Brian Bauman
CSC 452

Auction DB Documentation Outline

Introduction

For my final project, I decided to create an auction house **backed by an Oracle SQL DB**. This was an idea suggested in chapter 10 of the Oracle 10g Programming: A Primer book that we've been using for this class. The basics of this auction house (to which I'll now refer as Auction DB) are as follows. All visitors to the Auction DB **must either log in or create an account** to be able to view the items for auction. Once they've logged in, they can perform a few different tasks, including adding items for auction, **placing bids on items** that are already up for auction, and **searching Auction DB for relevant items** that are still for auction or have already sold. All of these features are available through a Java console front-end, which provides simple, clean navigation through the menus. In this report, I'll further detail the above boldfaced features.

Backed by an Oracle SQL DB

As mentioned, the entire application relies on an Oracle SQL DB for almost all of its functionality, from logging in to placing bids to searching for items. Because this SQL DB is hosted on CDM's servers and not locally on the client machine, it preserves information added and updated by users across sessions, devices, and time, just like an actual auction house. An item added by one user will appear in the search results of another user, whose bids will in turn be visible to the original user. The transactional integrity of a central SQL DB provides this consistency and durability. However, there are a few failings of this current implementation. First, there has been little to no effort invested in providing solid concurrency for users. If two users are interacting with the database at the same time, it may enter an inconsistent state. In high-frequency applications, such as investment trading, this is a critical design flaw. But, for the purposes of an auction house with few users (cough… the professor and myself…), it does the trick. Also, the application has little to no error handling for a scenario where the SQL DB itself is unavailable.

The database architecture is as follows:

Tables (PRIMARY KEYS, foreign keys):

- MEMBERS (USERID, password, name, address, phone, email)
- ITEMS (INO, category, title, description, sellerID, quantity, startPrice, bidIncrement, lastBidReceived, closeTime)
- BID (BUERYID, INO, PRICE, QTYWANTED, BIDTIME)

Packages:

- auction_db (contains all procedures needed for interacting with these tables)

User login and account creation

One nice feature of backing the application with a database is the ability to record and make use of user profiles. In almost all modern applications, users are required to identify themselves and authenticate before making changes to the application's underlying data and

this application is no different. Before being granted the ability to place bids or search for items, users must register an account and provide a username, password, and other personal information. Then, once they successfully log in, all of their actions (placing bids, adding items, etc.) are tied to their account. In further implementations, this information could also be used to record metadata such as a history or searches for a user, records of last login and failed password attempts, and activity data related to time spent browsing different areas of the application. One implementation of this metadata could involve the addition of triggers on the database tables!

One important failing to point out is the lack of security in this application's login process. All of the usernames and passwords are stored in plain text and the application itself is completely vulnerable to common security vulnerabilities like SQL injection. Were this application to be developed further, fixing these deficiencies would be a high priority.

Searching for items

One fun feature of this application is the ability to search for items. This was probably my favorite part to code because it could really be a project of its own — building a lightweight search engine. The user is prompted with a text entry screen and asked to enter a search argument, which is compared to the titles, categories, and descriptions of the items for auction. The first iteration was pretty clunky — the user was prompted to use SQL LIKE syntax and their argument was passed directly to a SQL query (note: See SQL injection worries above! Same story here.). However, this led to some weird outcomes, such as the implicit requirement to start and end the search argument with the "%" symbol, or the implicit case-sensitivity that would not return an item titled "Starry Night" if the user searched "%starry night%". So, to improve this, I added a "%" character to the front and end of the search argument and lowercased the entire string. Then, in the SQL SELECT statements that queried the ITEMS table, I also made sure to compare this user-entered string to the lowercased values in the title, description, and category columns. The result is a search that doesn't require any knowledge of SQL LIKE syntax (through throwing in a "%" or "_" in your query would work!) and resembles something that users are familiar with. Lastly, when the search does find results, it takes advantage of the data on the ITEMS table to label each result with a status (ITEM SOLD, OPEN AUCTION, etc.)

Bidding for items

As an auction house, Auction DB of course allows users to place bids. And, as with other features of this application, one of the most important prerequisites to an action is to verify that action prior to taking it. This ensures that the DB integrity is preserved. For example, you should not be able to place a bid on an item whose auction has ended! The process of placing a bid involved a few of these checks, passing one of 5 return codes to the calling Java application to inform it whether the prepared bid is valid, or, if invalid, why. The checks it performs are: does the item exist, is the item's auction still live, is the prepared bid high enough (must exceed the current bid plus the specified bid increment), and is the quantity requested equal to or less than the quantity available. Depending on the results of these questions (which are all answered and gathered in the auction_db.verify_bid procedure), the application will inform the user whether their bid succeeded or what about it was invalid. This allows the user a clear view into the process of placing a bid and reduces errors and confusion.

After a user places a bid, they can see the current price of their bid on the item in the search results page.

Conclusion

In conclusion, this application functions as a lightweight front-end to a simple but durable auction database. If desired, this could be built out without much trouble into a web front-end and scaled to many users at once. There are a few design flaws (see my notes on security and concurrency above), but the meat is there. I thought this final project was a great way to put everything we learned in class into action. I have only a passing familiarity with Java and Oracle PL/SQL, so the chance to build a "real" application from scratch was really exciting and solidified a lot of the class's concepts. I almost think the students that instead spent their time studying for a final exam should be jealous!

If I could add or change one thing, I think I would have spent more time adding triggers — this would allow for easy recording of user activity in log tables, which would be a neat feature.

References

1. Oracle 10g Programming: A Primer, Sunderraman
   - Introduced the project idea, suggested table structures, etc.
   - Very helpful references throughout, pertaining to procedures, in/out variables, and packages

2. https://stackoverflow.com (many, many times)
   - Helped solve many one-off problems (largely with the Java implementation)
     - Improperly referenced Oracle Types
     - Using a REF_SYSCURSOR to retrieve a cursor from a procedure
     - Doing Date arithmetic
     - Setting the appropriate classpath so that the Oracle driver was pulled in