A DATABASE DESIGN CASE STUDY: THE INTERNET SHOP

Advice for software developers and horse racing enthusiasts: Avoid hacks.

—Anonymous

We now present an illustrative, 'cradle-to-grave' design example. DBDudes Inc., a well-known database consulting firm, has been called in to help Barns and Nobble (B&N) with their database design and implementation. B&N is a large bookstore specializing in books on horse racing, and they've decided to go online. DBDudes first verify that B&N is willing and able to pay their steep fees and then schedule a lunch meeting—billed to B&N, naturally—to do requirements analysis.

A.1 REQUIREMENTS ANALYSIS

The owner of B&N has thought about what he wants and offers a concise summary:

"I would like my customers to be able to browse my catalog of books and to place orders over the Internet. Currently, I take orders over the phone. I have mostly corporate customers who call me and give me the ISBN number of a book and a quantity. I then prepare a shipment that contains the books they have ordered. If I don't have enough copies in stock, I order additional copies and delay the shipment until the new copies arrive; I want to ship a customer's entire order together. My catalog includes all the books that I sell. For each book, the catalog contains its ISBN number, title, author, purchase price, sales price, and the year the book was published. Most of my customers are regulars, and I have records with their name, address, and credit card number. New customers have to call me first and establish an account before they can use my Web site.

On my new Web site, customers should first identify themselves by their unique customer identification number. Then they should be able to browse my catalog and to place orders online."

DBDudes's consultants are a little surprised by how quickly the requirements phase was completed—it usually takes them weeks of discussions (and many lunches and dinners) to get this done—but return to their offices to analyze this information.

A.2 CONCEPTUAL DESIGN

In the conceptual design step, DBDudes develop a high level description of the data in terms of the ER model. Their initial design is shown in Figure A.1. Books and customers are modeled as entities and are related through orders that customers place. Orders is a relationship set connecting the Books and Customers entity sets. For each order, the following attributes are stored: quantity, order date, and ship date. As soon as an order is shipped, the ship date is set; until then the ship date is set to *null*, indicating that this order has not been shipped yet.

DBDudes has an internal design review at this point, and several questions are raised. To protect their identities, we will refer to the design team leader as Dude 1 and the design reviewer as Dude 2:

Dude 2: What if a customer places two orders for the same book on the same day?
Dude 1: The first order is handled by creating a new Orders relationship and the second order is handled by updating the value of the quantity attribute in this relationship.
Dude 2: What if a customer places two orders for different books on the same day?
Dude 1: No problem. Each instance of the Orders relationship set relates the customer to a different book.
Dude 2: Ah, but what if a customer places two orders for the same book on different days?
Dude 1: We can use the attribute order date of the orders relationship to distinguish the two orders.
Dude 2: Oh no you can't. The attributes of Customers and Books must jointly contain a key for Orders. So this design does not allow a customer to place orders for the same book on different days.
Dude 1: Yikes, you're right. Oh well, B&N probably won't care; we'll see.

DBDudes decides to proceed with the next phase, logical database design.

A.3 LOGICAL DATABASE DESIGN

Using the standard approach discussed in Chapter 3, DBDudes maps the ER diagram shown in Figure A.1 to the relational model, generating the following tables:

```
CREATE TABLE Books ( isbn            CHAR(10),
                     title           CHAR(80),
                     author          CHAR(80),
                     qty_in_stock    INTEGER,
                     price           REAL,
                     year_published  INTEGER,
                     PRIMARY KEY (isbn))
```

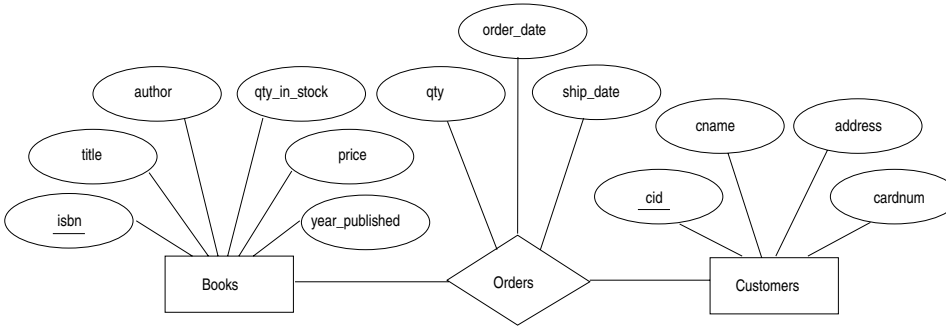

Figure A.1 ER Diagram of the Initial Design

```
CREATE TABLE Orders ( isbn          CHAR(10),
                      cid           INTEGER,
                      qty           INTEGER,
                      order_date    DATE,
                      ship_date     DATE,
                      PRIMARY KEY (isbn,cid),
                      FOREIGN KEY (isbn) REFERENCES Books,
                      FOREIGN KEY (cid) REFERENCES Customers )

CREATE TABLE Customers ( cid        INTEGER,
                         cname      CHAR(80),
                         address    CHAR(200),
                         cardnum    CHAR(16),
                         PRIMARY KEY (cid)
                         UNIQUE (cardnum))
```

The design team leader, who is still brooding over the fact that the review exposed a flaw in the design, now has an inspiration. The Orders table contains the field *order_date* and the key for the table contains only the fields *isbn* and *cid*. Because of this, a customer cannot order the same book on different days, a restriction that was not intended. Why not add the *order_date* attribute to the key for the Orders table? This would eliminate the unwanted restriction:

```
CREATE TABLE Orders (    isbn          CHAR(10),
                         ...
                         PRIMARY KEY (isbn,cid,ship_date),
                         ...)
```

The reviewer, Dude 2, is not entirely happy with this solution, which he calls a 'hack'. He points out that there is no natural ER diagram that reflects this design, and stresses

the importance of the ER diagram as a design document. Dude 1 argues that while Dude 2 has a point, it is important to present B&N with a preliminary design and get feedback; everyone agrees with this, and they go back to B&N.

The owner of B&N now brings up some additional requirements that he did not mention during the initial discussions: "Customers should be able to purchase several different books in a single order. For example, if a customer wants to purchase three copies of 'The English Teacher' and two copies of 'The Character of Physical Law,' the customer should be able to place a single order for both books."

The design team leader, Dude 1, asks how this affects the shippping policy. Does B&N still want to ship all books in an order together? The owner of B&N explains their shipping policy: "As soon as we have have enough copies of an ordered book we ship it, even if an order contains several books. So it could happen that the three copies of 'The English Teacher' are shipped today because we have five copies in stock, but that 'The Character of Physical Law' is shipped tomorrow, because we currently have only one copy in stock and another copy arrives tomorrow. In addition, my customers could place more than one order per day, and they want to be able to identify the orders they placed."

The DBDudes team thinks this over and identifies two new requirements: first, it must be possible to order several different books in a single order, and second, a customer must be able to distinguish between several orders placed the same day. To accomodate these requirements, they introduce a new attribute into the Orders table called *ordernum*, which uniquely identifies an order and therefore the customer placing the order. However, since several books could be purchased in a single order, *ordernum* and *isbn* are both needed to determine *qty* and *ship_date* in the Orders table.

Orders are assigned order numbers sequentially and orders that are placed later have higher order numbers. If several orders are placed by the same customer on a single day, these orders have different order numbers and can thus be distinguished. The SQL DDL statement to create the modified Orders table is given below:

```
CREATE TABLE Orders ( ordernum     INTEGER,
                      isbn         CHAR(10),
                      cid          INTEGER,
                      qty          INTEGER,
                      order_date   DATE,
                      ship_date    DATE,
                      PRIMARY KEY (ordernum, isbn),
                      FOREIGN KEY (isbn) REFERENCES Books
                      FOREIGN KEY (cid) REFERENCES Customers )
```

A.4 SCHEMA REFINEMENT

Next, DBDudes analyzes the set of relations for possible redundancy. The Books relation has only one key (*isbn*), and no other functional dependencies hold over the table. Thus, Books is in BCNF. The Customers relation has the key (*cid*), and since a credit card number uniquely identifies its card holder, the functional dependency *cardnum* $\rightarrow$ *cid* also holds. Since *cid* is a key, *cardnum* is also a key. No other dependencies hold, and so Customers is also in BCNF.

DBDudes has already identified the pair $\langle$*ordernum, isbn*$\rangle$ as the key for the Orders table. In addition, since each order is placed by one customer on one specific date, the following two functional dependencies hold:

ordernum $\rightarrow$ *cid*, and *ordernum* $\rightarrow$ *order_date*

The experts at DBDudes conclude that Orders is not even in 3NF. (Can you see why?) They decide to decompose Orders into the following two relations:

Orders(*ordernum*, *cid*, *order_date*, and
Orderlists(*ordernum*, *isbn*, *qty*, *ship_date*)

The resulting two relations, Orders and Orderlists, are both in BCNF, and the decomposition is lossless-join since *ordernum* is a key for (the new) Orders. The reader is invited to check that this decomposition is also dependency-preserving. For completeness, we give the SQL DDL for the Orders and Orderlists relations below:

```
CREATE TABLE Orders ( ordernum     INTEGER,
                      cid          INTEGER,
                      order_date   DATE,
                      PRIMARY KEY (ordernum),
                      FOREIGN KEY (cid) REFERENCES Customers )

CREATE TABLE Orderlists ( ordernum   INTEGER,
                          isbn       CHAR(10),
                          qty        INTEGER,
                          ship_date  DATE,
                          PRIMARY KEY (ordernum, isbn),
                          FOREIGN KEY (isbn) REFERENCES Books)
```

Figure A.2 shows an updated ER diagram that reflects the new design. Note that DBDudes could have arrived immediately at this diagram if they had made Orders an entity set instead of a relationship set right at the beginning. But at that time they did not understand the requirements completely, and it seemed natural to model Orders

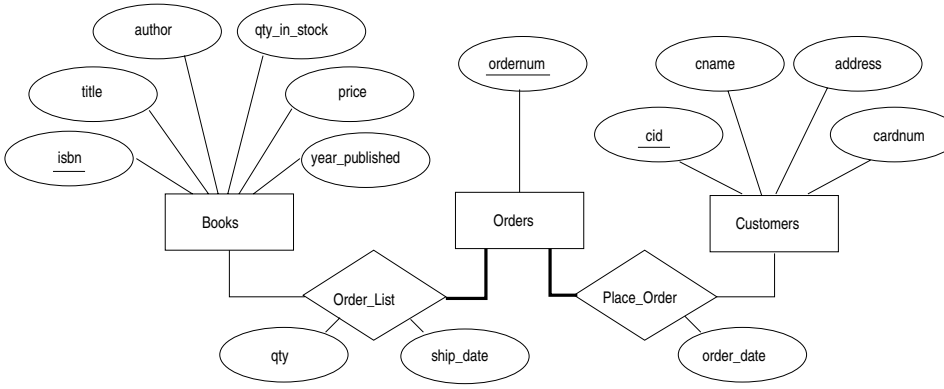

Figure A.2 ER Diagram Reflecting the Final Design

as a relationship set. This iterative refinement process is typical of real-life database design processes. As DBDudes has learned over time, it is rare to achieve an initial design that is not changed as a project progresses.

The DBDudes team celebrates the successful completion of logical database design and schema refinement by opening a bottle of champagne and charging it to B&N. After recovering from the celebration, they move on to the physical design phase.

A.5 PHYSICAL DATABASE DESIGN

Next, DBDudes considers the expected workload. The owner of the bookstore expects most of his customers to search for books by ISBN number before placing an order. Placing an order involves inserting one record into the Orders table and inserting one or more records into the Orderlists relation. If a sufficient number of books is available, a shipment is prepared and a value for the *ship_date* in the Orderlists relation is set. In addition, the available quantities of books in stocks changes all the time since orders are placed that decrease the quantity available and new books arrive from suppliers and increase the quantity available.

The DBDudes team begins by considering searches for books by ISBN. Since *isbn* is a key, an equality query on *isbn* returns at most one record. Thus, in order to speed up queries from customers who look for books with a given ISBN, DBDudes decides to build an unclustered hash index on *isbn*.

Next, they consider updates to book quantities. To update the *qty_in_stock* value for a book, we must first search for the book by ISBN; the index on *isbn* speeds this up. Since the *qty_in_stock* value for a book is updated quite frequently, DBDudes also considers partitioning the Books relation vertically into the following two relations:

BooksQty(*isbn*, *qty*), and
BookRest(*isbn*, *title*, *author*, *price*, `year_published`).

Unfortunately, this vertical partition would slow down another very popular query: Equality search on ISBN to retrieve full information about a book would require a join between BooksQty and BooksRest. So DBDudes decide not to vertically partition Books.

DBDudes thinks it is likely that customers will also want to search for books by title and by author, and decides to add unclustered hash indexes on *title* and *author*—these indexes are inexpensive to maintain because the set of books is rarely changed even though the quantity in stock for a book changes often.

Next, they consider the Customers relation. A customer is first identified by the unique customer identifaction number. Thus, the most common queries on Customers are equality queries involving the customer identification number, and DBDudes decides to build a clustered hash index on *cid* to achieve maximum speedup for this query.

Moving on to the Orders relation, they see that it is involved in two queries: insertion of new orders and retrieval of existing orders. Both queries involve the *ordernum* attribute as search key and so they decide to build an index on it. What type of index should this be—a B+ tree or a hash index? Since order numbers are assigned sequentially and thus correspond to the order date, sorting by *ordernum* effectively sorts by order date as well. Thus DBDudes decides to build a clustered B+ tree index on *ordernum*. Although the operational requirements that have been mentioned until know favor neither a B+ tree nor a hash index, B&N will probably want to monitor daily activities, and the clustered B+ tree is a better choice for such range queries. Of course, this means that retrieving all orders for a given customer could be expensive for customers with many orders, since clustering by *ordernum* precludes clustering by other attributes, such as *cid*.

The Orderlists relation mostly involves insertions, with an occasional update of a shipment date or a query to list all components of a given order. If Orderlists is kept sorted on *ordernum*, all insertions are appends at the end of the relation and thus very efficient. A clustered B+ tree index on *ordernum* maintains this sort order and also speeds up retrieval of all items for a given order. To update a shipment date, we need to search for a tuple by *ordernum* and *isbn*. The index on *ordernum* helps here as well. Although an index on $\langle$*ordernum, isbn*$\rangle$ would be better for this purpose, insertions would not be as efficient as with an index on just *ordernum*; DBDudes therefore decides to index Orderlists on just *ordernum*.

A.5.1 Tuning the Database

We digress from our discussion of the initial design to consider a problem that arises several months after the launch of the B&N site. DBDudes is called in and told that customer enquiries about pending orders are being processed very slowly. B&N has become very successful, and the Orders and Orderlists tables have grown huge.

Thinking further about the design, DBDudes realizes that there are two types of orders: *completed orders*, for which all books have already shipped, and *partially completed orders*, for which some books are yet to be shipped. Most customer requests to look up an order involve partially completed orders, which are a small fraction of all orders. DBDudes therefore decides to horizontally partition both the Orders table and the Orderlists table by *ordernum*. This results in four new relations: NewOrders, OldOrders, NewOrderlists, and OldOrderlists.

An order and its components are always in exactly one pair of relations—and we can determine which pair, old or new, by a simple check on *ordernum*—and queries involving that order can always be evaluated using only the relevant relations. Some queries are now slower, such as those asking for all of a customer's orders, since they require us to search two sets of relations. However, these queries are infrequent and their performance is acceptable.

A.6 SECURITY

Returning to our discussion of the initial design phase, recall that DBDudes completed physical database design. Next, they address security. There are three groups of users: customers, employees, and the owner of the book shop. (Of course, there is also the database administrator who has universal access to all data and who is responsible for regular operation of the database system.)

The owner of the store has full privileges on all tables. Customers can query the Books table and can place orders online, but they should not have access to other customers' records nor to other customers' orders. DBDudes restricts access in two ways. First, they design a simple Web page with several forms similar to the page shown in Figure 22.1 in Chapter 22. This allows customers to submit a small collection of valid requests without giving them the ability to directly access the underlying DBMS through an SQL interface. Second, they use the security features of the DBMS to limit access to sensitive data.

The Web page allows customers to query the Books relation by ISBN number, name of the author, and title of a book. The Web page also has two buttons. The first button retrieves a list of all of the customer's orders that are not completely fulfilled yet. The second button will display a list of all completed orders for that customer. Note that

customers cannot specify actual SQL queries through the Web; they can only fill in some parameters in a form to instantiate an automatically generated SQL query. All queries that are generated through form input have a `WHERE` clause that includes the *cid* attribute value of the current customer, and evaluation of the queries generated by the two buttons requires knowledge of the customer identification number. Since all users have to log on to the Web site before browsing the catalog, the business logic (discussed in Section A.7) must maintain state information about a customer (i.e., the customer identification number) during the customer's visit to the Web site.

The second step is to configure the database to limit access according to each user group's need to know. DBDudes creates a special `customer` account that has the following privileges:

```
SELECT ON Books, NewOrders, OldOrders, NewOrderlists, OldOrderlists
INSERT ON NewOrders, OldOrders, NewOrderlists, OldOrderlists
```

Employees should be able to add new books to the catalog, update the quantity of a book in stock, revise customer orders if necessary, and update all customer information *except the credit card information*. In fact, employees should not even be able to see a customer's credit card number. Thus, DBDudes creates the following view:

```
CREATE VIEW CustomerInfo (cid,cname,address)
       AS SELECT C.cid, C.cname, C.address
          FROM   Customers C
```

They give the `employee` account the following privileges:

```
SELECT ON CustomerInfo, Books,
          NewOrders, OldOrders, NewOrderlists, OldOrderlists
INSERT ON CustomerInfo, Books,
          NewOrders, OldOrders, NewOrderlists, OldOrderlists
UPDATE ON CustomerInfo, Books,
          NewOrders, OldOrders, NewOrderlists, OldOrderlists
DELETE ON Books, NewOrders, OldOrders, NewOrderlists, OldOrderlists
```

In addition, there are security issues when the user first logs on to the Web site using the customer identification number. Sending the number unencrypted over the Internet is a security hazard, and a secure protocol such as the SSL should be used.

There are companies such as CyberCash and DigiCash that offer electronic commerce payment solutions, even including 'electronic' cash. Discussion of how to incorporate such techniques into the Website are outside the scope of this book.

A.7 APPLICATION LAYERS

DBDudes now moves on to the implementation of the application layer and considers alternatives for connecting the DBMS to the World-Wide Web (see Chapter 22).

DBDudes note the need for session management. For example, users who log in to the site, browse the catalog, and then select books to buy do not want to re-enter their customer identification number. Session management has to extend to the whole process of selecting books, adding them to a shopping cart, possibly removing books from the cart, and then checking out and paying for the books.

DBDudes then considers whether Web pages for books should be static or dynamic. If there is a static Web page for each book, then we need an extra database field in the Books relation that points to the location of the file. Even though this enables special page designs for different books, it is a very labor intensive solution. DBDudes convinces B&N to dynamically assemble the Web page for a book from a standard template instantiated with information about the book in the Books relation.

This leaves DBDudes with one final decision, namely how to connect applications to the DBMS. They consider the two main alternatives that we presented in Section 22.2: CGI scripts versus using an application server infrastructure. If they use CGI scripts, they would have to encode session management logic—not an easy task. If they use an application server, they can make use of all the functionality that the application server provides. Thus, they recommend that B&N implement server-side processing using an application server.

B&N, however, refuses to pay for an application server and decides that for their purposes CGI scripts are fine. DBDudes accepts B&N's decision and proceeds to build the following pieces:

- The top level HTML pages that allow users to navigate the site, and various forms that allow users to search the catalog by ISBN, author name, or author title. An example page containing a search form is shown in Figure 22.1 in Chapter 22. In addition to the input forms, DBDudes must develop appropriate presentations for the results.
- The logic to track a customer session. Relevant information must be stored either in a server-side data structure or be cached in hte customer's browser using a mechanism like **cookies**. Cookies are pieces of information that a Web server can store in a user's Web browser. Whenever the user generates a request, the browser passes along the stored information, thereby enabling the Web server to 'remember' what the user did earlier.
- The scripts that process the user requests. For example, a customer can use a form called 'Search books by title' to type in a title and search for books with that

title. The CGI interface communicates with a script that processes the request. An example of such a script written in Perl using DBI for data access is shown in Figure 22.4 in Chapter 22.

For completeness, we remark that if B&N had agreed to use an application server, DBDudes would have had the following tasks:

- As in the CGI-based architecture, they would have to design top level pages that allow customers to navigate the Web site as well as various search forms and result presentations.
- Assuming that DBDudes select a Java-based application server, they have to write Java Servlets to process form-generated requests. Potentially, they could reuse existing (possibly commercially available) JavaBeans. They can use JDBC as a database interface; examples of JDBC code can be found in Section 5.10. Instead of programming Servlets, they could resort to Java Server Pages and annotate pages with special JSP markup tags. An example of a Web page that includes JSP commands is shown in Section 22.2.1.
- If DBDudes select an application server that uses proprietary markup tags, they have to develop Web pages by using such tags. An example using Cold Fusion markup tags can be found in Section 22.2.1.

Our discussion thus far only covers the 'client-interface', the part of the Web site that is exposed to B&N's customers. DBDudes also need to add applications that allow the employees and the shop owner to query and access the database and to generate summary reports of business activities.

This completes our discussion of Barns and Nobble. While this study only describes a small part of a real problem, we saw that a design even at this scale involved non-trivial tradeoffs. We would like to emphasize again that database design is an iterative process and that therefore it is very important not to lock oneself down early on in a fixed model that is too inflexible to accomodate a changing environment. Welcome to the exciting world of database management!