Data Storage Formats

Instructor: Matei Zaharia

Outline

Overview

Record encoding

Collection storage

C-Store paper

Indexes

Outline

Overview

Record encoding

Collection storage

C-Store paper

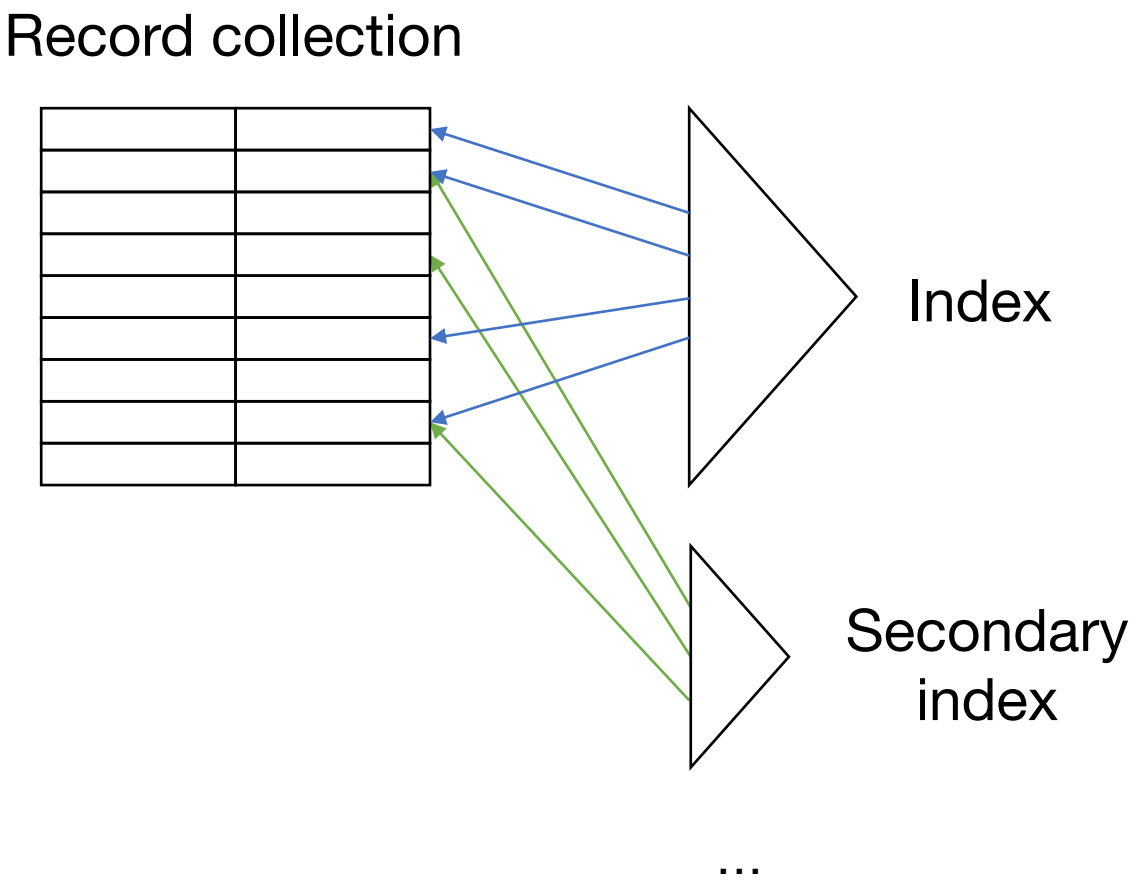
Indexes

Designing Storage Formats

Key concerns:

- » **Access time:** minimize # of random accesses, bytes transferred, etc
  - Main way: place co-accessed data together!
- » **Space:** storage costs $
- » **Ease of updates**

General Setup



Outline

Storage devices wrap-up

Record encoding

Collection storage

C-Store paper

Indexes

What Are the Data Items We Want to Store?

a salary

a name

a date

a picture

What Are the Data Items We Want to Store?

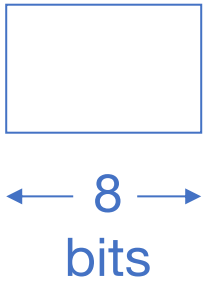
a salary

a name

a date

a picture

What we have available: bytes



Fixed-Length Items

Integer: fixed # of bytes (e.g., 2 bytes)

e.g., 35 is | 00000000 | 00100011 |

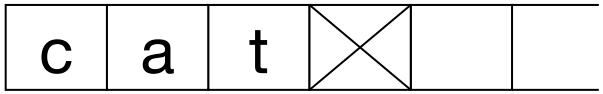
Floating-point: n-bit mantissa, m-bit exponent

Character: encode as integer (e.g. ASCII)

Variable-Length Items

String of characters:

» Null-terminated

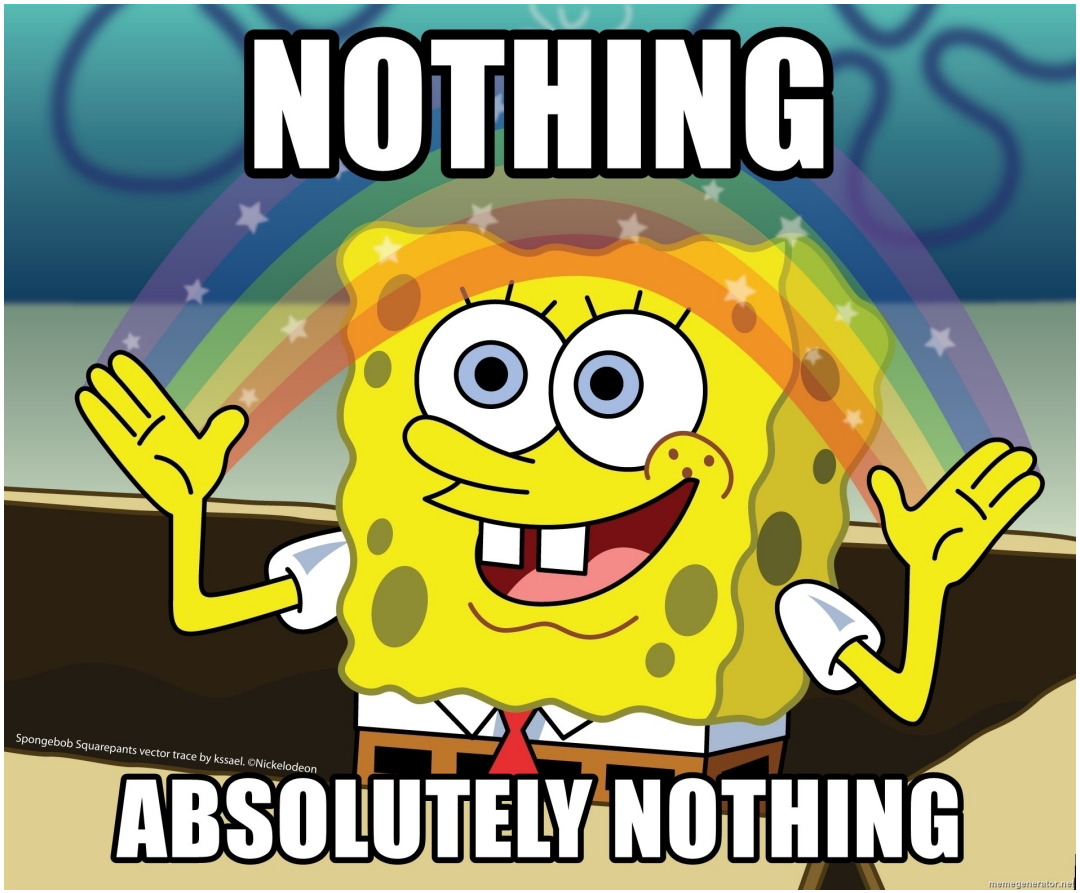


» Length + data

» Fixed-length

Bag of bits:

Representing



Representing Nothing

NULL concept in SQL (not same as 0 or “”)

Physical representation options:

- » Special “sentinel” value in fixed-length field
- » Boolean “is null” flag
- » Just skip the field in a sparse record format

Pretty common in practice!

Bigger Collections

Data Items

Records

Blocks

Files

Record: Set Data Items (Fields)

E.g. employee record:

- » name field
- » salary field
- » date-of-hire field
- » ...

Record Encodings

Fixed vs variable **format**

Fixed vs variable **length**

Fixed Format

A **schema** for all records in table specifies:

- # of fields
- type of each field
- order in record
- meaning of each field

Example: Fixed Format & Length

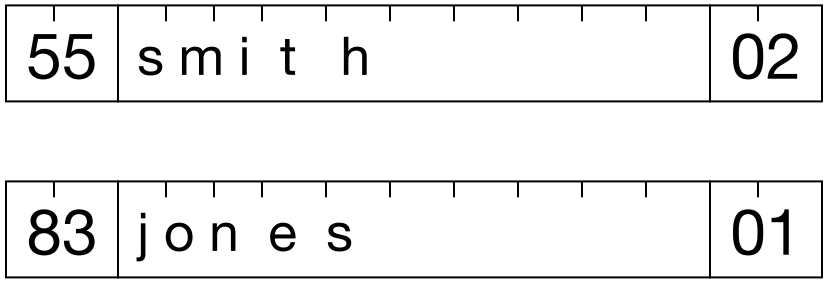
Employee record

(1) EID, 2 byte integer

(2) Name, 10 chars

(3) Dept, 2 byte code

Schema



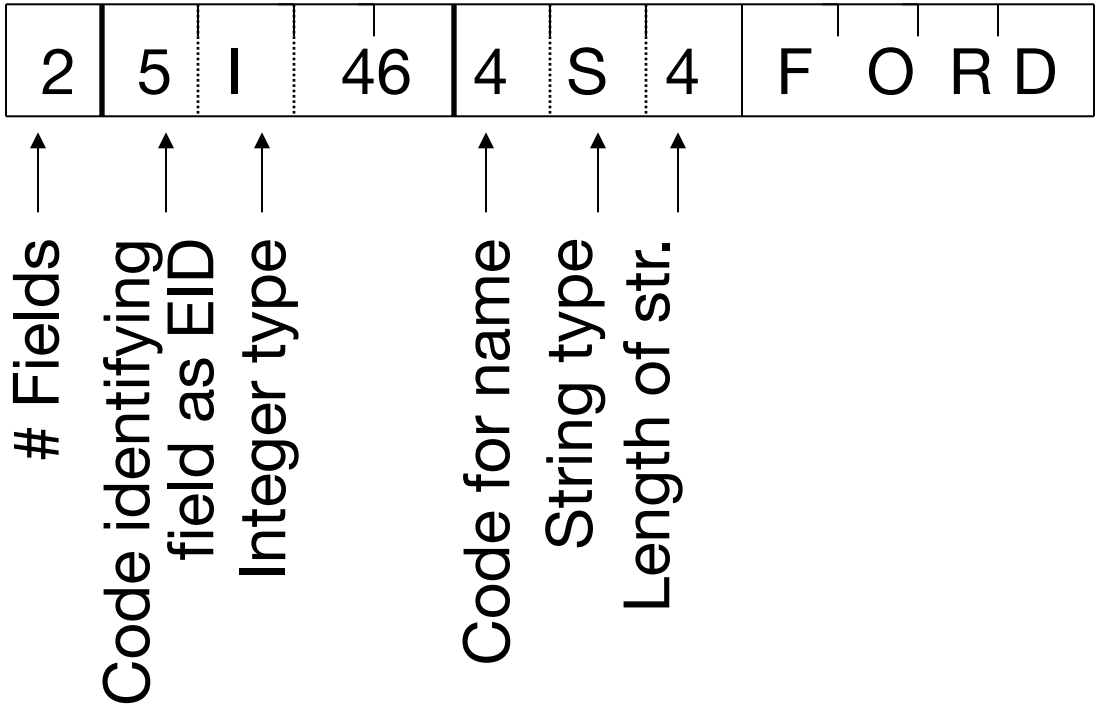
Records

Variable Format

Record itself contains format

"Self-describing"

Example: Variable Format & Length



Variable Format Useful For

“Sparse” records

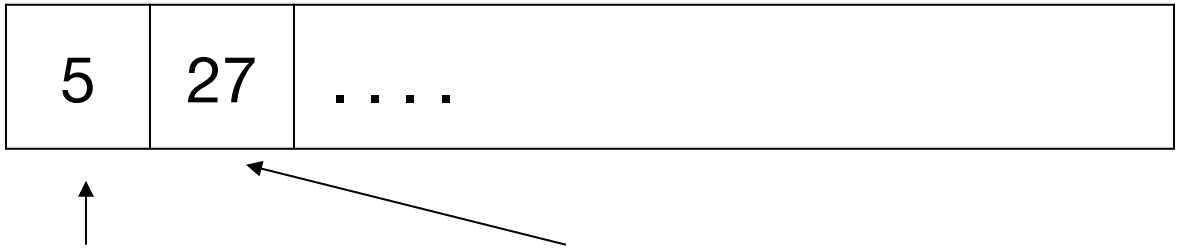
Repeating fields

Evolving formats

But may waste space...

Many Variants Between Fixed and Variable Format

Example: Include a **record type** in record



Type is a pointer to one of several schemas

Outline

Overview

Record encoding

Collection storage

C-Store paper

Indexes

Collection Storage Questions

How do we place data items and records for efficient access?

» **Locality** and **searchability**

How do we physically encode records in blocks and files?

Placing Data for Efficient Access

Locality: which items are accessed together

» When you read one field of a record, you're likely to read other fields of the same record

» When you read one field of record 1, you're likely to read the same field of record 2

Searchability: quickly find relevant records

» E.g. sorting the file lets you do binary search

Locality Example: Row Stores vs Column Stores

Row Store

name	age	state
Alex	20	CA
Bob	30	CA
Carol	42	NY
David	21	MA
Eve	26	CA
Frances	56	NY
Gia	19	MA
Harold	28	AK
Ivan	41	CA

Fields stored contiguously in one file

Column Store

name
Alex
Bob
Carol
David
Eve
Frances
Gia
Harold
Ivan

age
20
30
42
21
26
56
19
28
41

state
CA
CA
NY
MA
CA
NY
MA
AK
CA

Each column in a different file

Locality Example: Row Stores vs Column Stores

Row Store

name	age	state
Alex	20	CA
Bob	30	CA
Carol	42	NY
David	21	MA
Eve	26	CA
Frances	56	NY
Gia	19	MA
Harold	28	AK
Ivan	41	CA

Fields stored contiguously in one file

Column Store

name
Alex
Bob
Carol
David
Eve
Frances
Gia
Harold
Ivan

age
20
30
42
21
26
56
19
28
41

state
CA
CA
NY
MA
CA
NY
MA
AK
CA

Each column in a different file

Accessing all fields of one record: 1 random I/O for row, 3 for column

Locality Example: Row Stores vs Column Stores

Row Store

name	age	state
Alex	20	CA
Bob	30	CA
Carol	42	NY
David	21	MA
Eve	26	CA
Frances	56	NY
Gia	19	MA
Harold	28	AK
Ivan	41	CA

Fields stored contiguously in one file

Column Store

name
Alex
Bob
Carol
David
Eve
Frances
Gia
Harold
Ivan

age
20
30
42
21
26
56
19
28
41

state
CA
CA
NY
MA
CA
NY
MA
AK
CA

Each column in a different file

Accessing one field of all records: 3x less I/O for column store

Can We Have Hybrids Between Row & Column?

Yes! For example, colocated **column groups**:

name
Alex
Bob
Carol
David
Eve
Frances
Gia
Harold
Ivan

File 1

age	state
20	CA
30	CA
42	NY
21	MA
26	CA
56	NY
19	MA
28	AK
41	CA

File 2: age & state

Helpful if age & state are frequently co-accessed

Improving Searchability: Ordering

Ordering the data by a field will give:

» Closer I/Os if queries tend to read data with nearby values of the field (e.g. time ranges)

» Option to accelerate search via an ordered index (e.g. B-tree), binary search, etc

What's the downside of having an ordering?

Improving Searchability: Partitions

Just place data into buckets based on a field (but not necessarily fine-grained order)

E.g. Hive table storage over a filesystem:

```
/my_table/date=20190101/file1.parquet
                       /file2.parquet
         /date=20190102/file1.parquet
                       /file2.parquet
         /date=20190103/file1.parquet
                       ...
```

Easy to add, remove & list files in any directory

Can We Have Searchability on Multiple Fields at Once?

Yes! Many possible ways:

1) Multiple partition or sort keys (e.g., partition by date, then sort by userID)

2) Interleaved orderings such as Z-ordering

Z-Ordering

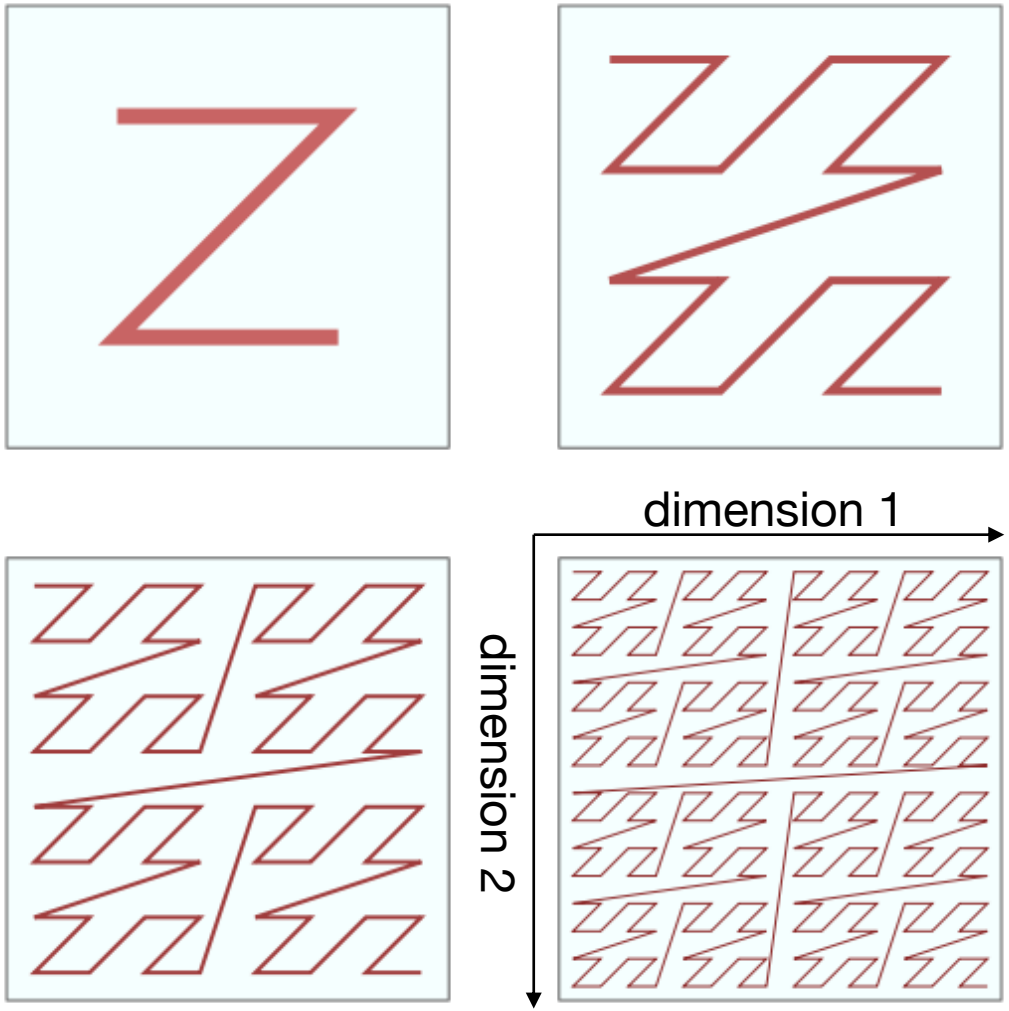
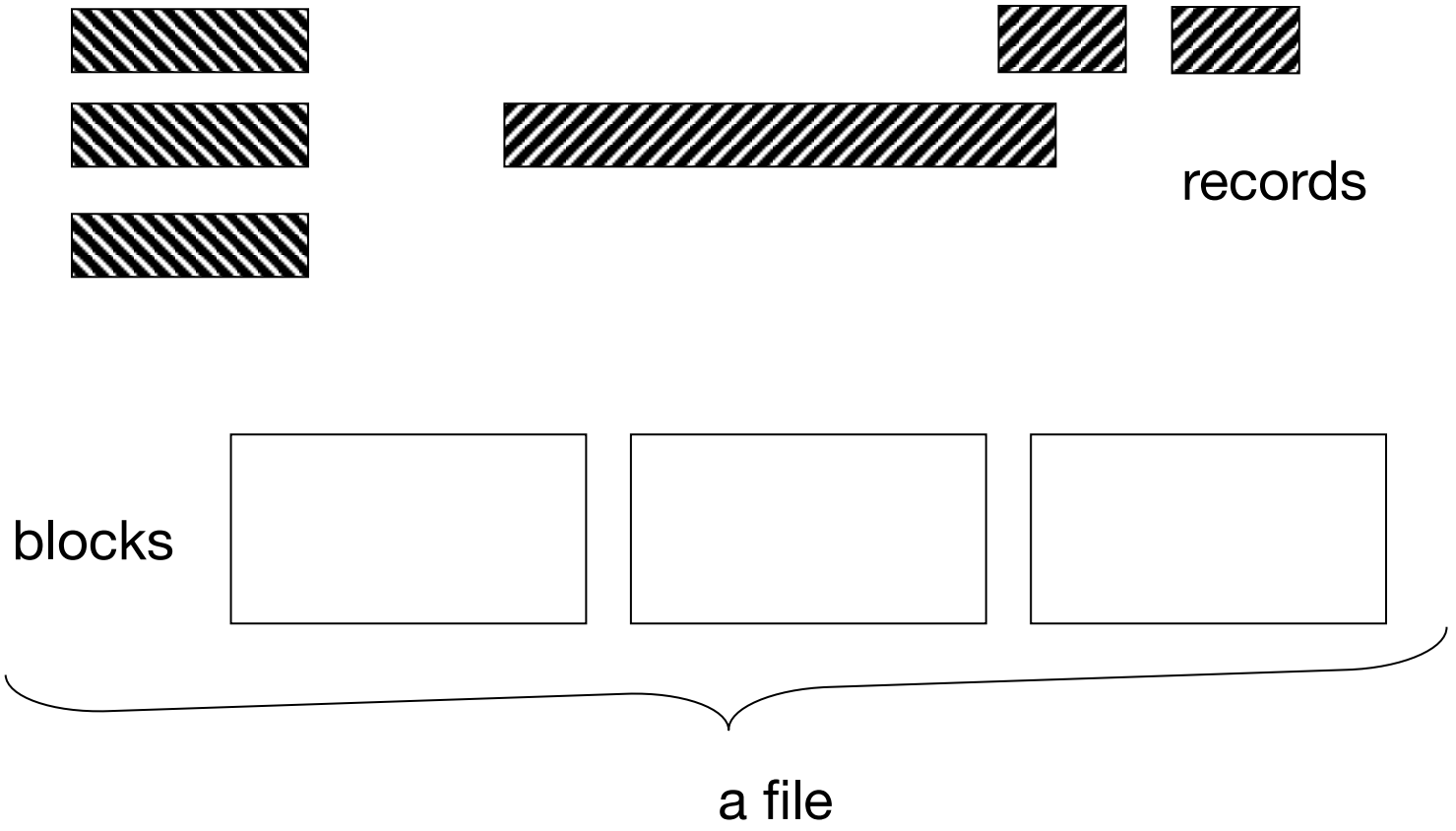


Image source: Wikipedia

How Do We Encode Records into Blocks & Files?



Questions in Storing Records

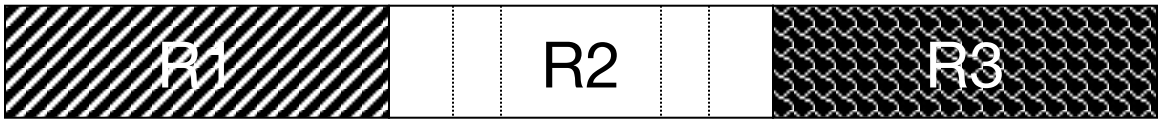
(1) separating records

(2) spanned vs. unspanned

(3) indirection

(1) Separating Records

Block



(a) no need to separate - fixed size recs.

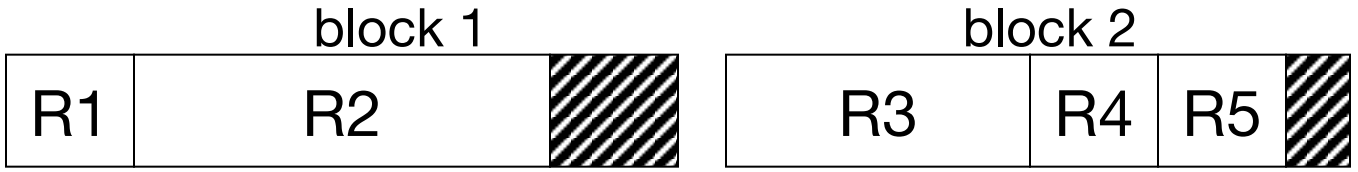
(b) special marker

(c) give record lengths (or offsets)

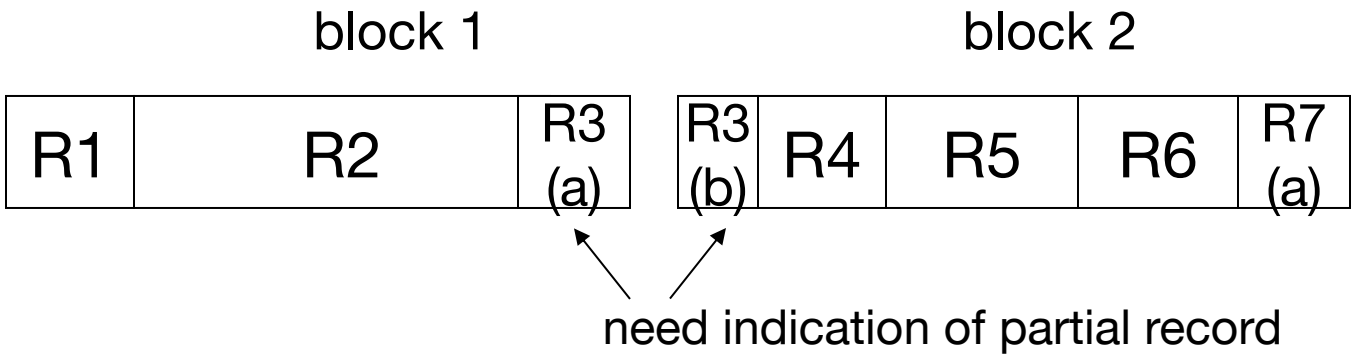
- within each record
- in block header

(2) Spanned vs Unspanned

Unspanned: records must be within one block



Spanned:



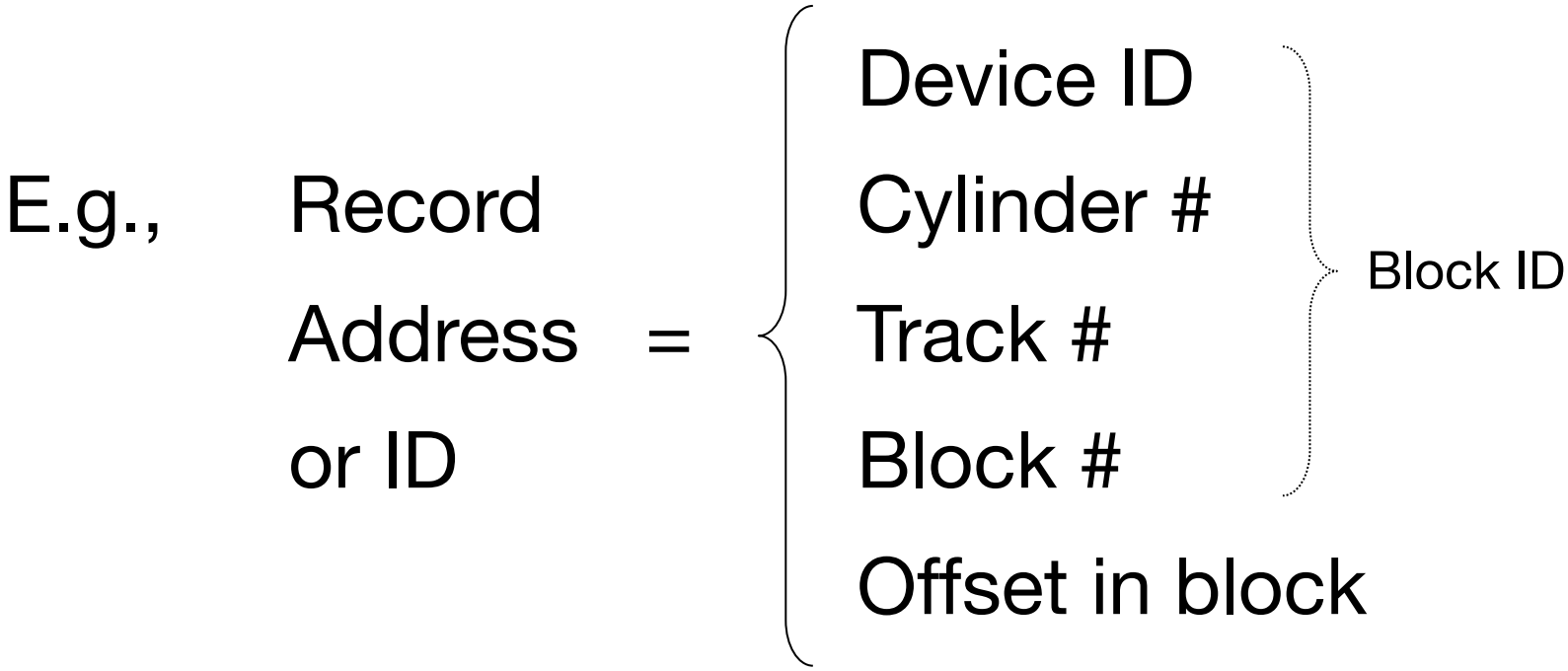
(3) Indirection

How does one refer to other records?

Many options:

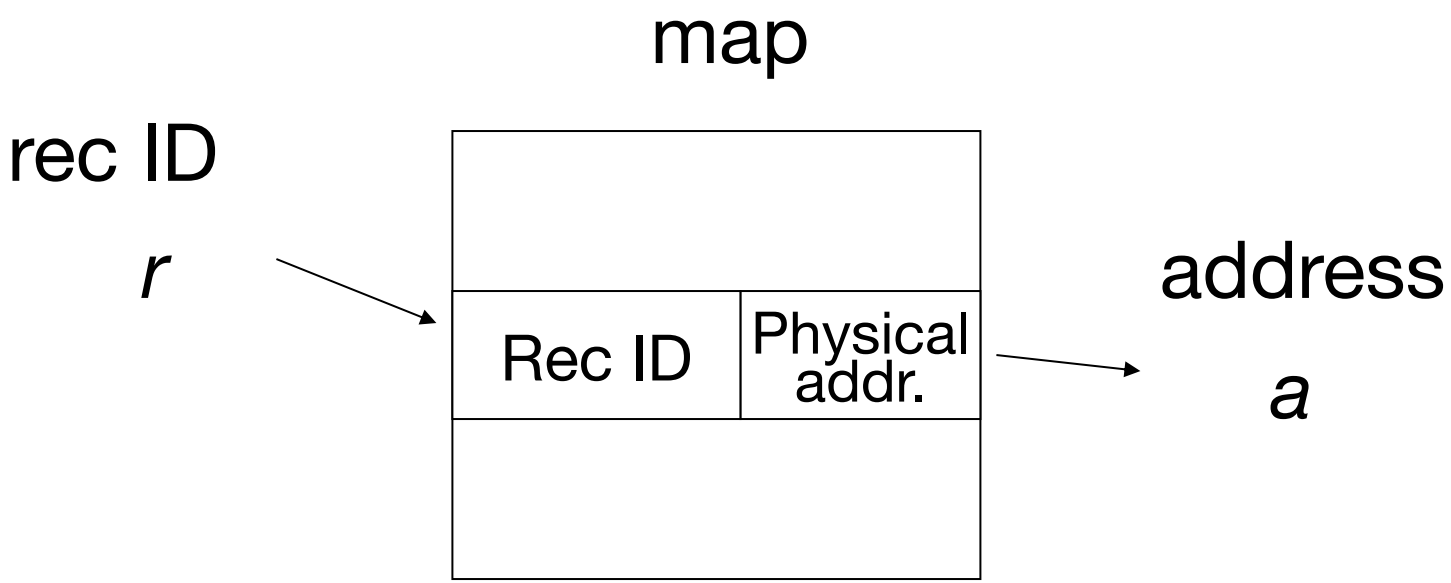
Physical ⟷ Indirect

Purely Physical



Fully Indirect

E.g., Record ID is arbitrary bit string



Tradeoff

Flexibility  Cost

to move records

of indirection

(for deletions, insertions)

Inserting Records

Easy case: records not ordered

» Insert record at end of file or in a free space

» Harder if records are variable-length

Hard case: records are ordered

» If free space close by, not too bad...

» Otherwise, use an **overflow** area and reorganize the file periodically

Deleting Records

Immediately reclaim space

OR

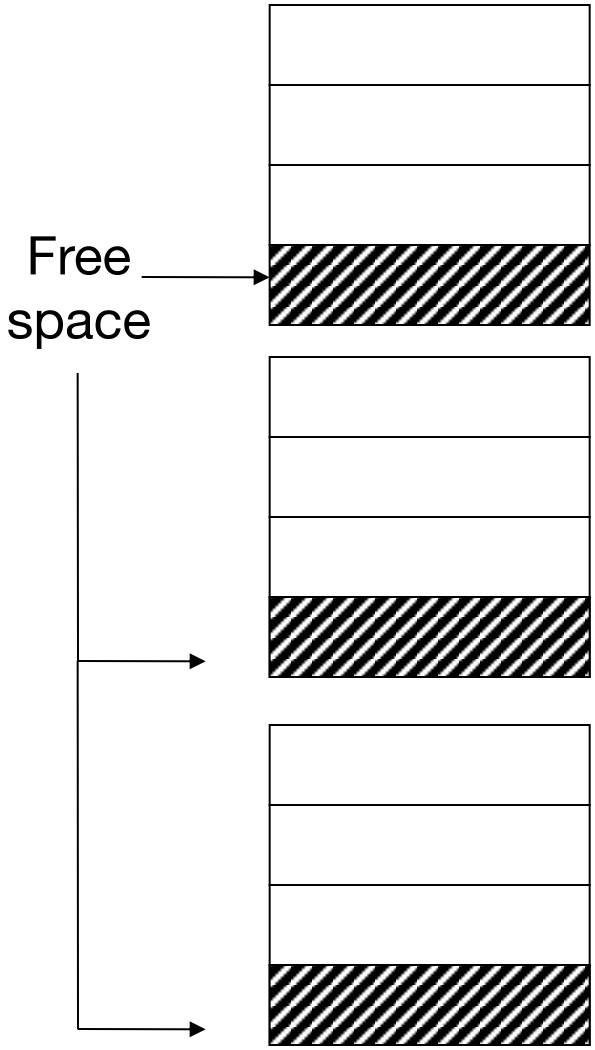
Mark deleted

– And keep track of freed spaces for later use

Interesting Problems

How much free space to leave in each block, track, cylinder, etc?

How often to reorganize file + merge overflow?



Compressing Collections

Usually for a block at a time

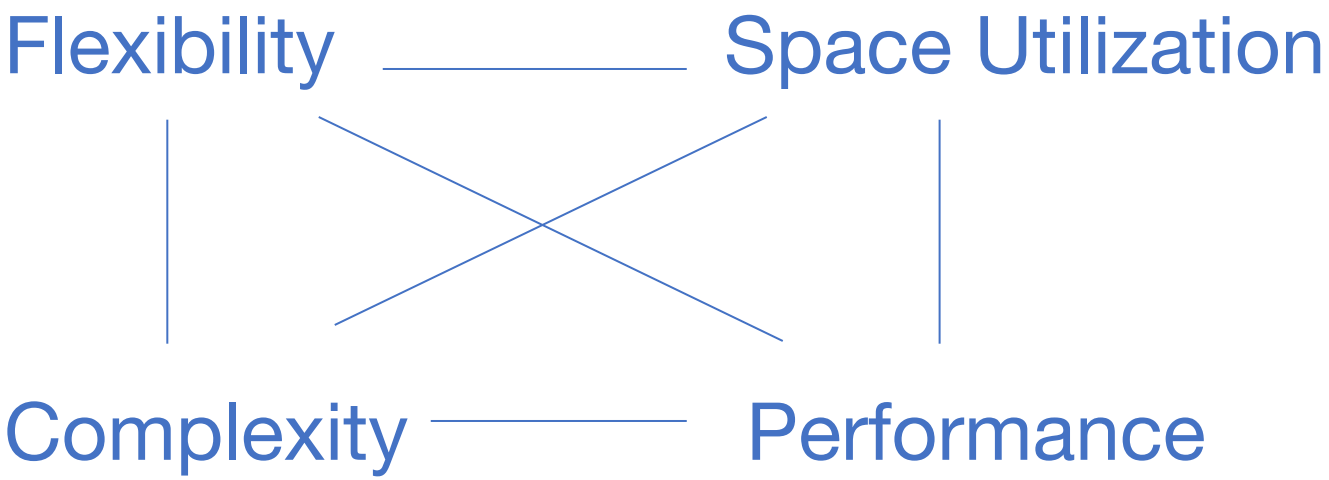
» Benefits from placing similar items together

Can be integrated with execution (C-Store)

Summary

There are many ways to organize data on disk

Key tradeoffs:



To Evaluate a Strategy, Compute:

Space used for expected data

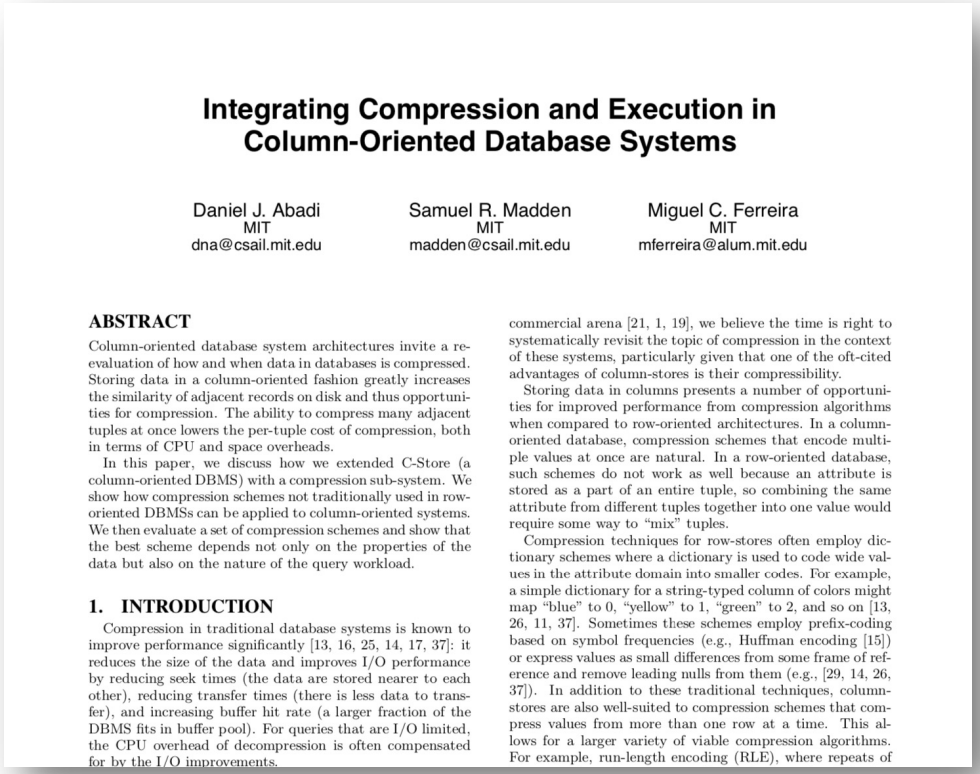
Expected time to

- fetch record given key
- read whole file
- insert record
- delete record
- update record
- reorganize file
- …

Reading for Next Class

"Integrating Compression and Execution in Column-Oriented Database Systems"

From the MIT C-Store project (led to Vertica)



Integrating Compression and Execution in Column-Oriented Database Systems

Daniel J. Abadi
MIT
dna@csail.mit.edu

Samuel R. Madden
MIT
madden@csail.mit.edu

Miguel C. Ferreira
MIT
mferreira@alum.mit.edu

ABSTRACT

Column-oriented database system architectures invite a re-evaluation of how and when data in databases is compressed. Storing data in a column-oriented fashion greatly increases the similarity of adjacent records on disk and thus opportunities for compression. The ability to compress many adjacent tuples at once lowers the per-tuple cost of compression, both in terms of CPU and space overheads.

In this paper, we discuss how we extended C-Store (a column-oriented DBMS) with a compression sub-system. We show how compression schemes not traditionally used in row-oriented DBMSs can be applied to column-oriented systems. We then evaluate a set of compression schemes and show that the best scheme depends not only on the properties of the data but also on the nature of the query workload.

1. INTRODUCTION

Compression in traditional database systems is known to improve performance significantly [13, 16, 25, 14, 17, 37]: it reduces the size of the data and improves I/O performance by reducing seek times (the data are stored nearer to each other), reducing transfer times (there is less data to transfer), and increasing buffer hit rate (a larger fraction of the DBMS fits in buffer pool). For queries that are I/O limited, the CPU overhead of decompression is often compensated for by the I/O improvements.

commercial arena [21, 1, 19], we believe the time is right to systematically revisit the topic of compression in the context of these systems, particularly given that one of the oft-cited advantages of column-stores is their compressibility.

Storing data in columns presents a number of opportunities for improved performance from compression algorithms when compared to row-oriented architectures. In a column-oriented database, compression schemes that encode multiple values at once are natural. In a row-oriented database, such schemes do not work as well because an attribute is stored as a part of an entire tuple, so combining the same attribute from different tuples together into one value would require some way to "mix" tuples.

Compression techniques for row-stores often employ dictionary schemes where a dictionary is used to code wide values in the attribute domain into smaller codes. For example, a simple dictionary for a string-typed column of colors might map "blue" to 0, "yellow" to 1, "green" to 2, and so on [13, 26, 11, 37]. Sometimes these schemes employ prefix-coding based on symbol frequencies (e.g., Huffman encoding [15]) or express values as small differences from some frame of reference and remove leading nulls from them (e.g., [29, 14, 26, 37]). In addition to these traditional techniques, column-stores are also well-suited to compression schemes that compress values from more than one row at a time. This allows for a larger variety of viable compression algorithms. For example, run-length encoding (RLE), where repeats of