CS 525: Advanced Database Organization

03: Disk Organization

Boris Glavic

Slides: adapted from a course taught by Hector Garcia-Molina, Stanford InfoLab
Topics for today

• How to lay out data on disk
• How to move it to/from memory
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

  00000000  00100011

  Endian! Could as well be

  00100011  00000000

- Real, floating point
  $n$ bits for mantissa, $m$ for exponent....
To represent:

- Characters

  → various coding schemes suggested, most popular is ASCII (1 byte encoding)

Example:

A:  1000001
a:  1100001
5:  0110101
LF: 0001010
To represent:

- Boolean
  
  e.g., TRUE  \[1111\ 1111\]
  FALSE  \[0000\ 0000\]

- Application specific
  
  e.g., enumeration
  
  RED \[\rightarrow\] 1  GREEN \[\rightarrow\] 3
  BLUE \[\rightarrow\] 2  YELLOW \[\rightarrow\] 4  ...
To represent:

• Boolean
e.g., TRUE 1111 1111
   FALSE 0000 0000

• Application specific
e.g., RED → 1   GREEN → 3
   BLUE → 2   YELLOW → 4 ...

➡️ 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
    (not YYMMDD! Why?)

• Time
  e.g.  - Integer, seconds since midnight
  - characters, HHMMSSFF
To represent:

- String of characters
  - Null terminated
e.g.,
  
  \[
  \text{c a t}
  \]

  - Length given
e.g.,
  
  \[
  3 \text{ c a t}
  \]

- Fixed length
To represent:

- Bag of bits

<table>
<thead>
<tr>
<th>Length</th>
<th>Bits</th>
</tr>
</thead>
</table>

Key Point

- Fixed length items
- Variable length items
  - usually length given at beginning
Also

• Type of an item: Tells us how to interpret (plus size if fixed)
Overview

Data Items

Records

Blocks

Files

Memory
Record - Collection of related data items (called 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
A SCHEMA (not record) contains following information
- # fields
- type of each field
- order in record
- meaning of each field
Example: fixed format and length

Employee record
(1) E#, 2 byte integer
(2) E.name, 10 char.
(3) Dept, 2 byte code

Schema

Records

55 smith 02
83 jones 01
Variable format

- Record itself contains format “Self Describing”
Example: variable format and 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...

Additional indirection...
• **EXAMPLE:** var format record with repeating fields

Employee → one or more → children

<table>
<thead>
<tr>
<th></th>
<th>E_name: Fred</th>
<th>Child: Sally</th>
<th>Child: Tom</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Note:** Repeating fields does not imply
- variable format, nor
- variable size

<table>
<thead>
<tr>
<th>John</th>
<th>Sailing</th>
<th>Chess</th>
<th>--</th>
</tr>
</thead>
</table>
Note: Repeating fields does not imply
- variable format, nor
- variable size

| John | Sailing | Chess | -- |

• Key is to allocate maximum number of repeating fields (if not used → null)
Many variants between fixed - variable format:

Example: Include record type in record

| 5 | 27 | . . . |

record type      record length

tells me what
to expect
(i.e. points to schema)
Record header - data at beginning that describes record

May contain:

- record type
- record length
- time stamp
- null-value bitmap
- other stuff ...
Other interesting issues:

• Compression
  – within record - e.g. code selection
  – collection of records - e.g. find common patterns

• Encryption

• Splitting of large records
  – E.g., image field, store pointer
Record Header – null-map

• SQL: NULL is special value for every data type
  – Reserve one value for each data type as NULL?

• Easier solution
  – Record header has a bitmap to store whether field is NULL
  – Only store non-NULL fields in record
Separate Storage of Large Values

- Store fields with large values separately
  - E.g., image or binary document
  - Records have pointers to large field content

- Rationale
  - Large fields mostly not used in search conditions
  - Benefit from smaller records
Next: placing records into blocks

blocks

...
Next: placing records into blocks

blocks

... 

a file

assume fixed length blocks

assume a single file (for now)
Options for storing records in blocks:

(1) separating records
(2) spanned vs. unspanned
(3) sequencing
(4) indirection
(1) Separating records

- within each record
- in block header
(2) Spanned vs. Unspanned

- **Unspanned**: records must be within one block

  block 1
  
  R1   R2
  
  block 2
  
  R3   R4   R5
  
  ...  ...  ...

- **Spanned**

  block 1
  
  R1   R2   R3 (a)
  
  block 2
  
  R3 (b) R4   R5   R6   R7 (a)
  
  ...  ...  ...


With spanned records:

need indication of partial record “pointer” to rest

need indication of continuation (+ from where?)
Spanned vs. unspanned:

- Unspanned is much simpler, but may waste space...
- Spanned essential if record size > block size
(3) Sequencing

- Ordering records in file (and block) by some key value

Sequential file \( \Rightarrow \) sequenced
Why sequencing?

Typically to make it possible to efficiently read records in order (e.g., to do a merge-join — discussed later)
Sequencing Options

(a) Next record physically contiguous

(b) Linked
Sequencing Options

(c) Overflow area

Records in sequence

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td></td>
</tr>
<tr>
<td>R4</td>
<td></td>
</tr>
<tr>
<td>R5</td>
<td></td>
</tr>
</tbody>
</table>
Sequencing Options

(c) Overflow area

Records in sequence

<table>
<thead>
<tr>
<th>header</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
</tr>
<tr>
<td>R2</td>
</tr>
<tr>
<td>R3</td>
</tr>
<tr>
<td>R4</td>
</tr>
<tr>
<td>R5</td>
</tr>
</tbody>
</table>

- R2.1
- R1.3
- R4.7
(4) Indirection

- How does one refer to records?

  \[ \text{Rx} \]
(4) Indirection

• How does one refer to records?

Many options:

Physical  ↔  Indirect

Rx
Purely Physical

E.g., Record Address = { Device ID, Cylinder #, Track #, Block #, Offset in block }
★ Fully Indirect

E.g., Record ID is arbitrary bit string

```
<table>
<thead>
<tr>
<th>Rec ID</th>
<th>Physical addr.</th>
</tr>
</thead>
</table>
```

map

```
rec ID
```

address
Tradeoff

Flexibility \rightarrow \text{Cost} \rightarrow \text{to move records of indirection} \rightarrow \text{(for deletions, insertions)}
Physical $\rightarrow$ Indirect

Many options in between ...
Block header - data at beginning that describes block

May contain:
- File ID (or RELATION or DB ID)
- This block ID
- Record directory
- Pointer to free space
- Type of block (e.g. contains recs type 4; is overflow, ...)
- Pointer to other blocks “like it”
- Timestamp ...
Example: Indirection in block

A block:

Header

Free space

R3

R4

R1

R2
Tuple Identifier (TID)

- TID is
  - Page identifier
  - Slot number
- Slot stores either record or pointer (TID)
- TID of a record is fixed for all time
TID Operations

• Insertion
  – Set TID to record location (page, slot)

• Moving record
  – e.g., update variable-size or reorganization
  – Case 1: TID points to record
    • Replace record with pointer (new TID)
  – Case 2: TID points to pointer (TID)
    • Replace pointer with new pointer
TID: Block 1, Slot 2

Block 1

Block 2
Move record to Block 2 slot 3 -> TID does not change!

TID: Block 1, Slot 2
Move record again to Block 2 slot 2
-> still one level of indirection

TID: Block 1, Slot 2
TID Properties

• TID of record never changes
  – Can be used safely as pointer to record (e.g., in index)

• At most one level of indirection
  – Relatively efficient
  – Changes to physical address - changing max 2 pages
Options for storing records in blocks:

(1) separating records
(2) spanned vs. unspanned
(3) sequencing
(4) indirection
Other Topics

(1) Insertion/Deletion
(2) Buffer Management
(3) Comparison of Schemes
Deletion

Block

Rx
Options:

(a) Immediately reclaim space
(b) Mark deleted
Options:

(a) Immediately reclaim space
(b) Mark deleted
   – May need chain of deleted records
     (for re-use)
   – Need a way to mark:
     • special characters
     • delete field
     • in map
As usual, many tradeoffs...

• How expensive is it 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

R1 → ?
Solution #1: Do not worry
Solution #2: Tombstones

E.g., Leave “MARK” in map or old location
Solution #2: Tombstones

E.g., Leave “MARK” in map or old location

• Physical IDs

A block

This space never re-used

This space can be re-used
Solution #2: Tombstones

E.g., Leave “MARK” in map or old location

• Logical IDs

<table>
<thead>
<tr>
<th>ID</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>7788</td>
<td>⚰️</td>
</tr>
</tbody>
</table>

Never reuse ID 7788 nor space in map...
Insert

Easy case: records not in sequence
→ Insert new record at end of file or in deleted slot
→ If records are variable size, not as easy...
Hard case: records in sequence
  → If free space “close by”, not too bad...
  → Or use overflow idea...
Interesting problems:

- How much free space to leave in each block, track, cylinder?
- How often do I reorganize file + overflow?
Buffer Management

- For Caching of Disk Blocks
- Buffer Replacement Strategies
  - E.g., LRU, clock
- Pinned blocks
- Forced output
- Double buffering
- Swizzling

\[\text{in Notes02}\]
Buffer Manager

- Manages blocks cached from disk in main memory
- Usually -> fixed size buffer (M pages)
- DB requests page from Buffer Manager
  - Case 1: page is in memory -> return address
  - Case 2: page is on disk -> load into memory, return address
Goals

• Reduce the amount of I/O
• Maximize the *hit rate*
  – Ratio of number of page accesses that are fulfilled without reading from disk
• -> Need strategy to decide when to
Buffer Manager Organization

- **Bookkeeping**
  - Need to map (hash table) page-ids to locations in buffer (*page frames*)
  - Per page store *fix count, dirty bit*, ...
  - Manage free space

- **Replacement strategy**
  - If page is requested but buffer is full
  - Which page to emit remove from buffer
FIFO

- First In, First Out
- Replace page that has been in the buffer for the longest time
- Implementation: E.g., pointer to oldest page (circular buffer)
  - Pointer->next = Pointer++ % M
- Simple, but not prioritizing frequently accessed pages
LRU

- Least Recently Used
- Replace page that has not been accessed for the longest time
- Implementation:
  - List, ordered by LRU
  - Access a page, move it to list tail
- Widely applied and reasonable performance
Clock

• Frames are organized clock-wise
• Pointer S to current frame
• Each frame has a reference bit
  – Page is loaded or accessed \( \rightarrow \) bit = 1
• Find page to replace (advance pointer)
  – Return first frame with bit = 0
  – On the way set all bits to 0
Clock Example

Reference bit

<table>
<thead>
<tr>
<th>S</th>
<th>Page 0</th>
<th>Page 1</th>
<th>Page 2</th>
<th>Page 3</th>
<th>Page 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Other Replacement Strategies

• LRU-K
• GCLOCK
• Clock-Pro
• ARC
• LFU
Swizziling

Memory

Disk

block 1

block 1

block 2

Rec A
Swizzling

Memory

Disk

block 1

Rec A

block 1

block 2

Rec A

block 2
Row vs Column Store

• So far we assumed that fields of a record are stored contiguously (row store)...  
• Another option is to store all values of a field together (column store)
Row Store

- Example: Order consists of
  - id, cust, prod, store, price, date, qty

<table>
<thead>
<tr>
<th>id</th>
<th>cust</th>
<th>prod</th>
<th>store</th>
<th>price</th>
<th>date</th>
<th>qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1</td>
<td>cust1</td>
<td>prod1</td>
<td>store1</td>
<td>price1</td>
<td>date1</td>
<td>qty1</td>
</tr>
<tr>
<td>id2</td>
<td>cust2</td>
<td>prod2</td>
<td>store2</td>
<td>price2</td>
<td>date2</td>
<td>qty2</td>
</tr>
<tr>
<td>id3</td>
<td>cust3</td>
<td>prod3</td>
<td>store3</td>
<td>price3</td>
<td>date3</td>
<td>qty3</td>
</tr>
</tbody>
</table>
Column Store

- Example: Order consists of
  - id, cust, prod, store, price, date, qty

<table>
<thead>
<tr>
<th>id1</th>
<th>cust1</th>
</tr>
</thead>
<tbody>
<tr>
<td>id2</td>
<td>cust2</td>
</tr>
<tr>
<td>id3</td>
<td>cust3</td>
</tr>
<tr>
<td>id4</td>
<td>cust4</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>id1</td>
<td>prod1</td>
</tr>
<tr>
<td>id2</td>
<td>prod2</td>
</tr>
<tr>
<td>id3</td>
<td>prod3</td>
</tr>
<tr>
<td>id4</td>
<td>prod4</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>id1</td>
<td>price1</td>
</tr>
<tr>
<td>id2</td>
<td>price2</td>
</tr>
<tr>
<td>id3</td>
<td>price3</td>
</tr>
<tr>
<td>id4</td>
<td>price4</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ids may or may not be stored explicitly
Row vs Column Store

• Advantages of Column Store
  – more compact storage (fields need not start at byte boundaries)
  – Efficient compression, e.g., RLE
  – efficient reads on data mining operations

• Advantages of Row Store
  – writes (multiple fields of one record) more efficient
  – efficient reads for record access (OLTP)
Comparison

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

Which is right for me?
Issues:

Flexibility  ———  Space Utilization

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Performance</td>
</tr>
</tbody>
</table>
To evaluate a given strategy, compute following parameters:

-> 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 complete file
- reorganize file
Example

How would you design Megatron 3000 storage system? (for a relational DB, low end)
  – Variable length records?
  – Spanned?
  – What data types?
  – Fixed format?
  – Record IDs?
  – Sequencing?
  – How to handle deletions?
Summary

- How to lay out data on disk

Data Items
  - Records
    - Blocks
      - Files
        - Memory
          - DBMS
How to find a record quickly, given a key