Implementing a Logical-to-Physical Mapping

Computer Science E-66
Harvard University
David G. Sullivan, Ph.D.

Recall: 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’re giving 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

- Creates an instance of a subclass of the class SQLStatement:

  SQLStatement
  
  CreateStatement  DropStatement  InsertStatement

  - SQLStatement is an abstract class.
    - contains fields and methods inherited by the subclasses
    - includes an abstract execute() method
      - just the method header, not the body

  - Each subclass implements its own version of execute()
    - you'll do this for some of the subclasses
SQLStatement 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();
    ...
}
```

Java's built-in ArrayList class. Use the Java API to see the available methods!

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 its methods static
    - this means 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.
- We’re using Berkeley DB Java Edition (JE)
- Note: We’re 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"
- BDB Java Edition always uses a B+tree.
  - other versions of BDB provide other index-structure options
- 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.

Opening/Creating a BDB Database

• The environment must be configured and opened first.

• Then you open/create the database.

• We give you the code for this in the DBMS framework:
  • DBMS.environmentInit() creates/opens the environment
  • CreateStatement.execute() creates a database
    for a new table
  • Table.open() opens the database for an existing table

• Because BDB Java Edition always uses a B+tree,
  we need keys for the key/value pairs.
  • if a primary key isn't specified, we use the first column

• Note: our framework assumes that primary keys never involve
  more than one attribute.
Opening/Creating a BDB Database

- The environment must be configured and opened first.

- Then you open/create the database.

- We give you the code for this in the DBMS framework:
  - `DBMS.environmentInit()` creates/opens the environment
  - `CreateStatement.execute()` creates a database for a new table
  - `Table.open()` opens the database for an existing table

- The DBMS framework uses the following index structures:
  - btree for tables that have a specified primary key
  - recno for tables that don't have one

- Note: our framework assumes that primary keys never involve more than one attribute.

Key/Value Pairs

- 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` s.
  - one for the key
  - 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`, ...
  - consult the Berkeley DB API for info on the methods!
Byte Arrays

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

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

• Your code will need to impose structure on these byte arrays.

Marshalling the Data

• When inserting a row, we need to turn a collection of fields into a key/value pair.
  • example:
    \[
    ('1234567', 'comp sci', 200)
    \]

  • In BDB, the key and value are each:
    • represented by a *DatabaseEntry* object
    • based on a byte array that we need to create

  • This process is referred to as *marshalling* the data.

  • The reverse process is known as *unmarshalling*. 
Recall: Format of Variable-Length Records

- Option 3: Put offsets and other metadata in a record header.

  Option 3 did:
  ('1234567', 'comp sci', 200)

  We'll do something a bit different:
  ('1234567', 'comp sci', 200)

  Our Recommended Record Format

  - Here's what option 3 did:

    ('1234567', 'comp sci', 200) → 16 23 31 35 1234567 comp sci 200

  - We'll do something a bit different:

    (1234567, 'comp sci', 200)

    - the primary-key value becomes the key in the key/value pair
    - the value is the other fields with a header of offsets
    - we use a special offset for the primary-key in the header
      (note: it won't always be the first column!)
    - what should the remaining offsets be in this case?
      (assume 2-byte offsets)
Our Recommended Record Format

• Here’s what option 3 did:
  (`'1234567', 'comp sci', 200) \rightarrow \begin{array}{c}
  16 \ 23 \ 31 \ 35 \ 1234567 \ comp \ sci \ 200
\end{array}

• We’ll do something a bit different:

\begin{array}{c}
  \text{key} \\
  1234567
\end{array}

\begin{array}{c}
  \text{value} \\
  -2 \ 8 \ 16 \ 20 \ comp \ sci \ 200
\end{array}

• the primary-key value becomes the key in the key/value pair
• the value is the other fields with a header of offsets
• we use a special offset for the primary-key in the header
  (note: it won't always be the first column!)
• what should the remaining offsets be in this case?
  (assume 2-byte offsets)

Marshalling Data on PS 2

• 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
• you can’t sort the resulting byte arrays

• Instead, you'll make use of classes that we've provided.
Classes for Manipulating Byte Arrays

- **RowOutput**: an output stream that writes into a byte array
  - inherits from Java’s **DataOutputStream**:
    - `writeBytes(String val)`
    - `writeShort(int val)` // can use for offsets!
    - `writeInt(int val)`
    - `writeDouble(double val)`
  - methods for obtaining the results of the writes:
    - `getBufferBytes()`
    - `getBufferLength()`
  - includes a `toString()` method that shows the current contents of the byte array

Classes for Manipulating Byte Arrays (cont.)

- **RowInput**: an input stream that reads from a byte array
  - methods that take an offset from the start of the byte array
    - `readBytesAtOffset(int offset, int length)`
    - `readIntAtOffset(int offset)`
    - etc.
  - methods that read from the current offset (i.e., from where the last read left off)
    - `readNextBytes(int length)`
    - `readNextInt()`
    - etc.
  - includes a `toString()` method that shows the contents of the byte array and the current offset
**Example of Marshalling**

('1234567', 'comp sci', 200)

- Marshalling this row could be done as follows:
  ```java
  RowOutput keyBuffer = new RowOutput();
  keyBuffer.writeBytes("1234567");
  RowOutput valueBuffer = new RowOutput();
  valueBuffer.writeShort(-2);
  valueBuffer.writeShort(8);
  valueBuffer.writeShort(16);
  valueBuffer.writeShort(20);
  valueBuffer.writeBytes("comp sci");
  valueBuffer.writeInt(200);
  ```

**Inserting Data into a BDB Database**

- Create the DatabaseEntry objects for the key and value:
  ```java
  // see previous slide for marshalling code
  byte[] bytes = keyBuffer.getBufferBytes();
  int numBytes = keyBuffer.getBufferLength();
  DatabaseEntry key = new DatabaseEntry(bytes, 0, numBytes);

  bytes = valueBuffer.getBufferBytes();
  numBytes = valueBuffer.getBufferLength();
  DatabaseEntry value = new DatabaseEntry(bytes, 0, numBytes);
  ```

- Use the appropriate Database "put" method:
  ```java
  Database db;  // assume it has been opened
  OperationStatus ret = db.putNoOverwrite(null, key, value);
  ```
  - the first argument can specify a transaction
    - null indicates no transaction
  - the return value is of type OperationStatus
    - used to indicate success or failure (see next slide)
BDB Database Methods for Inserting Data

• **put()**
  • if there is an existing key/value pair with the specified key, **put()** will replace the existing value with the new one
  • would be useful when implementing UPDATE commands

• **putNoOverwrite()** (see previous slide)
  • if there is an existing key/value pair with the specified key:
    • the insertion fails
    • the method returns `OperationStatus.KEYEXIST`

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 calls using a *cursor handle*.
  • an instance of the Cursor class
Opening and Using a Cursor

• Use the Database handle’s openCursor() method:
  ```
  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.

• A cursor is initialized by performing a method call to get a record.

• Cursor methods include:
  • `getNext()` – gets the next key/value pair
    • if a cursor is not yet initialized, attempts to get the next record will retrieve the first record
  • `getFirst()` – gets the first key/value pair
    • useful if you need to reset the cursor to the first record after reaching the end of the database

Opening and Using a Cursor (cont.)

• The key/value pairs are returned in "empty" DatabaseEntry objects that are passed as parameters to the cursor’s "get" method:
  ```
  DatabaseEntry key = new DatabaseEntry();
  DatabaseEntry value = new DatabaseEntry();
  OperationStatus ret = curs.getNext(key, value, null);
  ```

• We could also use a loop to iterate over the entire database:
  ```
  while (curs.getNext(key, value, null) == OperationStatus.SUCCESS) {
    // code to unmarshall each key/value pair goes here
  }
  ```
Table Iterators

- You will use a cursor to implement the `TableIterator` class.

- It will be used to iterate over the tuples in either:
  - an entire single table:
    
    ```
    SELECT *
    FROM Movie;
    ```
  
  - or the relation that is produced by applying a selection operator to the tuples of single table:
    
    ```
    SELECT *
    FROM Movie
    WHERE rating = 'PG-13' and year > 2010;
    ```

- A `TableIterator` will have:
  
  - fields for the current key/value pair
  
  - methods for accessing values in the current key/value pair
  
  - methods for advancing/resetting the underlying cursor

Simple Example of Unmarshalling

- Once we have retrieved the key/value pair for a given row, we could unmarshall the entire row as follows:

  ```
  DatabaseEntry key, value;
  // code to read in the key/value pair goes here
  RowInput keyIn = new RowInput(key.getData());
  int length = key.getSize();
  String id = keyIn.readNextBytes(length);
  RowInput valueIn = new RowInput(value.getData());
  int[] offsets = new int[4];
  for (int i = 0; i < offsets.length; i++) {
    offsets[i] = valueIn.readNextShort();
  }
  length = offsets[2] - offsets[1];
  String department = valueIn.readNextBytes(length);
  int enrollment = valueIn.readNextInt();
  ```
Retrieving a Single Field's Value

- On the problem set, you will not unmarshall the entire record all at once.

- Rather, you will write a TableIterator method that unmarshalls the value of a single column:
  
  ```java
  public Object getColumnVal(int colIndex) {
  
  • takes in a column index
  • returns the value of that column in the current row
  
  Thus, you should mostly use the "at offset" versions of the RowInput methods.
  • readBytesAtOffset, readIntAtOffset, etc.
  ```

Examples of Unmarshalling: Assumptions

- We have a simplified version of the Movie table from PS 1:
  
  Movie(id CHAR(7), name VARCHAR(64), runtime INT, rating VARCHAR(5), earnings_rank INT)

- We didn't specify a primary key when we created the table.
  • thus, id is the primary key – and the key in the key/value pair
  • the rest of the row is in the value portion of the key/value pair

- We're using 2-byte offsets.

- The cursor/iterator is currently positioned on this key/value pair:
  
  ```
  key 4975722  value 0 2 4 6 8 10 12 21 25 -1 26 Moonlight 111 R
  ```

- We'll use index notation to refer to particular offset or field value:
  • examples: offset[1], field[1] (but they are not really arrays!)
Example 1

\begin{verbatim}
Movie(id CHAR(7), name VARCHAR(64), runtime INT, 
    rating VARCHAR(5), earnings_rank INT)

• To retrieve the movie's name (field[1] – the second field):
  • determine that offset[1] is 1*2 = 2 bytes from the start
  • perform a read at offset 2 to obtain offset[1] \rightarrow 12
  • because name is a VARCHAR, read offset[2] \rightarrow 21 
    and compute this name's length = 21 – 12 = 9
  • read 9 bytes at an offset of 12 bytes \rightarrow 'Moonlight'
\end{verbatim}

Example 2

\begin{verbatim}
Movie(id CHAR(7), name VARCHAR(64), runtime INT, 
    rating VARCHAR(5), earnings_rank INT)

• To retrieve the earnings_rank (field[4])
  • determine that offset[4] is 4*2 = 8 bytes from the start
  • perform a read at offset 8 to obtain offset[4] \rightarrow -1
  • conclude that the value is NULL
\end{verbatim}
Example 3

```
value  0  2  4  6  8 10 12 21 25
-2  12 21 25 -1 26 Moonlight 111 R
```

`Movie(id CHAR(7), name VARCHAR(64), runtime INT, rating VARCHAR(5), earnings_rank INT)`

- To retrieve the rating (field[3]):
  - determine that offset[3] is 3*2 = 6 bytes from the start
  - perform a read at offset 6 to obtain offset[3] → 25
  - because rating is a VARCHAR:
    - read offset[4] → -1, so we need to keep going!
    - read offset[5] → 26
    - compute this rating's length = 26 – 25 = 1
  - read 1 byte at an offset of 25 → 'R'