B-Trees

11

This chapter covers one of the more important data structures of the last thirty years. B-Trees are primarily used by relational databases to efficiently implement an operation called *join*. B-Trees have other properties that are also useful for databases including ordering of rows within a table, fast delete capability, and sequential access.

11.1 Chapter Goals

This chapter introduces some terminology from relational databases to motivate the need for B-Trees. The chapter goes on to introduce the B-Tree data structure and its implementation. By the end of this chapter you should have an understanding of B-Trees, their advantages over other data structures, and you should be able to demonstrate your understanding by implementing a B-Tree that can be used to efficiently process *joins* in relational databases.

11.2 Relational Databases

While this is not a database text we will cover a bit of database terminology to demonstrate the need for a B-Tree and its use in a relational database. A relational database consists of entities and relationships between these entities. A database schema is a collection of entities and their relationships. A schema is specified by a *Entity Relationship* diagram, often abbreviated *ER*-diagram, or a Logical Data Structure [2]. Figure 11.1 provides an ER-diagram for a database called the *Dairy Database*. It is used to formulate rations for dairy cattle to maximize milk production.

Each box in Fig. 11.1 represents an entity in the database. Of particular interest in this text are the Feed, FeedAttribute, and FeedAttribType entities. A feed, like corn silage or alfalfa, is composed of many different nutrients. Nutrients are things like calcium, iron, phosphorus, protein, sugar, and so on. In the *Dairy Database*

© Springer International Publishing Switzerland 2015

K.D. Lee and S. Hubbard, *Data Structures and Algorithms with Python*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-13072-9_11

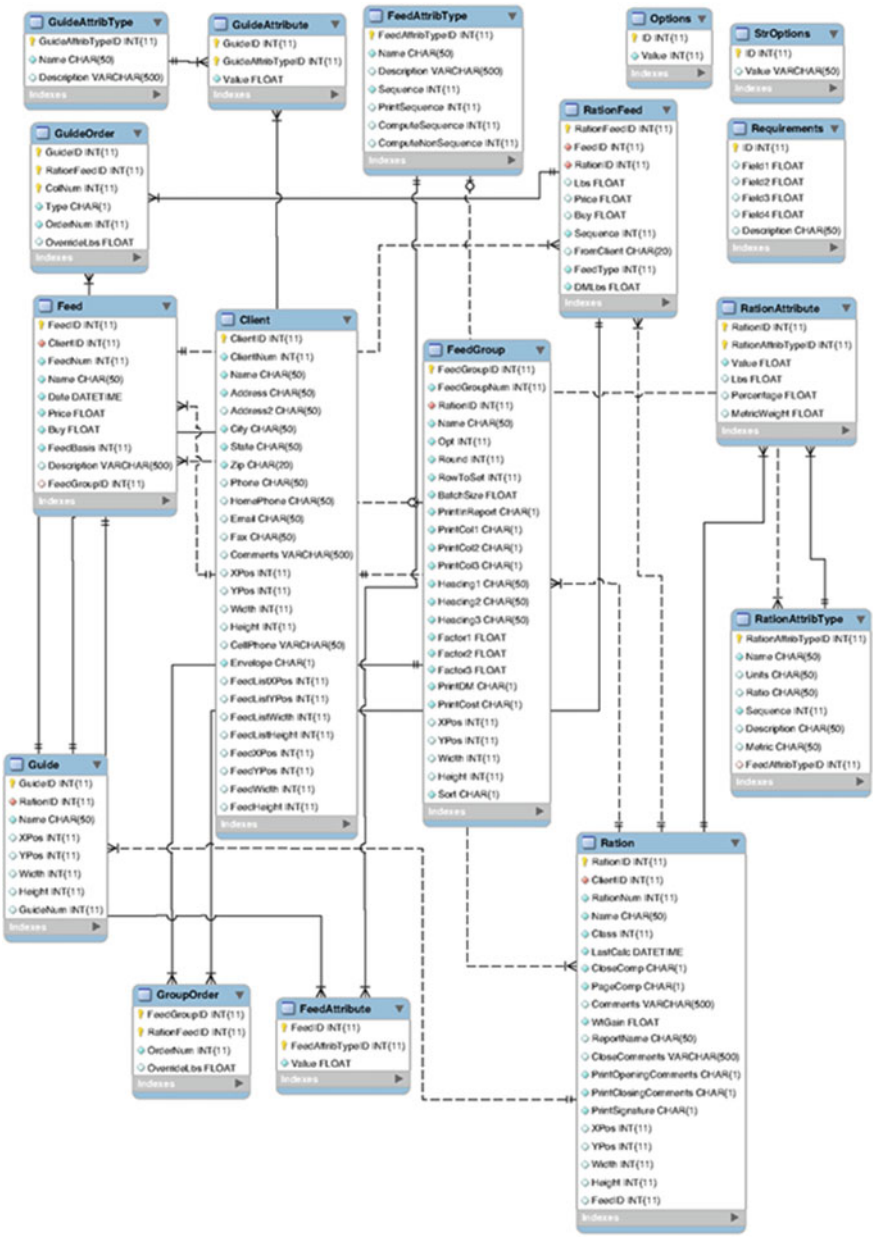

Fig. 11.1 Dairy Database Entity Relationship Diagram

these nutrients are called FeedAttribTypes. There is a many-to-many relationship between Feeds and FeedAttribTypes. A feed has many feed attributes, or nutrients. Each nutrient or feed attribute type appears in more than one feed. This relationship

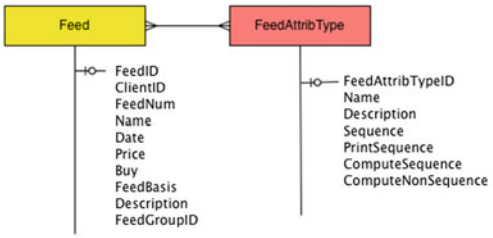

Fig. 11.2 A Many to Many Relationship

is depicted in Fig. 11.2. The forks on the two ends of the line represent the many-to-many relationship between feeds and feed attribute types.

Many-to-Many relationships cannot be represented in a relational database without going through a process called reification. *Reification* introduces new entities that remove many-to-many relationships. When a many-to-many relationship appears within a logical data structure it indicates there may be missing attributes. In this case, the quantity of each nutrient within a feed was missing. The new *FeedAttribute* entity eliminates the many-to-many relationship by introducing two one-to-many relationships. One-to-many relationships can be represented in relational databases.

Every entity in a relational database must have a unique identifier. In Fig. 11.3 the Feed entities are uniquely identified by their *FeedID* attribute. The other attributes are important, but do not have to be unique. Each *FeedID* must be unique and it cannot be null or empty for any feed. Likewise, a *FeedAttribTypeID* field uniquely identifies each feed nutrient. There is a unique FeedAttribTypeID for calcium, iron, and so on. The FeedAttribute entity has a unique id made up of two fields. Together, the *FeedID* and the *FeedAttribTypeID* identify a unique instance of a nutrient for a particular feed. The *Value* was the missing attribute in Fig. 11.2 that was introduced by reifying the many-to-many relationship as depicted in Fig. 11.3. The Logical Data Structure in Fig. 11.3 describes the schema for feeds and nutrients in the *Dairy Database*.

A relational database is composed of tables and the shema provides the definition of these tables. The *Feed* table consists of rows and columns. Each row in the *Feed* table describes one feed. The columns of the Feed table are each of the attributes of a feed provided in Fig. 11.3. The example in Sect. 11.2.1 provides a subset of this table

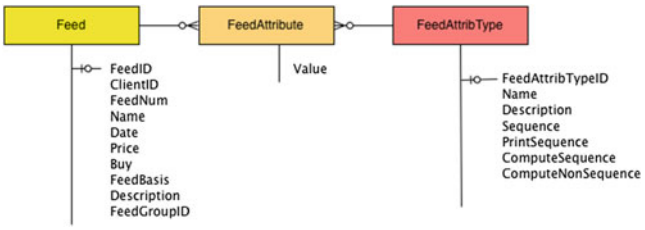

Fig. 11.3 Logical Data Structure

with a subset of the columns of this table. The ellipses (i.e. the …) indicate omitted rows within the database. The full table is available as *Feed.tbl* on the text website.

11.2.1 The Feed Table

```
...
1316        512         10'Corn Silag'                          2/28/2002 12:00:00 AM
1317        512         11'Almond Hul'                          7/15/1989 12:00:00 AM
1318        512         12'MolassWet'                           5/19/1989 12:00:00 AM
1319        512         13'LIQ CIT PL'                          3/2/2002 12:00:00 AM
1320        512         14'Whey'                                9/4/1997 12:00:00 AM
1321        512         16'SF CORN'                             9/29/1998 12:00:00 AM
1322        512         18'Dry Min'                             10/17/2002 12:00:00 AM
1323        512         19'Min Plts'                            11/17/2002 12:00:00 AM
1324        512         20'Mineral'                             2/28/2002 12:00:00 AM
1372        525          1'Hay lact'                            9/15/2003 12:00:00 AM
1373        525          2'DRY HAY'                             11/30/1999 12:00:00 AM
1374        525          3'Oat hay'                             11/10/1998 12:00:00 AM
1375        525          4'Hlg'                                 4/12/2004 12:00:00 AM
1376        525          5'CUPHay'                              9/17/2003 12:00:00 AM
1377        525          6'Hay #1'                              3/27/2001 12:00:00 AM
1378        525          8'BMR CSilage'                         4/1/2004 12:00:00 AM
1379        525          9'Wheat Sil'                           9/15/2003 12:00:00 AM
1380        525         10'Corn Silag'                          10/30/2003 12:00:00 AM
1381        525         11'Almond Hul'                          7/10/2000 12:00:00 AM
1382        525         14'ClosePlt'                            1/13/2003 12:00:00 AM
1383        525         16'Corn1%fat'                           9/3/2002 12:00:00 AM
1384        525         18'Dry Min'                             7/12/2000 12:00:00 AM
1385        525         19'Comm Mix'                            12/13/2003 12:00:00 AM
1386        525         20'On Farm'                             10/30/2003 12:00:00 AM
1438        529          1'Big Sq155'                           6/21/1999 12:00:00 AM
1439        529          2'Hay#1-200'                           2/1/2000 12:00:00 AM
1440        529          3'Hay#2-145'                           2/1/2000 12:00:00 AM
...
```

Normally a relational database would store a table like the *Feed* table in a binary format that would be unreadable except by a computer. The *Feed.tbl* file is written in ASCII format to be human readable with a simple text editor, but the principles are the same. Each row within the table represents one record of the table which is one instance of a feed in this case. The records are each the same size to make reading the table easy. Within any record we can find the name of the feed by going to the correct column for feed name, which is the fourth field within each record and starts 30 bytes or characters into each record. Ten bytes or characters are allocated to each integer field (the first column was edited to better fit on the page). There are 107 records or feeds within the sample *Feed.tbl* table provided on the text website.

11.2.2 The FeedAttribType Table

```
...
10'P'               'Phosphorus as % of DM'                     15
11'Ca'              'Calcium as % of DM'                        16
12'RFV'             'Relative Feed Value (calculated)'          17
13'S'               'Sulfur as % of DM'                         18
14'K'               'Potassium as % of DM'                      19
```

```
15'Mg'              'Magnesium as % of DM'                    20
16'Fat'             'Fat as % of DM'                          14
...
```

The table in Sect. 11.2.2 contains a subset of the records in the *FeedAttribType* table, available as *FeedAttribType.tbl* on the text website. The full table has 57 different rows each containing 7 fields. As with the *Feed* table, the *FeedAttribType* table is organized into rows and columns.

A subset of the *FeedAttribute* table is provided in Sect. 11.2.3. Each feed attribute is comprised of the corresponding *FeedID*, the *FeedAttribTypeID*, and the amount of that nutrient for the given feed which is called the *Value* column within the table.

11.2.3 The FeedAttribute Table

```
...
1316        10            0.250000
1316        11            0.210000
1316        12          128.000000
1316        13            0.150000
1316        14            1.200000
1316        15            0.200000
1316        16            3.000000
...
1317        10            0.110000
1317        11            0.220000
1317        12          129.000000
1317        13            0.110000
1317        14            0.530000
1317        15            0.130000
1317        16            2.000000
...
```

Storing the feed data this way is flexible. New nutrients can easily be added. Feeds can be added as well. Feed attributes can be stored if available or omitted. Occasionally, programs that use relational databases need access to data from more than one table but need to correlate the data between the tables. For instance, it may be convenient to temporarily construct a table that contains the feed number, feed name, nutrient name, and value of that nutrient for the corresponding feed into a table like that in Sect. 11.2.4. We may want to compute the average phosphorous content within all feeds. In fact, we may wish to calculate the average content for each nutrient type within the database. In that case a table like the one in Sect. 11.2.4 would be very useful.

11.2.4 A Temporary Table

```
...
10 Corn Silag P 0.25
10 Corn Silag Ca 0.21
10 Corn Silag RFV 128.0
10 Corn Silag S 0.15
```

```
10 Corn Silag K 1.2
10 Corn Silag Mg 0.2
10 Corn Silag Fat 3.0
...
11 Almond Hul P 0.11
11 Almond Hul Ca 0.22
11 Almond Hul RFV 129.0
11 Almond Hul S 0.11
11 Almond Hul K 0.53
11 Almond Hul Mg 0.13
11 Almond Hul Fat 2.0
...
```

Relational databases are often called *SQL* databases. *SQL* stands for *System Query Language*. SQL is a language for querying relational databases. SQL can be used to build temporary tables like the one in Sect. 11.2.4. The *SQL* statement to build this table would be written as

```
SELECT Feed.FeedNum, Feed.Name, FeedAttribType.Name, FeedAttribute.Value WHERE
        Feed.FeedID = FeedAttribute.FeedID AND
        FeedAttribute.FeedAttribTypeID = FeedAttribType.FeedAttribTypeID
```

This SQL statement is known as a *join* of three tables because three tables will be joined together to form the result. It is up to the relational database to translate this query into commands that read the three tables and efficiently construct a new temporary table as the result of the join.

If we were to implement our own relational database, the join operation for these three tables might be programmed similarly to the code appearing in Sect. 11.2.5. Don't be misled. Relational databases don't program specific joins like this one, but the joining of the three tables might be functionally equivalent to this code. The entire program is available as *joinquery.py* on the text's website. The *readField* function here in the text is abbreviated for space, but reads any type of field from a table file. The join algorithm picks one of the tables and read it from beginning to end. In this case, the FeedAttribute table is read from beginning to end. For each feed attribute, the matching feed id from the feed table must be located. In the code in Sect. 11.2.5 this involves reading, on average, half the feed table to supply the feed number and feed name for each line of the query. Likewise, to supply the feed attribute name, on average half the FeedAttribType table is read to supply the feed attribute name for each line of the query output.

The complexity of this operation is O(n*m) where n is the number of records in *FeedAttribute.tbl* and m is the maximum of the number of records in *FeedAttribType.tbl* and *Feed.tbl*. This is $O(n^2)$ performance if n is roughly equivalent to m. Whether the two are roughly equivalent or not, the performance of this query, even on our small sample table, is not great. It takes about 4.993 s to run the query as written on a 2.66 GHz Intel Core i7 processor with 8 GB of RAM and a solid state hard drive.

11.2.5 Programming the Joining of Tables

```
import datetime
def readField(record,colTypes,fieldNum):
```

```
    # fieldNum is zero based
    # record is a string containing the record
    # colTypes is the types for each of the columns in the record
    offset = 0
    for i in range(fieldNum):
        colType = colTypes[i]

        if colType == "int":
            offset+=10
        elif colType[:4] == "char":
            size = int(colType[4:])
            offset += size
        elif colType == "float":
            offset+=20
        ...
    return val
def main():
    # SELECT Feed.FeedNum, Feed.Name, FeedAttribType.Name, FeedAttribute.Value WHERE
    # Feed.FeedID = FeedAttribute.FeedID AND
    # FeedAttribute.FeedAttribTypeID = FeedAttribType.FeedAttribTypeID
    attribTypeCols = ["int","char20","char60","int","int","int","int"]
    feedCols = ["int","int","int","char50","datetime","float","float","int","char50","int"]
    feedAttributeCols = ["int","int","float"]
    before = datetime.datetime.now()
    feedAttributeTable = open("FeedAttribute.tbl","r")
    for record in feedAttributeTable:
        feedID = readField(record,feedAttributeCols,0)
        feedAttribTypeID = readField(record,feedAttributeCols,1)
        value = readField(record,feedAttributeCols,2)
        feedTable = open("Feed.tbl","r")
        feedFeedID = -1
        while feedFeedID != feedID:
            feedRecord = feedTable.readline()
            feedFeedID = readField(feedRecord,feedCols,0)
        feedNum = readField(feedRecord,feedCols,2)
        feedName = readField(feedRecord,feedCols,3)
        feedAttribTypeTable = open("FeedAttribType.tbl", "r")
        feedAttribTypeIDID = -1
        while feedAttribTypeIDID != feedAttribTypeID:
            feedAttribTypeRecord = feedAttribTypeTable.readline()
            feedAttribTypeIDID = readField(feedAttribTypeRecord,attribTypeCols,0)
        feedAttribTypeName = readField(feedAttribTypeRecord,attribTypeCols,1)
        print(feedNum,feedName,feedAttribTypeName,value)
    after = datetime.datetime.now()
    deltaT = after - before
    milliseconds = deltaT.total_seconds() * 1000
    print("Time for the query without indexing was",milliseconds,"milliseconds.")

if __name__ == "__main__":
    main()
```

The code in Sect. 11.2.5 suffers because the two tables, *Feed.tbl* and *FeedAttribType.tbl* are read sequentially each time through the outer loop to find the matching feed and feed attribute type, respectively. We can improve the efficiency of this query if we recognize that disk drives are *random access* devices. That means that we can position the read head of a disk drive anywhere within a file. We don't have to start at the beginning of a table to begin looking for a matching feed or feed attribute type. We can jump around within the table to find the matching record.

11.2.6 The readRecord Function

```
def readRecord(file,recNum,recSize):
    file.seek(recNum*recSize)
    record = file.read(recSize)
    return record
```

Python includes a *seek* method on files to position the read head of a disk to a byte offset within a file. The *read* method on files reads a given number of bytes and returns them as a string. To test this *readRecord* function, and the functionality of the *seek* method, a program was written to randomly access the records in the *FeedAttribute.tbl* file. The results of that experiment are shown in Fig. 11.4. The data shows that accessing any record within the file took about the same amount of time regardless of its position within the file. As with any experiment, there were a few anomalies. But, the vast majority of records were accessed in the same amount of time or nearly the same amount of time.

Let's say we were to organize the *Feed.tbl* and the *FeedAttribType.tbl* files so that the records were sorted in increasing order by their keys. The *Feed.tbl* file would be sorted by *FeedID* and the *FeedAttribType.tbl* would be sorted by *FeedAttribTypeID*. Then we could use binary search on these two files to find the matching records for each feed attribute in the code of Sect. 11.2.5. Since the tables are randomly accessible, the query time could be reduced from O(n*m) to O(n log m). However,

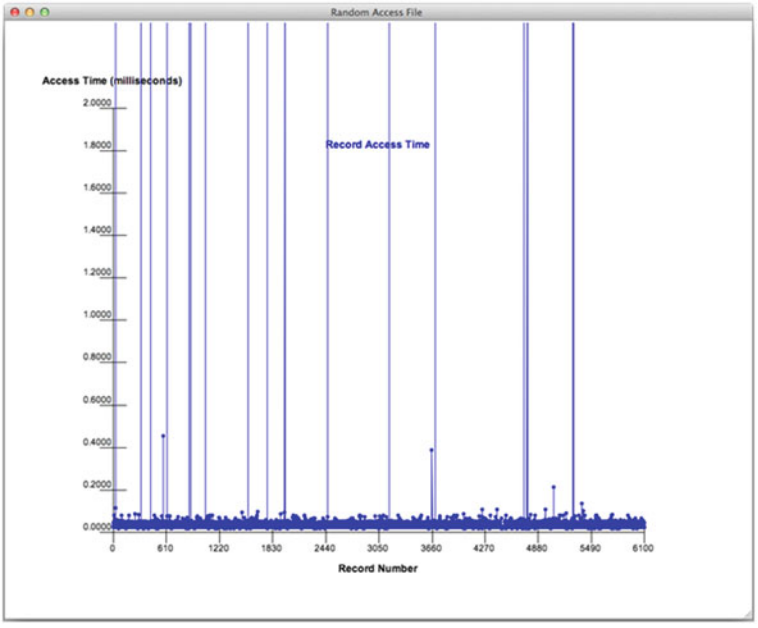

Fig. 11.4 Access Time for Randomly Read Records in a File

we can't assume that a database table will always, or ever, be sorted according to one field. Databases can have new records added and old records deleted at any time.

This is where the need for a B-Tree comes from. A B-Tree is a tree structure that is built over the top, so to speak, of a database table to provide O(log n) lookup time to any record within the database table. While the records themselves may be in any order, the B-Tree provides the O(log n) search complexity into the table.

A B-Tree is built by inserting records or items into the tree. Once built, the index provides the efficient lookup of any record based on the key value stored in the B-Tree. Consider the code in Sect. 11.2.7. Lines 18–23 build the *Feed.tbl* index and lines 39–44 build the *FeedAttribType.tbl* index. Once built, the indices are used when programming the query. The loop beginning on line 55 no longer contains two while loops to lookup the corresponding records in the two tables. Instead, the B-Trees are consulted to find the corresponding records in the two tables. When programmed this way, the query in Sect. 11.2.7 runs in approximately 1.628 s, three times faster than the original, non-indexed query. The sample query here uses relatively small tables. Imagine the speed up possible when either of the *Feed.tbl* or *FeedAttribType.tbl* tables contained millions of records. In that case, the original query would not have completed in an acceptable amount of time while the indexed query given here would have completed in roughly the same amount of time or perhaps a second longer at worst.

11.2.7 Efficient Join

```
1  def main():
2      # Select Feed.FeedNum, Feed.Name, FeedAttribType.Name, FeedAttribute.Value where
3      # Feed.FeedID = FeedAttribute.FeedID and FeedAttribute.FeedAtribTypeID = FeedAttribType.ID
4      attribTypeCols = ["int","char20","char60","int","int","int","int"]
5      feedCols = ["int","int","int","char50","datetime","float","float","int","char50","int"]
6      feedAttributeCols = ["int","int","float"]
7
8      feedAttributeTable = open("FeedAttribute.tbl","r")
9
10     if os.path.isfile("Feed.idx"):
11         indexFile = open("Feed.idx","r")
12         feedTableRecLength = int(indexFile.readline())
13         feedIndex = eval(indexFile.readline())
14     else:
15         feedIndex = BTree(3)
16         feedTable = open("Feed.tbl","r")
17         offset = 0
18         for record in feedTable:
19             feedID = readField(record,feedCols,0)
20             anItem = Item(feedID,offset)
21             feedIndex.insert(anItem)
22             offset+=1
23             feedTableRecLength = len(record)
24
25         print("Feed Table Index Created")
26         indexFile = open("Feed.idx","w")
27         indexFile.write(str(feedTableRecLength)+"\n")
28         indexFile.write(repr(feedIndex)+"\n")
29         indexFile.close()
30
31     if os.path.isfile("FeedAttribType.idx"):
32         indexFile = open("FeedAttribType.idx","r")
```

```
        attribTypeTableRecLength = int(indexFile.readline())
        attribTypeIndex = eval(indexFile.readline())
    else:
        attribTypeIndex = BTree(3)
        attribTable = open("FeedAttribType.tbl","r")
        offset = 0
        for record in attribTable:
            feedAttribTypeID = readField(record,attribTypeCols,0)
            anItem = Item(feedAttribTypeID,offset)
            attribTypeIndex.insert(anItem)
            offset+=1
            attribTypeTableRecLength = len(record)

        print("Attrib Type Table Index Created")
        indexFile = open("FeedAttribType.idx","w")
        indexFile.write(str(attribTypeTableRecLength)+"\n")
        indexFile.write(repr(attribTypeIndex)+"\n")
        indexFile.close()

    feedTable = open("Feed.tbl","rb")
    feedAttribTypeTable = open("FeedAttribType.tbl", "rb")
    before = datetime.datetime.now()
    for record in feedAttributeTable:

        feedID = readField(record,feedAttributeCols,0)
        feedAttribTypeID = readField(record,feedAttributeCols,1)
        value = readField(record,feedAttributeCols,2)

        lookupItem = Item(feedID,None)
        item = feedIndex.retrieve(lookupItem)
        offset = item.getValue()
        feedRecord = readRecord(feedTable,offset,feedTableRecLength)
        feedNum = readField(feedRecord,feedCols,2)
        feedName = readField(feedRecord,feedCols,3)

        lookupItem = Item(feedAttribTypeID,None)
        item = attribTypeIndex.retrieve(lookupItem)
        offset = item.getValue()
        feedAttribTypeRecord = readRecord(feedAttribTypeTable,offset,attribTypeTableRecLength)
        feedAttribTypeName = readField(feedAttribTypeRecord,attribTypeCols,1)

        print(feedNum,feedName,feedAttribTypeName,value)
    after = datetime.datetime.now()
    deltaT = after - before
    milliseconds = deltaT.total_seconds() * 1000
    print("Time for the query with indexing was",milliseconds,"milliseconds.")
```

Clearly we need the functionality of a B-Tree to make queries possible and efficient in relational database joins. The next section goes on to explain the organization of a B-Tree, the advantages of B-Trees, and how they are implemented.

11.3 B-Tree Organization

A B-Tree is a balanced tree. Each node in a B-Tree consists of alternating pointers and items as shown in Fig. 11.5. B-Trees consist of nodes. Each node in a B-Tree contains pointers to other nodes and items in an alternating sequence. The items in a node are arranged sequentially in order of their keys. In Fig. 11.5 the key is the first value in each tuple. A pointer to the left of an item points to another B-Tree node

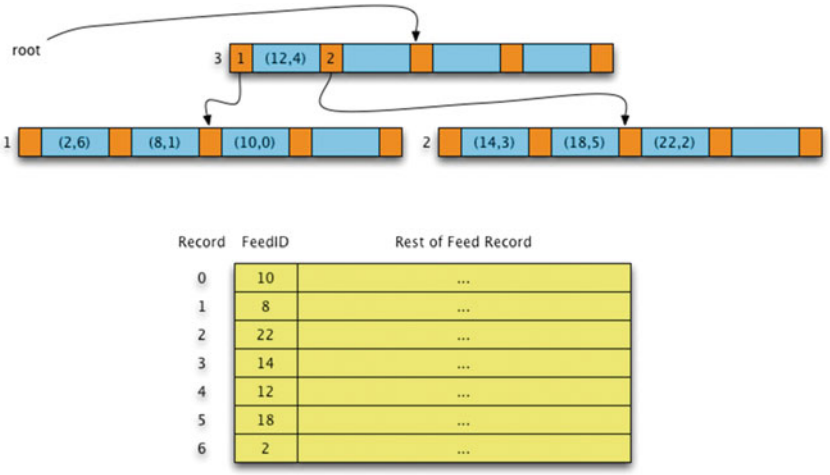

Fig. 11.5 A Sample B-Tree

that contains items that are all less than the item to the right of the pointer. A pointer to the right of an item points to a node where all the items are greater than the item. In Fig. 11.5 the items in node 1 are all less than 12 while the items in node 2 are all greater than 12.

B-Trees are always balanced, meaning that all the leaf nodes appear on the same level of the tree. A B-Tree may contain as many items and pointers as desired in each node. There will always be one more pointer than items in a node. B-Trees don't have to fill each node. The *degree* of a B-Tree is the minimum number of items that a B-Tree node may contain, except for the root node. The *capacity* of a node is always twice its *degree*. In Fig. 11.5 the degree is 2 and the capacity is 4.

The requirements of a B-Tree are as follows:

1. Every node except the root node must contain between *degree* and *2*degree* items.
2. Every node contains one more pointer than the number of items in the node.
3. All leaf nodes are at the same level within a B-Tree.
4. The items within a B-Tree node are ordered in ascending (or descending) order. All nodes have their items in the same order, either ascending or descending.
5. The items in the subtree to the left of an item are all less than that item.
6. The items in the subtree to the right of an item are all greater than that item.

To maintain these properties, inserting and deleting items from the tree must be done with some care. Inserting an item can cause splitting of a node. Deleting from a tree sometimes requires rebalancing of the tree. Looking up an item in a B-Tree is performed much the same way lookup is performed in a binary search tree. The node is examined to find the item. If it is not found, then the pointer is followed that lies between the items that are less than and greater than the item to be found. If this leads to a leaf node and the item is not found in the leaf node, the item is reported as not in the tree.

11.4 The Advantages of B-Trees

A B-Tree may contain entire records instead of just key/value pairs as appear in Fig. 11.5 where the key/value pairs are the FeedID and record number of each record in the Feed table. For instance, the entire record for FeedID 10 might be stored directly in the B-Tree where (10,0) currently appears. In the examples in this text the B-Tree and the database table are stored separately. This has the advantage that more than one B-Tree index could be built over the Feed table. The B-Tree in Fig. 11.5 is built over the FeedID field. Some other unique field might be used to build another B-Tree over the table if desired. By storing the B-Tree and the table separately, multiple indices are possible.

As mentioned earlier in the chapter, B-Trees provide $O(\log_d n)$ lookup time where d is the degree of the B-Tree and n is the number of items in the tree. Hash tables provide faster lookup time than a B-Tree. So why not use a hash table instead?

Unlike a hash table, a B-Tree provides ordered sequential access to the index. You can iterate over the items in a B-Tree much like binary trees provide iteration. Iteration over a B-Tree provides the items or keys in ascending (or descending) order. A hash table does not provide an ordering of its keys.

B-Trees provide O(log n) insert, delete, and lookup time as well. While not as efficient as hash tables in this regard, B-Trees nodes are often quite large providing a very flat tree. In this case, the time for these three operations often comes close to that of a hash table.

B-Trees are often constructed with literally millions of items. When a B-Tree reaches this size, holding all the nodes in memory at one time may consume a lot of RAM. This is a great advantage of B-Trees over hash tables. A B-Tree may be stored in a file itself. Since files are randomly accessible on a disk, a B-Tree's node may be thought of as a record in a file. Consider the B-Tree in Fig. 11.5. The nodes 1, 2, and 3 could be thought of as three records within a file. The record number are the pointer values, so to search the B-Tree it is only necessary to start with the root node in memory. Then, to search when a pointer is followed during search, the record corresponding to the new node is read into memory during the search. A search can proceed in this way, reading one record at a time from disk. Typically a pool of records would be held in memory for a B-Tree and records would be replaced in memory using some sort of node replacement scheme. In this way a fixed amount of RAM can be allocated to hold a B-Tree that would typically be much smaller than the total size of the tree.

In addition, since a B-Tree can be stored in a file, it is not necessary to reconstruct the B-Tree each time it is needed. The code in Sect. 11.2.7 stores the B-Trees in two files named *Feed.idx* and *FeedAttribType.idx* and reads the index from the file the next time the program is run.

Deleting a record from a table with a million records or more in it could be an expensive operation if the table has to be completely rewritten. If sequential access to the underlying table is handled through the B-Tree or if the entire file is stored in the nodes of the B-Tree, deletion of a row or record in the table gets much simpler. For instance, in Fig. 11.6 the feed with FeedID of 10 remains in the Feed.tbl file, but

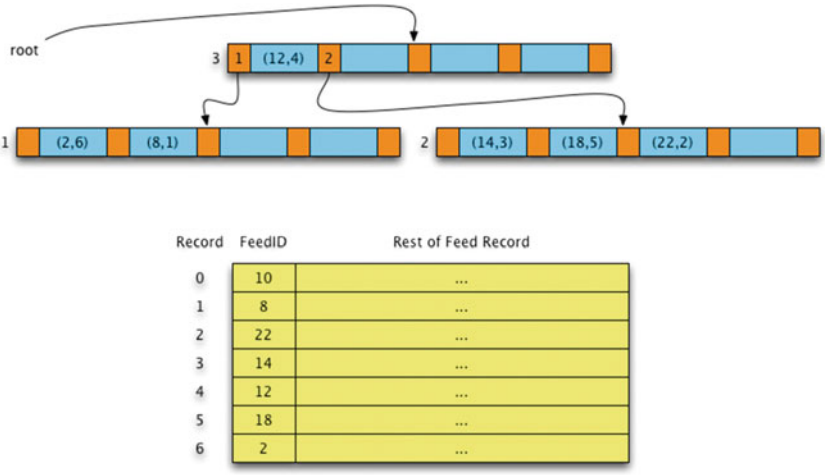

Fig. 11.6 A Sample B-Tree with Key 10 Deleted

has been deleted from the B-Tree. If sequential access is always handled through the B-Tree, it would appear that the feed with FeedID 10 has been deleted from the table. Deleting an item from the table in this way is a O(log n) operation while deleting by rewriting the entire file would take O(n) time. When *n* is millions of records, the difference between O(log n) and O(n) is significant.

The same goes for inserting a new row or record within the Feed table. Adding one new record to the end of a file can be done quickly, without rewriting the entire file. When a B-Tree is used the newly inserted item automatically maintains its sorted position within the file.

To summarize, B-Trees have several characteristics that make them attractive to use in relational databases and for providing access to large quantities or ordered data. These properties include:

- Ordered sequential access over the key value on O(n) time.
- O(log n) insert time, while maintaining the ordering of the items.
- O(log n) delete time of items within the B-Tree.
- If sequential access is handled through the B-Tree then O(log n) delete time is provided for the underlying table as well.
- B-Trees can be stored in a file and B-Tree nodes can be read on an as needed basis allowing B-Trees to be larger than available memory.
- A B-Tree index stored in a file does not have to be rebuilt each time it is needed in a program.

It is this final point that make B-Trees and their derivatives so valuable to relational database implementations. Relational databases need B-Trees and their derivative implementations to efficiently process *join* operations while also providing many of the advantages listed above.

11.5 B-Tree Implementation

Looking up a value in a B-Tree is relatively simple and is left as an exercise for the reader. Inserting and deleting values are where all the action is. Alan Tharp [7] provides a great discussion of both inserting and deleting values in a B-Tree. In this text we provide new examples and suggest both iterative and recursive implementations of both operations.

11.6 B-Tree Insert

Inserting an item in a B-Tree involves finding the leaf node which should contain the item. It may also involve splitting if no room is left in the leaf node. When a leaf node reaches its capacity, which is two times its degree and a new item is being inserted, the 2*degree+1 items are sorted and the median value (i.e. the middle value) is promoted up the tree to the parent node. In this way, splitting may cascade up the tree.

To see the splitting process in action, consider building the tree given in Fig. 11.5 with the keys given in this order [10, 8, 22, 14, 12, 18, 2, 50, 15]. The first item to be inserted is the 10. When this occurs, the B-Tree is empty, consisting of one empty node. The (10,4) item is added into that node as shown in Fig. 11.7.

The items with keys 8, 14, and 22 are inserted in a similar fashion as shown in Fig. 11.8. The node is now full. The next item to be inserted will cause a split.

The next item inserted is a 12 causing the node to split into two nodes. The left subtree node is the original node. The right subtree contains the new node. The middle value, 12 in this case, is promoted up to the parent. In this case, there is no parent since we split the root node. In this special case a new root node is created to hold the promoted value. After taking these steps, the tree appears as shown in Fig. 11.9.

The three values 18, 2, and 50 are inserted resulting in the tree as shown in Fig. 11.10.

When 15 is inserted B-Tree node number 2 is going to split and promote the middle value, 18 in this case, up to the parent. This time there is room in the parent so the new item is added resulting in the tree shown in Fig. 11.11.

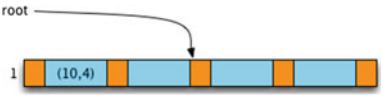

Fig. 11.7 Inserting 10 into an empty B-Tree

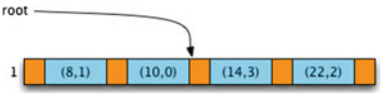

Fig. 11.8 After Inserting 8, 14, and 22

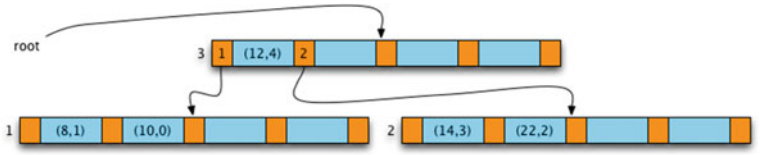

Fig. 11.9 After Splitting as a Result of Inserting 12

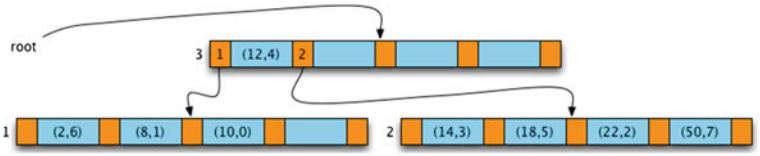

Fig. 11.10 After Inserting 18, 2, and 50

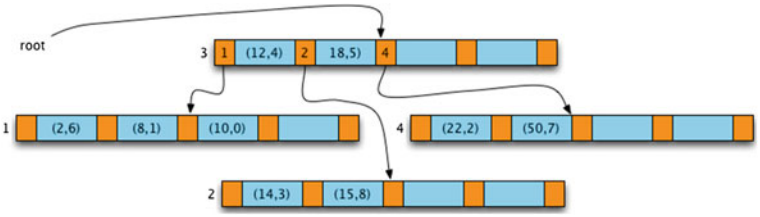

Fig. 11.11 Inserting 15 into the B-Tree Causes Splitting

Inserting an item causes one of two possible outcomes. Either the leaf node has room in it to add the new item or the leaf node splits resulting in a middle value and a new node being promoted to the parent. This suggests a recursive implementation is appropriate for inserting a new item. The recursive algorithm is given an item to insert and returns two values, the promoted key and the new right node if there is one and proceeds as follows.

1. If this is a leaf node and there is room for it, make room and store the item in the node.
2. Otherwise. if this is a leaf node, make a new node. Sort the new item and old items. Choose the middle item to promote to the parent. Take the items after the middle and put them into the new node. Return a tuple of the middle item and new right node.
3. If this is a non-leaf node, call insert recursively on the appropriate subtree. Consult the return value of the recursive call to see if there is a newly promoted key and right subtree. If so, take the appropriate action to store the new item and subtree pointer in the node. If there is no room to store the promoted value, split again as described in step 2.

Step 3 above automatically handles any cascading splits that must occur. After the recursive call the algorithm looks for any promoted value and handles it by either adding it into the node or by splitting again. An iterative version of insert would proceed in a similar manner as the recursive version except that the path to the newly inserted item would have to be maintained on a stack. Then, after inserting or splitting the leaf node, the stack of nodes on the path to the leaf would be popped one at a time, handling any promoted values, until the stack was emptied.

When writing insert as a recursive function it makes sense to implement it as a method of a B-Tree node class. Then the insert method on a B-Tree class can call the recursive insert on the B-Tree node class. In this way, if the root node is split, the B-Tree insert method can deal with this by creating a new root node from the promoted value and the left and right subtrees. Recall that the old root is the new left subtree in the newly created node.

11.7 B-Tree Delete

Deleting from a B-Tree can be written recursively or iteratively like the insert algorithm. When an item is deleted from a B-Tree there may be rebalancing required. Recall that every node, except the root node, of a B-Tree must contain at least *degree* items. There are just a few rules that can be followed to delete items from the tree while maintaining the balance requirements.

1. If the node containing the item is a leaf node and the node has more than degree items in it then the item may simply be deleted.
2. If the node containing the item is a leaf node and has *degree* or fewer items in it before deleting the value, then rebalancing is required.
3. If the node is a non-leaf node then the least value of the right subtree can replace the item in the node.

Rebalancing can be accomplished in one of two ways.

1. If a sibling of the unbalanced node contains more than *degree* items, then some of those items can be rotated into the current node.
2. If no rotation from a sibling is possible, then a sibling and the unbalanced node, along with the item that separates them in the parent, can be coalesced into one node. This reduces by one the number of items in the parent which in turn may cause cascading rotations or coalescing to occur.

Another example will help to illustrate the delete and rebalancing algorithm. Consider deleting the item containing 14 from the B-Tree in Fig. 11.11. This causes

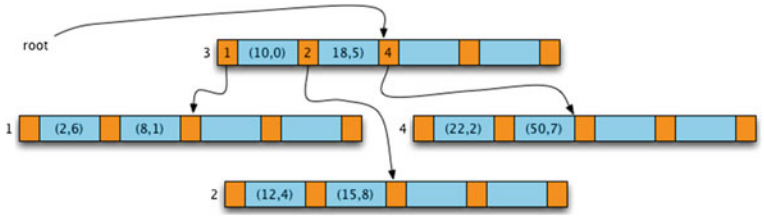

Fig. 11.12 After Deleting the Item Containing 14

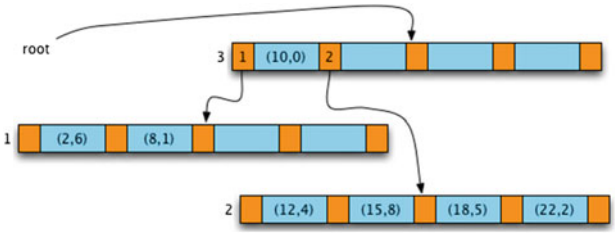

Fig. 11.13 After Deleting the Item Containing 50

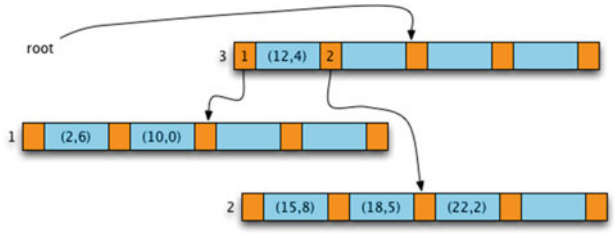

Fig. 11.14 After Deleting the Item Containing 8

the node containing 14 to become unbalanced. Rebalancing is accomplished by borrowing items from its left sibling. This is depicted in Fig. 11.12.

In Fig. 11.12 notice that the 10 rotates to the parent and the item containing 12 rotates into node 2 of the tree. This is necessary to maintain the ordering within the nodes. The rotation travels through the parent to redistribute the items between the two nodes. Next, consider deleting the item containing 50. In this case there is no sibling on the right and the sibling on the left doesn't have enough items to redistribute them. So, nodes 2 and 4 are coalesced into one node along with the item containing 18 from the root node, producing the B-Tree shown in Fig. 11.13.

Next, 8 is deleted from the B-Tree. This causes a left rotation with the right sibling resulting in the B-Tree depicted in Fig. 11.14.

Continuing the example assume that the item containing a key of 12 is deleted from the tree. The item is in a non-leaf node so in this case the least value from the right subtree replaces the item containing 12. This must be followed up with deleting

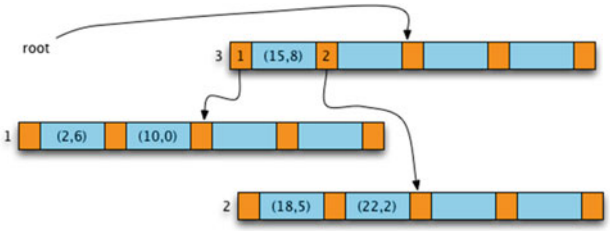

Fig. 11.15 After Deleting the Item Containing 12

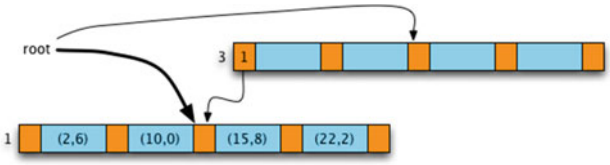

Fig. 11.16 After Deleting the Item Containing 18

that value, the item containing 15 in this case, from the right subtree. The result is depicted in Fig. 11.15.

Deleting 18 next causes the two sibling nodes to coalesce along with the separating item in the parent (the root in this case). The result is an empty root node as shown in Fig. 11.16. In this case, the delete method in the B-Tree class must recognize this situation and update the root node pointer to point to the correct node. B-Tree node 3 is no longer the root node of the B-Tree. Deleting any more of the nodes simply reduces the number of items in the root node.

Again, the delete method on B-Tree nodes may be implemented recursively. The B-Tree node delete method is given the item to delete and does not need to return anything. The recursive algorithm proceeds as follows.

1. If the item to delete is in the current node then we do one of two things depending on whether it is a leaf node or not.

    a. If the node is a leaf node, the item is deleted from the node without regard to rebalancing.
    b. If the node is a non-leaf node, then the smallest valued item from the right subtree replaces the item and the smallest valued item is deleted from the right subtree.

2. If the item is not in the current node then delete is called recursively on the correct subtree.
3. After delete returns, rebalancing of the child on the path to the deleted item may be needed. If the child node is out of balance first try rotating a value from a left

or right sibling. If that can't be done, then coalesce the child node with a left or right sibling.

If the algorithm is implemented iteratively instead of recursively a stack is needed to keep track of the path from the root node to the node containing the item to delete. After deleting the item the stack is emptied and as each node is popped from the stack rebalancing of the child node on the path may be required as described in the steps above.

11.8 Chapter Summary

B-Trees are very important data structures, especially for relational databases. In order for *join* operations to be implemented efficiently, indices are needed over at least some tables in a relational database. B-Trees are also important because they can be stored in record format on disk meaning that the entire index does not need to be present in RAM at any one time. This means that B-Trees can be created even for tables that consist of millions of records.

B-Trees have many important properties including O(log n) lookup, insert, and delete time. B-Trees always remain balanced, regardless of the order of insertions and deletions. B-Trees can also provide sequential access of records within a table in a sorted order, either ascending or descending.

Due to the balance requirement in B-Trees splitting of nodes may be required during item insertion. Rebalancing of nodes may be required during item deletion. Rebalancing takes the form of rotation of items or coalescing of nodes. Rotation to redistribute items is the preferred method of rebalancing.

Both the insert and delete operations may be implemented either recursively or iteratively. In either case the splitting or rebalancing may result in cascading splitting or rebalancing as the effects ripple up through the tree on the path taken to insert or delete the item. If implemented iteratively, both the insert and delete algorithms require a stack to record the path from the root node to the inserted or deleted item so that this ripple affect can be handled. In the recursive case no stack is required since the run-time stack remembers the path from the root node to the inserted or deleted item.

There are derivative implementations of B-Trees that have been created. B+-Trees and B#-Trees are two other variations that are not covered in this text. Alan Tharp [7], among others, covers both these derivative implementations.

11.9 Review Questions

Answer these short answer, multiple choice, and true/false questions to test your mastery of the chapter.

1. How does the use of an index improve the efficiency of the sample join operation presented in Sect. 11.2.7?
2. What advantages does a B-Tree have over a hash table implementation of an index?
3. What advantages does a hash table have over a B-Tree implementation of an index?
4. How can a B-Tree index be created over a table with millions of records and still be usable? What challenges could this pose and how does a B-Tree provide a means to deal with those challenges?
5. Starting with Fig. 11.13 insert an item with key 13 and draw a picture of the resulting B-Tree.
6. Starting with Fig. 11.10 delete the item containing 12 and draw a picture of the resulting B-Tree.
7. When does a node get coalesced? What does that mean? Provide a short example different from any example in the text.
8. When does a rotation correct imbalance in a node? Provide a short example different from any example in the text.
9. Insert the values 1 through 10 into an empty B-Tree of degree 4 to demonstrate your understanding of the insert algorithm. Draw pictures, but you can combine pictures that don't require splitting. At each split be sure to draw a completely new picture.
10. Delete the values 7, 8 and 9 from the tree you constructed in the previous review question showing the rebalanced tree after each deletion.

11.10 Programming Problems

1. Write a B-Tree class and a B-Tree node class. Implement the insert and delete algorithms described in this chapter. Implement a lookup method as well. Use this implementation to efficiently run the join operation presented in Sect. 11.2.7. Compare the time this algorithm takes to run to the time the non-indexed join, from Sect. 11.2.5, takes to run. Write the two methods recursively.
2. Write the B-Tree class with iterative, non-recursive, implementations of insert and delete. In this case the insert and delete methods of the B-Tree class don't necessarily have to call insert and delete on B-Tree nodes.
3. Since the example tables in this chapter are rather small, after completing exercise 1 or 2, run the query code again using a dictionary for the index. Compare the amount of time taken to implement the query in this way with the B-tree implementation. Comment on the experiment results.