Data Storage Formats

Instructor: Matei Zaharia

cs245.stanford.edu

Outline

Overview

Record encoding

Collection storage

Indexes

Outline

Overview

Record encoding

Collection storage

Indexes

Overview

Recall from last time: I/O slow compared to compute, random I/O ≪ sequential

Key concerns in storage:

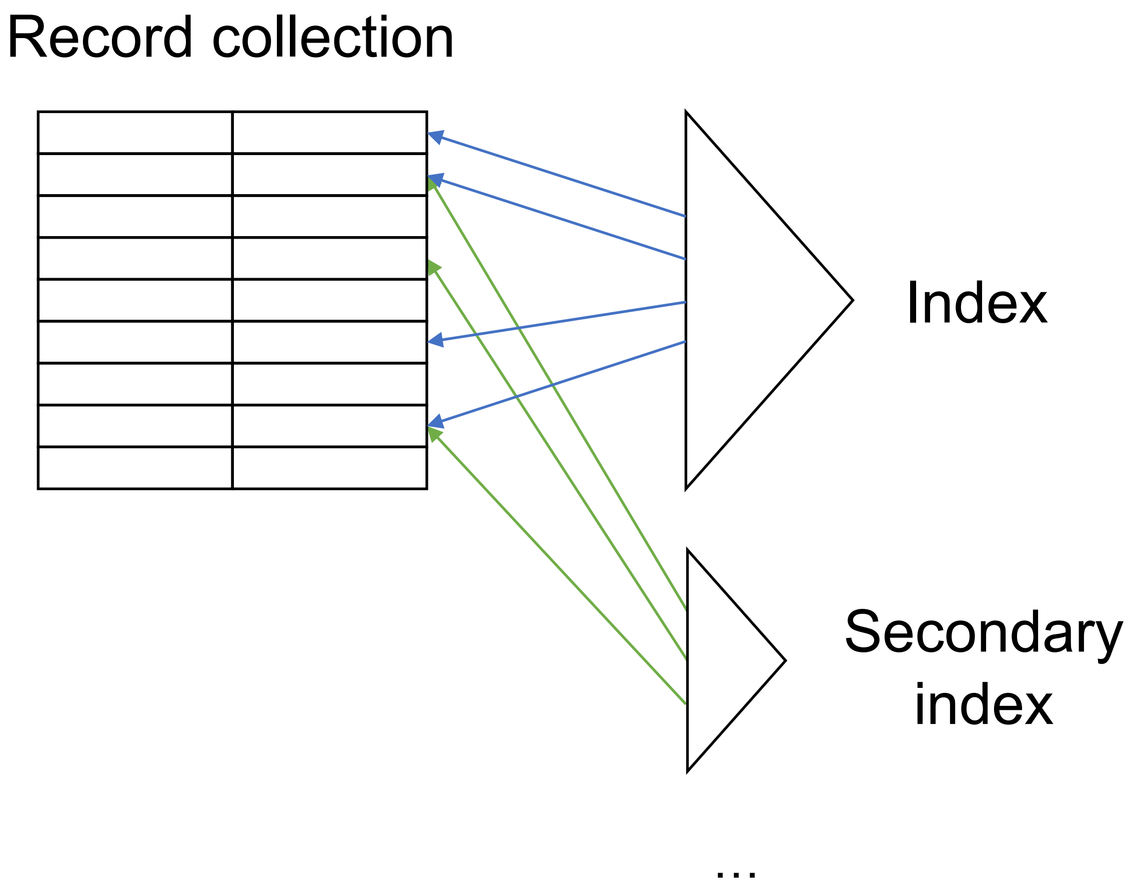
» **Access time:** minimize # of random accesses, bytes transferred, etc

- Main way: place co-accessed data together!

» **Size:** storage costs $

» **Ease of updates**

General Setup



Outline

Overview

Record encoding

Collection storage

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?

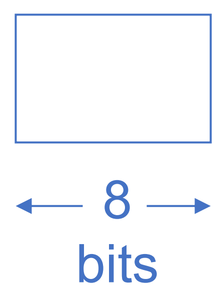
a salary

a name

a date

a picture

What we have available: **bytes**



To Represent:

Integer (short): 2 bytes

e.g., 35 is $\boxed{00000000}$ $\boxed{00100011}$

Real, floating point

n bits for mantissa, m for exponent….

To Represent:

Characters

→ Various coding schemes available

Example: ASCII

A: 1000001

a: 1100001

5: 0110101

LF: 0001010

To Represent:

Boolean

e.g., TRUE $\boxed{1111\ 1111}$

FALSE $\boxed{0000\ 0000}$

Application specific

e.g., RED $\rightarrow$ 1 GREEN $\rightarrow$ 3

BLUE $\rightarrow$ 2 YELLOW $\rightarrow$ 4 …

$\Rightarrow$ Can we use less than 1 byte/code?

Yes, but only if desperate...

To Represent:

Dates

e.g.: - Integer, # days since Jan 1, 1900

- 8 characters, YYYYMMDD
- 7 characters, YYYYDDD

Time

e.g. - Integer, seconds since midnight

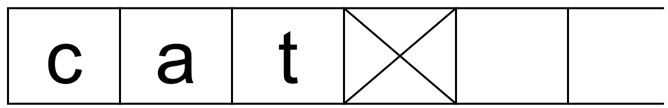
- characters, HHMMSSFF

To Represent:

String of characters

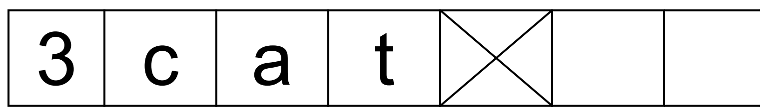
» Null terminated

e.g.,



» Length given

e.g.,



- Fixed length

To Represent:

Bag of bits

Length	Bits

To Represent:

To Represent: 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!

Key Point

- Fixed length items

- Variable length items
  - usually length given at beginning

Bigger Collections

Data Items

↓

Records

↓

Blocks

↓

Files

Record: Set of Related Data Items (“Fields”)

E.g.: Employee record:

name field,

salary field,

date-of-hire field, ...

Types of Records

Main choices:

» Fixed vs variable **format**

» Fixed vs variable **length**

Fixed Format

A **schema** (not record) contains following info:

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

Example: Fixed Format & Length

Employee record

(1) E#, 2 byte integer

(2) E.name, 10 char.

(3) Dept, 2 byte code

Schema

55	s m i t h	02

83	j o n e s	01

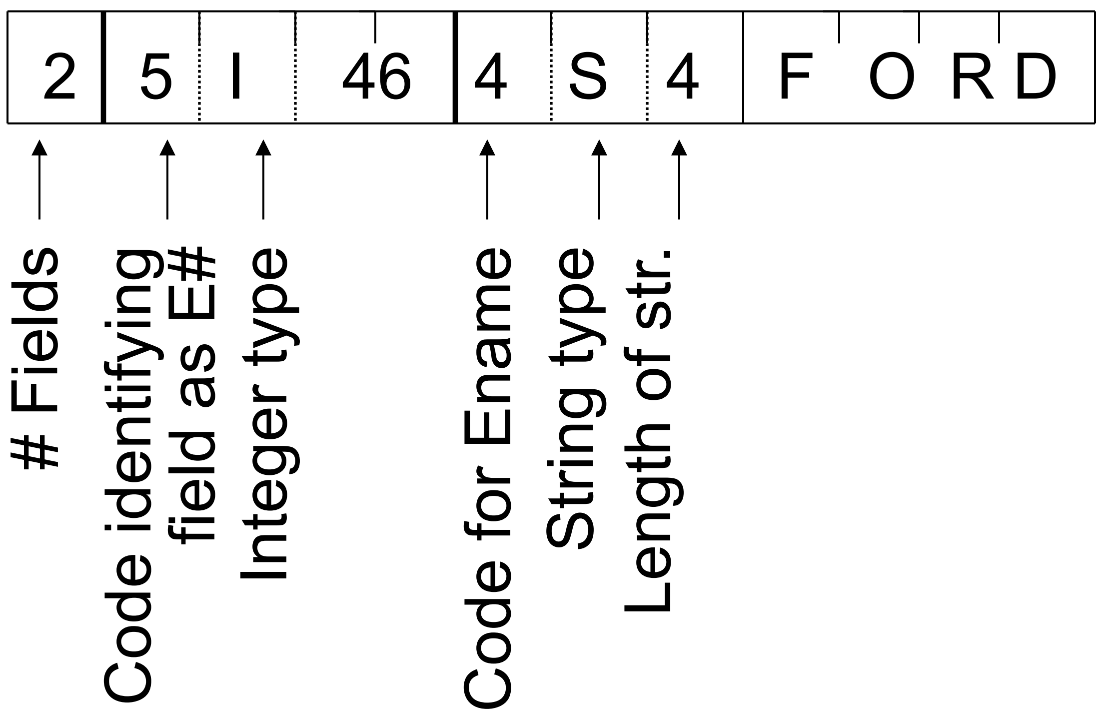
Records

Variable Format

Record itself contains format

“Self Describing”

Example: Variable Format & Length



Field name codes could also be strings, i.e. TAGS

Variable Format Useful For

"Sparse" records

Repeating fields

Evolving formats

But may waste space...

Example: Variable Format Record with Repeated Fields

Employee $\rightarrow$ one or more $\rightarrow$ children

3	E_name: Fred	Child: Sally	Child: Tom

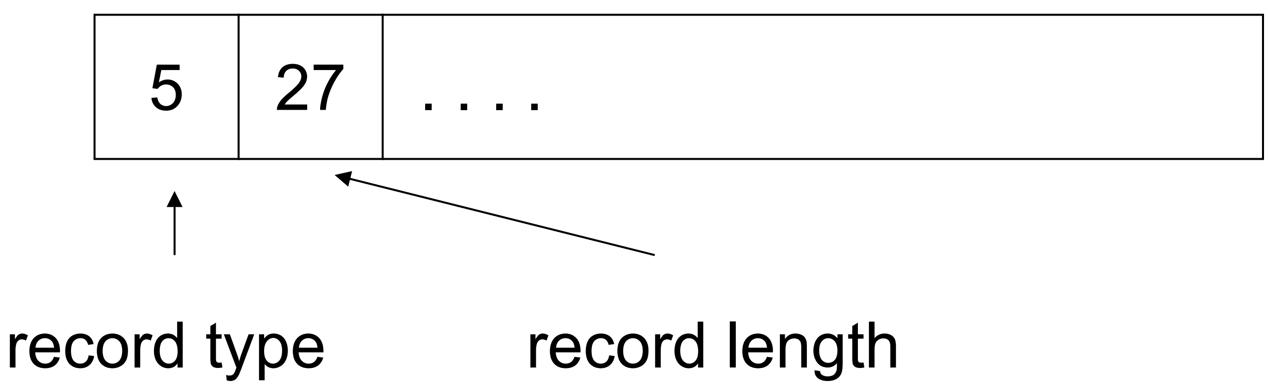
Note: Repeated Fields Does Not Imply Variable Format/Length

Could have fixed space for a max # of items and their sizes

John	Sailing	Chess	(null)

Many Variants Between Fixed and Variable Format

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



Type is a pointer to one of several schemas

Record Header: Data at Start that Describes a Record

May contain:

- record type
- record length
- timestamp
- concurrency stuff ...

Exercise: How to Store JSON Data?

```
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "address": {
     "streetAddress": "21 2nd Street",
     "city": "New York",
     "state": "NY",
     "postalCode": "10021”
   },
  "phoneNumbers": [
     { "type": "home", "number": "212-555-1234" },
     { "type": "fax", "number": "646-555-4567" }
  ]
}
```

Other Issues

Compression

» Within record: e.g. encoding selection

» Collection of records: use common patterns

Encryption

» Usually operates on large blocks

Outline

Overview

Record encoding

Collection storage

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 filesystem or S3:

```
/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 data by date, then sort by customer ID)

2) Interleaved orderings such as Z-ordering

Z-Ordering

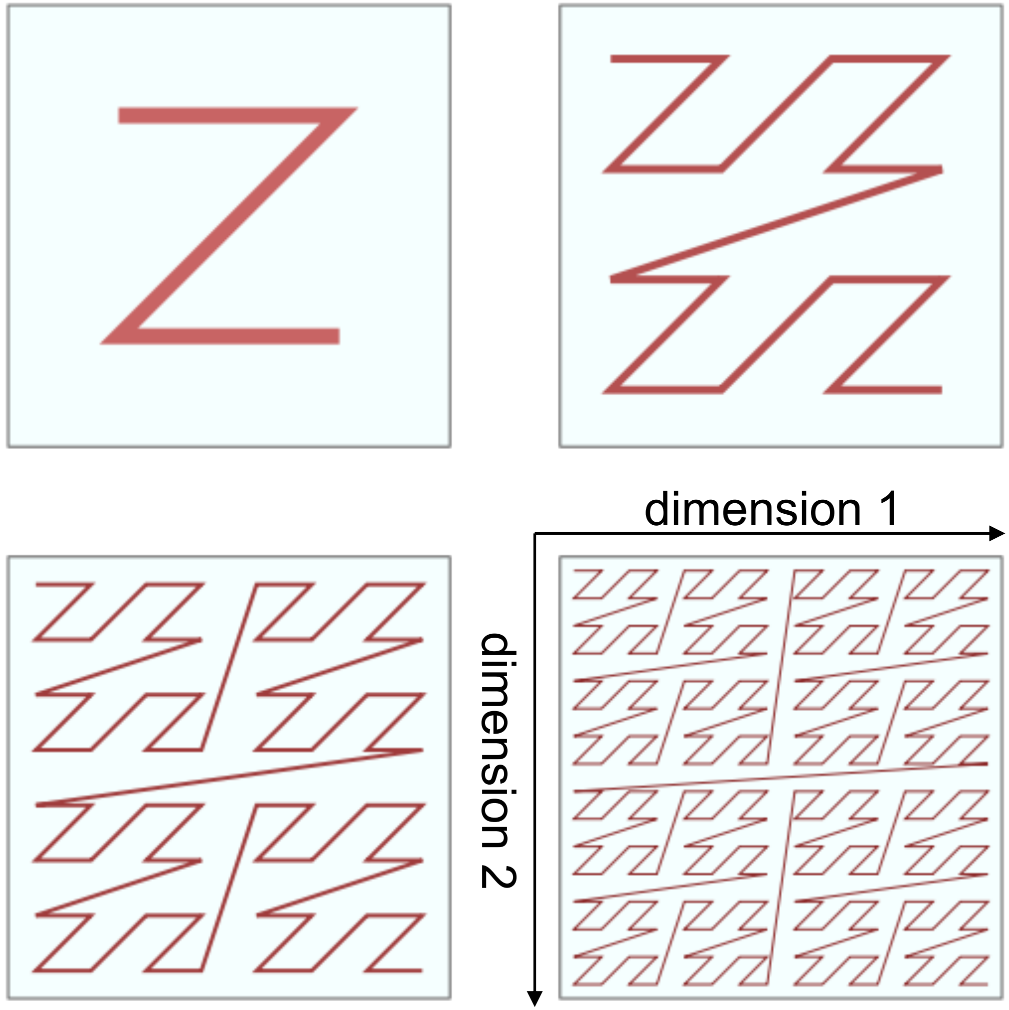
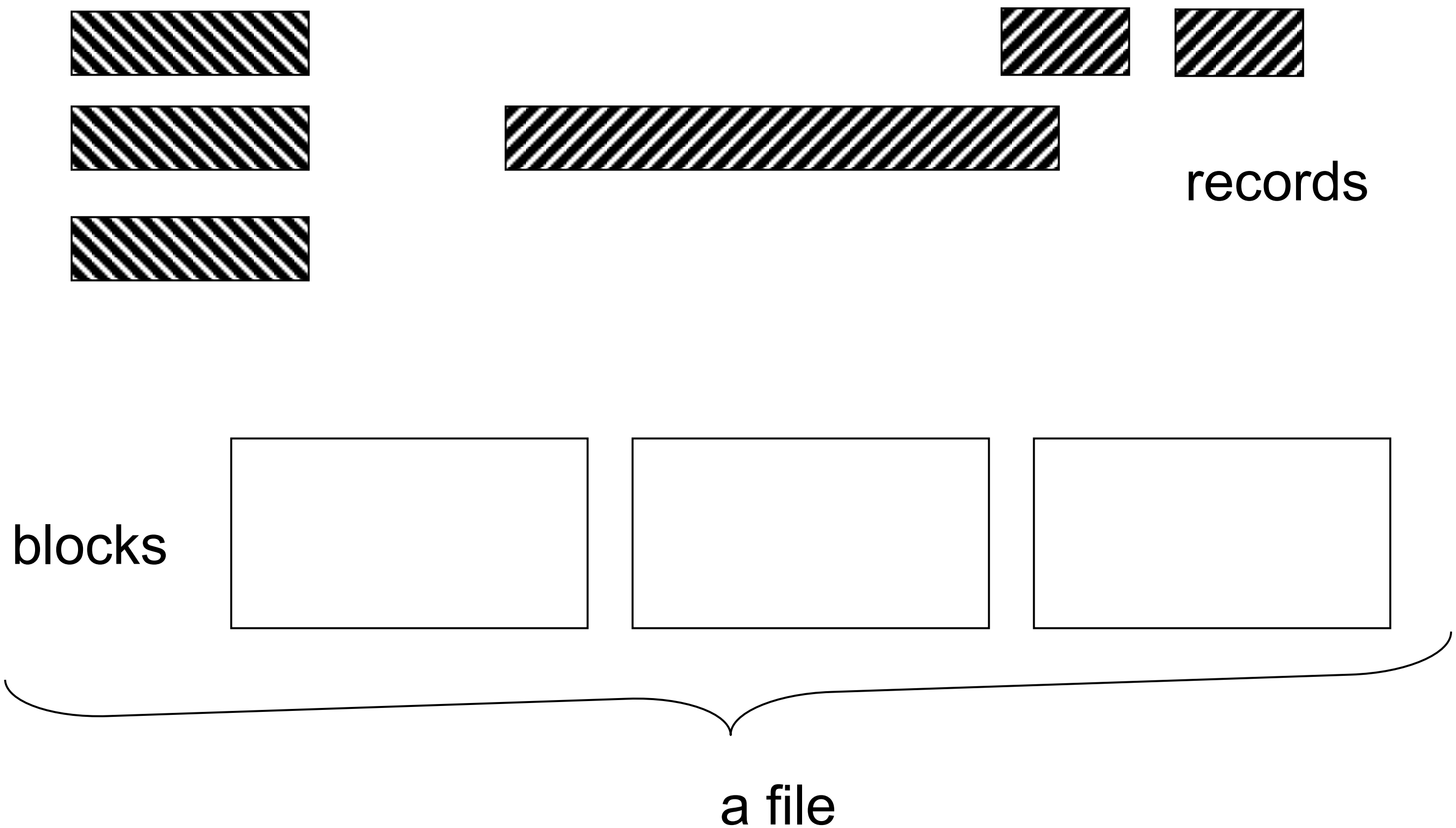


Image source: Wikipedia

How Do We Encode Records into Blocks & Files?

How Do We Encode Records into Blocks & Files?



Questions in Storing Records

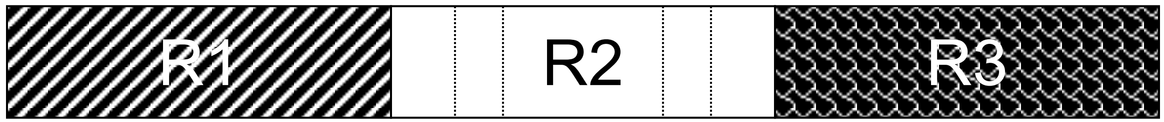
(1) separating records

(2) spanned vs. unspanned

(3) indirection

(1) Separating Records

Block



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

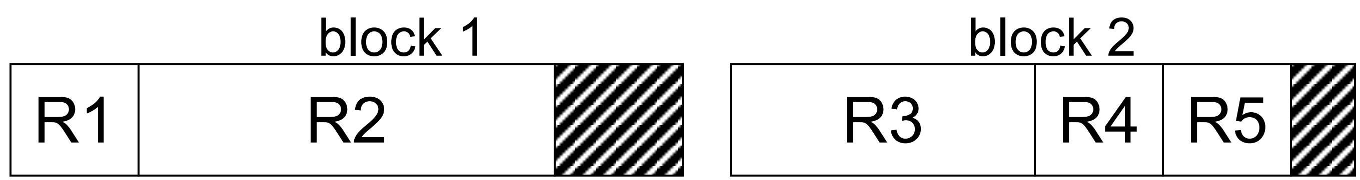
(b) special marker

(c) give record lengths (or offsets)

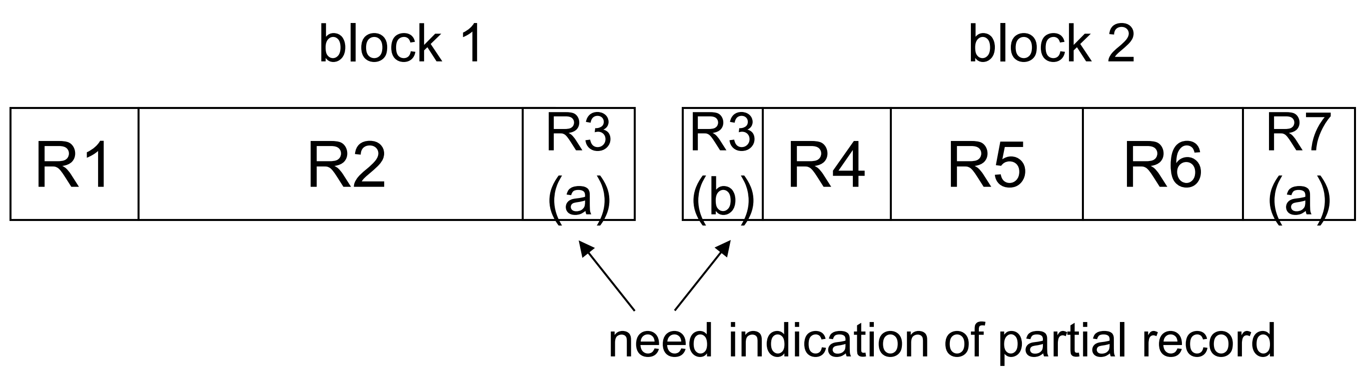
- within each record
- in block header

(2) Spanned vs Unspanned

Unspanned: records must be within one block



Spanned:



Spanned vs Unspanned

Unspanned is **much** simpler, but may waste storage space…

Spanned essential if record size > block size

(4) Indirection

How does one refer to specific records?
(e.g. in metadata or in other records)

(4) Indirection

How does one refer to records?

Many options:

Physical ⟺ Indirect

Purely Physical

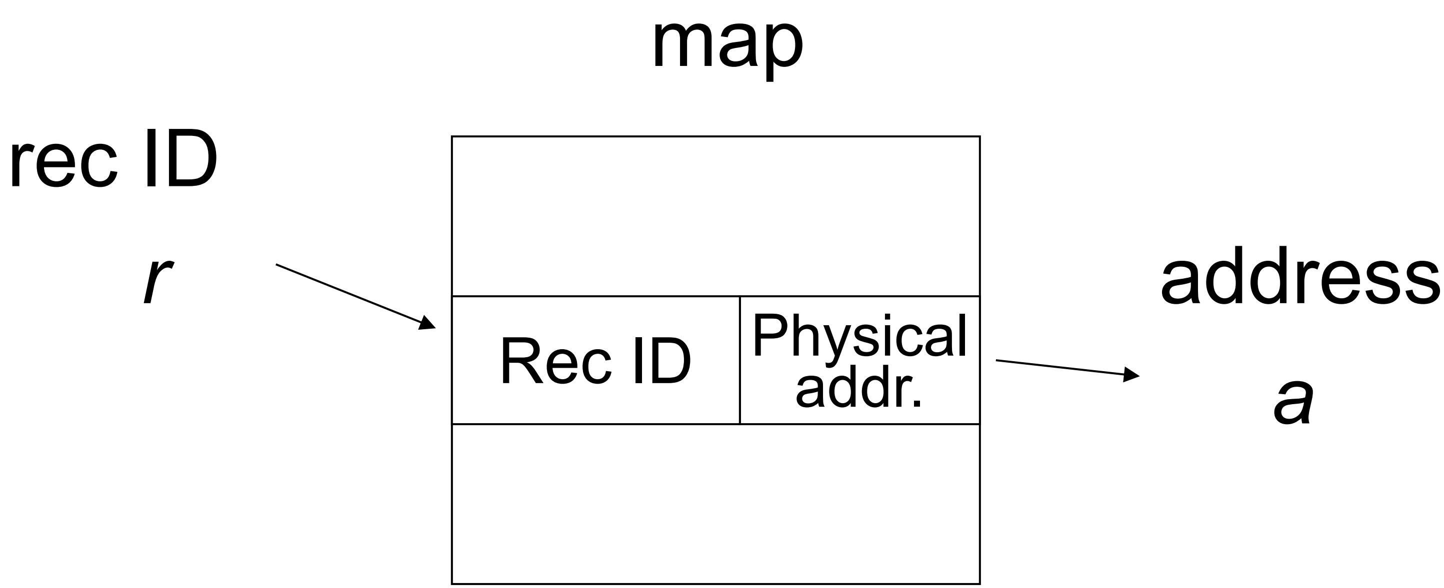
E.g., Record Address or ID =

- Device ID
- Cylinder #
- Track #
- Block #
- Offset in block

$\left.\begin{array}{l}\text{Device ID}\\ \text{Cylinder \#}\\ \text{Track \#}\\ \text{Block \#}\end{array}\right\}$ Block ID

Fully Indirect

E.g., Record ID is arbitrary bit string



Tradeoff

Flexibility ⟷ Cost

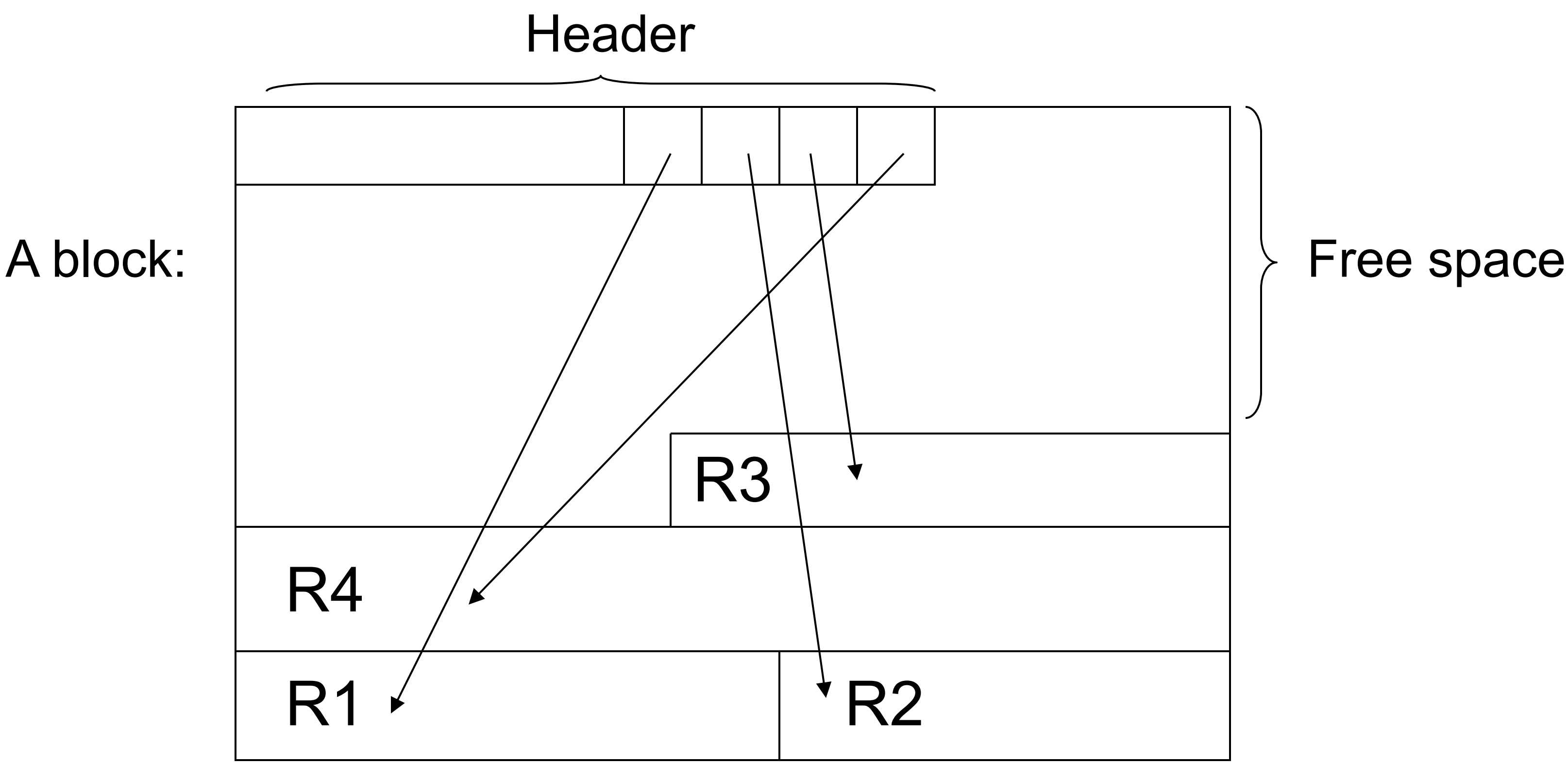
to move records of indirection

(for deletions, insertions)

Physical $\longleftrightarrow$ Indirect

Many options
in between …

Example: Indirection in Block

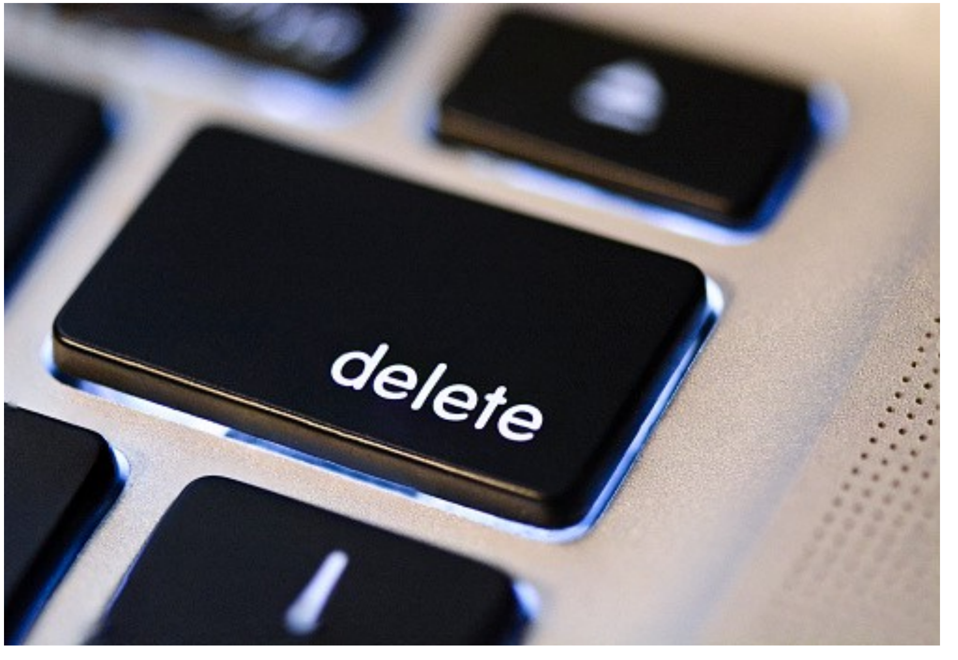


Block Header: Data at Start that Describes Block

May contain:

- File ID (or table or database ID)
- This block ID
- Record directory
- Pointer to free space
- Type of block (e.g. contains recs type 4)
- Pointer to other blocks “like it”
- Timestamp ...

Other Concern: Deletion!



Options

(a) Immediately reclaim space

(b) Mark deleted

Options

(a) Immediately reclaim space

(b) Mark deleted

- May need chain of deleted records (for space re-use)
- Need a way to mark:
  - special characters
  - delete field
  - entries in maps

As Usual, Many Tradeoffs

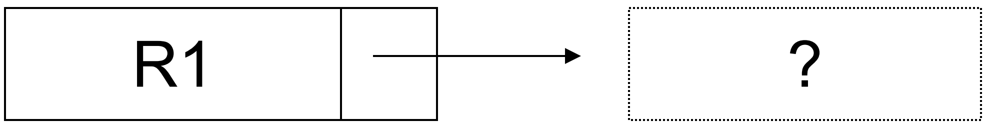
How expensive is to move valid record to free space for immediate reclaim?

How much space is wasted?

» e.g., deleted records, delete fields, free space chains,...

Concern with Deletions

Dangling pointers



Solution 1: Do Not Worry

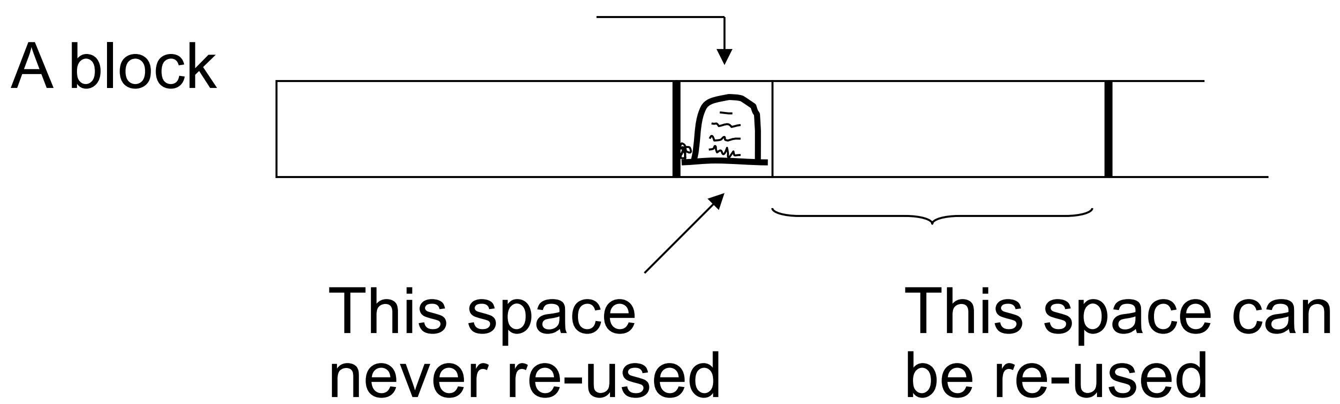
Solution 2: Tombstones

Special mark in old location or mappings

Solution 2: Tombstones

Special mark in old location or mappings

Physical IDs:



Solution 2: Tombstones

Special mark in old location or mappings

Logical IDs:

map

ID	LOC
7788	

Never reuse
ID 7788 nor
space in map...

Insertion

Easy case: records not ordered

→ Insert new record at end of file or in a deleted slot

→ If records are variable size, not as easy...

Insertion

Hard case: records are ordered

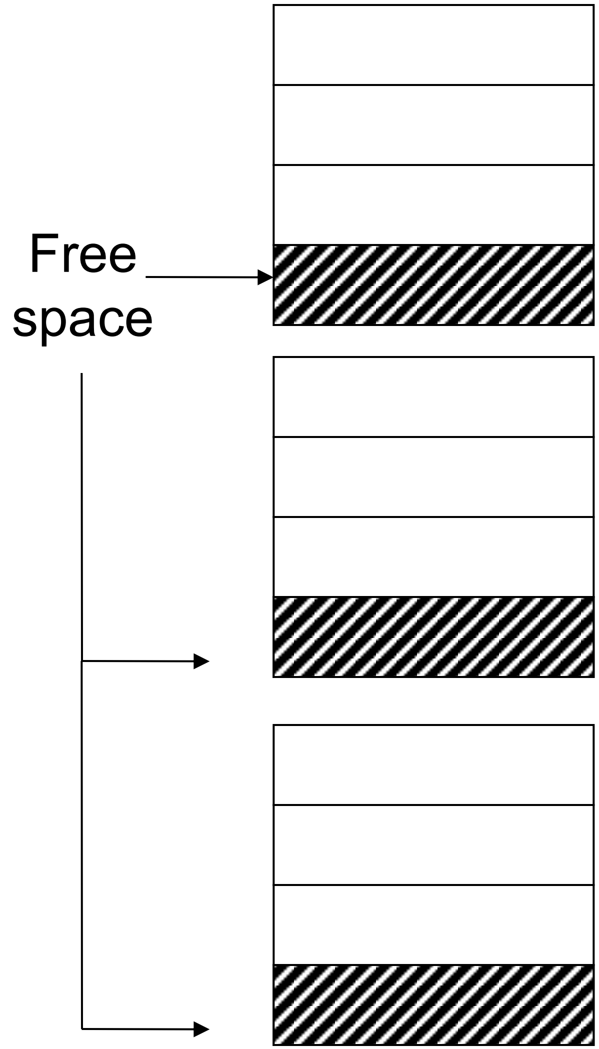
$\rightarrow$ If free space close by, not too bad...

$\rightarrow$ Otherwise, use an **overflow** area?

Interesting Problems

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

How often do I reorganize file + overflow?

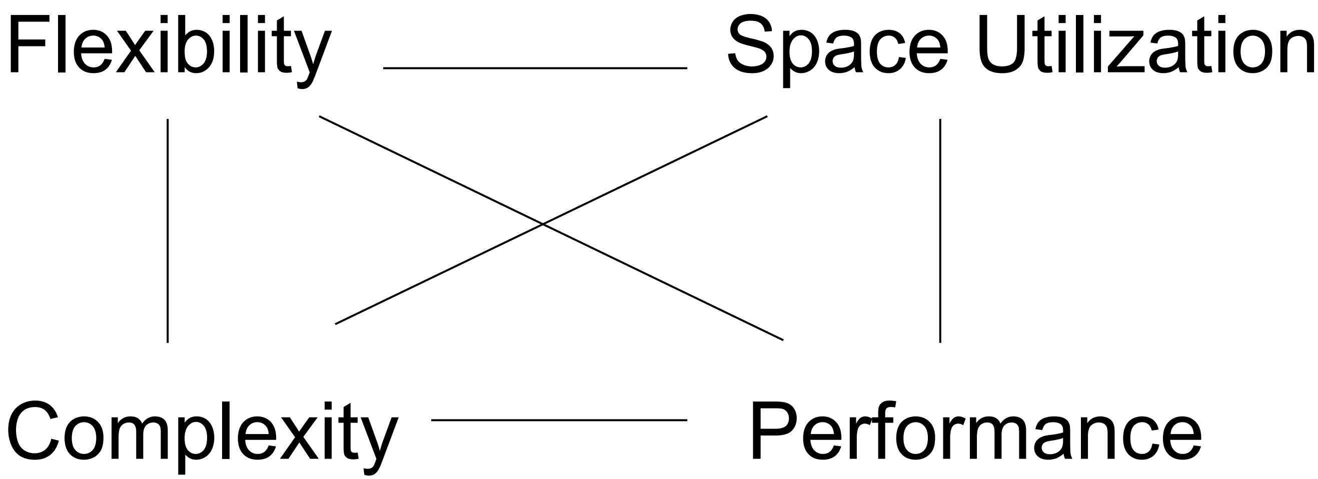


Summary

There are 10,000,000 ways to organize my data on disk…

Which is right for me?

Issues



To Evaluate a Strategy, Compute:

Space used for expected data

Expected time to

- fetch record given key
- fetch record with next key
- insert record
- append record
- delete record
- update record
- read all file
- 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