Chapter 9: Data Management Layer Design

Objectives

- Become familiar with several object-persistence formats.
- Be able to map problem domain objects to different object-persistence formats.
- Be able to apply the steps of normalization to a relational database.
- Be able to optimize a relational database for object storage and access.
- Become familiar with indexes for relational databases.
- Be able to estimate the size of a relational database.
- Understand the effect of nonfunctional requirements on the data management layer
- Be able to design the data access and manipulation classes.

Introduction

- Applications are of little use without data
  - Data must be stored and accessed efficiently
- The data management layer includes:
  - Data access and manipulation logic
  - Storage design
- Four-step design approach:
  - Selecting the format of the storage
  - Mapping problem-domain objects to object persistence format
  - Optimizing the object persistence format
  - Designing the data access & manipulation classes

Object Persistence Formats

- Files (sequential and random access)
- Object-oriented databases
- Object-relational databases
- Relational databases
- “NoSQL” data stores

Electronic Files

- Sequential access files
  - Operations (read, write and search) are conducted one record after another (in sequence)
  - Efficient for report writing
  - Inefficient for searching (an average of 50% of records have to be accessed for each search)
  - Unordered files add records to the end of the file
  - Ordered files are sorted, but additions & deletions require additional maintenance
- Random access files
  - Efficient for operations (read, write and search)
  - Inefficient for report writing

Application File Types

- Master Files
  - Store core information (e.g., order and customer data)
  - Usually held for long periods
  - Changes require new programs
- Look-up files (e.g., zip codes with city and state names)
- Transaction files
  - Information used to update a master file
  - Can be deleted once master file is updated
- Audit file—records data before & after changes
- History file—archives of past transactions

Relational Databases

- Most popular way to store data for applications
- Consists of a collection of tables
  - Primary key uniquely identifies each row
  - Foreign keys establish relationships between tables
    - Referential integrity ensures records in different tables are matched properly
    - Example: you cannot enter an order for a customer that does not exist
- Structured Query Language (SQL) is used to access the data
  - Operates on complete tables vs. individual records
  - Allows joining tables together to obtain matched data

Object-Relational Databases

- A standard relational database with ability to store objects added
- Must create a mapping from UML class diagrams to database schema is required.
- Accomplished using user-defined data types
  - SQL extended to handle complex data types
  - Support for inheritance varies

Object-Oriented Databases

- Two approaches:
  - Add persistence extensions to OO programming language
  - Create a separate OO database
- Utilize extents—a collection of instances of a class
  - A table associated with a class.
  - Each row an instance of the class, and having an Object ID
  - Object IDs relate objects together
  - Another primary key may still be desirable
- Inheritance is supported but is language dependent
- Represent a small market share due to its steep learning curve

NoSQL Data Stores

- Newest type; used primarily for complex data types
  - Does not support SQL
  - No standards exist
  - Support very fast queries
- Data may not be consistent since there are no locking mechanisms
- Types
  - Key-value data stores
  - Document data stores
  - Columnar data stores
- Immaturity of technology prevents traditional business application support

Selecting Persistence Formats

	Sequential and Random Access Files	Relational DBMS	Object Relational DBMS	Object-Oriented DBMS
Major Strengths	Usually part of an object-oriented programming language Files can be designed for fast performance Good for short-term data storage	Leader in the database market Can handle diverse data needs	Based on established, proven technology, e.g., SQL Able to handle complex data	Able to handle complex data Direct support for object orientation
Major Weaknesses	Redundant data Data must be updated using programs, i.e., no manipulation or query language No access control	Cannot handle complex data No support for object orientation Impedance mismatch between tables and objects	Limited support for object orientation Impedance mismatch between tables and objects	Technology is still maturing Skills are hard to find
Data Types Supported	Simple and Complex	Simple	Simple and Complex	Simple and Complex
Types of Application Systems Supported	Transaction processing	Transaction processing and decision making	Transaction processing and decision making	Transaction processing and decision making
Existing Storage Formats	Organization dependent	Organization dependent	Organization dependent	Organization dependent
Future Needs	Poor future prospects	Good future prospects	Good future prospects	Good future prospects

Mapping Problem-Domain Objects to Object-Persistence Formats

- Map objects to an OODBMS format
  - Each concrete class has a corresponding object persistence class
  - Add a data access and manipulation class to control the interaction
- Map objects to an ORDBMS format
  - Procedure depends on the level of support for object orientation by the ORDBMS
- Map objects to an RDBMS format

Mapping to an ORDBMS

Rule 1: Map all concrete Problem Domain classes to the ORDBMS tables. Also, if an abstract problem domain class has multiple direct subclasses, map the abstract class to an ORDBMS table.

Rule 2: Map single-valued attributes to columns of the ORDBMS tables.

Rule 3: Map methods and derived attributes to stored procedures or to program modules.

Rule 4: Map single-valued aggregation and association relationships to a column that can store an Object ID. Do this for both sides of the relationship.

Rule 5: Map multivalued attributes to a column that can contain a set of values.

Rule 6: Map repeating groups of attributes to a new table and create a one-to-many association from the original table to the new one.

Rule 7: Map multivalued aggregation and association relationships to a column that can store a set of Object IDs. Do this for both sides of the relationship.

Rule 8: For aggregation and association relationships of mixed type (one-to-many or many-to-one), on the single-valued side (1..1 or 0..1) of the relationship, add a column that can store a set of Object IDs. The values contained in this new column will be the Object IDs from the instances of the class on the multivalued side. On the multivalued side (1..* or 0..*), add a column that can store a single Object ID that will contain the value of the instance of the class on the single-valued side.

For generalization/inheritance relationships:

Rule 9a: Add a column(s) to the table(s) that represents the subclass(es) that will contain an Object ID of the instance stored in the table that represents the superclass. This is similar in concept to a foreign key in an RDBMS. The multiplicity of this new association from the subclass to the "superclass" should be 1..1. Add a column(s) to the table(s) that represents the superclass(es) that will contain an Object ID of the instance stored in the table that represents the subclass(es). If the superclasses are concrete, that is, they can be instantiated themselves, then the multiplicity from the superclass to the subclass is 0..1, otherwise, it is 1..1. An exclusive-or (XOR) constraint must be added between the associations. Do this for each superclass.

or

Rule 9b: Flatten the inheritance hierarchy by copying the superclass attributes down to all of the subclasses and remove the superclass from the design.*

*It is also a good idea to document this modification in the design so that in the future, modifications to the design can be maintained easily.

Mapping to an RDBMS

Rule 1: Map all concrete-problem domain classes to the RDBMS tables. Also, if an abstract Problem Domain class has multiple direct subclasses, map the abstract class to a RDBMS table.

Rule 2: Map single-valued attributes to columns of the tables.

Rule 3: Map methods to stored procedures or to program modules.

Rule 4: Map single-valued aggregation and association relationships to a column that can store the key of the related table, i.e., add a foreign key to the table. Do this for both sides of the relationship.

Rule 5: Map multivalued attributes and repeating groups to new tables and create a one-to-many association from the original table to the new ones.

Rule 6: Map multivalued aggregation and association relationships to a new associative table that relates the two original tables together. Copy the primary key from both original tables to the new associative table, i.e., add foreign keys to the table.

Rule 7: For aggregation and association relationships of mixed type, copy the primary key from the single-valued side (1..1 or 0..1) of the relationship to a new column in the table on the multivalued side (1..* or 0..*) of the relationship that can store the key of the related table, i.e., add a foreign key to the table on the multivalued side of the relationship.

For generalization/inheritance relationships:

Rule 8a: Ensure that the primary key of the subclass instance is the same as the primary key of the superclass. The multiplicity of this new association from the subclass to the "superclass" should be 1..1. If the superclasses are concrete, that is, they can be instantiated themselves, then the multiplicity from the superclass to the subclass is 0..1, otherwise, it is 1..1. Furthermore, an exclusive-or (XOR) constraint must be added between the associations. Do this for each superclass.

OR

Rule 8b: Flatten the inheritance hierarchy by copying the superclass attributes down to all of the subclasses and remove the superclass from the design.*

* It is also a good idea to document this modification in the design so that in the future, modifications to the design can be maintained easily.

Optimizing RDBMS-Based Object Storage

- Primary (often conflicting) dimensions:
  - Improve storage efficiency
    - Normalize the tables
    - Reduce redundant data and the occurrence of null values
  - Improve speed of access
    - De-normalize some tables to reduce processing time
    - Place similar records together (clustering)
    - Add indexes to quickly locate records

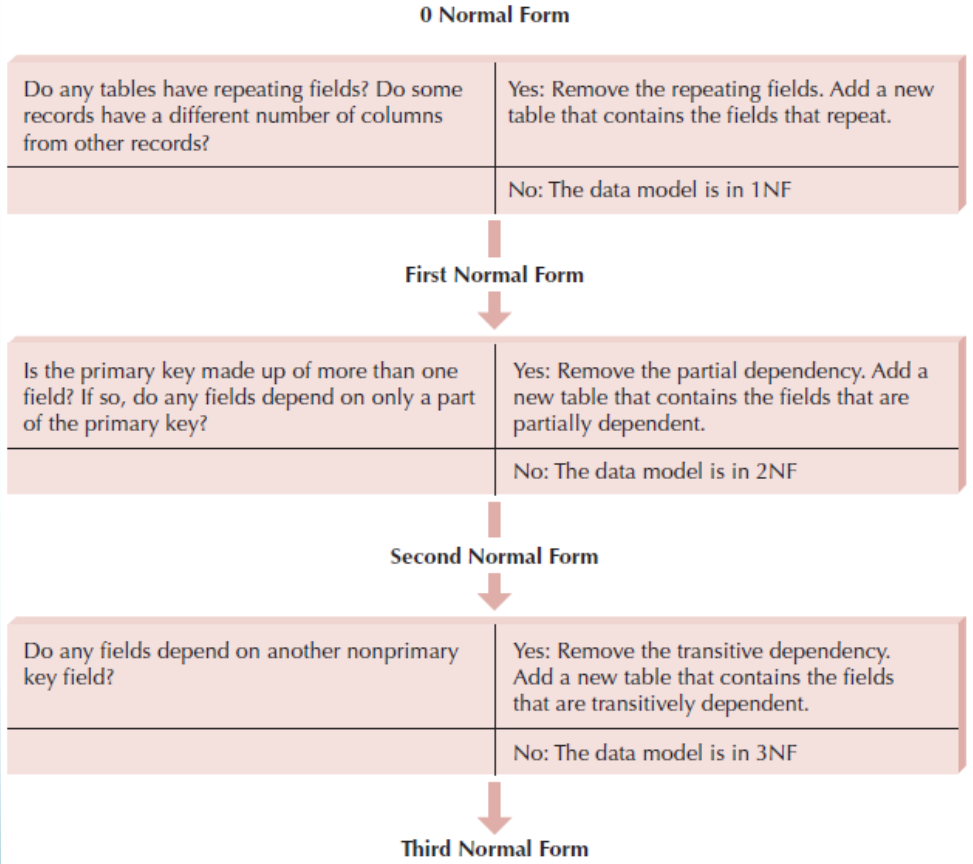
Normalization

- Store each data fact only once in the database
- Reduces data redundancies and chances of errors
- First four levels of normalization are
  - 0 Normal Form: normalization rules not applied
  - 1 Normal Form: no multi-valued attributes (each cell has only a single value)
    - Eliminate fields which are arrays or repeated.
    - They can be new tables.

Normalization

- First four levels of normalization are
  - 2 Fields depend on whole primary key
    - New order: primary key is order and product number
    - Description depends only on product
    - New table for product info
  - 3 None of the fields depend on the primary key
    - No field depends on non-primary key.
    - Tax rate depends on state.
    - Add table of states and rates.

Steps of Normalization



Optimizing Data Access Speed

- De-normalization
  - Table joins require processing
  - Add some data to a table to reduce the number of joins required (Increases data retrieval speed)
  - Creates redundancy and should be used sparingly
- Clustering
  - Place similar records close together on the disk
  - Reduces the time needed to access the disk
  - Ordering records in a table?

Optimizing Data Access Speed (cont.)

- Indexing
  - A small file with attribute values and a pointer to the record on the disk
  - Search the index file for an entry, then go to the disk to retrieve the record
  - Accessing a file in memory is much faster than searching a disk
  - Adds overhead to update; most efficient when lookup >> update.

Use indexes sparingly for transaction systems.

Use many indexes to increase response times in decision support systems.

For each table, create a unique index that is based on the primary key.

For each table, create an index that is based on the foreign key to improve the performance of joins.

Create an index for fields that are used frequently for grouping, sorting, or criteria.

Optimizing Data Access Storage

- Estimating Data Storage Size
  - Use volumetrics to estimate amount of raw data + overhead requirements
  - This helps determine the necessary hardware capacity

Field	Average Size
Order Number	8
Date	7
Cust ID	4
Last Name	13
First Name	9
State	2
Amount	4
Tax Rate	2
Record Size	49
Overhead	30%
Total Record Size	63.7
Initial Table Size	50,000
Initial Table Volume	3,185,000
Growth Rate/Month	1,000
Table Volume @ 3 years	5,478,200

Nonfunctional Requirements & Data Management Layer Design

- Operational requirements: affected by choice in hardware and operating system
- Performance requirements: speed & capacity issues
- Security requirements: access controls, encryption, and backup
- Cultural & political requirements: may affect the data storage (e.g., expected number of characters for data field, required format of a data field, local laws pertaining to data storage, etc…)

VERIFYING AND VALIDATING THE DATA MANAGEMENT LAYER

- Test the fidelity of the design before implementation
- Verifying and validating the design of the data management layer falls into three basic groups:
  1. Verifying and validating any changes made to the problem domain
  2. Dependency of the object persistence instances on the problem domain must be enforced
  3. The design of the data access and manipulation classes need to be tested

Summary

- Object Persistence Formats
- Mapping Problem-Domain Objects to Object-Persistence Formats
- Optimizing RDBMS-Based Object Storage
- Nonfunctional Requirements and Data Management Layer Design
- Designing Data Access and Manipulation Classes
- Nonfunctional Requirements & Data Management Layer Design
- Verifying and validating the data management layer