

Theoretically, looking up an item by searching a list of N items and making key comparisons will, on average, require $O(\log N)$ work.

As with sorting, we can “cheat” this result by organizing the data so that the search may be carried out with no (or very few) key comparisons.

In essence, we will store the data elements in a structure, known as a table, and provide that table with an efficient indexing scheme. The table index will allow us to rapidly look up a key value and immediately find the location of the corresponding record in the table. The table will support random access, or at least approximate that, so given a location we can then find the record in constant or nearly constant time.

All general-purpose programming languages provide native support for arrays, mimicking mathematical matrices or vectors.

Arrays provide constant time random access if an element's location is specified using the array index.

The use of arrays is familiar, and inherently boring at this stage. What we seek is a generalization of the idea to provide similar capabilities for scenarios in which an in-memory array of cells is inappropriate or inadequate.

It helps, however, to understand how the array index is transformed into an address in memory...

A 2-D Array Template

```
template <class Item> class Array2DT {
private:
    Item *Data;
    int  nRows;
    int  nCols;

    int  Map(int R, int C) const;
    bool ValidRC(int R, int C) const;

public:
    Array2DT();
    Array2DT(int R, int C);
    Array2DT(const Array2DT& Source);
    Array2DT& operator=(const Array2DT& Source);

    bool Set(int R, int C, Item Value);
    Item Get(int R, int C) const;

    ~Array2DT();
};
```

Physically, a 1-D array is used, but from the client's perspective the table is addressed as a 2-D structure.

The client specifies a 2-D location as row and column coordinates. These are validated against the logical table dimensions, and then translated into the appropriate index for the physical 1-D array.

Accessor and mutator members allow the client to retrieve and/or change the contents of any table location.

The Array2DT template provides a safe table that appears to the user as a nearly-normal 2-D array.

```
template <class Item>
bool Array2DT<Item>::Set(int R, int C, Item Value) {

    if ( ValidRC(R, C) ) {
        int Idx = Map(R, C);
        Data[Idx] = Value;
        return true;
    }
    return false;
}
```

Coordinates supplied by the client are validated against the specified size of the table...

... and translated into a linear index appropriate to the physical structure.

```
template <class Item>
int Array2DT<Item>::Map(int R, int C) const {

    return (R * nCols + C);
}
```

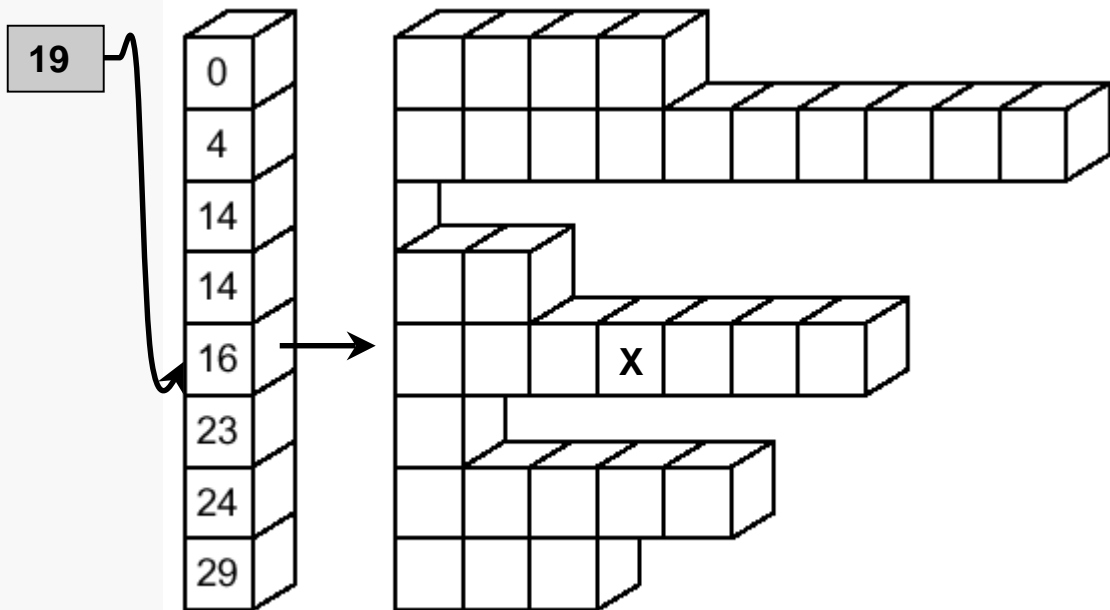
The Array2DT coordinate transformation is simple and reasonably quick, and very similar to the computation needed to find the physical memory address of the specified 1-D array cell.

In some cases, a table is naturally thought of as being composed of rows, but the rows contain wildly varying numbers of elements:

Given a (linear) index value, search the access array to find the appropriate table row...

... the access array cell will contain a "pointer" to the beginning of the corresponding table row.

The row can then be quickly searched for the desired entry.



Updating the access array entries when insertions or deletions are performed in the table is simple but costs $O(R)$ if the table has R rows.

In many cases, a table holding a collection of records must support efficient lookup by more than one key value. For example, we may have a set of customer records consisting of a name field, an address field and a phone number field. If we use a simple array, sorted by name, we have good support for searching on the name field, but not for address or phone number searches:

0	Baker, John S	17 King Street	2884285
1	Byers, Carolyn	118 Maple Street	4394231
2	Hill, Thomas M	39 King Street	2495723
3	Hill, Thomas M	High Towers #317	2829478
4	King, Barbara	High Towers #802	2863386
5	Moody, C L	High Towers #210	2822214
6	Roberts, L B	53 Ash Street	4372296

Inverted Tables

A solution is to provide an access array for each key value for which we need lookup capability:

2495723	2
2822214	5
2829478	3
2863386	4
2884285	0
4372296	6
4394231	1

**Phone
Number
Index**

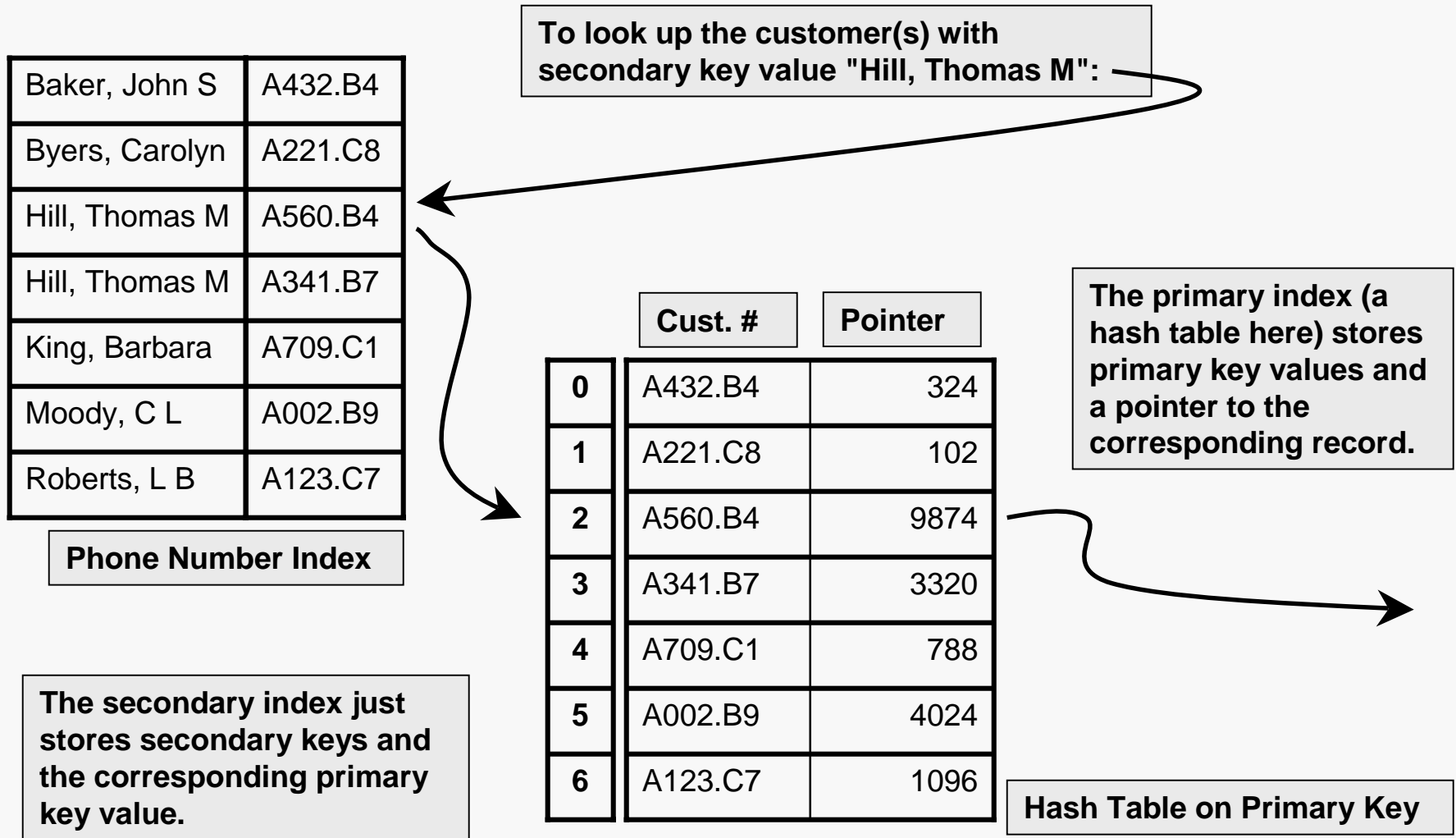
The “master” table doesn’t even need to be kept in sorted order. That’s a large improvement if there are many records and they are stored on disk.

Table of Records

0	Baker, John S	17 King Street	2884285
1	Byers, Carolyn	118 Maple Street	4394231
2	Hill, Thomas M	39 King Street	2495723
3	Hill, Thomas M	High Towers #317	2829478
4	King, Barbara	High Towers #802	2863386
5	Moody, C L	High Towers #210	2822214
6	Roberts, L B	53 Ash Street	4372296

Inverted Table as a Secondary Index

A solution is to provide an access array for each key value for which we need lookup capability:



Inverted Table as a Secondary Index

A solution is to provide an access array for each key value for which we need lookup capability:

Baker, John S	A432.B4
Byers, Carolyn	A221.C8
Hill, Thomas M	A560.B4
Hill, Thomas M	A341.B7
King, Barbara	A709.C1
Moody, C L	A002.B9
Roberts, L B	A123.C7

Phone Number Index

To look up the customer(s) with secondary key value "Hill, Thomas M":

Of course, the secondary index may find more than one match, in which case it must return a list of matching records.

A560.B4
A341.B7

QTP: why not store pointers in the secondary index to eliminate a lookup?

When the primary index is searched, and actual records are retrieved, the results may be filtered to extract the desired matches.

A table can be thought of as an abstract data type. Given a set of index values I , and a base type T , a table is a function M from I to T that supports the operations:

- access evaluate the function at any index value (retrieval)
- assignment modify the value of $M(I)$ for any index value I
- creation define a new function M
- clearing remove all elements from I , so M 's domain is empty
- insertion add a new value, x , to I and define $M(x)$
- deletion remove a value, x , from I (restricting M 's domain)