Implementing a Logical-to-Physical Mapping

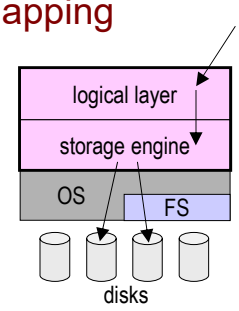
Harvard Extension School

Cody Doucette, Ph.D.

Lecture designed by David G. Sullivan

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*

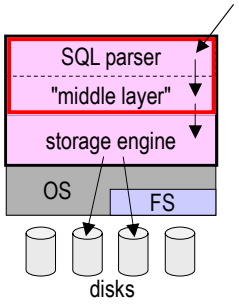


- The logical layer implements a mapping from the logical schema of a collection of data to its physical representation.
  - example: for the relational model, it maps:

attributes		fields
tuples	to	records
relations		files and index structures
selects, projects, etc.		scans, searches, field extractions

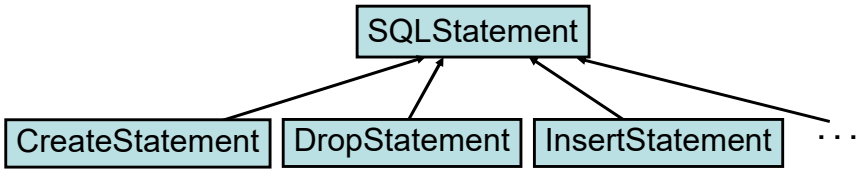
Your Task

- 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` 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:

```
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 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

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 for those databases
- RDBMS: related tables are grouped together into a database.
  BDB: related databases are grouped together into an environment.
- Files for a given environment are put in the same folder.
  - known as the environment's *home directory*

Opening/Creating a BDB Database

- We give you the code for this in the DBMS framework:
  - `CreateStatement.execute()` creates a database for a new table
  - `Table.open()` opens the database for an existing table
- Use the table's primary key for the keys in the key/value pairs.
  - if one wasn't specified when the table was created, we use the first column
  - can assume no multi-attribute primary keys

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* `DatabaseEntrys`.
  - 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!

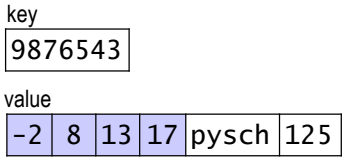
Byte Arrays

- 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:

('9876543', 'psych', 125)



- 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*.

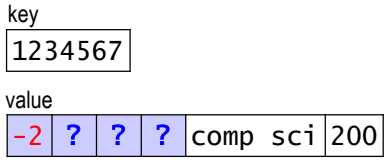
The Required Record Format

- Here's what option 3 did:

('1234567', 'comp sci', 200) → 8 | 15 | 23 | 27 | 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 and 4-byte integer values)

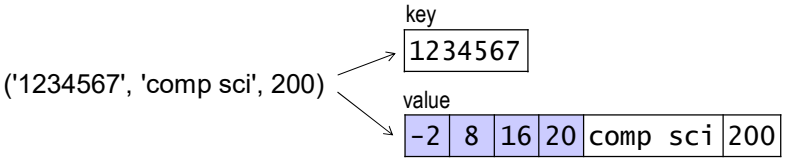
Classes for Manipulating Byte Arrays

- `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



- Marshalling this row could be done as follows:

```
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:

```
// 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 `Database putNoOverwrite` method:

```
Database db;  // assume it has been opened
OperationStatus ret = db.putNoOverwrite(null, key, value);
```

  - `null` because we are not using transactions
  - if there is an existing key/value pair with the specified key:
    - the insertion fails
    - the method returns `OperationStatus.KEYEXIST`
  - if the insertion succeeds, returns `OperationStatus.SUCCESS`

Cursors in Berkeley DB

- In general, a *cursor* is a construct used to iterate over records in a database file.
  - similar to an iterator for a collection class
- In BDB, cursors iterate over key/value pairs in a BDB database.
  - based on method calls using an instance of the `Cursor` class
- The key/value pairs are returned in "empty" `DatabaseEntry`s that are passed as parameters to the cursor's `getNext` method:

```
DatabaseEntry key = new DatabaseEntry();
DatabaseEntry value = new DatabaseEntry();
OperationStatus ret = curs.getNext(key, value, null);
```

Table Iterators

- In PS 2, a cursor is used to implement a `TableIterator` class.
- It can 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` has:
  - fields for the current key/value pair accessed by the cursor
  - methods for advancing/resetting the cursor
  - a method you'll implement for getting a column's value

Unmarshalling a Single Field's Value

- You will write a `TableIterator` method that unmarshalls the value of *a* single column from the current key/value pair.

```
public Object getColumnVal(int colIndex)
```

- First, you'll need to create the necessary `RowInput` objects:

```
RowInput keyIn = new RowInput(this.key.getData());
RowInput valueIn = new RowInput(this.value.getData());
```

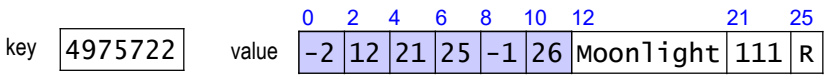
- Then you'll use `RowInput` methods to access the necessary offset(s) and value.
- You should *not* unmarshall the entire record – only the portions that are needed to get the value of the specified column.
- Thus, you should mostly use the "at offset" versions of the `RowInput` methods.
  - `readBytesAtOffset`, `readIntAtOffset`, etc.

Examples of Unmarshalling: Assumptions

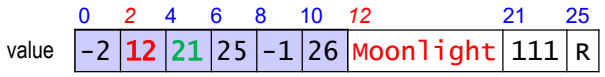
- We have a simplified version of the Movie table from PS 1:

```
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.
  - `-2` indicates the primary key
  - `-1` indicates a NULL value
- The cursor/iterator is currently positioned on this key/value pair:



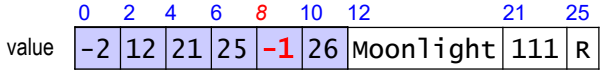
Example 1




```
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 an offset of 2 to obtain $offset_1$ → 12
  - because name is a VARCHAR, read $offset_2$ → 21 and compute this name's length = 21 – 12 = 9
  - read 9 bytes at an offset of 12 bytes → 'Moonlight'

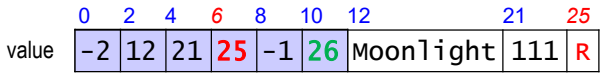
Example 2




```
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 an offset of 8 to obtain $offset_4$ → -1
  - conclude that the value is NULL

Example 3




```
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 an offset of 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'