Chapter 17

Indexes

9 March 2017

Chris Sexton

TOC

Indexes

- primary mechanism for improving performance on a database
- takes advantage of the persistent data structures on disk
- many implementation details and decisions must be accounted for

Utility

- Indexes avoid full table scans by allowing fast access to tuples
- This is a difference beetween O(n) and O(1) execution time, or...
- If you have 5,000,000 entries in a database with an access time of 1ms per entry:

W|A: 5,000,000ms in minutes (http://www.wolframalpha.com/input/?i=5,000,000+ms+in+minutes)

- Versus instantaneous O(1) constant time lookup of the entry with a hash table
- Or O(*lg n*) (Or, ~22 operations) access with a tree or binary search on an ordered file

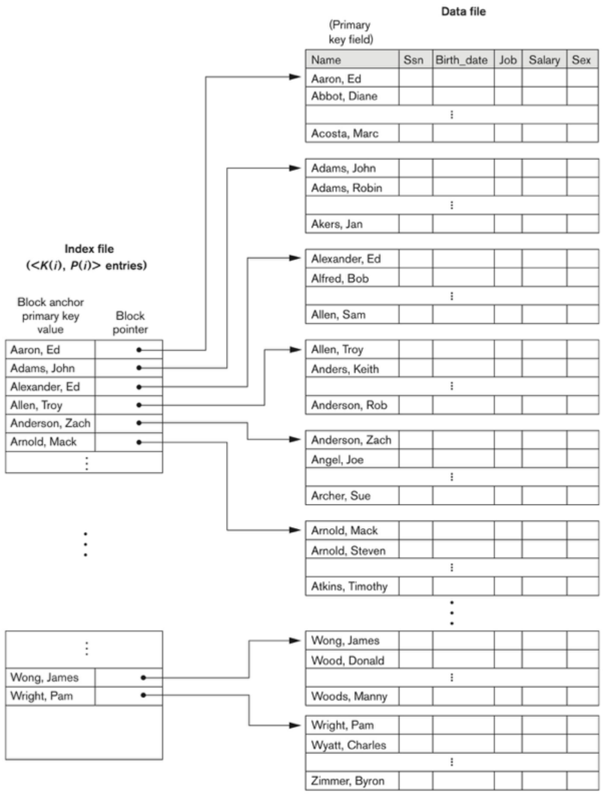
Primary Index

- A **primary index** is an ordered file with records of fixed length of two fields
- Used for primary key access where we are looking up an exact record
- Use binary search on the fixed-length ordered file to find entries
- Notation: <`K(i), P(i)`>: The first field, `K(i)` is the hashed key, `P(i)` is the pointer to the block containing the entry, `i`
- This looks up entries by block, meaning the index has fewer entries than the actual data file
- Since the data file is ordered by this key, retrieving a block corresponding to the key value is sufficient

```
<K(1) = (Aaron, Ed), P(1) = address of block 1>
<K(2) = (Adams, John), P(1) = address of block 2>
<K(3) = (Alexander, Ed), P(1) = address of block 3>
```

Primary Index

Figure 17.1 Primary index on the ordering key field of the file shown in Figure 16.7.



PEARSON

Primary Index

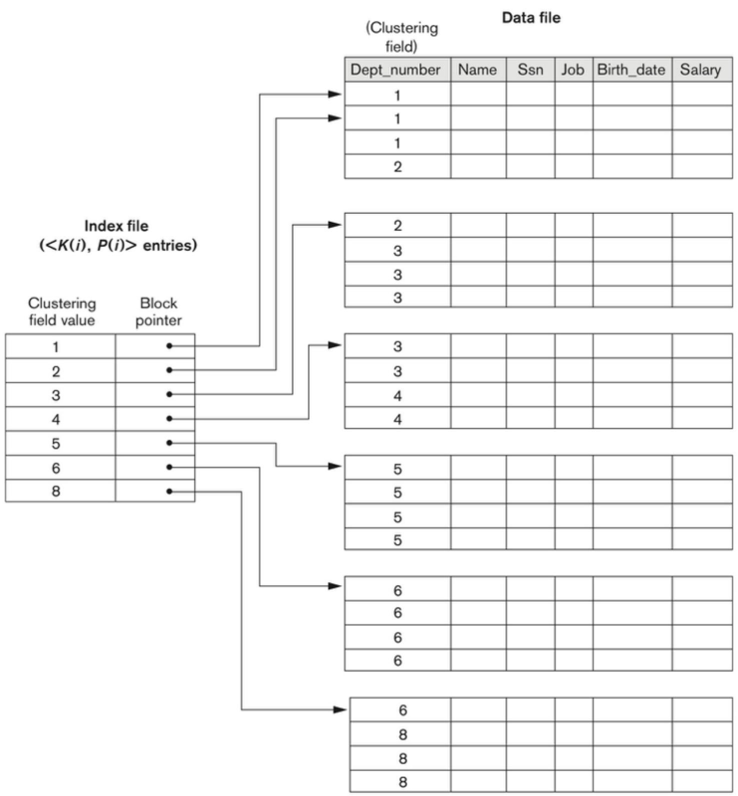
* Gains
  - The index file is strictly smaller than the data file
  - Searching the index file is ordered, so fewer disk/block accesses are necessary
* Issues
  - Insertion/Deletion is a problem in ordered files
  - We must "mark" entries for deletion
  - Insertion requires we move all entries "further" in the data file
  - Can use overflow linked lists or blocks to alleviate these issues

Clustering Indexes

- Used for non-key/non-unique fields
- Speeds up retrieval of records that have the same value for a particular field
- Utilizes the same method for storage as the primary index, except the block pointer points to an ordered data file

Clustering Indexes

Figure 17.2 A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.



Secondary Indexes

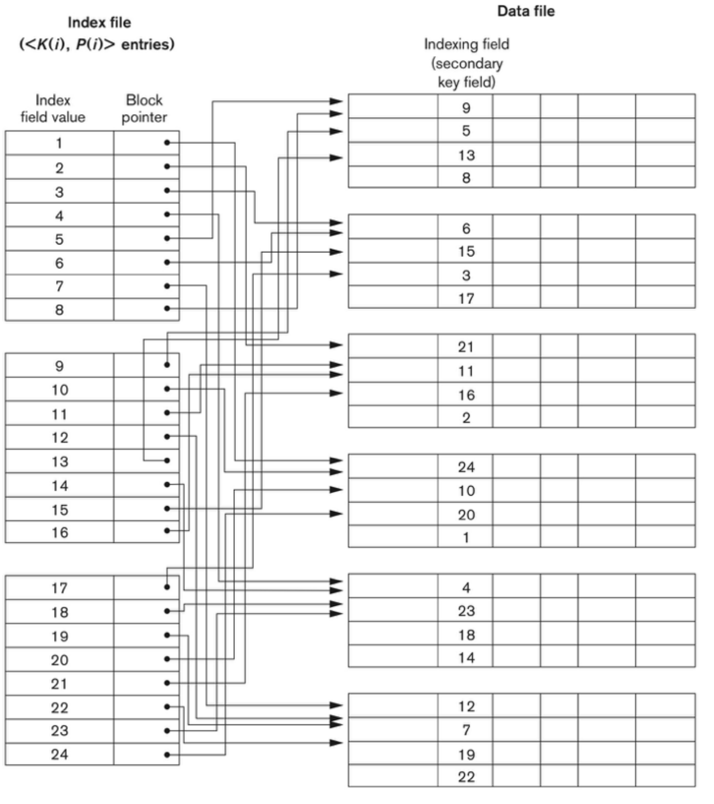
- Any field indexed that is not the primary means of accessing a particular set of data
- Could be ordered or unordered; unique or non-unique
- Can have many secondary indexes for any particular data set

Unique Secondary Indexes

- Called a **secondary key**
- Corresponds to any field with the `UNIQUE` attribute set
- Cannot use a block-level key mechanism since the data file is not ordered by these fields
- Therefore, uses more storage space than the primary key
- We would have to do a linear search on the data without a secondary key, but still can perform binary search with the key

Unique Secondary Indexes

Figure 17.4 A dense secondary index (with block pointers) on a nonordering key field of a file.

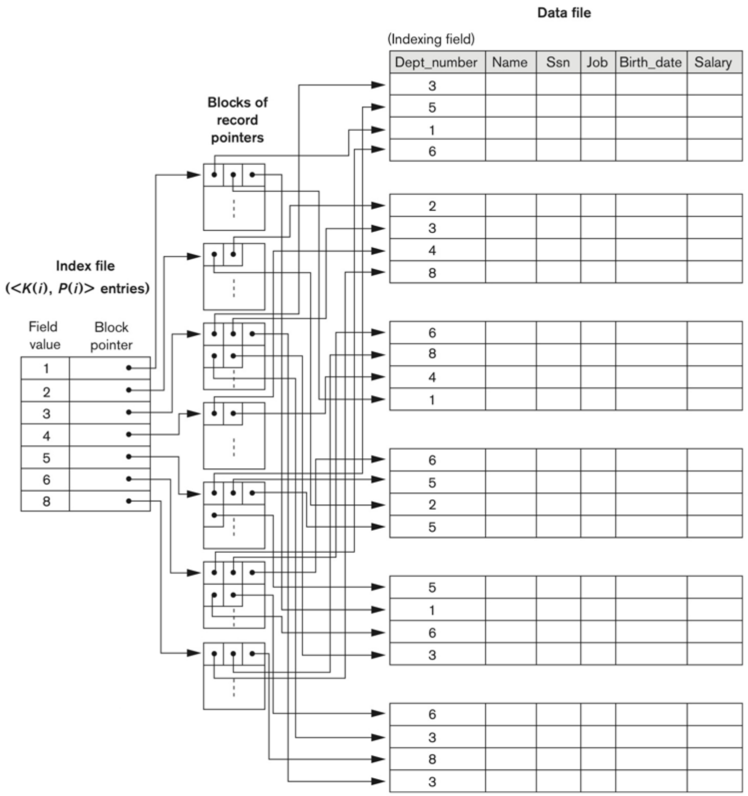


Non-Unique Secondary Indexes

- Secondary index on an entry which is not a primary key
- Uses a table of lookup values matching the field `K(i)` pointing to a collection of blocks related to that value
- Extra level of indirection, meaning extra time cost

Non-Unique Secondary Indexes

Figure 17.5 A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.



Index Types

Table 17.2 Properties of Index Types

Type of Index	Number of (First-Level) Index Entries	Dense or Nondense (Sparse)	Block Anchoring on the Data File
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no[a]
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records[b] or number of distinct index field values[c]	Dense or Nondense	No

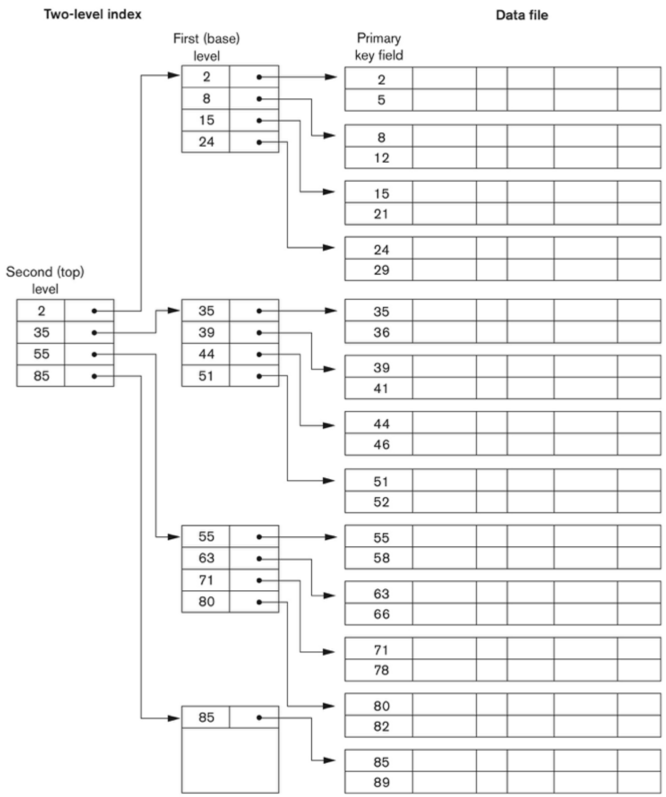
[a]Yes if every distinct value of the ordering field starts a new block; no otherwise.
[b]For option 1.
[c]For options 2 and 3.

Multilevel Indexes

- All previous indexes were on ordered index files with binary search
- A **multilivel index** reduces the search space even further by reducing disk reads
- Works for all previous types of indexes as long as the index has distinct values for `K(i)` and is of fixed-length

Multilevel Indexes

Figure 17.6 A two-level primary index resembling ISAM (indexed sequential access method) organization.



B-Trees

- A search tree is used to guide search for records given a particular field value
- Different from a multilevel index, a search tree allows ordering and dynamic insertion/deletion
- B-trees are balanced trees that minimize wasted space

* Complexity

- Space required is O(n)
- Insertion time: O($lg\ n$)
- Search time: O($lg\ n$)
- Removal time: O($lg\ n$)
- Ranged query (e.g. 1 < x < 5) of k elements: O($lg\ n+k$)

B+ Tree Basics (https://www.youtube.com/watch?v=CYKRMz8yzVU)

Implementation

* Balanced Trees

- B Trees and B+ trees
- Allow searching for values within an index

* Hash tables

- key/value constant lookup
- Useful with exact lookups

* Other methods

- many variations depending on DBMS

An example

```
select name
from student
where student_id = 38732
```

The student_id field is a unique primary key and has an automatic index.

An example

```
select name
from student
where name = 'George' and gpa > 3.5
```

- An index on the name may be a hash or tree
- An index on the GPA must be tree based to allow for searching values

An example

```
select s.name, d.name
from student s, department d
where s.dept = d.id
```

- Query planning & optimization allows for selection of available indexes
- Index on student's department and department's primary key
- Database must have had the student's department specified as an index

Downsides

- More space required for index storage
- Creation of index takes time
- Maintenance of index (on insert,update/delete) can take time and negate benefits

When to index

* Tuning indexes

- Must verify indexes are used
- Queries not hitting an index may use full table scans and take a long time to run
- Indexes on tables that change frequently may slow the system down due to index updates/rebuilds

* Choosing indexes is an exercise in judgement.

- How big is the table?
- What is the distribution of data?
- Is the data mostly being queried or updated? Do many deletes occur?

SQL Syntax for indexing

```
CREATE INDEX name ON table(attr)

CREATE INZDEX name ON table(attr1, attr2, ... attrN)

CREATE UNIQUE INDEX name ON table(attr)

DROP INDEX name
```

- Used when creating a table
- Will start an indexing process for previously created tables (and incur load)

Thank you

Chris Sexton