COSC2406/2407 Database Systems

Files, Pages and Records

Xiangmin (Emily) Zhou

RMIT University
Room : 14.11.04
Online Consultation via Collaborate Ultra(no appointment required): 10:20-11.20am Thursdays
Email : xiangmin.zhou@rmit.edu.au

Lecture 3

References: Ramakrishnan & Gehrke Chapter 9
Garcia-Molina et al. Chapter 12
Diagrams courtesy Ramakrishnan & Gehrke

Overview

In this lecture, we will:

- Discuss file, data, and record organisations in a DBMS
- Discuss page formats
- MongoDB and Apache Derby

Representing Data Elements

Last week, we discussed disks, their characteristics, and how DBMS buffer managers interact with disks.

We focused on disc blocks (corresponding to pages in memory), and now we focus on how data is stored in those pages:

1. Attributes are stored as variable- or fixed-length sequences of bytes (known as *fields*)
2. Fields are stored together to form logical *records*
3. Records are stored in blocks (pages)
4. Blocks of records of the same type are typically stored together to form a *file* (note that a file in a DBMS is different in concept to an operating system file)

Records

Records group together related fields. Logical records might be, for example, student records or, say, order records that store logically-grouped information about real-world objects.

Each type of record has a *schema* that is stored in the database and it lists the names, type, and offset of fields within the record.

A record can be thought of as a unit of storage, the length of which depends on the particular application.

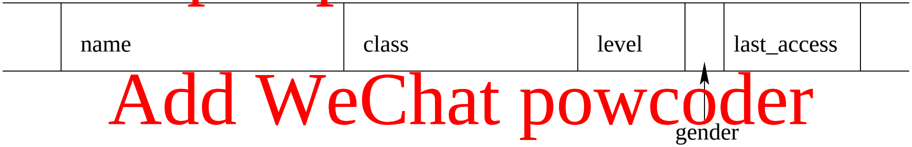
A Sample Schema

```
CREATE TABLE character (
    name CHAR(30),
    class VARCHAR(20),
    level INT,
    gender CHAR(1),
    last_access DATE
);
```

An Example Relation

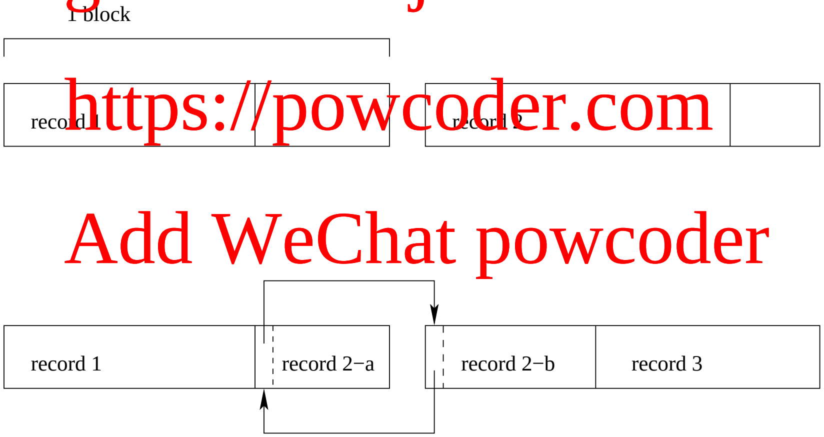
name	class	level	gender	last_access
Frost	Mage	5	M	13/01/04
Moon	Warrior	20	F	20/01/05
Lysa	Druid	13	F	01/12/04
Varra	Warrior	19	M	12/12/04

An Example Record



Spanning

A block on disk—a physical record—may contain more than one logical record. A logical record that is partly in one block, and partly in another is called a *spanning record*.



Blocking Factor

- The *blocking factor* is the block size divided by the record size.
- Suppose that a disk is formatted with a block size of 2048 bytes, and that we need to store records that have a size of 100 bytes.
- How many records can we store in 10 blocks if there is no spanning?

- How many records can we store in 10 blocks if spanning is allowed (not taking header information into account)?

BLOBs

- Each logical record may also take more than one physical block. For example, many modern applications store very large records, called BLOBs (Binary Large OBjects).
- Truly large objects include images (e.g. JPEG, GIF), movies (e.g. MPEG), and signals (e.g. audio, radar).
- BLOBs are often stored on a sequence of blocks for efficient retrieval. However, it might not always be necessary to retrieve a BLOB in its entirety (can use a linked list of blocks, e.g. for a movie).
- If it is necessary that parts of a BLOB can be retrieved (e.g. the third scene of a movie), an index is used.

File Organisation

File organisation deals with how the records are placed in a file.

The *file access method* is the technique used to get to records.

The factors that influence organisation and access method include:

- speed of access requirements
- storage space limitations
- file volatility

The operations that need be supported by the access methods might include:

- Scan the file and access all the records
- Search for records matching a value, or in a range of values
- Insert or delete a record

Record Identifiers (*rids*)

The DBMS needs a way to identify records. Each record is generally identified by a pair:

<page_id, slot_number>

Alternatively, each records can be assigned with a unique integer as its *rid*.

This allows for more flexible page formats, but has the drawback that a *mapping table* that lists the corresponding page and slot number for each record must be maintained. Due to the overhead of maintaining such a table, the first approach is more common.

Page Formats

A page consists of a collection of logical records.

How are (logical) records arranged in a page?

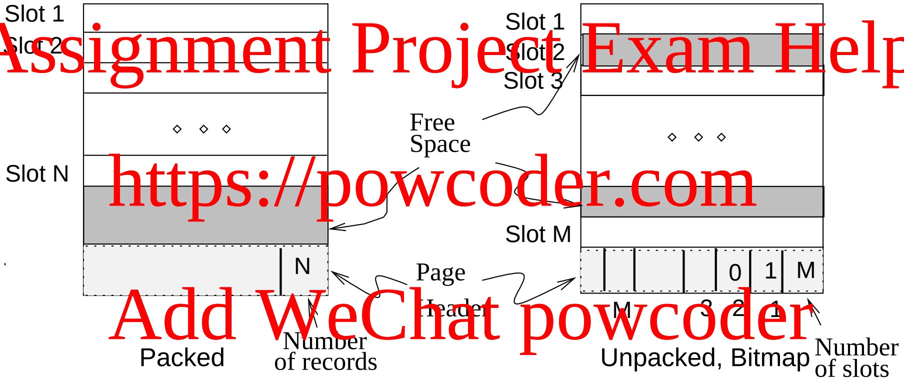
A page is thought of as having a number of slots, one for each record.

With fixed length records, there are two simple ways of arranging the records.

- Packed organisation: each page has a count of number of records in it. All the free space is at one end. Record deletion will involve moving records.

- Unpacked: Each page has a bit map indicating which slots are full and which are empty. Deleting a record simply involves setting the corresponding bit to zero.

Packed and Unpacked Organisations



Packed Organisation

In the packed page organisation, records are stored in the first N slots (where N is the number of records on a page). When a record is deleted, the last record on the page is moved to the empty slot.

Advantages:

- can obtain the position of the i^{th} record by a simple offset calculation
- all empty records are together at the end of the page

Disadvantage:

- Problems occur if there are external references to a record that is moved: its slot number will change, and the slot number is part of the *rid*

Unpacked Organisation

In the unpacked page organisation, an array of bits is used to keep track of free slot information (one bit per slot).

When a record is deleted, its bit is turned off. Locating records requires scanning the bit array to find slots whose bit is on. With this scheme, the *rid* is not affected when a record is deleted.

In both organisations, a page usually stores additional information in a page header. This includes file-level information, such as the id of the next page in the file.

Record Formats

Records may be fixed length or variable length.

If records are of a fixed-length, then compact storage is simple.

For variable-length records, the difficulty of compact storage depends on:

- frequency of insertion
- frequency of update and deletion
- whether system is available 24 hours
- how much “scratch space” is available
- whether record order is important

Representing Data Elements

All data is ultimately represented as a sequence of bytes.

Numeric types are represented by bit-strings that are specially interpreted by the machine's hardware. This allows the usual arithmetic operations to be performed on them.

INTEGER: typically 2 or 4 bytes

FLOAT: typically 4 or 8 bytes

Representing Data Elements...

Fixed-length character strings: CHAR(n).
These are represented using an array of n bytes. If the string is shorter than n, the array is filled with a pad character.
Example: assignment of the string “bat” to an attribute with type CHAR(6) would result in:

```
bat###
```

where # is a pad character whose 8-bit code is not a legal value for an SQL string.

Representing Data Elements...

Variable-length character strings: VARCHAR(n).

A possible implementation is to assign $n + 1$ bytes, no matter how long the string is. The extra byte is used to indicate the length of the string, and additional bytes are not used.

Example: assignment of the string “bat” to an attribute with type VARCHAR(6) would result in:

```
3bat
```

Note that 7 bytes are consumed (even though the last 3 are not actually used).

An alternative implementation would be to actually allocate a variable number of bytes. This would save space, but introduces added complexity through the creation of variable-length records.

Representing Data Elements...

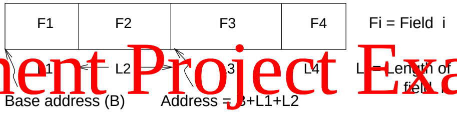
Dates: DATE

In SQL, the date type is a fixed-length string, so can be implemented like CHAR

Times: TIME An arbitrary-length string, can be implemented like VARCHAR (if an upper limit on length is imposed).

Bits: BIT(n) Can store 8 in a byte.

Fixed-length Records



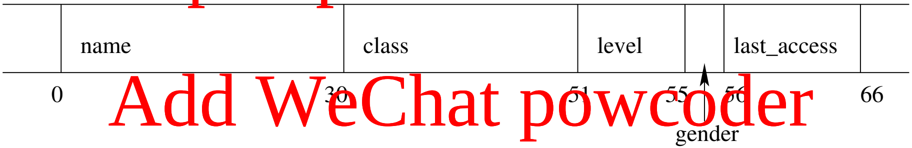
- Information about field types is usually the same for all records stored in file (if not, then a pointer to the schema can be stored with each record)

- All records have the same length, and fields are in the same order and at the same offsets in each record
- The address of the *i*th field can be calculated, allowing direct access

- Timestamps may be stored with each that indicate last modification or read time

(The structure of fixed-length records is discussed in Section 9.7.1.)

Fixed-length Records...



Variable-Length Records

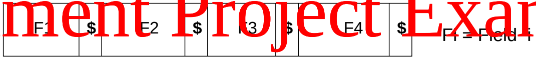
The size of fields may vary, fields may repeat, fields may be optional, fields may have variable formats that are not pre-declared before use and some fields may be large in size. (More details of field formats are in Section 9.7.2)

These constraints may lead to variable-length records.

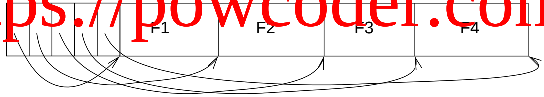
If fixed- and variable-length fields are stored in one record, then fixed-length fields can be stored first and then:

- The length of the record is stored in the header of the record
- Pointers are stored to the beginning of each of the variable-length fields (or variable-length fields are separated by a special character, that is, a delimiter)

Variable-Length Records



Fields Delimited by Special Symbol $



Array of Field Offsets

Two approaches: delimiters and direct field access using pointers.
Variable-length records are discussed further in Section 9.7.2.

Field Access Using Pointers

In the second approach, an array of integer offsets is stored at the start of the record.

The i^{th} item in the array gives the start address of field i, relative to the start of the record itself.

Advantages of this approach:

- Direct access to any field
- Easy to deal with NULL values

Variable-Length Records ...

We have seen that if there are variable length fields, we can use characters such as ? or $ or % as separator characters between the fields.

When the number of possible fields is large, but the number of fields in a typical record is small, we can store `<type,value>` instead of just storing values.

```
NAME=Smith%ADDR=124 Ann St.%WORK=fireman$
```

This can be improved further by storing codes such as “01” instead of “NAME”.

Page Formats and Variable-length Records

When records are of variable length, the previous approach of dividing a page into a fixed number of slots is no longer feasible.

When a new record is inserted, it is necessary to find space of the correct length to accommodate it.

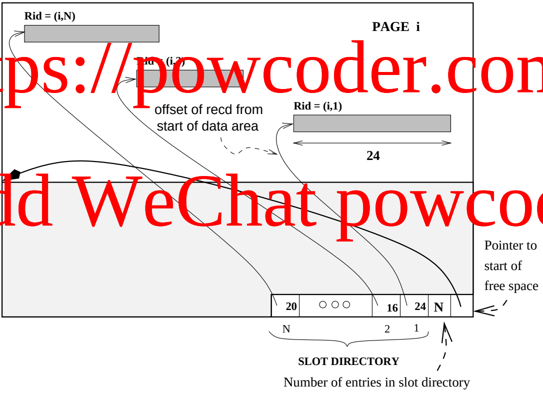
When a record is deleted, the hole that is created must be filled, so that all of the free space on the page is in one contiguous region.

We therefore need to be able to move records around on the page.

Page Formats

Variable length records are handled by having a directory of slots in each page.

This organisation enables moving of records, expansion of records, and so on.


Rid = (i,N)
PAGE i
Rid = (i,2)
offset of recd from start of data area
Rid = (i,1)
24
Pointer to start of free space
20
16
24
N
N
2
1
SLOT DIRECTORY
Number of entries in slot directory

Managing Free Space

Free space can be managed by maintaining a pointer to the start of the free space region.

When a new record is too large to fit into available space, records on the page are re-arranged so that the space created from earlier deletions is merged. After reorganisation, all records should be in contiguous order, followed by free space.

This organisation can also be used for fixed-length records (and is useful if the records need to be moved frequently).

Record Formats in Commercial Systems

IBM DB2:

- Fixed-length fields are at fixed offsets from the record's starting address
- Variable-length fields have offset and length stored in the fixed-length part of the record. Variable-length fields follow the fixed-length fields.

Oracle 8:

- Records are sequences of length and data pairs. That is, all fields are treated as if they are of variable length.

Page Format in Apache Derby

- tables (and indexes) stored in Containers
- data is stored in pages in Container
- a data page contains
  - formatId (4 bytes)
  - a page header (56 bytes)
  - a set of records
    accessed by either "slot" (defines order of records on page) or "id" (defines identity of record on page)
  - slot offset table
  - checksum (8 bytes)

http://db.apache.org/derby/papers/pageformats.html

Record Formats in Apache Derby

Apache Derby uses a variable length record, containing:

- a record header (1 byte) with bits indicating status of record, e.g.
  - whether record deleted
  - overflow for long rows
- one or more fields containing
  - a status byte (indicating e.g. NULL, overflow for long columns, ...)
  - fieldDataLength (a variable length CompressedInt)
  - fieldData

`http://db.apache.org/derby/papers/pageformats.html`

Slot Offset Table in Apache Derby

Slot Offset Table contains of 6 bytes (12 bytes when pagesize $\geq$ 64KiB) per record:

- 2 bytes page offset for record
- 2 bytes length of record on this page
- 2 bytes length of the reserved number of bytes for this record on this page

`http://db.apache.org/derby/papers/pageformats.html`

Record Format in MongoDB

- represents JSON documents in binary format called BSON
- supports multiple storage engines, e.g. WiredTiger, MMAPv1, In-Memory, Encrypted

`https://www.mongodb.com/json-and-bson`

BSON basic types

The following basic types are used as terminals in the rest of the grammar. Each type must be serialized in little-endian format.

```
byte 1 byte (8-bits)
int32 4 bytes (32-bit signed integer, two's complement)
int64 8 bytes (64-bit signed integer, two's complement)
uint64 8 bytes (64-bit unsigned integer)
double 8 bytes (64-bit IEEE 754-2008 binary floating point)
decimal128 16 bytes (128-bit IEEE 754-2008 decimal floating poi
```

http://bsonspec.org/spec.html

BSON grammar (incomplete extract of)

```
document ::= int32 e_list "\x00"   BSON Document:
                                   int32=total bytes in document
e_list ::= element e_list |   ""
element ::= "\x01" e_name double   64-bit binary floating point
     | "\x02" e_name string        UTF-8 string
     | "\x03" e_name document      Embedded document
     | "\x04" e_name document      Array
     | "\x05" e_name binary        Binary data
     | ...
e_name ::= cstring                 Key name
string ::= int32 (byte*) "\x00"    String: int32=number bytes in
                                   (byte*) +1 (for trailing '\x00')
                                   (byte*) is 0 or more UTF-8 chars
cstring ::= (byte*) "\x00"         0 or more modified UTF-8 chars
                                   followed by '\x00'
                                   (byte*) MUST NOT contain '\x00'
                                   hence not full UTF-8
binary ::= int32 subtype (byte*)   Binary:
                                   int32=number of bytes in (byte*)
```

BSON example

The following are some example documents in JSON and their corresponding BSON representations.

```
{"hello": "world"} ->

  \x16\x00\x00\x00                   // total document size
  \x02                               // 0x02 = type String
  hello\x00                          // field name
  \x06\x00\x00\x00world\x00          // field value
  \x00                  // 0x00 = type EOO ('end of object')
```

http://bsonspec.org/faq.html

BSON another example

```
{"BSON": ["awesome", 5.05, 1986]} ->

\x31\x00\x00\x00
\x04BSON\x00
\x26\x00\x00\x00
\x02\x30\x00\x08\x00\x00\x00awesome\x00
\x01\x31\x00\x33\x33\x33\x33\x33\x33\x14\x40
\x10\x32\x00\xc2\x07\x00\x00
\x00
\x00
```

http://bsonspec.org/faq.html

Records and Files

Database applications process logical records that must be read from secondary storage into memory.

When a logical record is required, its physical block is read into a memory buffer.

The logical record is unpacked from the physical record and passed to the DBMS.

Because a physical record may contain more than one logical record, every request for a logical record may not require a disk read of a physical block.

Similarly, a write may not necessarily cause a page to be written to disk. The record may be added to a buffer that is written to disk when a complete physical record has been constructed, or when dictated by buffer management policy.

Records and Files ...

Typically the DBMS must carry out the following operations:

- Find a record, transferring the corresponding block to the memory buffer and setting the found record as current
- Read a record: get physical block into buffer cache, unpack record, and transfer it to the DBMS
- Find the next record
- Delete a record
- Modify a record; may expand or reduce a record
- Insert a record

Basic Principles of Minimising Retrieval Cost

1. Data should be organised such that logically associated records are stored together—this is known as *clustering*. Other types of clustering (such as by frequency of access are possible).

2. Sort disk accesses, to maximise cache hits and minimise disk head movement. Almost essential in database environments.
3. Try to avoid long sequential reads. Index data if possible (more on indexing later).

4. Store data as densely as possible by organising data onto nearby tracks, using compact storage to increase the likelihood of retrieving several records in a disk block.

Summary

In this lecture, we have covered:

- Files and file organisation
- Records and record formats
- Pages and page formats
- Apache Derby record and page formats
- MongoDB record format