COSC2406/2407 Database Systems

File Organisations and Indexing

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 4

References: Ramakrishnan & Gehrke Chapter 8
Garcia-Molina et al. Chapter 13
Elmasri & Navathe Chapters 5 & 6

Slot Offset Table in Apache Derby [from last lecture]

Slot Offset Table contains of 6 bytes (12 bytes when pagesize $>$ 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

Note: 1 KiB (kibibyte) $=$ 1024 bytes
similarly 1MiB (mibibyte) $=$ 1024^2 bytes
to avoid confusion with 1MB (megabyte) $=$ 1000^2 bytes
etc

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

Overview: Week 4

In the first part of this lecture, we will:

- Introduce the cost model
- Analyse three common file organisations:
  - *heap files*
  - *files sorted on some fields*
    (see Section 8.4.3 of Ramakrishnan & Gehrke)
  - *files hashed on some fields*

In the second part of this lecture, we will continue with a discussion of indexes.

- Discuss properties of an index
- Discuss alternatives for data entries in an index

Cost Model for Our Analysis

In our discussion we will use a simple cost model.
The cost metric is the number of disk-block I/Os.
Usually the number of I/Os is the dominant cost in database applications. We ignore CPU costs and the use of *pre-fetching* of blocks (*blocked access*).
We express the costs of basic operations in terms of:

- *B*: the number of data blocks (or pages)
- *D*: (average) time to transfer a disk block

(The average-case analyses here are based on several simplistic assumptions.)

Simplifying Assumptions

Aside from *CPU costs* and *blocked access*, in our cost model we will ignore:

- time to do an equality comparison
- time to apply a hash function

In 2003 these were in the order of 100 nanoseconds, while I/O is in the order of 15 milliseconds. Therefore, I/O is the dominant cost.

These trends will continue to diverge: CPU speeds are rising much more quickly than disk access speeds — both have increased by a factor of over 100 since 2003.

Example

We will consider a file that stores data from the following `Character` relation:

NAME	LEVEL	CLASS
Frost	38	Mage
Moon	20	Warrior
Lysa	13	Druid
Varra	19	Warrior
Meerkat	18	Rogue
Shaka	2	Shaman
Cass	15	Mage
Otho	24	Hunter

Basic Operations for File Organisations

The operations we analyse are those identified last lecture:

- Scan:

Basic Operations for File Organisations

The operations we analyse are those identified last lecture:

- Scan: Fetch all records in the file

Basic Operations for File Organisations

The operations we analyse are those identified last lecture:

- Scan: Fetch all records in the file
- Search with equality selection:

Basic Operations for File Organisations

The operations we analyse are those identified last lecture:

- Scan: Fetch all records in the file
- Search with equality selection: Fetch all records that satisfy an equality selection

Basic Operations for File Organisations

The operations we analyse are those identified last lecture:

- Scan: Fetch all records in the file
- Search with equality selection: Fetch all records that satisfy an equality selection
  "Find the record for a character with *name* 'Lysa'."

Basic Operations for File Organisations

The operations we analyse are those identified last lecture:

- Scan: Fetch all records in the file
- Search with equality selection: Fetch all records that satisfy an equality selection
  "Find the record for a character with *name* ’Lysa’."
- Search with range selection:

Basic Operations for File Organisations

The operations we analyse are those identified last lecture:

- Scan: Fetch all records in the file
- Search with equality selection: Fetch all records that satisfy an equality selection
  "Find the record for a character with *name* 'Lysa'."
- Search with range selection: Fetch all records that satisfy a range selection

Basic Operations for File Organisations

The operations we analyse are those identified last lecture:

- Scan: Fetch all records in the file
- Search with equality selection: Fetch all records that satisfy an equality selection
  "Find the record for a character with *name* 'Lysa'."
- Search with range selection: Fetch all records that satisfy a range selection
  "Find all records of characters with *level* greater than 22."

Basic Operations for File Organisations

The operations we analyse are those identified last lecture:

- Scan: Fetch all records in the file
- Search with equality selection: Fetch all records that satisfy an equality selection
  "Find the record for a character with *name* 'Lysa'."
- Search with range selection: Fetch all records that satisfy a range selection
  "Find all records of characters with *level* greater than 22."
- Insert:

Basic Operations for File Organisations

The operations we analyse are those identified last lecture:

- Scan: Fetch all records in the file
- Search with equality selection: Fetch all records that satisfy an equality selection
  "Find the record for a character with *name* ’Lysa’."
- Search with range selection: Fetch all records that satisfy a range selection
  "Find all records of characters with *level* greater than 22."
- Insert: Insert a single, new record into the file

Basic Operations for File Organisations

The operations we analyse are those identified last lecture:

- Scan: Fetch all records in the file
- Search with equality selection: Fetch all records that satisfy an equality selection
  "Find the record for a character with *name* 'Lysa'."
- Search with range selection: Fetch all records that satisfy a range selection
  "Find all records of characters with *level* greater than 22."
- Insert: Insert a single, new record into the file
- Delete:

Basic Operations for File Organisations

The operations we analyse are those identified last lecture:

- Scan: Fetch all records in the file
- Search with equality selection: Fetch all records that satisfy an equality selection
  "Find the record for a character with *name* ’Lysa’."
- Search with range selection: Fetch all records that satisfy a range selection
  "Find all records of characters with *level* greater than 22."
- Insert: Insert a single, new record into the file
- Delete: Delete a single record specified by its record-id `rid`

Example (again)

We will consider a file that stores data from the following `Character` relation:

NAME	LEVEL	CLASS
Frost	38	Mage
Moon	20	Warrior
Lysa	13	Druid
Varra	19	Warrior
Meerkat	18	Rogue
Shaka	2	Shaman
Cass	15	Mage
Otho	24	Hunter

Heap File

Records are unordered:

Frost	38	Mage
Moon	20	Warrior
Lysa	13	Druid
Varra	19	Warrior
Meerkat	18	Rogue
Shaka	2	Shaman
Cass	15	Mage
Otho	24	Hunter

Heap Files

Remember that in a heap file, records in the file are unorganised. Here, for simplicity, we assume insertions are always at the end of file. Equality selections are on a unique key that is, we have exactly one match. Access costs on average:

- Scan: BD
- Equality Search: $0.5BD$
- Range Search: BD
- Insert: $2D$
- Delete: $Search + D$

To ensure a compact heap file, we need to keep and update a free space list for deletions and insertions (using the structures we discussed last week).

Linear Search

We wish to search for a data entry with key value 73 in the following array of 16 items:

Slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Key	7	12	21	22	26	34	56	67	68	69	71	73	89	91	92	94

For a linear search, the average cost is:

$$\frac{N}{2} = \frac{16}{2} = 8$$

Binary Search

Suppose that we again wish to search for a data entry with key value 73 in the following array of 16 items, this time using *binary search*:

Slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Key	7	12	21	22	26	34	56	61	68	69	71	73	89	91	92	94

For a binary search, the average cost is:

$$\log_2 N = \log_2 16 = 4$$

Sorted File

Records are sorted on name:

Cass	15	Mage
Frost	38	Mage
Lysa	13	Druid
Meerkat	18	Rogue
Moon	20	Warrior
Otho	24	Hunter
Shaka	2[illegible]	Shaman
Varra	19	Warrior

Sorted Files

A sorted file is like a heap file, but the file is sorted on a sequence of fields, which we call the *search key*.

A *search key* need not uniquely identify records. We can locate a record using a binary search on the search key.

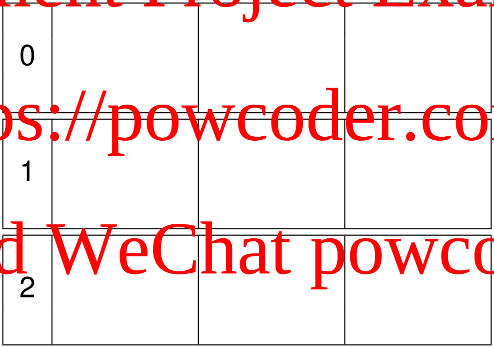
I/O cost on average:

- Scan: BD
- Equality Search: $D \log_2 B$
- Range Search: $D(\log_2 B +$ number of pages with matches$)$
- Insert: Search $+BD$
- Delete: Search $+BD$

Inserting and expanding records can be problematic.

Hashed File

Hash function: `level` mod 3



Hashed File

Hash function: `level` mod 3

0	Cass	15	Mage
	Meerkat	18	Rogue
	Otho	24	Hunter
1	Lysa	13	Druid
	Varra	19	Warrior
2	Frost	38	Mage
	Moon	20	Warrior
	Shaka	2	Shaman

Page Occupancy

Suppose that 100 records are to be stored in a file, and that 10 records can fit on one page.

- How many pages are needed?

Now suppose that an (initial) maximum occupancy of 80% is imposed.

- How many records fit on one page?

- How many pages are needed in total?

Static Hashed Files

The pages in a hashed file are grouped into buckets. We can apply a hash function to the search key to find out the bucket number to which a record belongs. We assume that we do not have overflow buckets. The page occupancy is assumed to be 80%. I/O cost on average:

- Scan: $1.25BD$ ($1.25 = \frac{1}{0.8}$; you need $1.25B$ blocks to store the records)
- Equality Search: D
- Range Search: $1.25BD$
- Insert: $2D$
- Delete: $2D$

Overflowing buckets decrease the performance of a static hashed file. Dynamic hash structures such as *Linear Hashing*, and *Extendible Hashing* address this problem.

Cost of Operation : Summary

Access Method	Heap File	Sorted File	Hashed File
Scan	BD	BD	$1.25BD$
Equality	$0.5BD$	$D \log_2 B$	D
Range Search	BD	$D(\log_2 B +$ $\#$ of match pages)	$1.25BD$
Insert	$2D$	Search $+BD$	$2D$
Delete	Search $+D$	Search $+BD$	$2D$

No file organisation is uniformly superior in all situations. *Indexes* are used to speed up operations that are not efficiently supported by the basic organisation.

Review: Alternative File Organisations

- Heap files: suitable when the typical access is a file scan to retrieve all records
- Sorted or *sequential files*: best if records must be retrieved in some order, or only a “range” of records are needed
- Hashed files: good for selecting records that match equality conditions
    - File is a collection of *buckets*;
      Bucket = *primary* page plus zero or more *overflow* pages
    - *Hashing function h* maps a record *r* into a bucket. *h* looks only at some of the fields of *r*, called the *search key*

Each file organisation works well for some situations but not for all.

Indexes

An *index* on a file speeds up selections on the *search key*

- Any subset of the fields of a relation can be the search key of an index.
- A *search key* is not the same as a *unique key* of a relation that uniquely identifies a record in a relation.

An index contains a collection of *data entries*, and supports efficient retrieval of all data entries $k*$ with a given search key value k.
There are three alternatives for a data entry $k*$ in an index.

Alternatives for the Data Entry $k*$ in an Index

Three alternatives:

1. Data record with search key value k
2. (k, `rid` of data record with search key value k)
3. (k, list of `rids` of data records with search key k)

The choice of an alternative for data entries is independent of the index technique used to locate data entries with a given value k. Any indexing technique can use one of the three alternatives above. Examples of indexing techniques include B+-trees and hash-based structures.

Typically, an index contains auxiliary information that directs searches to the desired data entries (for example, index entries in index pages in a B+-tree).

Alternatives for Data Entries ...

- Alternative 1:
  - If this is used, the index structure is in fact a file organisation for data records (e.g. like sorted file).
  - At most, one index on a given collection of data records can use Alternative 1. (Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency.)
  - If data records are very large, the number of pages containing data entries is high. This typically implies that the size of auxiliary information in the index is also large.

An Alternative 1 Index

An Alternative 1 index on `level`:

Shaka	2	Shaman
Lysa	13	Druid
Cass	15	Mage

Meerkat	18	Rogue
Varra	19	Warrior
Moon	20	Warrior

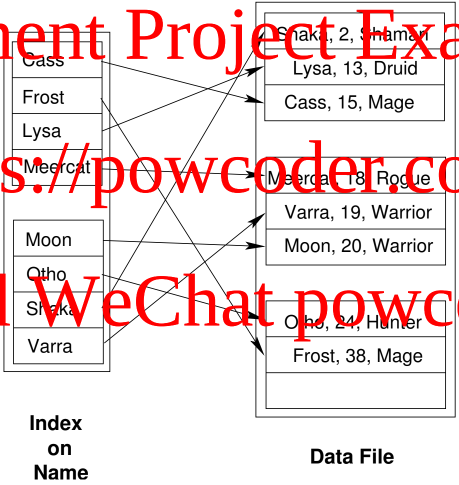
Otho	24	Hunter
Frost	38	Mage

Alternatives for Data Entries ...

- Alternatives 2 and 3:
  - Index data entries are typically much smaller than data records.
  - Therefore, more storage efficient than Alternative 1.
  - If more than one index is required on a file, only one can use Alternative 1, and remainder must use Alternatives 2 or 3.
  - Alternative 3 is most compact, but the variable size of the index entries is harder to handle (lists can grow and shrink in size).

Alternative 2 Index

An Alternative 2 index on `name`:



Cass

Frost

Lysa

Meercat

Moon

Otho

Shaka

Varra

Index on Name

Shaka, 2, Shaman

Lysa, 13, Druid

Cass, 15, Mage

Meercat, 18, Rogue

Varra, 19, Warrior

Moon, 20, Warrior

Otho, 24, Hunter

Frost, 38, Mage

Data File

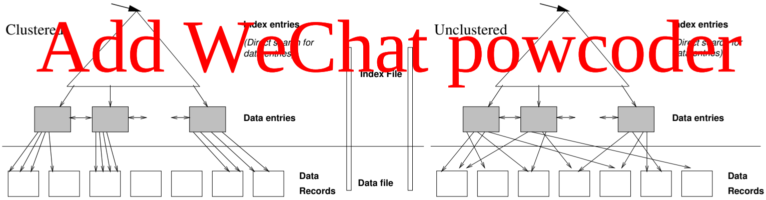
Index Classification

- *Primary* vs. *secondary*: If the search key contains the primary key, then the index is the primary index.
  - *Unique* index: Search key contains a candidate key.
- *Clustered* vs. *unclustered*: If order of data records is the same as, or “close to” the order of the data entries, then the index is a clustered index.
  - Using alternative 1 implies a clustered index, but not vice-versa
  - A file can be clustered at most on one search key
  - The cost of retrieving data records greatly depends on whether index is clustered or not

Clustered vs. Unclustered Indexes

Consider using alternative 2 used for the data entries and storing the data records in a heap file.

- To build clustered index, first sort the heap file (leaving free space on each page for future inserts).
- Overflow pages are used later for inserts. Thus, order of data records is 'close to' (but not identical) the sort order.



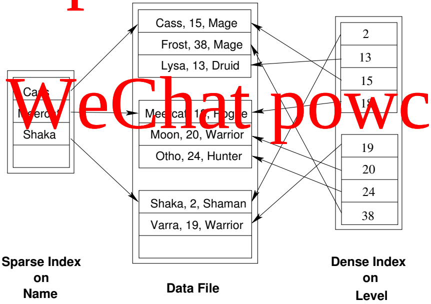
Dense vs. Sparse Indexes

- *Dense* vs *sparse*: If there is (at least) one data entry in the index per search key value then the index is dense. In a sparse index, we may have one data entry in the index for a page or set of records.
- Implications:
  - Alternative 1 always leads to a dense index
  - Every sparse index is clustered (otherwise we would ignore the order)
  - There is only one sparse index (since we can have only one clustered index)
  - Sparse indexes are smaller; however, some useful optimisations are based on dense indexes (refer to Section 12.5.2 of Ramakrishnan & Gehrke)

Dense vs. Sparse Indexes ...

The first index shown below is a sparse, clustered index on *name*. The order of data entries in the index corresponds to the order of records in the data file. There is one data entry per page of records.

The second index is dense, unclustered index on *level*. The order of data entries in the index differs from the order of data records. (There is one data entry in the index per record in the data field [Alternative 2].)



Cass
Meercat
Shaka

Cass, 15, Mage
Frost, 38, Mage
Lysa, 13, Druid
Meercat, 18, Rogue
Moon, 20, Warrior
Otho, 24, Hunter
Shaka, 2, Shaman
Varra, 19, Warrior

2
13
15
18
19
20
24
38

Sparse Index on Name

Data File

Dense Index on Level

Advantages and Disadvantages

- Clustered index: good for range queries. *Rids* of qualifying index entries point to a contiguous collection of records, hence few page I/Os
- Unclustered index: could lead to as many page I/Os as there are matching index entries. However, if we need more than one index, additional indexes must be unclustered
- Dense index: especially advantageous when index can fit into memory; can find a record with one I/O. Can determine from index alone whether a record exists
- Sparse index: smaller than a dense index, so can fit more into memory and can be searched quickly. However, may need to to an I/O just to check whether a record exists

Review

- Many alternative file organisations exist, each is appropriate for particular situations
- If selection queries are frequent, sorting the file (or building an index) is important
  - Hash-based files (or indexes) are only good for equality search
  - Sorted files (and tree-based indexes) are best for range searches, and also good for equality searches
- An index is a collection of data entries, plus a way to quickly find entries with given key values

Review ...

- Index data entries can be actual data records, (key, `rid`) pairs, or (key, `rid`-list) pairs.
    - The choice is independent of the indexing techniques used to locate data entries with a given key value
- There can be several indexes on a given file of data records, each with different search key
- Indexes can be classified as clustered or unclustered, primary or secondary, and dense or sparse. Differences have important consequences for utility and performance

Summary: File Organisations and Properties of Indexes

We have discussed:

- A file-access cost model based on the number of disk page I/Os as the cost metric.
- Three basic file organisations and their costs for common operations

- the properties of indexes:
  - *clustered* vs. *unclustered*;
  - *dense* vs. *sparse*;
  - *primary* vs. *secondary*.

- Alternatives for the index data entries $k*$ in an index.

In the next few lectures, we will cover hash-based indexing and tree-based indexing techniques like the B+ tree. We will also discuss a related topic, the external merge sort.