File Structures & Storage Manager

COM 3563: Database Implementation

Avraham Leff

Yeshiva University

avraham.leff@yu.edu

COM3563: Fall 2020

Today's Lecture: Overview

1. Page Storage Architecture
2. Tuple Storage Architecture: Fixed-Length Records
3. Tuple Storage Architecture: Variable-Length Records
4. Data Dictionary
5. Bonus: Denormalized Data

Some material from lectures created by Professor Pavlo (CMU), used with permission

Context

- Previous lectures discussed how a DBMS manages the reality of physical storage
  - We focused on characteristics & performance implications of various physical media
  - The existence of, and approaches for dealing with, the memory hierarchy
  - Motivated the need for a DBMS-specific buffer manager
- Today: we go a level deeper!
  - How are database data actually stored on disk?
  - We'll see that a hierarchy exists: records, pages, files
  - DBMS storage manager is responsible for organizing this data in a way that results in good performance

Terminology: Records, Pages and Files

- Higher-levels of DBMS deal with records
  - Not pages
  - Previous lectures discussed “pages”
- Lower-levels of DBMS store records in pages
- Lots of records require multiple pages: these are aggregated into files
- Storage manager responsibilities
  - Maintaining a database’s files
  - Track the data that’s read from, and written to, pages
  - Track the available space to facilitate new storage operations

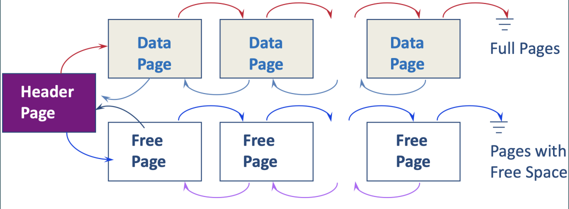
Page Storage Architecture

Introduction

- At this point, we're not thinking about "what's inside a page?"
- Just: "how do we organize pages within a file?"
- Note: however we organize these pages, the "file" abstraction must (at least) support operations such as:
  - CUD operations at a per-record granularity
  - Read a particular record by "record id"
  - Iterate over all records (possibly with some conditions on the records to be retrieved)
- Let's start with a heap file organization: store a record anywhere that the file has space
  - Straightforward to implement ☺
  - Or: maybe not quite so simple ☹

Heap Files: Lists of Pages

- Records in heap file pages do not follow any particular order
- As a heap file grows and shrinks, disk pages are allocated and deallocated
  - Records are stored in random order
  - Pages are unordered within a file
- One implementation approach: a heap file is a doubly linked list of pages



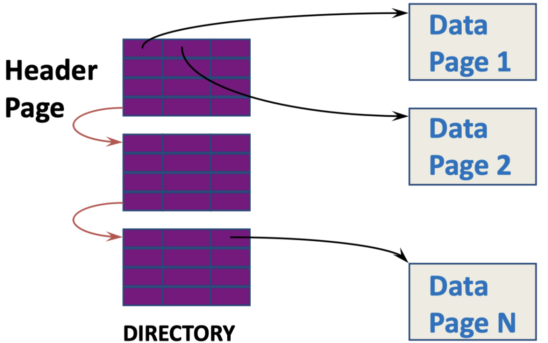
- A special "header" page is stored in a known disk location, contains minimal, but important information
  - Name of the heap file
  - Address of the first page's address

Heap Files: Problems With The “List” Approach

- It is likely that every page will contain at least a few free bytes
- Result: almost all pages in a file will be on the free list ☹
- When the storage manager wants to insert a record, will have to scan multiple pages on the free list before finding one with enough free space
- This problem can be addressed with a different implementation approach for heap files: a directory-based organization

Heap Files: Page Directory

- In this approach, the storage manager maintains a directory of pages
- Each directory entry identifies a page in the heap file



- Directory entries contain information about
  - Does a given page have any free space?
  - How much space is free in a given page?
- Note: benefits of this approach are obvious, but incurs the cost of having to keep directory entries "in sync" with page contents

Alternative Approaches To File Organization

- We've just discussed one approach: heap file organization
  - No organization at all ☺
- Other approaches are used as well
- Hashing technique (not this lecture)
  1. Compute a hash function on some attribute of each record
  2. The hash value determines the block in which the record will be placed
- B^+-Tree organization: (not this lecture)
  - Addresses weaknesses in sequential organization (below) with respect to many CUD operations
  - Very efficient "lookup by search key" (and range queries too) ☺
- Sequential technique: store records in sequential order, based on the value of a singe, pre-specified search key for the set of records
  - Advantage: can use binary search when searching on the "search key" attribute

Sequential File Organization

- This technique is suitable for applications that typically access the file's records in sequential fashion
- DBMS orders records in the file based on the value of some search-key
- Q: What's the search key in the Figure below?

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Naive Approach Won't Work!

- Q: can the sequential organization approach layout records statically within a page?
  - Recall: the records must be sequenced according to the "search key" attribute
- A: a "static" approach isn't feasible because CUD operations will "break" the physical layout ☹
- Q: what approach will work?
- A: a dynamic (pointer-based) approach ☺
- Deletion: link all free records on a free list
- Insertion: locate the position where the record is to be inserted
  1. If there is free space, insert in that location
  2. Otherwise: insert the record in an overflow block (we'll drill down on this technique later)
  3. Then: update the pointer chain

Sequential File Organization: Dynamic Version

Note the use of pointer-chain <u>and</u> overflow blocks to keep these records organized sequentially

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

32222	Verdi	Music	48000	

- Don't forget the physical storage constraints!
- If the physical layout of the records are "too randomly" located on disk, DBMS will pay a performance penalty because of "too many" disk block accesses
- Solution: periodically (physically) reorganize the records within the file to have physical and logical order realign

“Multitable Clustering” File Organization: Introduction

- We’ve tacitly assumed that every relation is stored in its own file
  - Makes intuitive sense ☺
  - Advantage: file organization algorithms are simple
- It can make sense to use a more complex approach in which the storage manager clusters records from multiple relations within a single file
  - Perhaps even within a single block
- Q: can you suggest why this would ever make sense?
  - Answer on next slide

Multitable Clustering File Organization: I

- Motivation: the DBMS “knows” that records of different types are often accessed together
- Scenario: joins on related tables

```sql
select department_name, building, budget, ID, salary
from department natural join instructor
```

department

dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Physics	Watson	70000

instructor

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

Multitable Clustering File Organization: II

```sql
select department_name, building, budget, ID, salary
from department natural join instructor
```

- DBMS must locate *instructor* tuples that share the same value for a given value of *department_name*
- If *instructor* tuples and *department_name* tuples are in different files, DBMS will have to do a separate block transfer for each record in the query
- Solution: Store tuples from both relations in same file
- Solution: Store *instructor* tuples for a given *ID* in same or near block of *department* tuples that have the same *department_name*

Comp. Sci.	Taylor	100000	
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
Physics	Watson	70000	
33456	Gold	Physics	87000

No Silver Bullet!

- Multitable clustering is a good strategy for specific scenarios
- In our example: has improved processing of a specific type of query
  - Good for queries involving *department* ⋈ *instructor*
  - Good for queries involving a single department and its instructors
- Inherently: degrades performance for other types of queries
  - Example: queries that only involve *instructor*
  - Example: queries that only involve *department*
- DBMS better do some serious statistical analysis of access patterns (periodically?) before taking this step ☺
- Other disadvantages: impossible to use fixed-length records for this file (see later discussion)

Multitable Clustering File Organization: III

- Can use "pointers" to mitigate the problem of "single-relation" queries
- Layer pointer-chaining on top of multi-table clustering approach
- Example: use pointers to link *department* records together

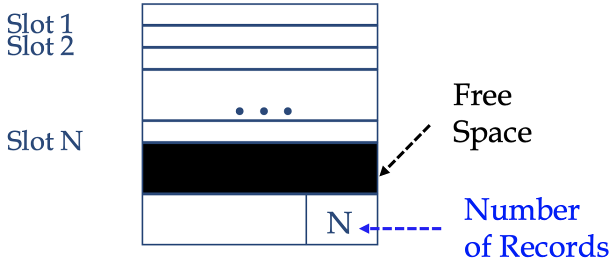
Comp. Sci.	Taylor	100000
45564	Katz	75000
10101	Srinivasan	65000
83821	Brandt	92000
Physics	Watson	70000
33456	Gold	87000

Segue To Tuple Storage Architecture (I)

- At this point we know that a database file (typically storing all tuples in a single relation) is comprised of n pages
  - Remember: "tuples" at the higher levels of the DBMS are "records" at the storage manager level
- In addition to containing "data" (m records), every page contains a meta-data header with information data about the page's contents
  - Page size
  - Checksum of page's data
  - etc
- Only remaining issue is how to store records in a given page
- We'll start with a simplification: fixed-length records
  - Then: consider how to deal with variable-length records

Segue To Tuple Storage Architecture (II)

- Think of a page as a collection of “slots”, each of which contains a record



- A record can now be identified using a duple: $\{page_i, slot_j\}$
- This duple is typically referred to as a record id (or RID)
- With fixed-length records, slots are uniform and can be arranged consecutively

Fixed-Length Records: Easiest Approach to Implement

- Store *record*$_i$ such that
  - If n is the (uniform) size of all records, the start address of *record*$_i$ is at byte $n \times i - 1$
- Only complication with accessing a record is: *"what if a record layout crosses block boundaries?"*
- "Solution": modify the algorithm to prevent records from cross block boundaries
  - Just stop allocating records in *block*$_i$ if not enough space ☺
- Fixed-length records have many advantages:
  - Data are as compact as can be
  - Finding i_{th} record does not require scan of file
  - Finding i_{th} field does not require scan of record
  - Information about field types same for all records in a file; stored in system catalogs

Fixed-Length Records: Example

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

- *Account* record
  - Consists of *account-number char(10), branch-name char(20), balance real*
  - Every record same size: *38 bytes*
- Store records sequentially
  - $Record_1$ at position 0, $record_2$ at position 38, $record_3$ at position 76

One Non-Trivial Issue With “Fixed-Length” Records

- Assuming we do have fixed-length records, the storage-manager algorithms seem trivial ☺
- Q: can you spot the problem?
- A: how do we with record deletion?
- One issue: “nulling” out the deleted slot wastes space
- Also: how does the DBMS know that this slot is now occupied by a deleted record?
- We’ll start with a naive algorithm for deleting record i
  - Record compaction: move records $i+1, \ldots, n$ to slots $i, \ldots n-1$

"Record Compaction" Algorithm

Algorithm applied to scenario: delete record #3

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

- Note: before deletion, file had twelve records, after compaction, has eleven records
- Potentially very expensive operation: on average, must move (copy) $n/2$ records ☹
- Q: can you suggest a simple improvement with big payoff for this record compaction algorithm?
- A: move only record n to slot i

“Move Last Record” Algorithm

Algorithm applied to scenario: delete record #3

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000

- This algorithm simply moves the n_{th} record in the file to the space formerly occupied by record #3
- Q: can we further improve the performance of this algorithm?
- A: yes, but at some increase in complexity

What If We Don't Move Any Records At All?

- Key idea: don't moving any records into the deleted slot!
  - Note: we're assuming that even the cost of copying a single record is expensive
  - Leave slot empty, reuse it when DBMS inserts a new record!
- Q: can you spot a problem with this approach?
- A: the algorithm handles delete operations, but does badly for record insertion!
  - When doing record insertion we need an efficient lookup of "next available slot"
- In other words: algorithm needs a better way to store the "deleted slot" information
- Note: these sort of issues are straight out of Data Structures ☺

“Free-List” Algorithm: I

Key idea: Maintain “address of first deleted record” information in a file header

- deleted $record_i$ stores the address of deleted $record_{i+1}$
- This algorithm (in effect) creates a free-list

header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

This Figure shows the file after records 1, 4, and 6 were deleted

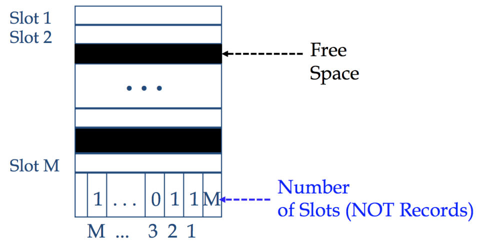
“Free-List” Algorithm: II

1. Deletion: Chain down the free list, add the newly deleted record to the end of the list
   - Textbook ignores this cost, probably because it assumes that insertions are more frequent than record deletions
2. Insertion:
   2.1 If free list contains any records, insert in first element of free list & adjust file header to point to the next element of the free list
   2.2 Otherwise: add new record to end of file

Before We Leave "Fixed-Length" Records (I)

- We've just discussed approaches for dealing with memory compaction issues that are raised by "record deletion"
- Don't ignore the issue of the RID!
  - Recall: a RID ("record id") is a very "low-level" concept
  - Consists of $\{page_i, slot_j\}$
- When moving a record to another slot, the storage manager must also update its RID (as well as any data structures that were pointing to that RID)
- Good news: the free-list approach or an equivalent bit-array implementation (next slide) also handles the RID issue

Before We Leave "Fixed-Length" Records (II)



- The bit-array approach handles "deletions" by using an array of bits
- When a record is deleted, its bit is turned off
- The RID of non-deleted records don't have to be changed

Tuple Storage Architecture: Variable-Length Records

The Problem

> The algorithms just discussed for record insertion and deletion cannot be applied to files that store variable-length records

- Is this an artificial problem?
- Absolutely not! Variable-length records occur whenever a DBMS
  - Store multiple record types in a single file (see "Multitable Clustering" discussion above)
  - Stores record types that allow variable lengths for fields such as strings (`varchar`)
  - Stores record types that allow repeating fields such as arrays or multi-sets
    - Was amused: Silberschatz writes "(used in some older data models)"
    - Ahem: and in NOSQL databases ☺
- Before we drill down into "variable-length" records, let's see why they pose new problems for the storage manager

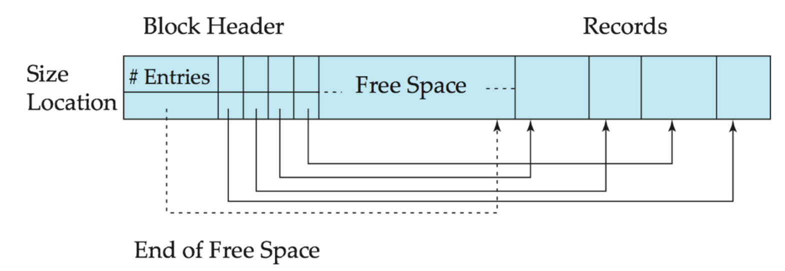
- Storage manager cannot divide the page into a fixed collection of n slots
- Inserting a record: storage manager must find an empty slot with enough space to hold the new record
  - Note: must figure out how to find the i_{th} record efficiently …
  - And: must also be able to find the j_{th} field of the i_{th} record efficiently
- Deleting a record: storage manager must ensure that free space is contiguous
  - Otherwise: the “insertion” problem becomes even worse, eventually impossible
- How can we moving a record without changing its RID?

Finding the $i_t h$ Record: I

- Naive approach: append a special "end-of-record" symbol to the end of each database record
  - Example: $\perp$
- Store each record as a string of bytes: hence byte string representation
- Some problems:
  - Difficult to reclaim space left by deleted record
  - Difficult to update a record: record may need to expand, can be expensive to "copy & grow" especially if record is currently pinned by the DBMS
- Approach is not generally used, mentioning in case you thought that approach might work ☺

Finding the i_th Record: II

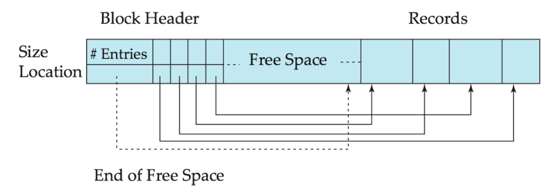
Slotted-Page ("Block") Structure Approach



Slotted page header contains:

- Number of record entries
- End of free space in the block
- Location and size of each record (as a header array, one element per record)

Finding the i_th Record: Slotted-Page Structure Algorithm (I)



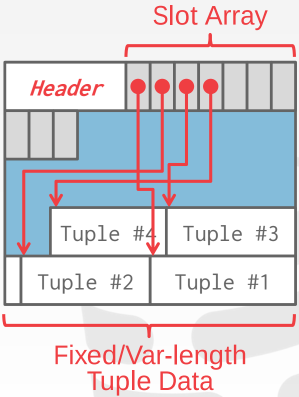
- Free space is a contiguous block, located between the last entry in the header array and the first actual record
- Record insertion:
  - Records are allocated contiguously, starting from the end of the "free space" block
  - Allocate space for the record at the current "end of free space" (moving "backwards")
  - Allocate a new header entry (specified record's size and location) to the current "beginning of free space"

SLOTTED PAGES

The most common layout scheme is called slotted pages.

The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:
→ The # of used slots
→ The offset of the starting location of the last slot used.

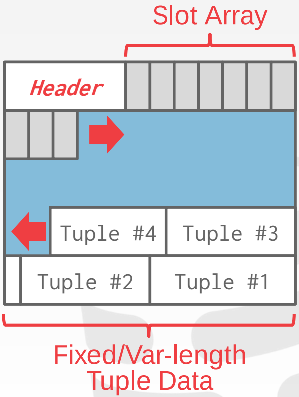


SLOTTED PAGES

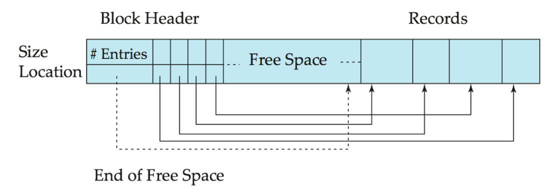
The most common layout scheme is called slotted pages.

The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:
→ The # of used slots
→ The offset of the starting location of the last slot used.



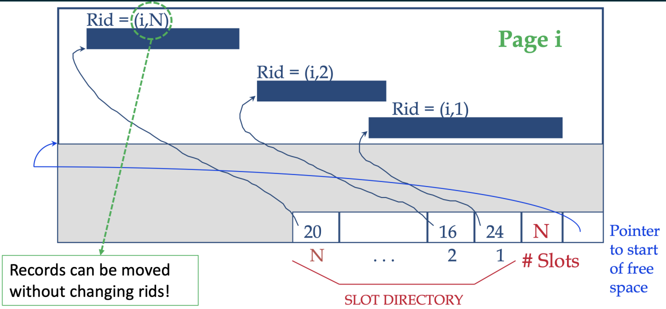
Finding the i_th Record: Slotted-Page Structure Algorithm (II)



- Record deletion
  1. Set "record size" to sentinel value (e.g., -1)
  2. Coalesce records to maintain invariant that *all free space is between last array header entry and first actual record*
  3. Adjust the "end of free-space" pointer
- Textbook claims that cost of moving records around is small because block sizes are small …

Slotted-Pages Approach: RID

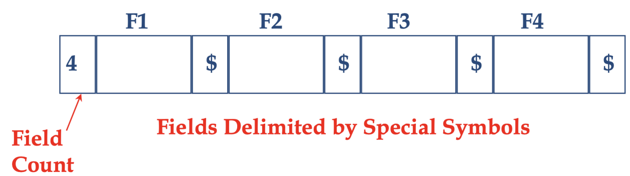
- We mentioned before how the requirement to move records around in a page (to keep free space contiguous) complicates the implementation of a RID
  - The RID can't be based on the actual page location anymore!
- The slotted-pages approach solves the RID issue using a CS classic: "pointer indirection" ☺
  - A RID never points (directly) to actual record location
  - Instead: point to record entry in header array (which can then be updated safely)



- A tuple is essentially a sequence of bytes: it's the job of the DBMS to interpret those bytes into attribute types and values
- Record fields can be either of:
  - Fixed-Length: each field has a fixed length and the number of fields is also fixed
  - Variable-Length: The number of fields are fixed but the fields themselves can have variable length
- We've focused on the implications of variable length records on page structure: *"how to store (and efficiently manage) multiple such records in a single page?"*
- Now: focus on a single such record: *"how to store (and efficiently access) a given field within this overall sequence of bytes?"*

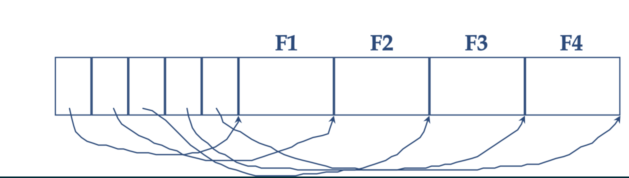
Variable-Length Fields: Don't Do This

- You might think of the following approach for implementing variable-length fields: *"simply layout the fields consecutively, and separate fields by special delimiters"*



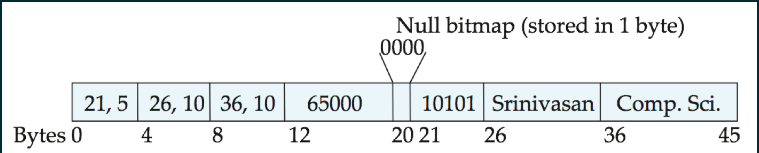
- Q: can you spot the flaw with this approach?
- A: it requires the storage manager to do a linear scan of the entire record whenever it needs to access a specific field ☹

Variable-Length Fields: Better Approach



- The figure above maintains an array of field offsets "meta-data" header in each record
- Advantage: storage manager now has direct access to individual record fields

Finding the j_th Field in a Record: (I)



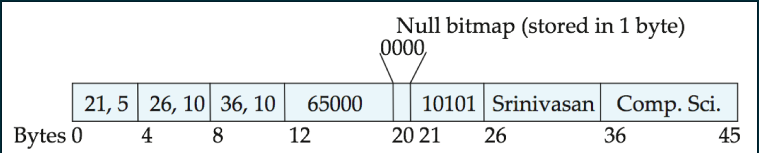
- Variable-length attributes: meta-data represented by a (offset, length) duple
  - *offset*: where does data for that field begin (within the record)?
  - *length*: how many bytes does that field occupy?
- All variable-length meta-data stored at beginning of record
- Example: first three fields of *instructor* record
  - ID, NAME, DEPT_NAME (of type *varchar*)
- Fixed-length attributes: stored after variable-length meta-data
  - Example: SALARY

Finding the j_th Field in a Record: (II)

Advantages of this approach

- Initial portion of record is "conceptually" fixed-length
  - Whether or not the field itself is fixed or variable length
- Variable-length data stored after the "fixed-length" part of the (variable-length) record
- Variable-length portion can be accessed (and updated) using initial (fixed-length) meta-data

Finding the j_th Field in a Record: (III)



- This approach solves another problem as well: allows attributes (whether "fixed" or "variable-length") to have NULL values
- Null values represented by a null-value bitmap
- Size of bitmap must be large enough to accommodate the attribute cardinality
  - Bit-pattern turns on bit_i *iff* $attribute_i$ is NULL
  - DBMS then ignores the actual field "values": knows they're NULL

Alternative Solutions for the “Finding the j_th Field in a Record” Problem

- Pointer method: represent a variable-length record using a list of fixed-length records
  - Chain the list elements together using pointers

- Advantage: “pointer method” can be used even if the maximum record length is unknown
- Disadvantage: record space (in addition to the pointer field) must be allocated for all records: even if a record doesn’t need to be chained

0	Perryridge	A-102	400	
1	Round Hill	A-305	350	
2	Mianus	A-215	700	
3	Downtown	A-101	500	
4	Redwood	A-222	700	
5		A-201	900	
6	Brighton	A-217	750	
7		A-110	600	
8		A-218	700	

Improving the "Pointer Method"

- Improve the data-structure by allowing two kinds of blocks
  - Anchor block: contains the first records of chain
  - Overflow block: contains "spill-over" records
    - Data that doesn't fit into the "anchor" block of a given record

Storing Large Objects

- One assumption that we haven't (yet) emphasized: *"a record must be smaller than a page (or block)"*
  - If this assumption is violated, all of the "file structure" architectures no longer work ☹
- Q: can you give examples of familiar RDB constructs that routinely violate this assumption?
- A: sure, consider the **BLOB** and CLOB data-types
- Approaches for handling "large objects"
  - Store separately as files in file systems (or managed by database)
  - Break into pieces and store in multiple tuples in separate relation
    - Note: we just used this approach in the context of representing any variable-length record using a list of fixed-length records
    - Example: POSTGRESQL uses this approach in its **TOAST** ("The Oversized-Attribute Storage Technique") mechanism

Data Dictionary

What Is a Data Dictionary?

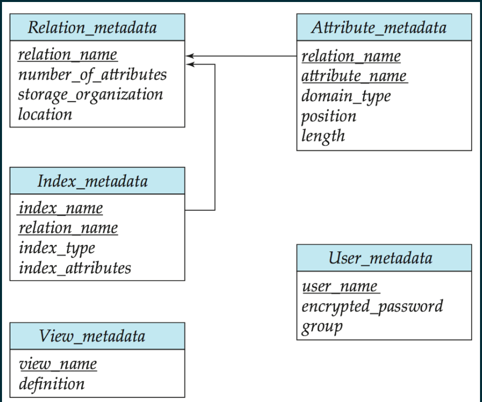
Data dictionary (also called system catalog) stores DBMS metadata

- Information about relations
  - Names of relations
  - Names, types and lengths of attributes of each relation
  - Names and definitions of views
  - Integrity constraints
- User and accounting information, including passwords
- Statistical and descriptive data
- Physical file organization information
  - How relation is stored (heap/sequential/hash)
  - Physical location of relation
  - Information about indices (topic for another set of lectures)

Storing a Data Dictionary

- Data dictionaries are a miniature database
  - Need to perform CRUD operations efficiently!
- Can use specialized data-structures and algorithms
- But: better (from "software engineering" perspective) to treat as "yet another" set of relations, and store in the relational database itself ☺

Data Dictionary as a Set of Relations



- Typically, not treated "exactly like other relations"
- Example: in the *Index_metadata* relation, note how *index_attributes* property is not normalized!
  - It's a list of attribute names so it's not even in 1NF …
- Could be normalized: typically more efficient not to normalize it for faster access
- Remember: cannot have turtles all the way down ☺

Bonus: Denormalized Data

- We've repeatedly stressed the distinction between "logical" and "physical" representations of data
- Example: the relational ("logical") data model absolutely forbids data to be "denormalized"
  - At its worst (e.g., "an array of phone numbers"), data aren't even in 1NF ☹
- However: at the storage manager layer, the normalized data may actually be stored in denormalized format
- Key point #1: that's fine if it improves performance
- Key point #2: that's fine so long as it doesn't affect the data representation at the logical level

DENORMALIZED TUPLE DATA

Can physically ***denormalize*** (e.g., "pre join") related tuples and store them together in the same page.

→ Potentially reduces the amount of I/O for common workload patterns.

→ Can make updates more expensive.

```
CREATE TABLE foo (
  a INT PRIMARY KEY,
  b INT NOT NULL,
);
```

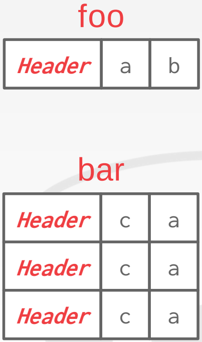
```
CREATE TABLE bar (
  c INT PRIMARY KEY,
  a INT
    REFERENCES foo (a),
);
```

DENORMALIZED TUPLE DATA

Can physically ***denormalize*** (e.g., "pre join") related tuples and store them together in the same page.

→ Potentially reduces the amount of I/O for common workload patterns.

→ Can make updates more expensive.

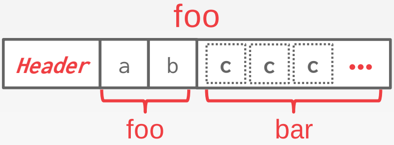


DENORMALIZED TUPLE DATA

Can physically ***denormalize*** (e.g., "pre join") related tuples and store them together in the same page.

→ Potentially reduces the amount of I/O for common workload patterns.

→ Can make updates more expensive.

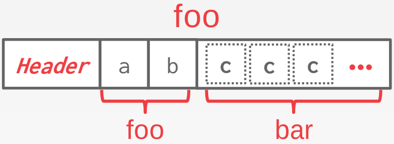


DENORMALIZED TUPLE DATA

Can physically ***denormalize*** (e.g., "pre join") related tuples and store them together in the same page.

→ Potentially reduces the amount of I/O for common workload patterns.

→ Can make updates more expensive.



Not a new idea.

→ IBM System R did this in the 1970s.

→ Several NoSQL DBMSs do this without calling it physical denormalization.

Page Storage Architecture

Tuple Storage Architecture: Fixed-Length Records

Tuple Storage Architecture: Variable-Length Records

Data Dictionary

Bonus: Denormalized Data

Readings

- Textbook: Chapter 13.1-13.5