

### **Indexing and Hash Look Up**

Hi! In this chapter I am going to discuss with you about the usage of indexing in DBMS. Anyway Databases are used to store information. The principle operations we need to perform, therefore, are those relating to

- (a) Creation of data,
- (b) Changing some information, or
- (c) Deleting some information which we are sure is no longer useful or valid.

We have seen that in terms of the logical operations to be performed on the data, relational tables provide a beautiful mechanism for all of the three above tasks. Therefore the storage of a Database in a computer memory (on the Hard Disk, of course), is mainly concerned with the following issues:

1. The need to store a set of tables, where each table can be stored as an independent file.
2. The attributes in a table are closely related, and therefore, often accessed together.

Therefore it makes sense to store the different attribute values in each record contiguously. In fact, the attributes **MUST** be stored in the same sequence, for each record of a table.

It seems logical to store all the records of a table contiguously; however, since there is no prescribed order in which records must be stored in a table, we may choose the sequence in which we store the different records of a table.

### **A Brief Introduction to Data Storage on Hard Disks**

Each Hard Drive is usually composed of a set disks. Each Disk has a layer of magnetic material deposited on its surface. The entire disk can contain a large amount of data, which is organized into smaller packages called **BLOCKS** (or **pages**). On most computers, one block is equivalent to 1 KB of data (= 1024 Bytes).

A block is the smallest unit of data transfer between the hard disk and the processor of the computer. Each block therefore has a fixed, assigned, address. Typically, the computer processor will submit a read/write request, which includes the address of the block, and the address of RAM in the computer memory area called a buffer (or cache) where the data must be stored/taken from. The processor then reads and modifies the buffer data as required, and, if required, writes the block back to the disk.

### **How are tables stored on Disk?**

We realize that each record of a table can contain different amount of data. This is because in some records, some attribute values may be 'null'. Or, some attributes may be of type varchar (), and therefore each record may have a different length string as the value of this attribute.

Therefore, the record is stored with each subsequent attribute separated by the next by a special ASCII character called a field separator. Of course, in each block, there we may place many records. Each record is separated from the next, again by another special ASCII character called the record separator. The figure below shows a typical arrangement of the data on a disk.

### **How indexes improve the performance?**

Indexes improve the performance of queries that select a small percentage of rows from a table. As a general guideline, create indexes on tables that are queried for less than 2% or 4% of the table's rows. This value may be higher in situations where all data can be retrieved from an index, or where the indexed columns and expressions can be used for joining to other tables.

This guideline is based on these assumptions:

- Rows with the same value for the key on which the query is based are uniformly distributed throughout the data blocks allocated to the table

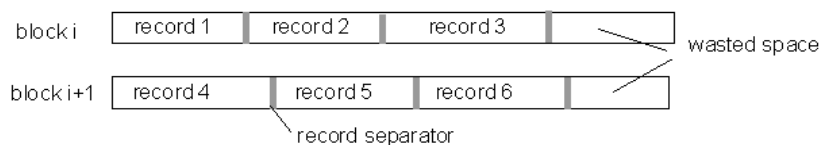
- Rows in the table are randomly ordered with respect to the key on which the query is based
- The table contains a relatively small number of columns
- Most queries on the table have relatively simple WHERE clauses
- The cache hit ratio is low and there is no operating system cache

If these assumptions do not describe the data in your table and the queries that access it, then an index may not be helpful unless your queries typically access at least 25% of the table's rows

	Name	SSN	Salary	Department
record 1	John Smith	1112223333	1000	Research
record 2	William Arnold	9998887777	30000	Administration

field separator

The records of such a table may be stored, typically, in a large file, which runs into several blocks of data. The physical storage may look something like the following:



The simplest method of storing a Database table on the computer disk, therefore, would be to merely store all the records of a table in the order in which they are created, on contiguous blocks as shown above, in a large file. Such files are called HEAP files, or a PILE.

### **Storage methods in terms of the operations**

We shall examine the storage methods in terms of the operations we need to perform on the Database:

***In a HEAP file:***

Operation: Insert a new record

Performance: Very fast

Method: The heap file data records the address of the first block, and the file size (in blocks). It is therefore easy to calculate the last block of the file, which is directly copied into the buffer. The new record is inserted at the end of the last existing record, and the block is written back to the disk.

Operation: Search for a record (or update a record)

Performance: Not too good ( on an average,  $O(b/2)$  blocks will have to be searched for a file of size  $b$  blocks.)

Method: Linear search. Each block is copied to buffer, and each record in the block is checked to match the search criterion. If no match is found, go to the next block, etc.

Operation: Delete a record

Performance: Not too good.

Method: First, we must search the record that is to be deleted (requires linear search). This is inefficient. Another reason this operation is troublesome is that after the record is deleted, the block has some extra (unused) space. What should we do about the unused space?

To deal with the ***deletion problem***, two approaches are used:

(a) Delete the space and rewrite the block. At periodic intervals (say few days), read the entire file into a large RAM buffer and write it back into a new file.

(b) For each deleted record, instead of re-writing the entire block, just use an extra bit per record, which is the 'RECORD\_DELETED' marker. If this bit has a value 1, the record is ignored in all searches, and therefore is equivalent to deleting the record. In this case, the deletion operation only requires setting one bit of data before re-writing the block (faster). However, after fixed intervals, the file needs to be updated just as in case (a), to recover wasted space.

Heaps are quite inefficient when we need to search for data in large database tables. In such cases, it is better to store the records in a way that allows for very fast searching.

The simplest method to do so is to organize the table in a Sorted File. The entire file is sorted by increasing value of one of the fields (attribute value). If the ordering attribute (field) is also a key attribute, it is called the ordering key.

The figure below shows an example of a sorted file of n blocks.

	Name	SSN	Job	Salary
Block 1	Aaron			
	Abbot			
	Acosta			
Block 2	Adams			
	Akers			
Block 3	Alex			
	Allen			
Block 4	Anders			
	Anderson			
Block 5	Arnold			
	Atkins			
Block n	Wong			
	Zimmer			

In addition, most DBMS's provide another mechanism to quickly search for records, at the cost of using some extra disk space. This is the use of INDEXES.

### What is an INDEX?

In a book, the index is an alphabetical listing of topics, along with the page number where the topic appears. The idea of an INDEX in a Database is similar. We will consider two popular types of indexes, and see how they work, and why they are useful.

### Ordered Indices


1. In order to allow fast **random** access, an index structure may be used.
2. A file may have several indices on different search keys.
3. If the file containing the records is sequentially ordered, the index whose search key specifies the sequential order of the file is the **primary index**, or **clustering**

- index.** Note: The search key of a primary index is usually the primary key, but it is not necessarily so.
4. Indices whose search key specifies an order different from the sequential order of the file are called the **secondary indices**, or **nonclustering indices**.

### Primary Indexes

*Index-sequential files:* Files are ordered sequentially on some search key, and a primary index is associated with it

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

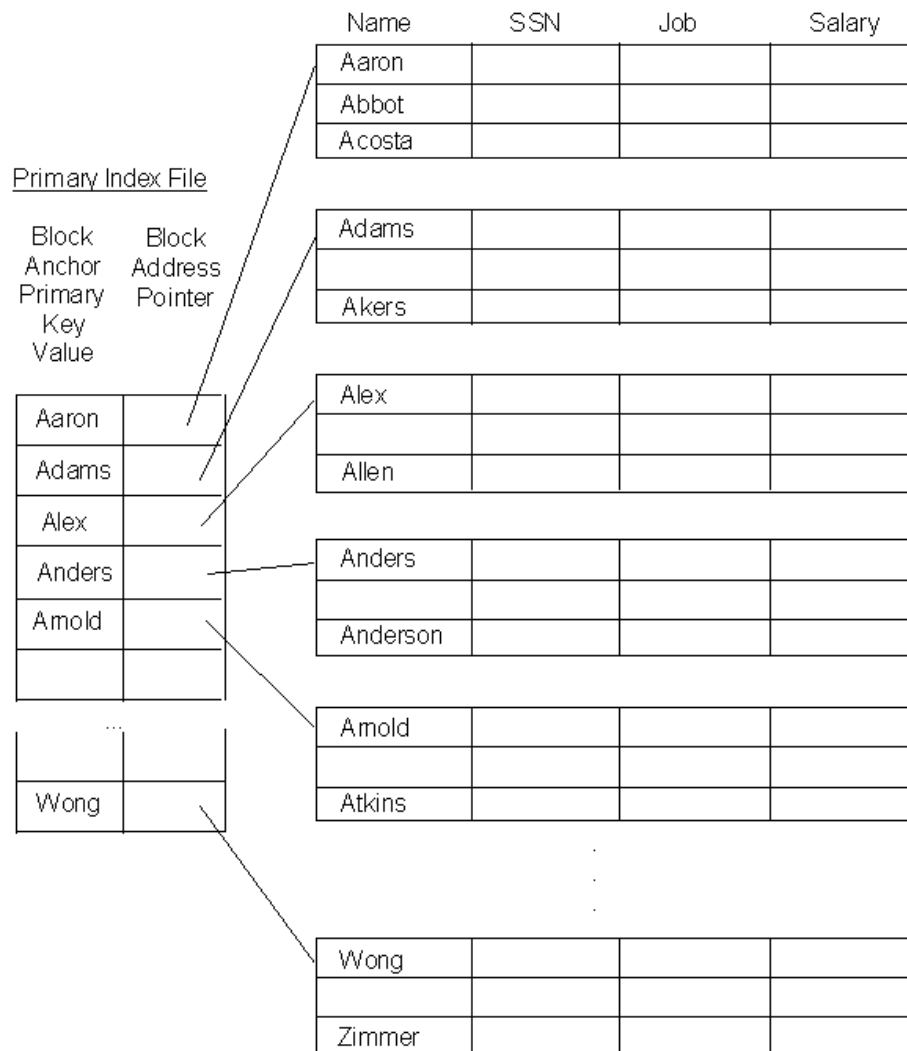


Consider a table, with a Primary Key Attribute being used to store it as an ordered array (that is, the records of the table are stored in order of increasing value of the Primary Key attribute.)

As we know, each BLOCK of memory will store a few records of this table. Since all search operations require transfers of complete blocks, to search for a particular record, we must first need to know which block it is stored in. If we know the address of the block where the record is stored, searching for the record is VERY FAST!

Notice also that we can order the records using the Primary Key attribute values. Thus, if we just know the primary key attribute value of the first record in each block, we can determine quite quickly whether a given record can be found in some block or not.

This is the idea used to generate the Primary Index file for the table. We will see how this works by means of a simple example.



### ***Again a problem arises...***

The Primary Index will work only if the file is an ordered file. What if we want to insert a new record?

### ***Answer for that question is***

We are not allowed to insert records into the table at their proper location. This would require (a) finding the location where this record must be inserted, (b) Shifting all records at this location and beyond, further down in the computer memory, and (c) inserting this record into its correct place.

Clearly, such an operation will be very time-consuming!

### ***So what is the solution?***

The solution to this problem is simple. When an insertion is required, the new record is inserted into an unordered set of records called the overflow area for the table. Once every few days, the ordered and overflow tables are merged together, and the Primary Index file is updated.

Thus any search for a record first looks for the INDEX file, and searches for the record in the indicated Block. If the record is not found, then a further, linear search is conducted in the overflow area for a possible match.

### Dense and Sparse Indices

1. There are Two types of ordered indices:

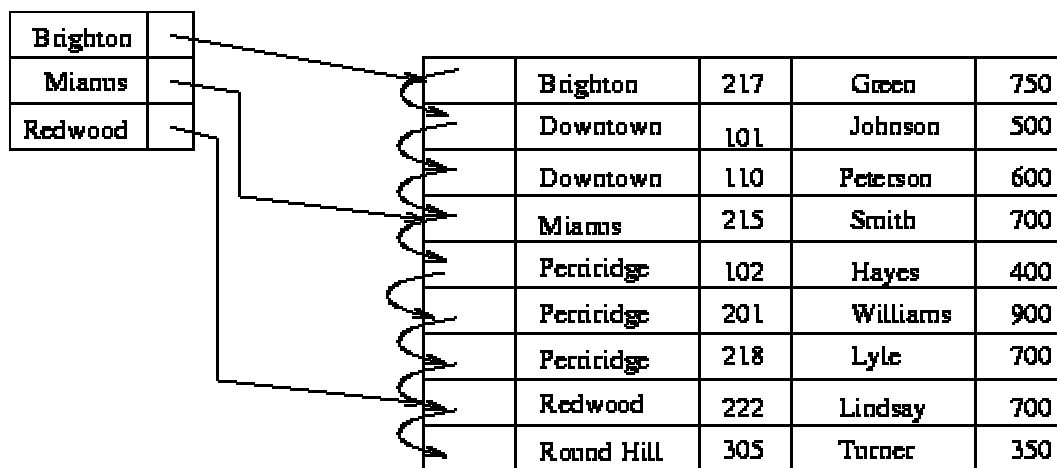
#### **Dense Index:**

- An index record appears for **every** search key value in file.
- This record contains search key value and a pointer to the actual record.

#### **Sparse Index:**

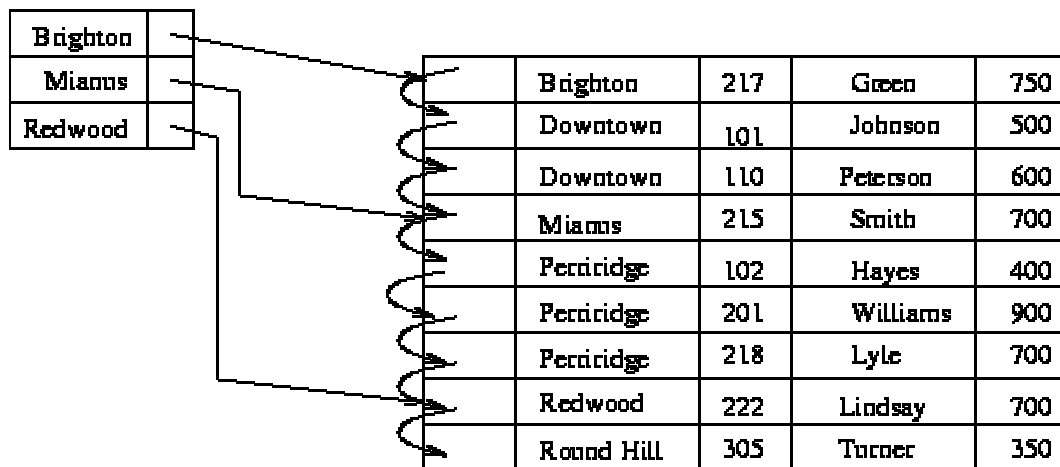
- Index records are created only for **some** of the records.
- To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.
- We start at that record pointed to by the index record, and proceed along the pointers in the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

2. Figures 11.2 and 11.3 show dense and sparse indices for the deposit file.



Dense Index



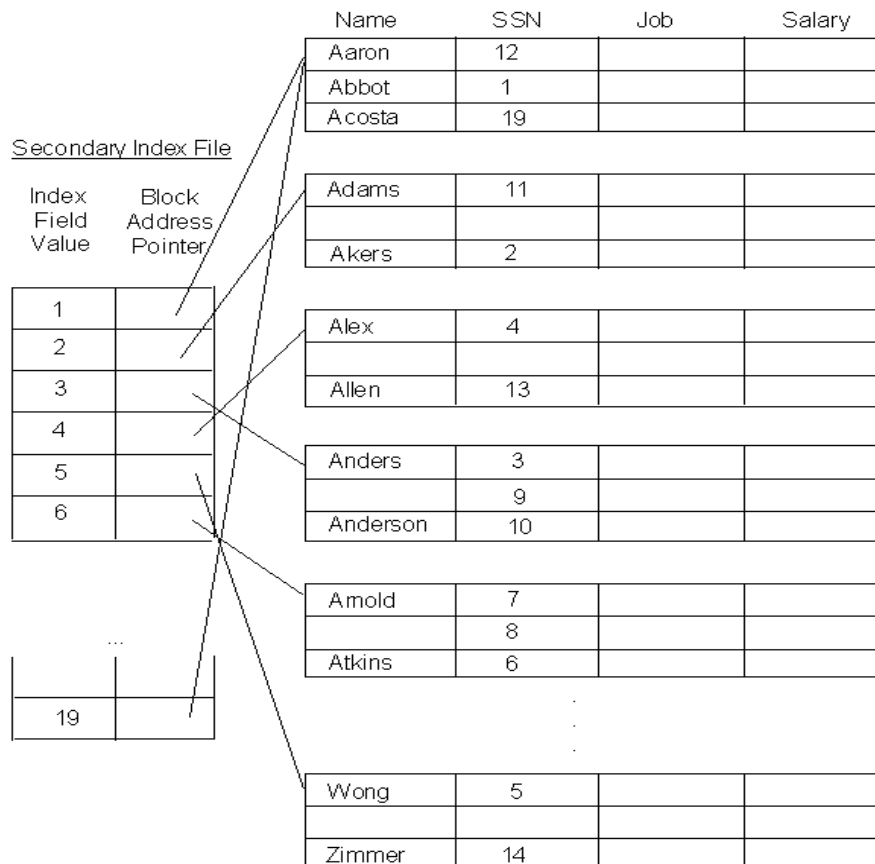
Sparse Index**Secondary Indexes**

Apart from primary indexes, one can also create an index based on some other attribute of the table. We describe the concept of Secondary Indexes for a Key attribute (that is, for an attribute which is not the Primary Key, but still has unique values for each record of the table). In our previous example, we could, for instance, create an index based on the SSN.

The idea is similar to the primary index. However, we have already ordered the records of our table using the Primary key. We cannot order the records again using the secondary key (since that will destroy the utility of the Primary Index !)

Therefore, the Secondary Index is a two column file, which stores the address of EVERY tuple of the table !

The figure below shows the secondary index for our example:



Unlike the Primary Index, where we need to store only the Block Anchor, and its Block Address, in the case of Secondary Index Files, we need to store one entry for EACH record in the table. Therefore, secondary index files are much larger than Primary Index Files. You can also create Secondary Indexes for non-Key attributes. The idea is similar, though the storage details are slightly different.

You can create as many indexes for each table as you like !

### Creating Indexes using SQL

Example:

Create an index file for Lname attribute of the EMPLOYEE Table of our Database.

Solution:

**CREATE INDEX myLnameIndex ON EMPLOYEE( Lname);**

This command will create an index file which contains all entries corresponding to the rows of the EMPLOYEE table sorted by Lname in Ascending Order.

Example:

You can also create an Index on a combination of attributes:

**CREATE INDEX myNamesIndex ON EMPLOYEE( Lname, Fname);**

Finally, you can delete an index by using the following SQL command:

Example:

**DROP INDEX myNamesIndex;**

which will drop the index created by the previous command?

Note that every index you create will result in the usage of memory on your Database server. This memory space on the hard Disk is used by the DBMS to store the Index File(s). Therefore, the advantage of faster access time that you get by creating indexes also causes the usage of extra memory space.

*Now that was all about Indexing, I would like to proceed to one of the similar kind of a topic like indexing, which is hashing.*

### **What is hashing?**

A hashed data structure, typically referred to as a *hash table* provides the following performance pattern:

1. insertion, deletion, and search for a specific element
2. search for a successive item in sorted order

A hash table is useful in an application that needs to rapidly store and look up collection elements, without concern for the sorted order of the elements in the collection.

A hash table is not good for storing a collection in sorted order.

### **Lookup tables**

Before we look into the details of hash tables, we should consider the more general top of a *lookup table* -- a collection in which information is located by some lookup key.

In the collection class examples we've studied so far, the elements of the collection have been simple strings or numbers. In real-world applications, the elements of a collection are frequently more complicated than this, i.e., they're some form of record structure. For example, some applications may need a collection to store a simple list of string names, such as

```
[ "Baker", "Doe", "Jones", "Smith" ]
```

In many other cases, there may be additional information associated names, such as age, id, and address; e.g.,

```
[ {"Baker, Mary", 51, 549886295, "123 Main St."},  
  {"Doe, John", 28, 861483372, "456 1st Ave."},  
  ...  
]
```

In such a structure, collection elements are typically called *information records*.

### **Information records.**

Information records can be implemented using a class in the normal way, e.g.,

```
class InformationRecord {  
    String name;    // Person name  
    int age;        // Age  
    int id;         // Unique id  
    String address; // Home address  
}
```

When one needs to search for information records in a collection, one of the fields is designated as a *unique* key. This key uniquely identifies each record so it can be located unambiguously. In the above InformationRecord, the id field is a good choice for unique key. A collection structure that holds keyed information records of some form is generally referred to as a *lookup table*.

The unique key is used to lookup an entry in the table. If an entry of a given key is found, the entire entry is retrieved. The implementations we have studied for linked lists and trees can be used as lookup tables, since the type of element has been Object. In the case of a hash table, the structure is specifically suited for use as a lookup table.

### **The basic idea of hashing.**

Suppose we have a collection of personal information records of the form shown above in the InformationRecord class, where the size of the collection is a maximum of 10,000 records. Suppose further that we want rapid access to these records by id. A linked list would be a pretty poor choice for implementing the collection, since search by id would take  $O(N)$  time. If we kept the collection sorted by id, a balanced search tree would give us  $O(\log N)$  access time. If we put the records in an array of 1,000,000,000 elements we could get  $O(1)$  access by id, but we'd waste a lot of space since we only have 10,000 active records.

*Is there some way that we could get  $O(1)$  access as in an array without wasting a lot of space?*

**The answer is hashing, and it's based on the following idea:**

**Allocate an array of the desired table size and provide a function that maps any key into the range 0 to `TableSize-1`. The function that performs the key mapping is called the *hashing function*.**

This idea works well when the hashing function evenly distributes the keys over the range of the table size. To ensure good performance of a hash table, we must consider the following issues:

- choosing a good hashing function that evenly maps keys to table indices;
- choosing an appropriate table size that's big enough but does not waste too much space;
- deciding what to do when the hashing function maps two different keys to the same table location, which condition is called a *collision*.

### **A simple example**

Suppose again we need a table of 10,000 InformationRecords with the id field used as the lookup key. We'll choose a hash table size of 10,000 elements. For the hashing function, we'll use the simple modulus computation of  $\text{id} \bmod 10000$ ; if keys are randomly distributed this will give a good distribution. To resolve collisions, we'll use the simple technique of searching down from the point of collision for the first free table entry.

If we insert the entries listed above for Mary Baker and John Doe. The hashing function computes the indices 6295 and 3372, respectively, for the two keys. The records are placed at these locations in the table array.

Suppose were next to add the record {"Smith, Jane", 39, 861493372, "789 Front St."}

In this case, the hashing function will compute the same location for this record as for Mary Baker, since the id keys for Mary Baker and Jane Smith happen to differ by exactly 10,000. To resolve the collision, we'll put the Jane Smith entry at the next available location in the table, which is 6296.

### **Things that can go wrong.**

In the preceding example, things worked out well, given the nature of the keys and the bounded table size. Suppose, however, some or all of the following conditions were to arise: The number of records grew past 10,000. Due to some coincidence of locale, a large number of ids differed by exactly 10,000. e wanted to use the name field as the search key instead of id. In such cases, we need to reconsider one or all of our choices for hashing function, table size, and/or collision resolution strategy.

### **Choosing a good hash function.**

The choice of hash function depends significantly on what kind of key we have.

In the case of numeric key with random distribution, the simple modulus hashing function works fine. However, if numeric keys have some non-random properties, such as divisibility by the table size, the modulus hashing function does not work well at all. If we use a non-numeric key, such as a name string, we must first convert the string into a number of some form before applying mod. In practical applications, lookup keys are

frequently strings, hence some consideration of good string-valued hash functions is in order.

### **Good hashing of string-valued keys**

***Approach 1: add up the character values of the string and then compute the modulus.***

The advantage of this approach is that it's simple and reasonably fast if the number of characters in a string is not too large. The disadvantage is that it may not distribute key values very well at all. For example, suppose keys are eight characters or fewer (e.g., UNIX login ids) and the table size is 10,000. Since ASCII string characters have a maximum value of 127, the summing formula only produces values between 0 and  $127 \times 8$ , which equals 1,016. This only distributes keys to a bit more than 10% of the table.

***Approach 2: use a formula that increases the size of the hash key using some multiplier.***

This approach is also simple and fast, but it may also not distribute keys well. E.g., one formula could be to sum the first three characters of a key string as follows:

```
char[0] + (27 * char[1]) + (729 * char[2])
```

**and then compute the modulus.**

The rationale for the number 27 is that it's the number of letters in the alphabet, plus one for a space character; 729 is  $27^2$ .

If string name characters are equally likely to occur, this distributes keys in the range 0 to  $26^3 = 17,576$ .

However, empirical analysis of typical names shows that for the first three characters, there are only 2,851 combinations, which is not good coverage for a 10,000-element table.

***Approach 3: sum all key characters with a formula that increases the size and mixes up the letters nicely.***

An empirically derived formula to do this is the following:

$$(37 * \text{char}[0]) + (37^2 * \text{char}[1]) + \dots + (37^{(l-1)} * \text{char}[l])$$

where 37 is the empirically-derived constant and  $l$  = the string length of the key.

This formula, plus similar ones with variants on the constant multiplier, has been shown to do a good job of mixing up the string characters and providing good coverage even for large table.

### **Review Questions**

1. What is the use of indexing?
2. What are primary index and secondary index?
3. Explain hashing?
4. What are look up tables?

### **References**

<http://www.microsoft-accesssolutions.co.uk>

<http://www.cs.sfu.ca/CC/354>