Principles of Database Management Systems

Pekka Kilpeläinen
(after Stanford CS245 slide originals by Hector Garcia-Molina, Jeff Ullman and Jennifer Widom)

DBMS 2001 Notes 3: Representing Data

• How to represent data on disk and in memory

Executive Summary:
• Principles are rather simple, but there are lots of variations in the details

Principles of Data Layout

• Attributes of relational tuples (or objects) represented by sequences of bytes called fields
• Fields grouped together into records – representation of tuples or objects
• Records stored in blocks
• File: collection of blocks that forms a relation (or the extent of an object class)

Overview

Data Items
→ Records
→ Blocks
→ Files
→ Memory

What are the data items we want to store?
• a salary, a name
• a date, a picture
• ...

What we have available: Bytes

8 bits

To represent:
• Integer (short): 2 bytes (-32000...+32000)
  e.g., 35 is 00000000 00100011
• Integer (long): 4 bytes (-2x10^9...+2x10^9)
• Real, floating point (SQL FLOAT)
  4 or 8 bytes
• arithmetic interpretation by hardware
To represent:

• Characters
  various coding schemes suggested,
  most popular is ASCII
  **Example** (8 bit ASCII):
  A: 01000001
  a: 01100001
  5: 00110101
  LF: 00001010

To represent:

• Boolean
  e.g., TRUE 11111111
  FALSE 00000000

• Application specific
  e.g., RED → 1  GREEN → 2
  BLUE → 3  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 chars: YYYYMMDD
  - 7 chars: YYYYDDD
  - 10 chars: YYYY-MM-DD (SQL2)

  (not YYMMDD! Why?)

• Time, e.g.
  - Integer: seconds since midnight
  - chars: HH:MM:SS[.FF…] (SQL2)

To represent:

• String of characters
  - Null terminated
    e.g.,
  - Length given
    e.g.,
  - 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
- Type of an item: Tells us how to interpret (plus size if fixed)
  - normally indicated in the record schema (see in a moment)

**Types of records:**

- Main choices:
  - FIXED vs VARIABLE FORMAT
  - FIXED vs VARIABLE LENGTH

**Example: fixed format and length**

Employee record

1. E#, 2 byte integer
2. E.name, 10 char.
3. Dept, 2 byte code

<table>
<thead>
<tr>
<th>02</th>
<th>01</th>
</tr>
</thead>
<tbody>
<tr>
<td>83</td>
<td>01</td>
</tr>
<tr>
<td>46</td>
<td>02</td>
</tr>
</tbody>
</table>

**Fixed format**

A schema (not record) contains following information

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

**Variable format**

- Record itself contains format; “Self Describing”
Variable format useful for:

- "sparse" records
  - e.g. patient records with thousands of possible tests
- repeating fields
- information integration from heterogeneous sources

Field name codes could also be strings, i.e. tags (∴ XML as a data interchange format)

Variable format useful for:

- "sparse" records
  - e.g. patient records with thousands of possible tests
- repeating fields
- information integration from heterogeneous sources

Field name codes could also be strings, i.e. tags (∴ XML as a data interchange format)

Many variants between fixed and variable format:

Ex. #1: Include record type in record

<table>
<thead>
<tr>
<th>record type</th>
<th>record length</th>
</tr>
</thead>
<tbody>
<tr>
<td>tells me what to expect (i.e. points to schema)</td>
<td></td>
</tr>
</tbody>
</table>

Record header - data at beginning that describes record

May contain:
- indication of record type
- record length
- time stamp
- other stuff ...

Note: Repeating fields does not imply
- variable format, nor
- variable size

- Key is to allocate maximum number of repeating fields (if not used → NULL)

• EXAMPLE: var format record with repeating fields
Employee → one or more children

<table>
<thead>
<tr>
<th>3</th>
<th>F_name: Fred</th>
<th>Child: Sally</th>
<th>Child: Tom</th>
</tr>
</thead>
</table>

Note: Repeating fields does not imply
- variable format, nor
- variable size

- Key is to allocate maximum number of repeating fields (if not used → NULL)

• EXAMPLE: var format record with repeating fields
Employee → one or more children

<table>
<thead>
<tr>
<th>3</th>
<th>F_name: Fred</th>
<th>Child: Sally</th>
<th>Child: Tom</th>
</tr>
</thead>
</table>

**Many variants between fixed and variable format:**

Ex. #1: Include record type in record

<table>
<thead>
<tr>
<th>record type</th>
<th>record length</th>
</tr>
</thead>
<tbody>
<tr>
<td>tells me what to expect (i.e. points to schema)</td>
<td></td>
</tr>
</tbody>
</table>

Record header - data at beginning that describes record

May contain:
- indication of record type
- record length
- time stamp
- other stuff ...

Note: Repeating fields does not imply
- variable format, nor
- variable size

- Key is to allocate maximum number of repeating fields (if not used → NULL)
Ex #2 of variant between FIXED/VAR format

- Hybrid format
  - one part is fixed, other variable

E.g.: All employees have E#, name, dept; Other fields vary.

| # of var fields | Smith | Toy | 2 | hobby: chess retired |

Also, many variations in internal organization of record

Just to show one:

| length of field | 1 | b | F1 | F2 | b | F3 |
| total size | F1 | F2 | F3 |
| offsets | 0 | 1 | 15 | 15 | 20 |

Next: placing records into blocks

blocks

blocks ... assume fixed length blocks

a file: assume a single relation (for now)

Issues in storing records in blocks:

(1) separating records
(2) spanned vs. unspanned
(3) mixed record types – clustering
(4) split records
(5) sequencing
(6) addressing records

(1) Separating records

Block

(a) fixed size recs. -> no need to separate
(b) special marker
(c) give record lengths (or offsets)
  - within each record
  - in block header (see later)

(2) Spanned vs. Unspanned

- Unspanned: records are within one block
  R1 R2 R3 R4 R5 R6

- Spanned: records span block boundaries
  block 1 block 2
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 necessary if record size > block size (e.g., fields containing large "BLOB"s for, say, MPEG video clips)

Example (of unspanned records)

<table>
<thead>
<tr>
<th>Block 1</th>
<th>Block 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R2</td>
</tr>
<tr>
<td>2050 bytes</td>
<td>wasted 2046</td>
</tr>
<tr>
<td>R3</td>
<td>R4</td>
</tr>
<tr>
<td>2050 bytes</td>
<td>wasted 2046</td>
</tr>
<tr>
<td>R5</td>
<td>R6</td>
</tr>
<tr>
<td>R7</td>
<td></td>
</tr>
</tbody>
</table>

- Space used about 4 x 10^9 B, about half wasted

Example

Q1: `select DEPT.Name, EMP.Name, ... from DEPT, EMP where DEPT.Name = EMP.DeptName`

<table>
<thead>
<tr>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPT</td>
</tr>
<tr>
<td>EMP</td>
</tr>
</tbody>
</table>

Why do we want to mix?
Answer: **CLUSTERING**
Records that are frequently accessed together should be in the same block

(3) Mixed record types
- Mixed - records of different types (e.g. DEPT, EMPLOYEE) allowed in same block

<table>
<thead>
<tr>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPT</td>
</tr>
<tr>
<td>EMP</td>
</tr>
</tbody>
</table>
• If Q1 frequent, clustering is good
• But consider Q2:

```sql
SELECT *
FROM DEPT
```
If Q2 is frequent, clustering is counter-productive

(4) Split records

- Fixed part in one block
- Typically for hybrid format
- Variable part in another block

(5) Sequencing

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

```
R1  Next (R1) ...
```

(b) Linked

```
R1  Next (R1)
```
6) Addressing records

- How does one refer to records?

Many options:
- Physical
- Indirect

### Purely Physical

E.g., Record Address or ID =

\[
\text{Device ID} \
\text{Cylinder #} \
\text{Track #} \
\text{Block #} \
\text{Offset in block}
\]

### Fully Indirect

E.g., Record ID is arbitrary byte string (say, an object ID)

map table

rec ID

Physical

Indirect

Tradeoff

Flexibility
Cost

to move records
of indirection

(for deletions, insertions)

Physical

Indirect

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

May contain:
- File ID (or RELATION or DB ID): the ID of this block
- Record directory: Pointer to free space
- Type of block (e.g. contains records of type 4; is overflow, ...)
- Pointer to other blocks "like it" (say, if part of an index structure)
- Timestamp ...

### Issues in storing records in blocks

1. Separating records
2. Spanned vs. Unspanned
3. Mixed record types - Clustering
4. Split records
5. Sequencing
6. Addressing records

### Other Topics

1. Insertion/Deletion
2. Pointer Swizzling
3. Comparison of Schemes

### Options:

(a) Immediately reclaim space
(b) Mark deleted
   - May need chain of deleted records (for re-use)
   - If there are pointers to the record, either need to set them to NULL or indicate the record space reclaimed

### Ex #1 Indirection in block

A block:

- **Header**
- **Free space**

R4
R3
R2
R1

### Deletion

**Block**

Rx
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,...

A solution: Tombstones

(a) Leave “MARK” in old location

• with physical IDs:

This space never re-used
This space can be re-used

A Solution: Tombstones

(b) Leave “MARK” in map table

• with 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...

Easy case: records not in sequence

→ Insert new record at end of file, or in any block with space for it

Hard case: records in sequence

→ If free space “close by”, not too bad...
  - can "slide" records to preceding/succeeding blocks
  - pointers to moved records require forwarding addresses
→ Or use additional blocks as an overflow area

Concern with deletions

Dangling pointers

R1 → ?
Interesting problems:
- How much free space to leave in each block, track, cylinder?
- How often to reorganize file + overflow?

(2) Pointer Swizzling

- Two kinds of record addresses:
  - **DB address**
    - physical location on secondary storage
  - **Memory address**
    - record location when loaded into (main or virtual) memory
- Which one to use, and when?

**Pointer Swizzling**

![Diagram: Memory and Disk with block 1 and 2, Rec A in Memory and Disk]

**One Option:**
- For all records copied to memory

**Another Option:**
- In memory pointers - need “type” bit
  - to disk
  - to memory

**Swizzling**
- Automatic ("eager")
- On-demand ("lazy")
- No swizzling / program control
Comparison

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

Which one is right for me?

Issues:

Flexibility  Space Utilization

Complexity  Performance

To evaluate a given strategy, estimate following parameters:
- space used for expected data
- expected time to
  - fetch record given its key
  - fetch record with next key
  - insert/delete/update record
  - scan all records in the 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