Logical-to-Physical Mapping

- Recall our earlier diagram of a DBMS, which divides it into two layers:
  - the logical layer
  - the storage layer or storage engine

- The logical layer implements a mapping from the logical schema of a collection of data to its physical representation.
  - example: for the relational model, it maps:
    - attributes to fields
    - tuples to records
    - relations to files and index structures
    - selects, projects, etc. to scans, searches, field extractions
Your Task

- On the homework, you will implement portions of the logical-to-physical mapping for a simple relational DBMS.
- We will give you:
  - a SQL parser
  - a storage engine: Berkeley DB
  - portions of the code needed for the mapping, and a framework for the code that you will write
- In a sense, we've divided the logical layer into two layers:
  - a SQL parser
  - everything else – the "middle layer"
    - you'll implement parts of this

The Parser

- Takes a string containing a SQL statement and returns a reference to an object that is an instance of a subclass of the class `SQLStatement`.
- `SQLStatement` is abstract.
  - contains fields and methods inherited by the subclasses
  - includes an abstract `execute()` method
- You'll implement the `execute()` method for some of the subclasses.
**SQL Statement Class**

- Looks something like this:
  ```java
  public abstract class SQLStatement {
      private ArrayList<Table> tables;
      private ArrayList<Column> columns;
      private ArrayList<Object> columnVals;
      private ConditionalExpression where;
      private ArrayList<Column> whereColumns;

      public abstract void execute();
  }
  ...
  ```

**Other Aspects of the Code Framework**

- **DBMS**: the "main" class
  - methods to initialize, shutdown, or abort the system
  - methods to maintain and access the state of the system
  - to allow access to the DBMS methods from other classes, we make all methods static, so that the class name can be used to invoke them

- Classes that represent relational constructs, including:
  - **Table**
  - **Column**
  - **InsertRow**: a row that is being prepared for insertion in a table

- **Catalog**: a class that maintains the per-table metadata
  - here again, the methods are static
  - `putMetadata()`, `getMetadata()`, `removeMetadata()`
The Storage Engine: Berkeley DB (BDB)

- An embedded database library for managing key/value pairs
  - fast: runs in the application’s address space, no IPC
  - reliable: transactions, recovery, etc.

- One example of a type of noSQL database known as a key-value store.

- You will use the Berkeley DB Java API.

- We are also *not* using the Berkeley DB SQL interface.
  - we’re writing our own!

Berkeley DB Terminology

- A *database* in BDB is a collection of key/value pairs that are stored in the same index structure.
  - BDB docs say "key/data pairs" instead of "key/value pairs"

- Possible index structures: btree, hash, record-number, queue
  - referred to in the BDB documentation as *access methods*

- A database is operated on by making method calls using a *database handle* – an instance of the *Database* class.

- We will use one BDB database for each table/relation.
Berkeley DB Terminology (cont.)

- An *environment* in BDB encapsulates:
  - a set of one or more related BDB databases
  - the state associated with the BDB subsystems (caching, logging, transactions, locking) for those databases

- RDBMS: related *tables* are grouped together into a *database*. BDB: related *databases* are grouped together into an *environment*.

- Typically, the files for the BDB databases associated with a given environment are put in the same directory.
  - known as the environment’s *home directory*.

- An environment is operated on by making method/function calls using an *environment handle* – an instance of the `Environment` class.

---

Opening/Creating a BDB Database

- The environment must be configured and opened first.
  - we’ll give you the code for this

- Then you open/create the database.
  - create an object that specifies the database’s configuration:
    ```java
    DatabaseConfig config = new DatabaseConfig();
    config.setType(DatabaseType.BTREE);
    config.setAllowCreate(true);
    ...
    ```

  - use the environment handle to open the database, passing the `DatabaseConfig` object as an argument:
    ```java
    Environment dbenv = DBMS.getEnv();
    Database db = dbenv.openDatabase(null, "movie.db", null, config);
    ```
Key/Value Pairs

- In Berkeley DB, the on-disk keys and values are *byte arrays* – i.e., collections of bytes.

- Berkeley DB does *not* attempt to interpret them.

- Your code will need to impose structure on these byte arrays.
  - see the earlier notes on record formats, and the later notes on marshalling data

Key/Value Pairs (cont.)

- When manipulating keys and values within a program, we represent them using a `DatabaseEntry` object.

- For a given key/value pair, we need two `DatabaseEntry` objects.
  - one for the key, and one for the value

- Each `DatabaseEntry` encapsulates:
  - a reference to the collection of bytes (the *data*)
  - the *size* of the data (i.e., its length in bytes)
  - some additional fields
  - methods: `getData`, `getSize`, ...
Inserting Data into a BDB Database

- Create the `DatabaseEntry` objects for the key and value:
  ```java
  String keyStr = "cscie268";
  String valueStr = "Maxwell Dworkin G-135";
  DatabaseEntry key =
      new DatabaseEntry(keyStr.getBytes("UTF-8"));
  DatabaseEntry value =
      new DatabaseEntry(valueStr.getBytes("UTF-8"));
  ```

  - if we didn’t specify UTF-8, the JVM would use the default encoding, and the default encoding could subsequently change.

- Use the `Database` handle's `put` method:
  ```java
  Database db;  // assume it has been opened
  OperationStatus ret = db.put(null, key, value);
  ```

  - the first argument to `put` can be used to specify the transaction in which the put is occurring
  - `null` indicates no transaction

Retrieving a Single Key/Value Pair

- Create the `DatabaseEntry` objects for the key and value:
  ```java
  String keyStr = "cscie268";
  DatabaseEntry key =
      new DatabaseEntry(keyStr.getBytes("UTF-8"));
  DatabaseEntry value = new DatabaseEntry(); // none yet
  ```

- Use the `Database` handle's `get` method:
  ```java
  Database db;  // assume it has been opened
  OperationStatus ret = db.get(null, key, value, null);
  if (ret == OperationStatus.NOTFOUND) {
      System.err.println("no item with this key");
  } else if (ret == OperationStatus.SUCCESS) {
      byte[] valueBytes = value.getData();
      String valueStr = new String(valueBytes, "UTF-8");
      System.out.println(keyStr + " meets in " + valueStr);
  }
  ```
Cursors in Berkeley DB

• In general, a cursor is a construct used to iterate over the records in a database file.

• In BDB, a cursor iterates over key/value pairs in a BDB database.

• Cursor operations are performed by making method/function calls using a cursor handle (an object or struct).
  • an instance of the Cursor class

Opening a Cursor

• Use the Database handle's openCursor() method:
  
  ```java
  Database db;  // assume it has been opened
  Cursor curs = db.openCursor(null, null);
  ```

  • A cursor that has just been opened is not yet pointing to any record.
    • attempts to get the current record will fail

  • A cursor is initialized by performing a method/function call to get one of the records.
• if a cursor is not yet initialized, attempts to get the next record will retrieve the first record
• after a delete, the position of the cursor doesn’t change, so you need to reposition it before trying to do something with the current record

Summary of Important Cursor Methods

<table>
<thead>
<tr>
<th>operation</th>
<th>method</th>
</tr>
</thead>
<tbody>
<tr>
<td>get the first record</td>
<td>getFirst()</td>
</tr>
<tr>
<td>get the record with the specified key</td>
<td>getSearchKey()</td>
</tr>
<tr>
<td>get the next record</td>
<td>getNext()</td>
</tr>
<tr>
<td>get the current record</td>
<td>getCurrent()</td>
</tr>
<tr>
<td>modify the current record</td>
<td>putCurrent()</td>
</tr>
<tr>
<td>delete current record</td>
<td>delete()</td>
</tr>
<tr>
<td>close the cursor</td>
<td>close()</td>
</tr>
</tbody>
</table>

Iterating Over All Records Using a Cursor

• The key/value pairs are returned in DatabaseEntry that are passed as parameters to the getter method.
• Assume we have an open cursor whose handle is curs.
• Here’s an example:

```java
databaseEntry key = new databaseEntry();
databaseEntry value = new databaseEntry();
while (curs.getNext(key, value, null) == OperationStatus.SUCCESS) {
    String keyStr = new String(key.getData(), "UTF-8");
    String valueStr = new String(value.getData(), "UTF-8");
    System.out.println(keyStr + " meets in " + valueStr);
}
```
Marshalling Data

- We need to be able to take a collection of fields and store them in a key/value pair – each component of which is a byte array
  - example:
    
    ```
    ('1234567', 'comp sci', 200)  
    ```

- This process is referred to as *marshalling* the data.
- The reverse process is known as *unmarshalling*.

Marshalling Data with BDB’s Java API

- In theory, we could use Java serialization to convert objects to byte arrays and back.
  - produces unnecessarily large records, because class information is stored in each record!
  - it’s also slow, and you can’t sort the resulting byte arrays
- Instead, we’ll make use of some of the classes from BDB’s *Bind API*.
  - provide methods for storing various types of data into byte arrays, and vice versa
**Classes from the BDB Bind API**

- **TupleOutput**: an output stream with methods that write values into a byte array (similar to Java's `DataOutputStream`)
  - `writeByte(int val)`
  - `writeBytes(String val)`
  - `writeInt(int val)`
  - `writeDouble(double val)`

- **TupleInput**: an input stream with methods that read values from a byte array (similar to Java's `DataInputStream`)
  - `readByte()`
  - `readBytes(int length)`
  - `readInt()`
  - `readDouble()`

- We'll give you more info about using the Bind API in the assignments.