Technical Solution/Design Document:

Unique Bid Blind Online Auction System

Version: 1.0
Date: March 27, 2025
Author: Jared Scott

Table of Contents:

1. Overview and Stakeholders

This document presents the technical design for a Unique Bid Blind Online Auction system. The system aims to provide a platform for conducting auctions where participants submit bids within a specific timeframe. The winner will be the individual who placed the highest bid that is uniquely offered. In the event of a tie for the highest bid, the current auction will be terminated, and a new auction will be initiated with the minimum acceptable bid increased by $1. The design also considers a future enhancement to incorporate a timestamp-based tie-breaking mechanism, awarding the bidder who submitted the highest tied bid earliest. The primary stakeholder for this document and the described system is Jared Scott, who is responsible for the design and potential implementation of this auction platform.

1.1. Context and Goals

The concept of a Unique Bid Blind Online Auction offers a distinct and engaging auction format. Unlike traditional auctions where the highest bidder wins, this format introduces an element of strategic bidding, as participants aim to place the highest bid that no other participant has offered. The development of this system is driven by an interest in exploring and implementing this specific auction mechanism. The primary functional goals of this system are to enable registered bidders to submit integer-valued bids in Canadian dollars for auctioned items within a defined bidding period, enforce a minimum bid requirement for all submissions, accurately identify the highest bid that is uniquely placed at the conclusion of the auction, implement a clear process for handling ties at the highest bid by restarting the auction with an incremented minimum bid, and (as a future goal) provide an alternative mechanism for resolving ties based on the timestamp of bid submission. The key technical goals include designing a system that can efficiently store and process a potentially large number of bids, utilizing data structures and algorithms that ensure acceptable performance and scalability, and creating a modular and well-documented design that facilitates future development and maintenance.

2. Requirements

- Efficient Bid Storage: The system must store bids effectively, allowing for quick retrieval and processing.
- Handling Large Bid Volumes: The solution should be scalable to handle thousands or even millions of bids.
- Unique Highest Bid Identification: The system must be able to efficiently identify the highest bid placed by only one bidder.
- Tie Detection and Restart Logic: The system needs to detect ties for the highest bid and trigger a restart of the auction with an incremented minimum bid.
- placeBid(Bid) Functionality: The system will receive bids one at a time through a placeBid(Bid) function, which will record the bid information.
- Tie-Breaking by Timestamp: If the tie-breaking rule changes, the system should award the win to the bidder who placed the highest tied bid earliest.

3. Proposed Solution

To handle all these bids coming in and figure out the winner efficiently, there are a couple key data structures. The approach revolves around keeping the bids organized in a way that makes it easy to find the highest and identify unique bids quickly.

3.1. Data Structures

First up, proposing an ordered map (std::map) in Python, a dictionary that we can treat as ordered or sort when needed. This will be called bidsByAmount.

- **Key:** The actual bid amount, as an integer in Canadian dollars.
- **Value:** For each bid amount, we'll store a set of the integer IDs of all the bidders who placed a bid at that exact amount. Using a set here is smart because it automatically takes care of duplicates. So, even if someone submits the same $25 bid five times, their ID will only appear once in the set for the $25 key. Plus, the ordered nature of the map will be helpful later when we need to look at the bids from highest to lowest.

Next, to keep track of what each bidder has bid, (bidsByBidder) is created to keep track of what they chose.

- **Key:** The unique ID of the bidder (an integer).
- **Value:** The value will be a set to all the *unique* bid amounts that this bidder in particular has submitted. While there might not be a direct use, this is for finding the unique winner in the first scenario, it could be useful for other features or analysis down the line.

3.2. placeBid(Bid) Algorithm

When a new bid comes in through the placeBid(Bid) function (which will contain the bidderID and the bidAmount), The outcome will be:

- **Minimum Bid Check***:* First the bids need to make sure the bidAmount is actually higher than the current minimum bid. If not, just discard it. There is no point in keeping bids that don't meet the basic requirement.
- **Recording in bidsByAmount:**
  - View bidsByAmount map and find the entry for the bidAmount of this new bid.
  - Then simply add the bidderID to the set of bidders associated with that bidAmount. It is a set, if the bidder has already bid that amount, their ID won't be added again.
- **Recording in bidsByBidder:**
  - Similarly, go to the bidsByBidder map and find the entry for this bidderID.
  - Then add the bidAmount of this new bid to the set of bid amounts for these bidders. Now the set will handle any duplicate bid amounts from the same bidder.

3.3. Algorithm for Determining the Winner

Once the bidding window closes, there needs to be a way to figure out who the winner is. Here's the approach:

- **Iterating Through Bids from Highest to Lowest:** The bidsByAmount map is ordered by the bid amount, this easily allows one to look at the bids starting from the highest to lowest. This can be done by iterating through the map in reverse order.
- **Checking the Number of Bidders at Each Amount:** For each bid amount encounter:
  - Once the set of bidder IDs associated with that amount has been discovered.
  - **If there's only one bidder in the set:** That means this is the highest bid that was placed by only one person. That bidder is our winner, and can return their ID.
  - **If there is more than one bidder in the set:** This means, there is a tie at the current highest bid. According to the rules, the auction needs to be restarted. This should set off a signal that activates a restart and also provides the new minimum bid, which would be the current tied bid amount plus $1.
- **What if No Unique Bid is Found?** It's a bit of an edge case, but if we go through all the bid amounts and don't find any with just a single bidder (maybe everyone tied at the initial minimum), there will be a need to handle that. Perhaps declare no winner for those particular rounds or have a specific rule for that scenario.

3.4. Handling Large Numbers of Bids

The selection of std::map and std::set (or their equivalents) is expected to enable efficient handling of a large number of bids. The primary justification for this lies in the logarithmic time complexity (O(log N), where N represents the number of unique bid amounts or bidders) associated with their core operations, such as insertion, deletion, and lookup. This characteristic ensures that even with bid volumes reaching into the thousands or millions, the performance of these operations remains relatively fast. Furthermore, the utilization of sets prevents the redundant storage of identical bids submitted by the same user, thereby optimizing memory usage.

4. Changes for Tie-Breaking by Timestamp

If the tie-breaking rule changes to awarding the win to the bidder who placed the highest tied bid first, the bidsByAmount data structure needs to be modified to store the timestamp of each bid.

4.1. Modified Data Structure(bidsByAmount):

- **Key:** The bid amount (integer).
- **Value:** A list (or vector) of pairs. Each pair contains:
  - The timestamp of the bid (time.time() in Python).
  - The bidderID (integer).

4.2. Modified placeBid(Bid) Algorithm:

- **Minimum Bid Check:** Same as before.
- **Record in bidsByAmount:**
  - Get the current timestamp when the bid is received.
  - Access the list associated with the bidAmount in the bidsByAmount map.
  - Append a new pair (timestamp, bidderID) to this list.
- **Record in bidsByBidder:** This remains the same.

4.3. Modified Algorithm for Determining the Winner

- **Iterate through bidsByAmount in Descending Order of Bid Amount:** Same as before.
- **Check Bid Count for Each Amount:** For each bid amount:
  - Retrieve the list of (timestamp, bidderID) pairs associated with that amount.
  - **If the size of the list is 1:** This is the highest unique bid. Return the bidderID from the single pair.
  - **If the size of the list is greater than 1:** This is a tie. Sort the list of pairs based on the timestamp in ascending order. The bidder who placed the bid earliest will be the first element in the sorted list. Return the bidderID from the first element of the sorted list.
- **No Unique Bid:** Handle as before.

4.4. Impact on Efficiency

Storing timestamps will increase the memory footprint slightly. In the case of a tie, sorting the list of bids by timestamp will take O(M log M) time, where M is the number of tied bids. However, since ties are expected to be less frequent than unique bids, this added complexity should not significantly impact the overall performance, especially with efficient sorting algorithms.

5. Conclusion

The proposed solution utilizing ordered maps and sets provides an efficient way to store and process bids for the Unique Bid Blind On-line Auction. The logarithmic time complexity of the core operations ensures scalability for a large number of bids. The modification to include timestamps allows for handling the alternative tie-breaking rule based on the bid submission time.

GitHub Repository

Winter2024-NSCC-ECampus/final-project-online-auction-solution-Autoscotty43: final-project-online-auction-solution-Autoscotty43 created by GitHub Classroom