Hashing and Indexing
{Part 2: Indexing}

UNIVERSITY OF VIRGINIA
DATA SCIENCE
INSTITUTE

Indexing – Preface

- We all have heard the word **index** since our childhood: the index at the back of a book and the thumb index of a dictionary
- We can clearly conclude that indexing is **something to facilitate search**
- In real life search, an index can be a number, a string, a set of strings, an algorithm or a formula. The purpose is the same: to use a simple signal to find something bigger and more elaborate (*Recall*: "Key-Value" pairs)

Indexing and Databases

- Our discussion of indexing will describe its uses in the context of **databases** (database searches)
- The purpose of indexing here, as elsewhere, is to improve the speed of data retrieval, especially when the database has grown significantly (larger table = harder to find things!)
- In databases, search complexities get worse especially when doing **join operations** (resulting tables are very large)
- *Indexing is a way to speed up searches*

Indexing – Two main types

- In MySQL when a **primary key** is selected for a table, it will reorganize the records by creating a pointer-chain to have them in "cluster" order -- an **ordered index**
- Ordered index - the basic index structure used by *most* DBs

- The other type of index that many DBs use is the **hash index**

(The topic of Hashing has already been covered and will not be revisited here. The use of Hashing, as introduced earlier, is similar in the database context)

Ordered Indices

Primary Indices
Secondary Indices
Sparse Indices
Dense Indices

Primary vs. Secondary Indices

- **Primary index**
  - *Index-sequential files*: the file containing the records is sequentially ordered (e.g. city)
  - Clustering index (primary index): the index whose search key specifies the sequential order of the file

Brighton	217	750
Downtown	101	500
Downtown	110	600
Mianus	215	700
Perryridge	102	400
Perryridge	201	900
Perryridge	218	700
Redwood	222	700
Round Hill	305	350

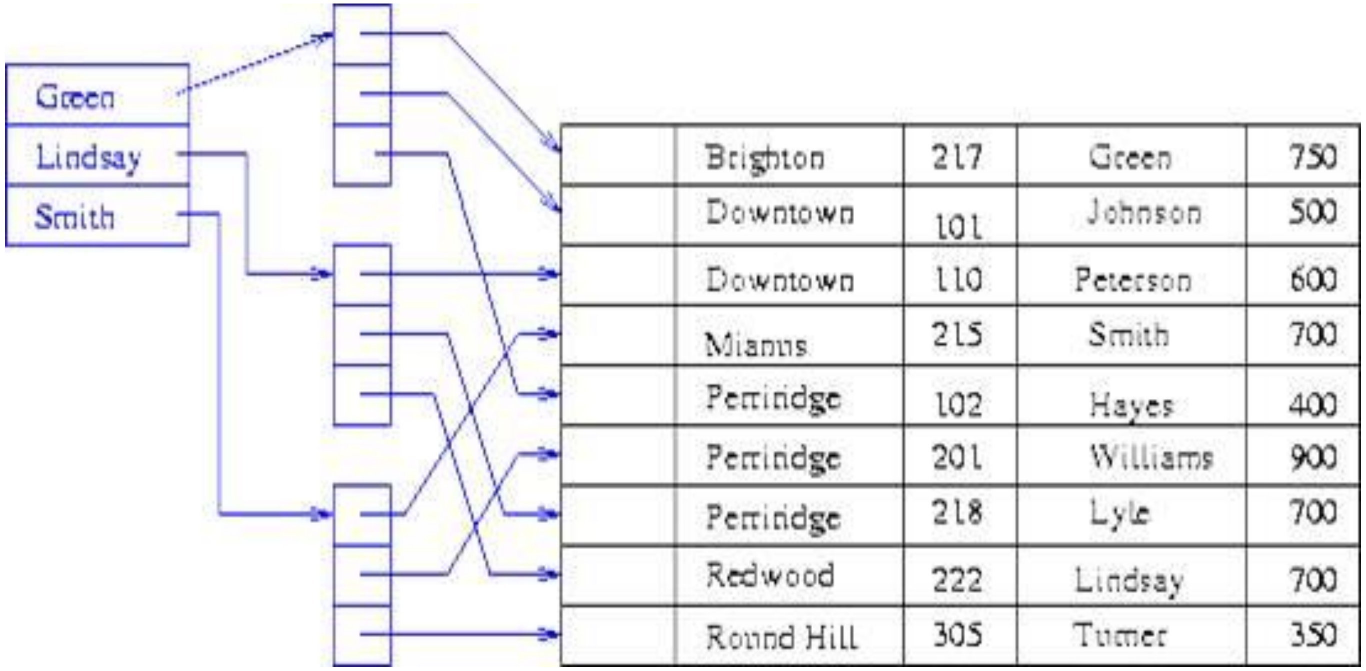
Primary vs. Secondary Indices

- **Secondary Index**
  - Indices whose search key *specifies an order different from the sequential order of the* ***file*** are called the **secondary indices**, or **non-clustering indices**
  - If the search key of a secondary index is not a **candidate key**, it is not enough to point to just the first record with each search-key value because the remaining records with the same search-key value could be *anywhere* in the file. Therefore, a secondary index must contain pointers to *all* the records (must be ***dense***)

Recall: *"Order different from the sequential order of the **file**"*

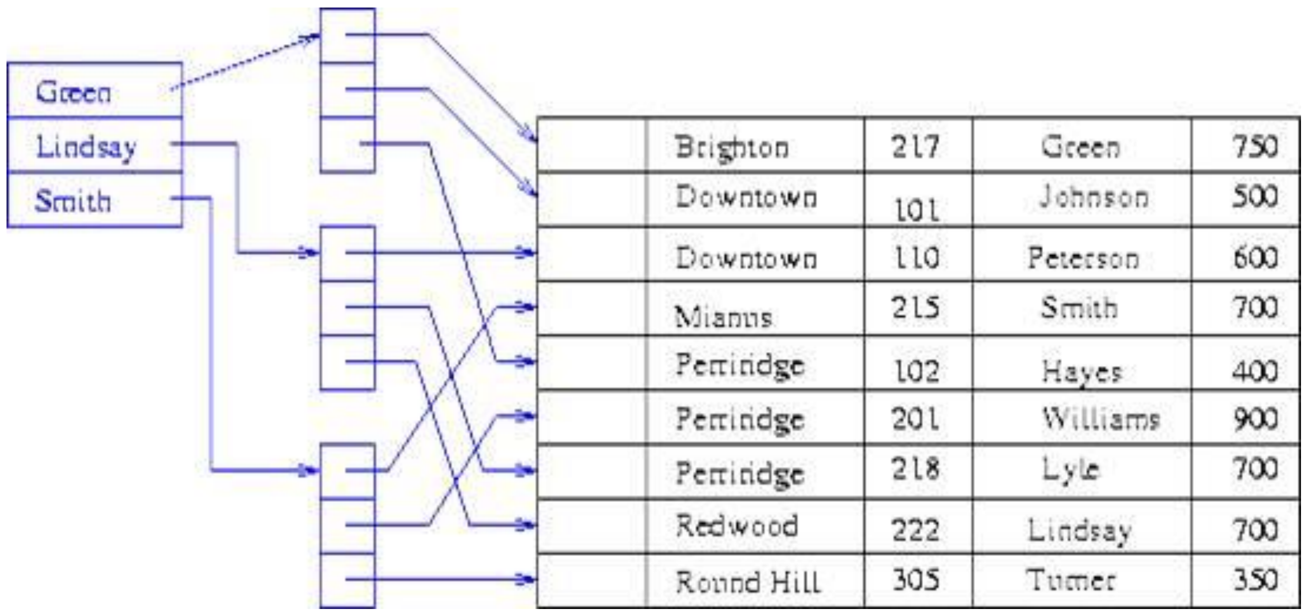
Primary vs. Secondary Indices

- Secondary index on *cname*
- Secondary Index – we can use an **extra-level of indirection** to implement secondary indices *on search keys that are not candidate keys*. A pointer does not point directly to the file but to a bucket that contains pointers to the file



Primary vs. Secondary Indices

- To perform a lookup on *Peterson*, we must read all three records pointed to by entries in bucket 2
- Only one entry points to a Peterson record, but three records need to be read
- As file is not ordered physically by *cname*, this may take 3 block accesses



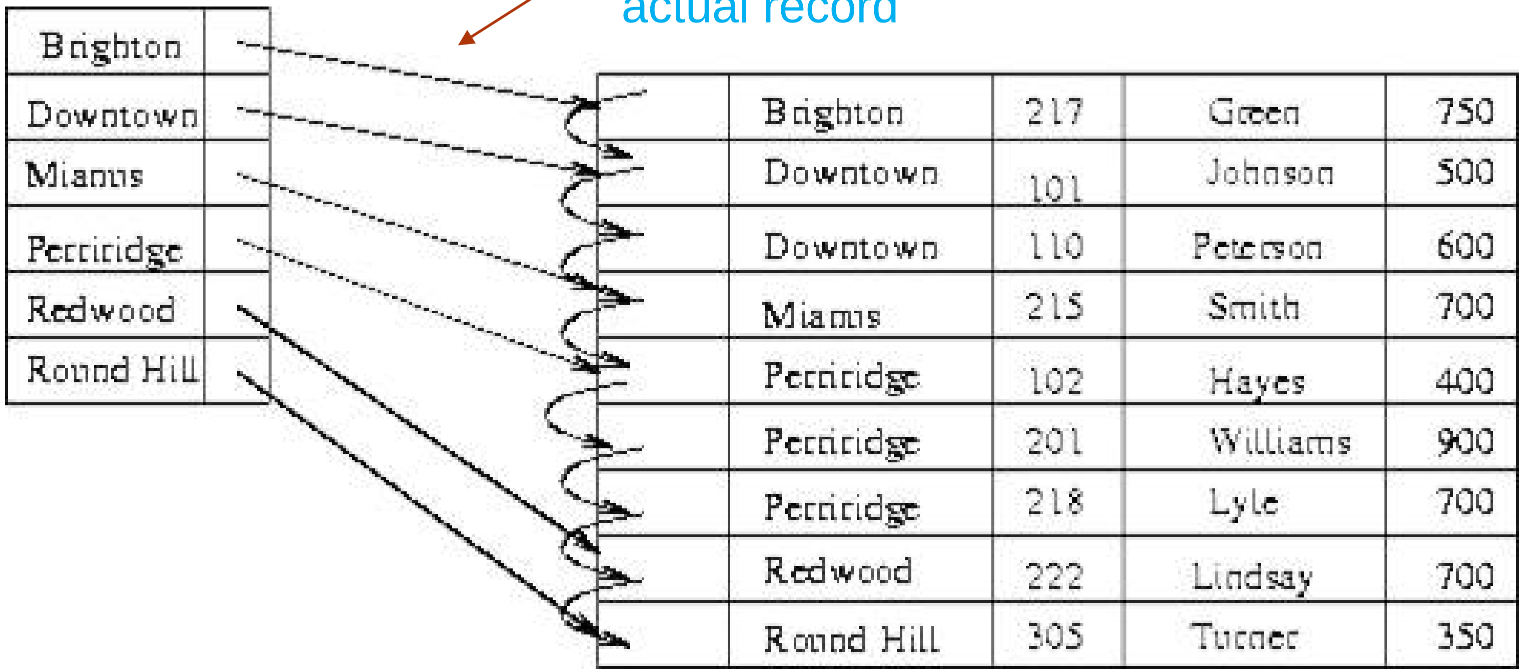
Overhead and Performance Issues

- Secondary indices improve the performance of queries on non-primary keys
- They also impose serious **overhead** on database modification: whenever a file is updated, *every index must be updated*
- Designer must decide whether to use secondary indices or not

Dense Index Example

Search Key

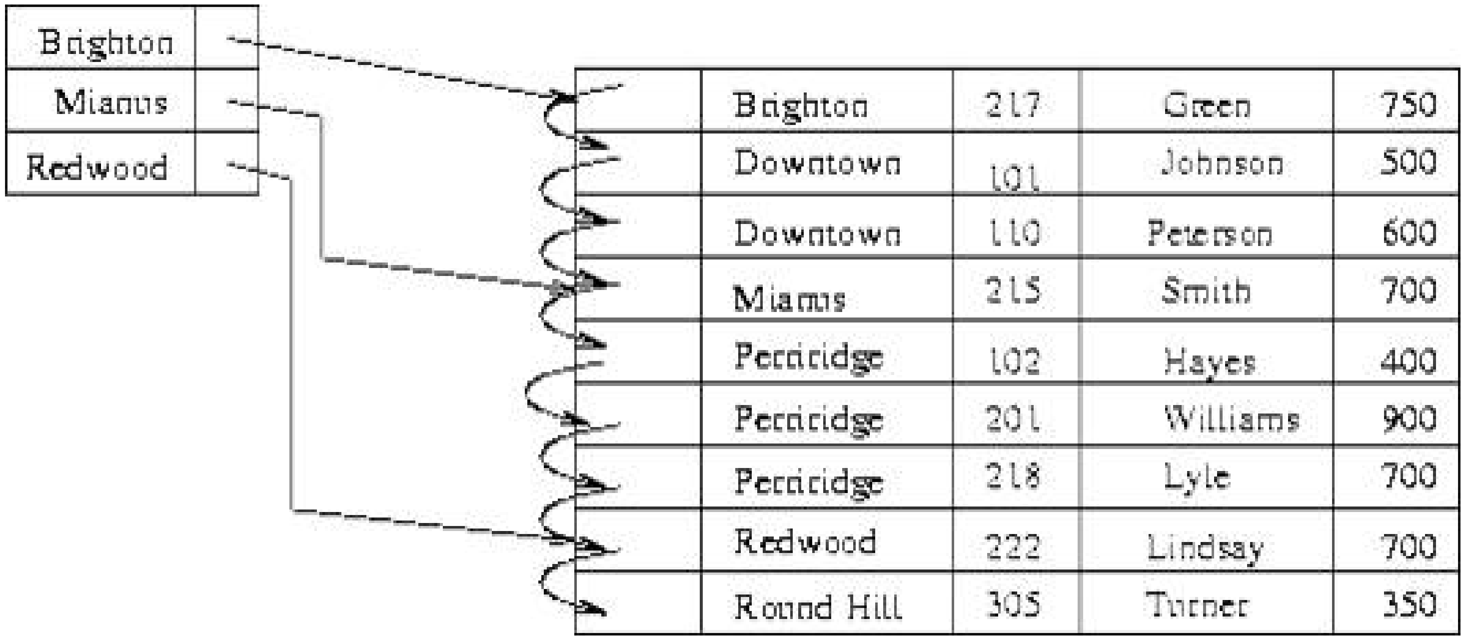
Pointers to the actual record



An index record appears for **every** search key value in file

Sparse Index Example

- Index records are created only for **some** of the records
- To locate a record, we find the index record with the largest search key value **less than or equal to** the search key value we are looking for
- We start at that record pointed to by the index record, and proceed along the pointers in the file (that is, sequentially) until we find the desired record



Discussion

- **Dense** indices don't give us much of a bonus on an ordered table – a primary index that is dense on an ordered table is <u>*redundant*</u>*!*

- Therefore a **primary** index on an *ordered* table is always sparse

- Otherwise (unordered) use dense or secondary indices

- **Dense** indices are **faster** in general, but **sparse** indices require **less space** and impose less maintenance for insertions and deletions

Discussion

- When you create a primary key for a table, it will create a record structure such that you have an ordered table based on that primary key
- It will then start building **sparse indices** – at least one sparse index on that record
- If you declare something as **unique** *(a property you can assign to a particular attribute)*– it will create a secondary index
- General: helps in "for where" clauses, helps for foreign keys, helps for "select… where…" queries

Discussion – How much Indexing To Do?

- However, for every index you create you are introducing more data and more **overhead** (especially when doing inserts / deletes / updates)
- As DB admin – how much indexing to do?
  - Read-heavy DBs – can index a lot (*if got the memory to do it*)
  - Write-heavy DBs – index sparingly! (*take a balanced approach*)
  - Write-ONLY DBs – one or no index (*e.g. log table*)

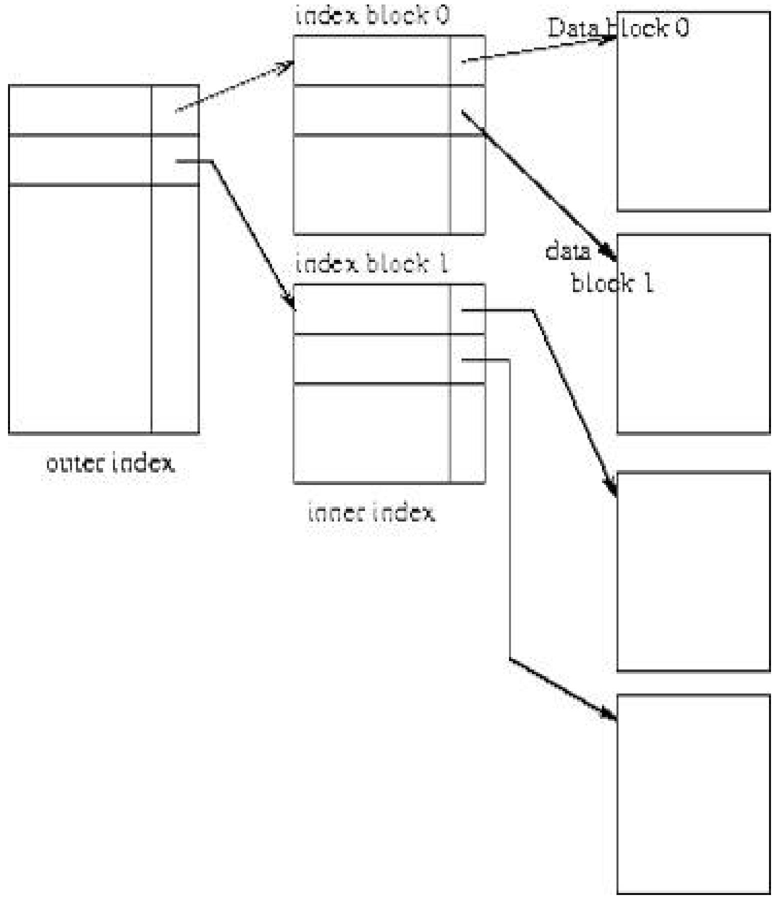
Supplementary Information

Multi-Level Indices

- Even with a sparse index, index size may still grow too large. For 100,000 records, 10 per block, at one index record per block, that's 10,000 index records! Even if we can fit 100 index records per block, this is 100 blocks
- If index is too large to be kept in main memory, one search results in several disk reads
  - If there are no overflow blocks in the index, we can use **binary search**
  - If index has overflow blocks, then **sequential search** typically used
- Solution: Construct a sparse index on the index

Two-level Sparse Index Example

- Use binary search on outer index. Scan **index block** until correct index record found. Use index record as before - scan block pointed to for desired record
- For very large files, additional levels of indexing may be required
- Indices must be updated at all levels when insertions or deletions require it
- Frequently, each level of index corresponds to a unit of physical storage (e.g. indices at the level of track, cylinder and disk)



Index Update

- Regardless of what form of index is used, every index must be updated whenever a record is either inserted into or deleted from the file
- **Deletion**:
  - Find (look up) the record
  - If it is the last record with a particular search key value, **delete** that search key value from index
  - For dense indices, this is like deleting a record in a file
  - For sparse indices, delete a key value by replacing key value's entry in index by next search key value. If that value already has an index entry, delete the entry
- **Insertion**:
  - Find place to insert
  - Dense index: insert search key value if not present
  - Sparse index: no change unless new block is created. (In this case, the first search key value appearing in the new block is inserted into the index)

From Theoretical to Actual: B+Trees

- Primary disadvantage of index-sequential file organization is that performance degrades as the file grows. This can be remedied by costly re-organizations
- B+ tree file structure maintains its efficiency despite frequent insertions and deletions. It imposes some *acceptable* update and space overheads
- A B+ tree index is a *balanced tree* in which every path from the root to a leaf is of the same length