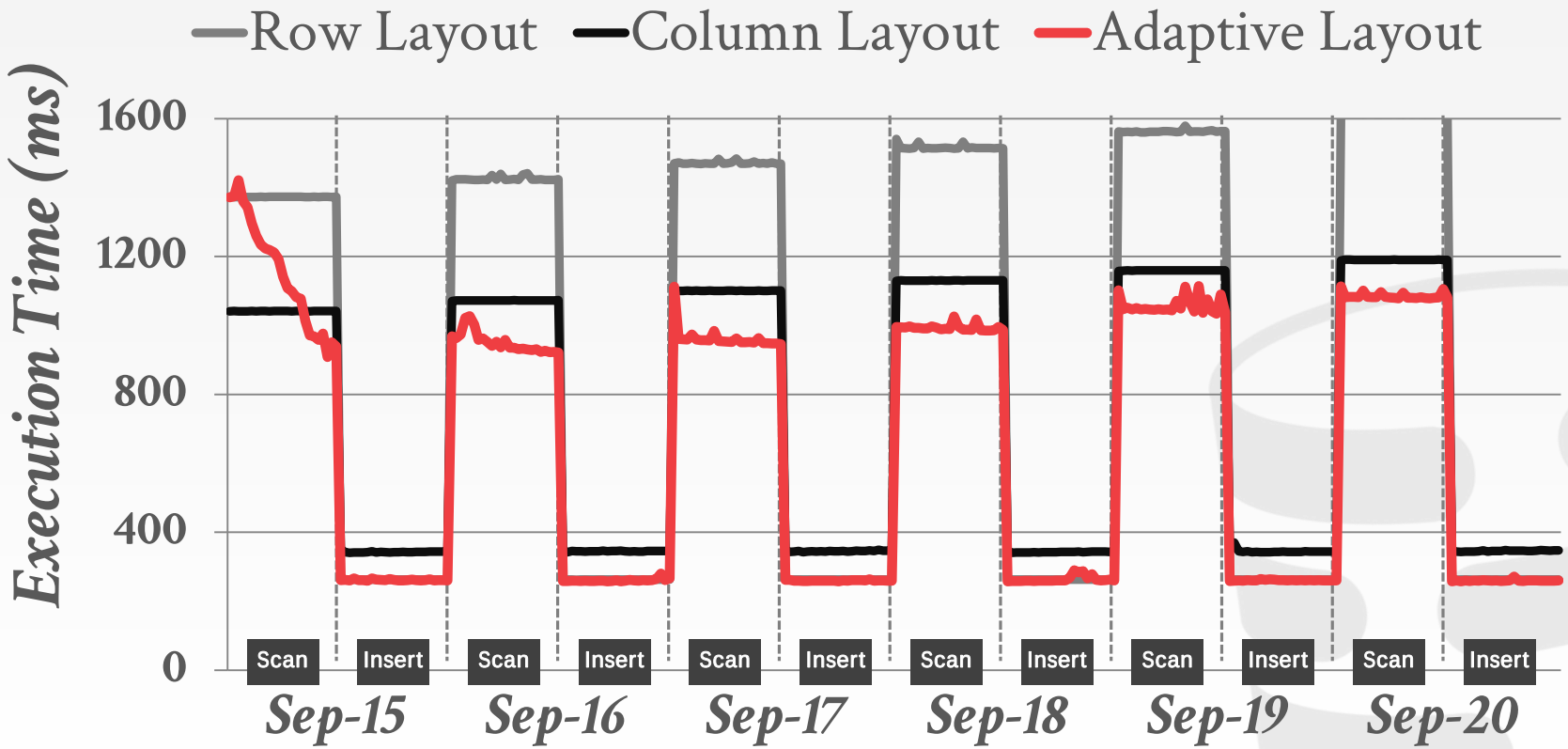


TILE ARCHITECTURE

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



PELOTON ADAPTIVE STORAGE



SYSTEM CATALOGS

Almost every DBMS stores their a database's catalog in itself.

→ 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 txns.

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 is allowed to happen.
Depends on default values.

INDEXES

CREATE INDEX:

→ Scan the entire table and populate the index.
→ Have to 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.