DBMS Architecture

• A DBMS can be viewed as a composition of two layers.

• At the bottom is the storage layer or storage engine, which takes care of storing and retrieving the data.

• Above that is the logical layer, which provides an abstract representation of the data.
  • includes some type of query language or API

• Different approaches select different options for these layers.
### Logical Schema

- The *logical schema* is the abstract representation of a database.
- Based on the data model.
- In the relational model, the schema includes:
  - relations/tables
  - tuples/rows
  - attributes/columns, including their types
- To be model-neutral, we'll use these terms instead:
  - *field* for an individual data value
  - *record* for a group of fields
  - *collection* for a group of records

### Logical-to-Physical Mapping

- The logical layer implements a mapping between:
  - the *logical schema* of a database
  - its *physical representation*

- Issues include:
  - how should we format the on-disk records?
  - what *metadata* is needed and where should it be stored?
    - example: the types and lengths of the fields
    - may need both *per-record* and *per-collection* metadata
  - how should the logical operations be implemented?
Example of a Logical-to-Physical Mapping

- Let's say we're using Berkeley DB for the storage layer.

- Most likely, each record would correspond to which BDB construct?

- Issues include:
  - how should we format the key and the data item?
  - which BDB index structures should we use?
  - where should we store the metadata?
  - what BDB operations are needed for each logical operation?

Mapping Records

- Design issues include:
  - whether to use fixed- or variable-length records
  - how to represent the field values in each record
    - and their associated metadata (if any)
  - how to handle NULL values
Fixed- or Variable-Length?

• This choice depends on:
  • the types of fields that the records contain
  • the number of fields per record, and whether it can vary

• Simple case: use fixed-length records when
  • all fields are fixed-length (e.g., CHAR or INTEGER),
  • there is a fixed number of fields per record

Fixed- or Variable-Length? (cont.)

• The choice is less straightforward when you have either:
  • variable-length fields (e.g., VARCHAR)
  • a variable number of fields per record
    • example: in an XML database

• Two options:
  1. fixed-length records: always allocate the maximum possible total length
     • pros and cons?  
       ... | comp | sci | ...
       ... | math | ... |

  2. variable-length records: only allocate the space needed for a given record
     • pros and cons?  
       ... | comp | sci | ...
       ... | math | ... |
Format of Fixed-Length Records

- With fixed-length records, the fields can be stored consecutively.
- If a fixed-length record contains a variable-length field, we allocate the max. length and store the actual length.

<table>
<thead>
<tr>
<th>field1</th>
<th>field2</th>
<th>field3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234567</td>
<td>comp sci</td>
<td>200</td>
</tr>
<tr>
<td>9876543</td>
<td>math</td>
<td>125</td>
</tr>
</tbody>
</table>

**computing offsets:**
- field1: \( O_1 = 0 \) (zero)
- field2: \( O_2 = L_1 \)
- field3: \( O_3 = O_2 + L_2 \)
- field\( n \): \( O_n = O_{n-1} + L_{n-1} \)

- To find the position of a field, use the per-collection metadata.
  - if metadata stores the offsets of the fields, use the offset
  - if metadata only stores the field lengths, compute the offset

---

Format of Variable-Length Records

- With variable-length records, we need per-record metadata to determine the locations of the fields.
- We’ll assume all records of a given type have the same # of fields.
- Options:
  1. terminate field values by a special delimiter character
     - scan to find a given value
     - pros and cons?

  2. precede each field by its length
     - allows us to “jump” over fields while scanning
     - pros and cons?
Format of Variable-Length Records (cont.)

- Options (cont.):
  3. put offsets and other metadata in a record header
    - pros and cons?
      
      | 0 | 4 | 8 | 12 | 16 | 23 | 31 |
      |---|---|---|----|----|----|----|
      | 16| 23| 31| 35 | 12 | 34 | 56 |
      | comp| sci| 200 |

    - what operation requires recomputing offsets in all records?
    - how can we avoid this?

4. store metadata in a separate file
   - key disadvantage?

Representing Null Values

- Option 1: use per-record metadata to indicate the presence of a NULL value.
  - 1a: for some fields, a length of 0 could indicate NULL when wouldn't this work?
  - 1b: could add additional metadata to the record for the purpose of indicating null
    - pros and cons?

- Option 2: add an "out-of-band" value for every type.
  - pros and cons?

- Option 3: use a separate structure to track NULL values.
  - pros and cons?
Marshalling Data

- Regardless of the record format that you use, you need to transform a record's fields into a form that can be stored on disk.
- Referred to as *marshalling* the data.
- The reverse process is known as *unmarshalling*.
- Example: using BDB for the storage engine
  - marshalling entails:
    - storing the field(s) for the key in one byte array
    - storing the full record in another byte array
    - include any necessary metadata
  - unmarshalling: extracting fields from the byte arrays

Marshalling Data (cont.)

- In C, we can use functions like `memcpy()` to copy field values to and from a byte array.
- In an OOP language like Java, we can:
  - represent the fields to be marshalled as an object
  - use *serialization* to convert an object into a byte array
  - drawbacks:
    - produces unnecessarily large records, because class information is stored in each record
    - slow, and you can’t sort the resulting byte arrays
- BDB’s Java API includes a *Bind API* with objects and methods that be can be used for marshalling and unmarshalling.
Per-Collection Metadata

- Includes things like:
  - names of the fields
  - types of the fields
  - which field(s) make up the primary key
  - how to compare values of a given field

- Where should we store it?
  - in application logic
    - e.g., an application with a fixed set of BDB databases
  - in the collection itself
    - e.g., have the first record describe the fields
  - in a separate file for each collection
  - in a catalog – one or more files that span all collections
    - one option: have the catalog be a database!
      - challenge of this approach?

Per-Collection Metadata (cont.)

- Exactly what metadata should we store?
  - example: fixed-length records

```
1234567 8 comp sci 200
9876543 4 math 125
```

- option 1: store field offsets
  - compute lengths from offsets
- option 2: store field lengths
  - compute offsets from lengths
- which would you choose?

- Consider the operations that use the metadata, and let that guide what you store and how you store it.
Review: Index Structures

- Index structures allow for more efficient access to data records.
  - given the value of one or more key field(s), we can find the appropriate record in a small # of disk reads
  - B-tree, hash table
  - covered earlier in the semester

- Now we'll briefly consider how index structures are used in the logical-to-physical mapping.

- If records are stored randomly (outside an index structure), the resulting file is sometimes called a heap file.
  - managed somewhat like the heap memory region

Primary Index

- A given collection of data records is typically stored within an index structure.
  - a clustered index or internal index
  - a primary index, because it is typically based on the primary key

- Example:
  - logical: Customer(id, name, street, city, state, zip)
  - physical: a BDB B-tree database with:
    - key = id
    - data = a full tuple
Secondary Indices

• In addition to the clustered/primary index, there can be one or more additional indices based on other fields.
  • an unclustered or external index
  • a secondary index

• Typically consist of (index value, primary key) pairs.
  • example: Customer(id, name, street, city, state, zip)
    • primary index: see previous slide
    • secondary index: key = name, data = id

• Need two lookups:

  secondary index value     secondary index value
  'Ted Codd'                '012345'
  primary key               primary key
  ('012345', 'Ted Codd', ...)
  full record

Organizing Collections on Disk

• The storage engine may use the filesystem, or it may bypass it and use its own disk manager.
  • pros and cons of each approach?

• Assume that we have directories or their equivalent.

• How should we organize the collections of records?
  • one directory for each collection?
  • one directory for each database (i.e., each collection of related collections)?
Example Organization: MySQL

- One directory for the catalog, which is itself a database.
  - consists of tables specifying privileges – who can access what
    - database-level privileges
    - table-level privileges
    - etc.

- One directory for each user database.

- Each table is represented by three files:
  - one for the per-table metadata
  - one for the data
  - one for any indices on the table