Exchange Market for Complex Goods: Theory and Experiments

Eugene Fink
eugene@csee.usf.edu

Josh Johnson
jojohnso@csee.usf.edu

Jenny Hu
jyhu@csee.usf.edu

Computer Science and Engineering
University of South Florida
Tampa, Florida 33620

Abstract

The modern economy includes a variety of markets, and the Internet has opened opportunities for efficient on-line trading. Researchers have developed algorithms for various auctions, which have become a popular means of on-line sales. They have also designed algorithms for exchange markets, which support fast-paced trading of standardized goods. On the other hand, they have done little work on exchanges for complex nonstandard goods, such as used cars.

We propose a formal model for trading complex goods, and present an exchange system that allows traders to describe desirable purchases and sales by multiple attributes; for example, a car buyer can specify a model, options, color, and other properties of a desirable vehicle. Furthermore, a trader can enter complex constraints on the acceptable items; for instance, a buyer can specify a set of desirable vehicles and their features. The system supports markets with up to 300,000 orders, and generates hundreds of trades per second.

Keywords: E-commerce, electronic trading, multi-attribute goods.

1 Introduction

The growth of the Internet has led to the development of on-line markets, which include bulletin boards, auctions, and exchanges. Electronic bulletin boards vary from sale catalogs to newsgroup postings, which help buyers and sellers find each other; however, they often require customers to invest significant time into reading multiple ads, and many buyers prefer on-line auctions, such as eBay (www.ebay.com). Auctions have their own problems, including high computational costs, lack of liquidity, and asymmetry between buyers and sellers. Exchange markets support fast-paced trading and ensure symmetry between buyers and sellers, but they require rigid standardization of tradable items. For example, the New York Stock Exchange allows trading of about 3,000 stocks, and a buyer or seller has to indicate a specific stock.

For most goods, the description of a desirable trade is more complex; for instance, a car buyer needs to specify a model, options, color, and other features of a desirable vehicle. She may also want to include preferences; for example, she may indicate that a red car is preferable to a white car. An exchange for nonstandard goods should allow complex constraints in specifications of purchases and sales, support fast-paced trading for markets with millions of participants, and include optimization techniques that maximize the traders' satisfaction.

Economists and computer scientists have long realized the importance of auctions and exchanges, and studied a variety of trading models. Their work has led to successful Internet auctions, such as eBay (www.ebay.com) and Yahoo Auctions (auctions.yahoo.com), as well as standardized on-line exchanges, such as Island (www.island.com) and NexTrade (www.nextrade.org).

Recently, researchers have developed efficient systems for combinatorial auctions, which allow buying and selling sets of goods rather than individual items. Rothkopf *et al.* [1998] gave a detailed analysis of these auctions and described semantics of combinatorial bids that allowed fast matching. Nisan discussed alternative semantics, formalized the problem of identifying optimal and near-optimal matches, and proposed a linear-programming solution [Nisan, 2000; Lavi and Nisan, 2000]. Gonen and Lehmann [2000, 2001] studied branch-and-bound heuristics for processing combinatorial bids and integrated them with linear programming. Sandholm and his colleagues developed combinatorial auctions that supported markets with several thousand bids [Sandholm, 2000; Sandholm and Suri, 2000; Sandholm *et al.*, 2001a]. They also described special cases of bid processing that allowed polynomial solutions, proved the NP-completeness of more general cases, and tested various heuristics for NP-complete cases [Sandholm and Suri, 2001; Sandholm *et al.*, 2001b]. Although the developed systems can efficiently process several thousand bids, their running time is superlinear in the number of bids, and they do not scale to larger markets.

Several researchers have studied techniques for specifying the dependency of item price on the number and quality of items. Che [1993] analyzed auctions that allowed negotiating not only the price but also the quality of goods. A bid in these auctions was a function that specified a desired trade-off between price and quality. Cripps and Ireland [1994] considered a similar setting and suggested several strategies for bidding on price and quality. Bichler discussed a market that allowed negotiations on any attributes of goods [Bichler *et al.*, 1999; Bichler, 2000]. Sandholm and Suri [2001] studied combinatorial auctions that allowed bulk discounts; that is, they enabled a bidder to specify a dependency between item price and transaction size. Lehmann *et al.* [2001] also considered the dependency of price on transaction size, showed that the problem of finding the best matches was NP-hard, and developed a greedy

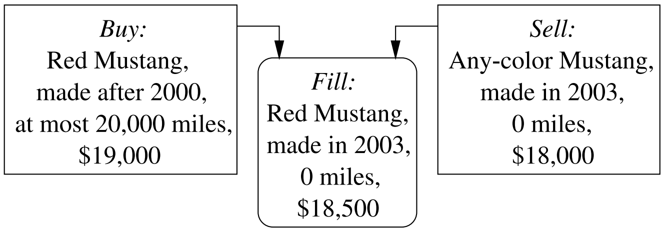


Figure 1: Matching orders and the resulting trade. When the exchange system finds a match between two orders, it generates a fill, which is a trade that satisfies both parties.

approximation algorithm. This initial work leaves many open problems, including the use of complex constraints with general preference functions, symmetric treatment of buyers and sellers, and design of efficient matching algorithms for advanced semantics.

Some auction researchers have studied exchange markets; they have viewed exchanges as a variety of auction markets, called continuous double auctions. Wurman *et al.* [1998a] proposed a theory of exchange markets and implemented a general-purpose system for auctions and exchanges. Sandholm and Suri [2000] developed an exchange for combinatorial purchases and sales, which supported markets with one thousand participants. Kalagnanam *et al.* [2000] investigated techniques for identifying matches between purchases and sales with complex constraints, which scaled to several thousand participants. A related open problem is to develop a scalable exchange system for large combinatorial markets.

A recent project at the University of South Florida has been aimed at building an automated exchange for complex goods [Johnson, 2001; Hu, 2002; Gong, 2002; Hershberger, 2003]. We have defined related trading semantics and developed an exchange system that supports large-scale markets. We give a formal model of a market for complex goods (Section 2), explain representation of purchases and sales (Section 3), describe data structures for identifying matches between buyers and sellers (Section 4), and show how the system's performance depends on the market size (Section 5).

2 General exchange model

We begin with an example of complex goods, and then define the notions of buy orders, sell orders, and matches between them.

Example. We consider an exchange for trading new and used cars; to simplify this example, we assume that a trader can describe a car by four attributes: model, color, year, and mileage. A prospective buyer can place a *buy order,* which specifies a desired car and a maximal acceptable price; for instance, she may indicate that she is looking for a red Mustang, made after 2000, with less than 20,000 miles, and she is willing to pay $19,000. Similarly, a seller can place a *sell order;* for example, a dealer may offer a brand-new Mustang of any color for $18,000. An exchange system must identify matches between buy and sell orders, and generate *fills,* that is, transactions that satisfy both buyers and sellers (Figure 1). If the system finds several matches for an order, it should choose the match with the best price; for instance, the buy order in Figure 2(a) should trade with the cheaper of the two sell orders. If two matching orders have the same price, the system should give preference to the earlier order (Figure 2b).

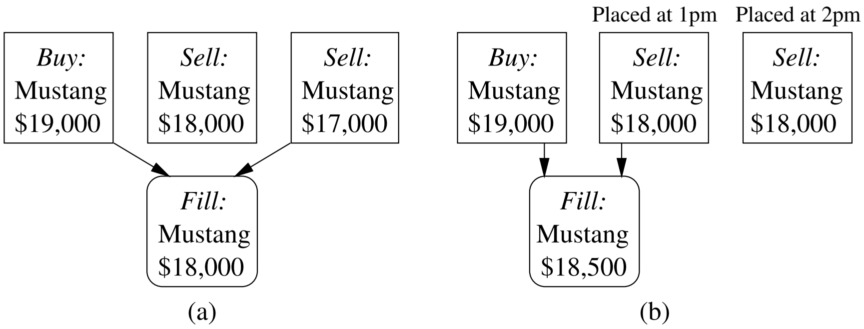


Figure 2: Selection among alternative matches. When the exchange system finds several matches for an order, it should choose the match with the best price (a). If two matches have the same price, it should prefer the earlier order (b).

Orders. When a trader makes a purchase or sale, she has to specify a set of acceptable items, denoted I, which stands for *item set.* For example, if a customer needs a sports car, she may be willing to buy any of several models, such as a Mustang or Camaro. In addition, a trader should specify a limit on the acceptable price. For instance, a car buyer may be willing to pay \$18,500 for a Mustang, but only \$17,500 for a Camaro; furthermore, she may offer an extra \$500 if a car is red, and subtract \$1 for every ten miles on its odometer. Formally, a price limit is a real-valued function on the set I; for each item $i \in I$, it gives a certain limit $Price(i)$. For a buyer, $Price(i)$ is the maximal acceptable price; for a seller, it is the minimal acceptable price. We use the term *buy order* to refer to a buyer's set of constraints; similarly, a *sell order* is a seller's set of constraints. When a buyer or seller announces her desire to trade, we say that she has *placed* an order. A buy order $(I_b, Price_b)$ *matches* a sell order $(I_s, Price_s)$ if the buyer's constraints are consistent with the seller's; that is, there exists an item $i \in I_b \cap I_s$ such that $Price_s(i) \leq Price_b(i)$.

Quality functions. A trader may have preferences among acceptable transactions, which depend on an item i and its price p; for instance, a car buyer may prefer a Mustang for \$18,000 to a Camaro for \$17,000. We represent preferences by a real-valued function $Qual(i, p)$ that assigns a numeric quality to each pair of an item and price. Larger values correspond to better transactions; that is, if $Qual(i_1, p_1) > Qual(i_2, p_2)$, then trading i_1 at price p_1 is better than i_2 at p_2. Each trader can use her own quality functions and specify different functions for different orders. Note that buyers look for low prices, whereas sellers prefer to get as much money as possible, which means that quality functions must be monotonic on price:

- *Buy monotonicity:* If $Qual_b$ is a quality function for a buy order, and $p_1 \leq p_2$, then, for every item i, we have $Qual_b(i, p_1) \geq Qual_b(i, p_2)$.
- *Sell monotonicity:* If $Qual_s$ is a quality function for a sell order, and $p_1 \leq p_2$, then, for every item i, we have $Qual_s(i, p_1) \leq Qual_s(i, p_2)$.

We do not require a trader to specify a quality function for each order; by default, quality is defined through price. This default function is the difference between the price limit and actual price, divided by the price limit:

- *For buy orders:* $Qual_b(i, p) = (Price(i) - p)/Price(i)$.
- *For sell orders:* $Qual_s(i, p) = (p - Price(i))/Price(i)$.

Order sizes. If a trader wants to buy or sell several identical items, she can include their number in the order specification, which is called an *order size.* We assume that a size is a positive integer, thus enforcing discretization of continuous goods, such as orange juice. The trader can specify not only an overall order size but also a minimal acceptable size of a transaction. For instance, suppose that a Ford wholesale agent is selling one hundred cars, and she works only with dealerships that are buying at least twenty cars. Then, she may specify that the overall size of her order is one hundred, and the minimal size is twenty. In addition, the trader can indicate that a transaction size must be divisible by a certain number, called a *size step;* for instance, a wholesale agent may sell cars in blocks of ten. To summarize, an order includes six elements:

- Item set, I.
- Price function, $Price \colon I \to \mathbf{R}$.
- Quality function, $Qual \colon I \times \mathbf{R} \to \mathbf{R}$.
- Overall order size, *Max*.
- Minimal acceptable size, *Min*.
- Size step, *Step*.

Fills. When a buy order matches a sell order, the corresponding parties can complete a trade; we use the term *fill* to refer to the traded items and their price (Figure 1). We define a fill by a specific item i, its price p, and the number of traded items, denoted *size*. If $(I_b, Price_b, Max_b, Min_b, Step_b)$ is a buy order, and $(I_s, Price_s, Max_s, Min_s, Step_s)$ is a matching sell order, then a fill must satisfy the following conditions:

- $i \in I_b \cap I_s$.
- $Price_s(i) \leq p \leq Price_b(i)$.
- $\max(Min_b, Min_s) \leq size \leq \min(Max_b, Max_s)$.
- $size$ is divisible by $Step_b$ and $Step_s$.

If both the buyer and seller specify a set of items, the resulting fill can contain any item $i \in I_b \cap I_s$; similarly, we may have some freedom in selecting the price and size of the fill.

- *Item choice:* If $I_b \cap I_s$ includes several items, we choose an item that maximizes the *surplus,* that is, the difference between $Price_b(i)$ and $Price_s(i)$.

- *Price choice:* The default strategy is to split the price difference between the buyer and seller (Figure 3). Another option is to favor either the buyer or seller; that is, we may always use $p = Price_b(i)$ or, alternatively, we may always use $p = Price_s(i)$.

- *Size choice:* We assume that the buyer and seller are interested in trading at the maximal size, or as close to the maximal size as possible; in Figure 4, we give an algorithm that finds the fill size for two matching orders.

After getting a fill, the trader may keep the initial order, reduce its size, or remove the order; the default option is the size reduction. If the reduced size is zero, the system removes the order from the market. If the size remains positive but drops below the minimal acceptable size *Min*, the order is also removed.

FILL-PRICE($Price_b, Price_s, i$)
The algorithm inputs the price functions of a buy and
sell order, and an item i that matches both orders.
It returns the fill price for these orders.

If $Price_b(i) \geq Price_s(i)$, then return $(Price_b(i) + Price_s(i))/2$
Else, return NONE (no acceptable price)

Figure 3: Computing the fill price for two matching orders.

FILL-SIZE($Max_b, Min_b, Step_b, Max_s, Min_s, Step_s$)
The algorithm inputs the size specification of a buy order, Max_b, Min_b, and $Step_b$,
along with the size specification of a matching sell order, Max_s, Min_s, and $Step_s$.
It returns the fill size for these orders.

Let *step* be the least common multiple of $Step_b$ and $Step_s$
$size := \lfloor \min(Max_b, Max_s)/step \rfloor \cdot step$
If $size \geq \max(Min_b, Min_s)$, then return *size*
Else, return NONE (no acceptable size)

Figure 4: Computing the fill size for two matching orders.

3 Order representation

We next describe the representation of orders in the developed exchange system.

Market attributes. A specific market includes a certain set of items that can be bought and sold, defined by a list of attributes. As a simplified example, we describe a car by four attributes: model, color, year, and mileage. An attribute may be a set of explicitly listed values, such as the car model; an interval of integers, such as the year; or an interval of real values, such as the mileage. This representation is based on the simplifying assumption that all items have the same attributes. Some markets do not satisfy this assumption; for example, if we trade cars and trucks on the same market, we need two lists of attributes.

Cartesian products. When a trader places an order, she has to specify a set of acceptable values for each attribute, which is called an *attribute set.* She specifies some set I_1 of values for the first attribute, some set I_2 for the second attribute, and so on. The resulting set I of acceptable items is the Cartesian product $I = I_1 \times I_2 \times ... \times I_n$. For example, suppose that a car buyer is looking for a Mustang or Camaro, the acceptable colors are red and white, the car should be made after 2000, and it should have at most 20,000 miles; then, the item set is $I = \{\texttt{Mustang}, \texttt{Camaro}\} \times \{\texttt{red}, \texttt{white}\} \times [2001..2003] \times [0..20{,}000]$.

Attribute sets. A trader can use specific values or ranges for each attribute; note that ranges work only for numeric attributes, such as year and mileage. A market specification may include certain *standard sets* of values, such as "all sports cars" and "all American cars," and a trader can use them in her orders. We must include all standard sets in the market description, and traders cannot define new sets. We specify a standard set by a list of values or numeric ranges; for example, a set of American cars is a list of specific models: $\{\texttt{Camaro}, \texttt{Mustang}, ...\}$. A trader can also use intersections and unions in the specification of attribute sets; the system allows an arbitrary nesting of intersections and unions. For instance, suppose that a buyer is interested in Mustangs, Camaros, and Japanese sports cars. Suppose further that we have defined a standard set of all Japanese cars, and another standard set of all sports cars. Then, the buyer can represent the desired set of models as $\{\texttt{Mustang}, \texttt{Camaro}\} \cup (\texttt{Japanese-Cars} \cap \texttt{Sports-Cars})$. To summarize, an attribute set is one of the following:

- Specific value, such as Mustang or 2001.
- Range of values, such as [2001..2003].
- Standard set of values, such as all Japanese cars.
- Intersection of several attribute sets.
- Union of several attribute sets.

Unions and filters. A trader can define an item set I as the union of several Cartesian products. For example, if a buyer is looking for either a used red Mustang or a new red Camaro, she can specify the set $I = \{\texttt{Mustang}\} \times \{\texttt{red}\} \times [2001..2003] \times [0..20{,}000] \cup \{\texttt{Camaro}\} \times \{\texttt{red}\} \times \{2003\} \times [0..200]$. Furthermore, a trader can indicate that she wants to avoid certain items; for instance, a superstitious buyer may want to avoid black cars with 13 miles on the odometer. In this case, she must use a *filter function* that prunes undesirable items. This filter is a Boolean function on the set I, encoded by a C++ procedure, which gives FALSE for unwanted items. To summarize, the representation of an item set consists of two parts: a union of Cartesian products and a filter function.

Orders. An order includes an item set, price function, quality function, size, and optional expiration time. If the price function is a constant, it is specified by a numeric value; else, it is a C++ procedure that inputs an item and outputs the corresponding price limit. The representation of a quality function is also a C++ procedure, which inputs an item description and price, and outputs a numeric quality value. A trader can use different functions for different orders; if she does not provide any quality function, the system uses the default quality measure defined through the price function (see Section 2). The size specification includes three integers: the overall size, minimal acceptable size, and size step. A trader can specify whether the system should preserve the minimal size in the case of a partial fill; if not, the system removes the minimal size after a partial fill. A trader can also specify the expiration time of an order with one-second precision. If the system does not find a match by the specified time, it cancels the order. A trader can cancel her order manually before the expiration time. She can also modify her order without removing it from the market; for example, if a car buyer has placed an order to purchase a red Mustang for \$19,000, she can later increase the price limit to \$20,000 or change the item description to "any-color Mustang or Camaro."

4 Matcher engine

We outline the architecture of the exchange system, and then describe the indexing structure and algorithms for fast identification of matches between buy and sell orders.

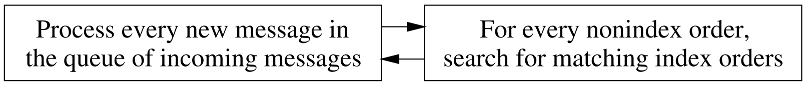


Figure 5: Main loop of the matcher.

4.1 Architecture

The system consists of a central matcher and multiple user interfaces that run on separate machines. The traders enter their orders through interface machines, which send the orders to the matcher. The system supports three types of messages to the matcher: placing, modifying, and cancelling an order.

The matcher maintains a central structure for indexing of orders with fully specified items, which do not include ranges, standard sets, intersections, or unions. If we can put an order into this structure, we call it an *index order.* If an order includes a set of items, rather than a fully specified item, it is added to an unordered list of *nonindex orders.* This indexing scheme allows fast retrieval of index orders that match a given order; however, the system does not identify matches between two nonindex orders.

The system alternates between processing new messages and identifying matches for old orders (Figure 5); we show the main steps of its top-level loop in Figure 6(a). When the system receives a message with a new order, it immediately identifies matching index orders (Figure 6b). If there are no matches, and the new order is an index order, then the system adds it to the indexing structure. Similarly, if the system fills only part of a new index order, it stores the remaining part in the indexing structure. If it gets a nonindex order and does not find a complete fill, it adds the unfilled part to the list of nonindex orders.

When the system gets a cancellation message, it removes the specified order from the market. When it receives a modification message, it makes changes to the specified order (Figure 6c). If the changes can potentially lead to new matches, it immediately searches for index orders that match the modified order. For example, if a seller reduces the price of her order, the system immediately identifies new matches. On the other hand, if the seller increases her price, the system does not search for matches.

After processing all messages, the system tries to fill old nonindex orders (Figure 6a); for each nonindex order, it identifies matching index orders. For example, suppose that the market includes an order to buy any red Mustang, and that a dealer places a new order to sell a red Mustang, made in 2003, with zero miles. If there are no matching index orders, the system adds this new order to the indexing structure. After processing all messages, it tries to fill the nonindex orders, and determines that the dealer's order is a match for the old order to buy any red Mustang.

4.2 Indexing structure

We have implemented an indexing structure for orders with fully specified items, which consists of two identical trees: one is for buy orders, and the other is for sell orders.

Indexing tree. In Figure 7, we show a tree for sell orders; its height equals the number of attributes, and each level corresponds to one of the attributes. The root node encodes the first attribute, and its children represent different values of this attribute. The nodes at the second

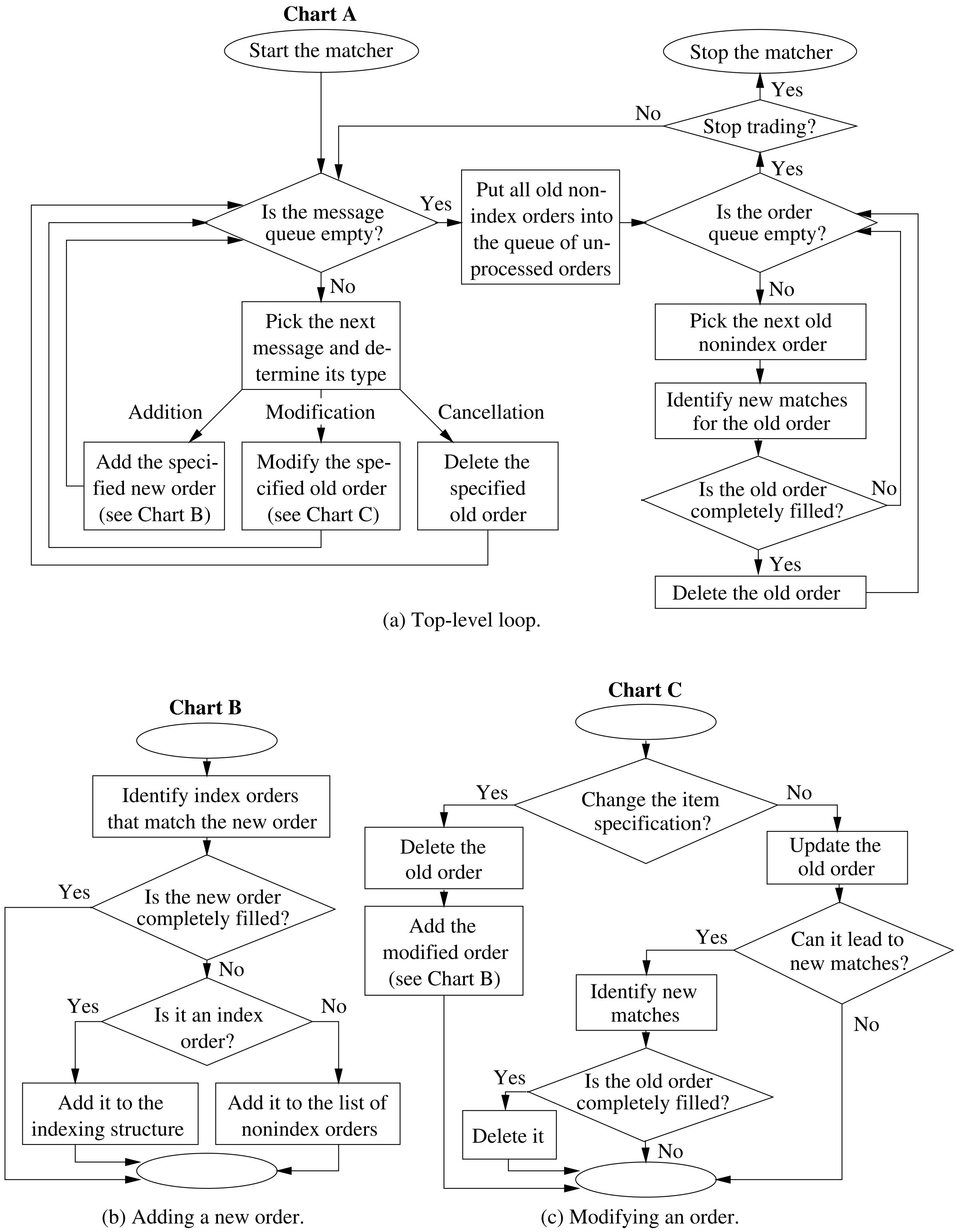


Figure 6: Processing of orders in the matcher.

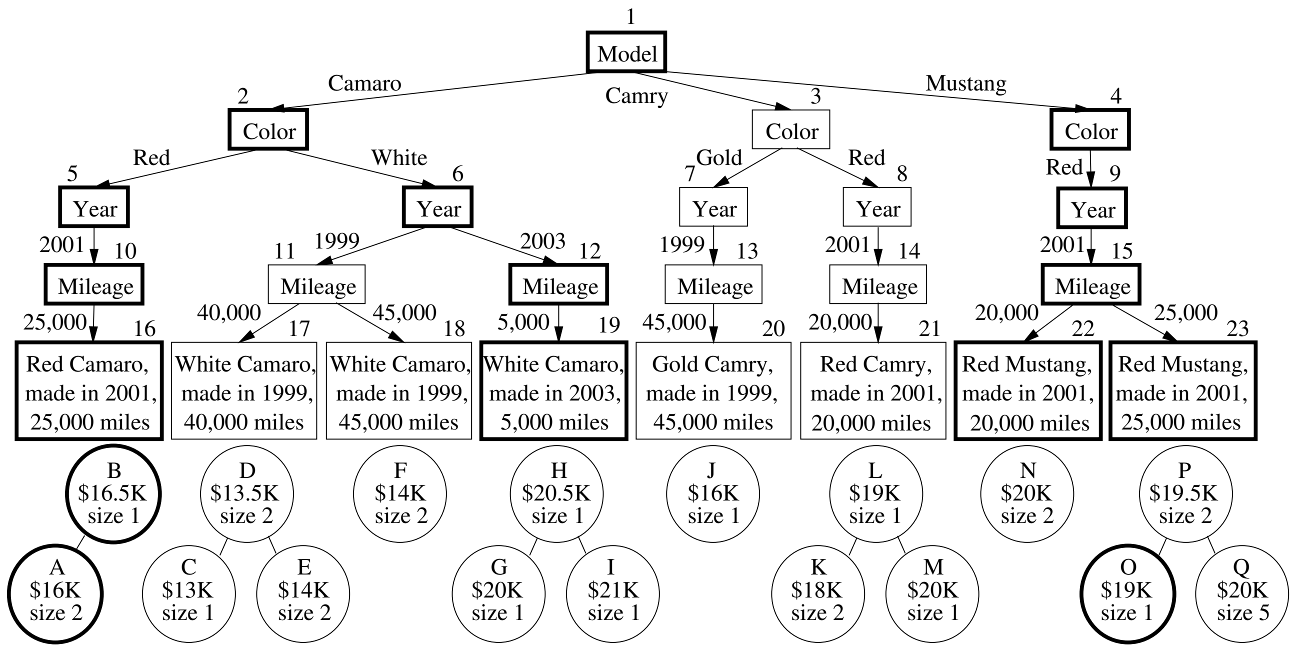


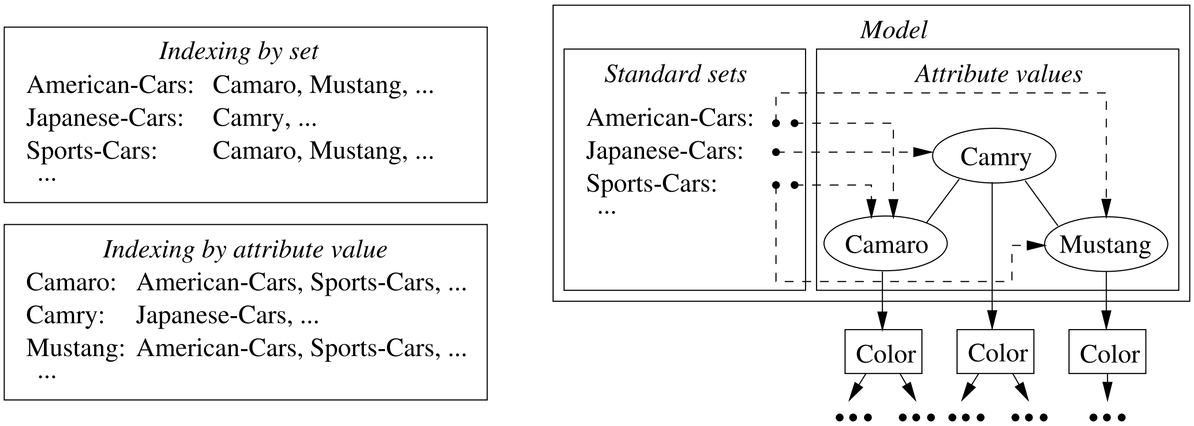
Figure 7: Indexing tree with seventeen orders. We illustrate the retrieval of matches for an order to buy four Mustangs or Camaros made after 2000. We show the matching nodes by thick boxes, and the retrieved orders by thick circles.

level divide the orders by the second attribute, and each node at the third level corresponds to specific values of the first two attributes. In general, a node at level i divides orders by the values of the ith attribute, and each node at level $(i+1)$ corresponds to all orders with specific values of the first i attributes. If some items are not currently on sale, the tree does not include the corresponding nodes. Every nonleaf node includes a red-black tree that allows fast retrieval of its children with given values; for example, the root node in Figure 7 includes a red-black tree that indexes its children by model values.

Standard sets. If a market includes standard sets of values, such as "all sports cars" and "all American cars," traders can use them in specifying their orders. For every attribute, the system maintains a central table of standard sets, which consists of two parts (Figure 8a). The first part includes a sorted list of values for every set; it allows determining whether a given value belongs to a specific set, by the binary search in the corresponding list. The second part includes all values that belong to at least one set; for each value, we store a sorted list of sets that include it. Every node of an indexing tree also contains a table of standard sets; for example, the root node in Figure 7 contains a table of sets for the first attribute (Figure 8b). Every set in the table includes a list of pointers to its elements in the node's red-black tree; for instance, the `American-Cars` set points to `Corvette` and `Mustang`.

Basic operations. The nonleaf-node structure (Figure 8b) supports fast addition and deletion of children, as well as fast retrieval operations:

- Retrieval of a child with a given attribute value.
- Retrieval of all children in a given standard set.
- Retrieval of all children in a given range.



(a) Central table of standard sets. (b) Standard sets in a node of the indexing tree.

Figure 8: Standard sets of values. The matcher includes a central table of sets (a), and every node in the indexing tree includes a table of sets for the respective attribute (b).

The system can also retrieve all children that belong to an intersection or union of several attribute sets. Given an intersection, the system identifies the matching children for one of its elements, and then prunes the children that do not belong to the other elements. Given a union, the system identifies matching children for each of its elements, and then generates the union of the resulting sets of children.

Leaf nodes. A leaf node includes orders with identical items, which are sorted by price, from the best to the worst; that is, the system sorts buy orders from the highest to the lowest price limit, and sell orders from the lowest to the highest price. If several orders have the same price, they are sorted by their placement time, from the earliest to the latest. This sorting ensures that the matcher gives preference to the earliest among equally priced orders, which is a standard requirement in the financial industry. We use a red-black tree to maintain this sorting, which allows fast insertion of a new order, deletion of an order, and change of an order's price and placement time.

Time stamps. The system keeps track of the "age" of each order, and uses it to avoid repetitive search for matches among the same orders. Every order has two time stamps; the first is the time of placing the order, and the second is the time of the last search for matches. Furthermore, for each node of the indexing tree, the system keeps the time of the last addition of an order to the corresponding subtree. If the system repeats the search for matches for some nonindex order, it skips the subtrees that have no new index orders (see Section 4.3).

Adding and deleting an order. When a trader places an index order, the system adds it to the corresponding leaf; for example, if a dealer places an order to sell a red Camaro, made in 2001, with 25,000 miles, the system adds it to node 16 in Figure 7. If the leaf is not in the tree, the system adds the appropriate new branch. After adding an order, it updates the time stamps of the ancestor nodes; for instance, if it adds a new order to node 16, then it updates the time stamps of nodes 1, 2, 5, 10, and 16. When the system fills an index order, or a trader cancels her old order, the system removes it from the corresponding leaf. If the leaf does not include other orders, the system deletes it from the tree; if the deleted node is the only leaf in some subtree, the system removes this subtree. For example, the deletion of order J in Figure 7 leads to the removal of nodes 7, 13, and 20.

Modifying an order. If a trader changes the size or expiration time of an order, it does not affect the indexing tree. If she modifies the price of an order, the system changes the position of the order in the red-black tree of the leaf. Finally, if she changes the item specification, the system treats it as the deletion of an old order and addition of a new one. If the modification can potentially lead to new matches, the system changes the order-placement time, as if it were a new order, and then updates the time stamps of the ancestor nodes. For example, if a seller reduces her price limit, the system updates the time stamps. On the other hand, if she increases her price, the system does not change the time stamps because it cannot result in new matches.

4.3 Search for matches

We give a two-step algorithm that identifies matches for a given order; it first finds the indexing-tree leaves that match the item set of the given order, and then selects the highest-quality matches in these leaves. In Figure 9, we present the notation for the order and leaf-node structures; in Figures 10 and 11, we give pseudocode for the two main steps of the algorithm.

Matching leaves. The algorithm in Figure 10 retrieves matching leaves for the item set of a given order; recall that we represent the item set by a union of Cartesian products and an optional filter function.

The DFS subroutine finds all matches for one Cartesian product using depth-first search in the indexing tree; it identifies all children of the root that match the first element of the Cartesian product, and then recursively processes the respective subtrees. For example, suppose that a buyer is looking for a Mustang or Camaro made after 2000, with any color and mileage, and the tree of sell orders is as shown in Figure 7. The subroutine determines that nodes 2 and 4 match the model, and then processes the two respective subtrees. It identifies three matching nodes for the second attribute, three nodes for the third attribute, and finally four matching leaves; we show these nodes by thick boxes.

If the system already tried to find matches for a given order during the previous execution of the main loop, it skips the subtrees that have not been modified since the previous search. If the order includes the union of several Cartesian products, the system calls the DFS subroutine for each product. If the order includes a filter function, the system uses the filter to prune inappropriate leaves.

If an order matches a large number of leaves, the retrieval may take considerable time. To prevent this problem, we can impose a limit on the number of retrieved leaves; in this case, the search terminates after finding the given number of matching leaves. For instance, if we allow at most three matches, and a trader places an order to buy any Camaro, then the system retrieves the three leftmost leaves in Figure 7. We use this limit to control the trade-off between the speed and quality of matches; a small limit ensures the efficiency but reduces the chances of finding the best match.

Best matches. After the system identifies matching leaves, it selects the best matching orders in these leaves, according to the quality function of the given order. In Figure 11, we give an algorithm that identifies the highest-quality matches and completes the respective trades. It

Elements of the order structure:

Price[*order*]	price function
Qual[*order*]	quality function
Filter[*order*]	filter function
Max[*order*]	overall order size
Min[*order*]	minimal acceptable size
Step[*order*]	size step
Place-Time[*order*]	time of placing the order
Search-Time[*order*]	time of the last search for matches

Elements of the leaf-node structure:

Item[*leaf*]	item in the leaf's nodes
Current-Order[*leaf*]	best-price unprocessed order in the leaf
Quality[*leaf*]	quality of the best-price unprocessed order
Place-Time[*leaf*]	time of placing the best-price unprocessed order

Figure 9: Notation for the main elements of the structures that represent an order and a leaf node. We use this notation in the matching-algorithm pseudocode in Figures 10 and 11.

MATCHING-LEAVES(*order*, *root*)
The algorithm inputs a given order and the root of an indexing tree.
It returns the leaves that match the item set of the order.

Initialize an empty set of matching leaves, denoted *leaves*
For each Cartesian product $I_1 \times I_2 \times ... \times I_n$ in *order*'s item set:
  Call DFS($I_1 \times I_2 \times ... \times I_n$, *Filter*[*order*], *Search-Time*[*order*], *root*, 1, *leaves*)
Return *leaves*

DFS($I_1 \times I_2 \times ... \times I_n$, *Filter*, *Search-Time*, *node*, k, *leaves*)
The subroutine inputs a Cartesian product $I_1 \times I_2 \times ... \times I_n$, a filter function, the previous-search time, a node of the indexing tree, the node's depth in the tree, and a set of leaves. It finds the matching leaves of the subtree rooted at the given node, and adds them to the set of leaves.

If *Search-Time* is larger than *node*'s time of the last order addition, then terminate
If *node* is a leaf and *Filter*(*Item*[*node*]) = TRUE, then add *node* to *leaves*
If *node* is not a leaf, then:
  Identify all children of *node* that match I_k
  For each matching *child*:
    Call DFS($I_1 \times I_2 \times ... \times I_n$, *Filter*, *Search-Time*, *child*, $k+1$, *leaves*)

Figure 10: Retrieval of matching leaves. The algorithm identifies the leaves of an indexing tree that match the item set of a given order. The DFS subroutine uses depth-first search to retrieve matching leaves for one Cartesian product.

BEST-MATCHES(*order*, *leaves*)

The algorithm inputs a given order and matching leaves of an indexing tree.
It identifies the highest-quality matches for the order in these leaves.

Initialize an empty priority queue of matching leaves, denoted *queue*,
  which prioritizes the leaves by the quality of the best-price unprocessed order
For each *leaf* in *leaves*:
  Set *Current-Order*[*leaf*] to the first order among *leaf*'s orders, sorted by price
  Call LEAF-PRIORITY(*order*, *leaf*, *queue*)
Repeat while $Max[order] \geq Min[order]$ and *queue* is nonempty:
  Set *leaf* to the highest-priority leaf in *queue*, and remove it from *queue*
  $match := Current\text{-}Order[leaf]$
  Set *Current-Order*[*leaf*] to the next order among *leaf*'s orders, sorted by price
  Call TRADE(*order*, *match*)
  Call LEAF-PRIORITY(*order*, *leaf*, *queue*)
If $Max[order] < Min[order]$, then remove *order* from the market
Else, set *Search-Time*[*order*] to the current time

LEAF-PRIORITY(*order*, *leaf*, *queue*)

The subroutine inputs the given order, a matching leaf, and the priority queue of leaves. If the order's price matches the price of the leaf's best-price unprocessed order, the leaf is added to the queue.

$match := Current\text{-}Order[leaf]$
If $match = \text{NONE}$, then terminate (no more orders in *leaf*)
If *order* is a buy order, then $price := \text{FILL-PRICE}(Price[order], Price[match], Item[leaf])$
Else, $price := \text{FILL-PRICE}(Price[match], Price[order], Item[leaf])$
If $price = \text{NONE}$, then terminate (the buyer's price does not match the seller's price)
$Quality[leaf] := Qual[order](Item[leaf], price)$
$Place\text{-}Time[leaf] := Place\text{-}Time[match]$
Add *leaf* to *queue*, prioritized by *Quality* with ties broken by *Place-Time*

TRADE(*order*, *match*)

The subroutine inputs the given order and the highest-quality order with a matching item and price.
If the sizes of these two orders match, it completes the trade between them.

If $Search\text{-}Time[order] > Place\text{-}Time[match]$, then terminate
$size := \text{FILL-SIZE}(Max[order], Min[order], Step[order], Max[match], Min[match], Step[match])$
If $size = \text{NONE}$, then terminate
Complete the trade between *order* and *match*
$Max[order] := Max[order] - size$
$Max[match] := Max[match] - size$
If $Max[match] < Min[match]$, then remove *match* from the market

---

Figure 11: Retrieval of matching orders. The algorithm finds the best matches for a given order and completes the corresponding trades. The LEAF-PRIORITY subroutine adds a given leaf to the priority queue, arranged by the quality of a leaf's best-price unprocessed match. The TRADE subroutine completes the trade between the given order and the best available match.

arranges the leaves in a priority queue by the quality of the best unprocessed match in a leaf; if several matches have the same quality, they are prioritized by their placement time. At each step, the algorithm processes the best available match; if several matching orders have the same quality, it gives preference to the order with the earliest placement time. The algorithm terminates after it fills the given order or runs out of matches. For example, consider the tree in Figure 7, and suppose that a buyer places an order for four Mustangs or Camaros made after 2000. We suppose further that she uses the default quality function, which depends only on price. The system first retrieves order A, with price $16,000 and size 2, then order B with price $16,500, and finally order O with price $19,000; we show these orders by thick circles.

5 Performance

We describe experiments with artificial market data and two real-world markets, on a 400-MHz Pentium computer with one-gigabyte memory. A more detailed report of the experimental results is available in the masters theses by Johnson [2001] and Hershberger [2003].

5.1 Artificial markets

We have implemented an experimental setup that allows control over the number of orders, rate of incoming orders, number of attributes, number of values per attribute, and average number of matches per order.

Old orders: We have varied the number of old orders in the market from one to 300,000, which is the maximal possible number for one-gigabyte memory. We have randomly generated these orders, which include an equal number of buy and sell orders.

New orders: Recall that the system's main loop involves processing messages and matching old orders (Figure 5). The number of messages with new orders in the beginning of the loop is proportional to the rate of placing new orders. We have directly controlled this number, and experimented with 300, 10,000, and 300,000 new orders.

Attributes and values: We have considered markets with one, three, ten, thirty, and one hundred attributes, and varied the number of values per attribute from two to 1,000.

Matching density: We define the *matching density* as the mean percentage of sell orders that match a given buy order; in other words, it is the probability that a randomly selected buy order matches a randomly chosen sell order. We have considered five matching-density values: 0.0001, 0.001, 0.01, 0.1, and 1.

For each setting of the control variables, we have measured the main-loop time, throughput, and response time. The *main-loop time* is the time of one pass through the system's main loop (Figure 5), which includes processing new orders and matching old orders. The *throughput* is the maximal acceptable rate of placing new orders; if the system gets more orders per second, the number of unprocessed orders keeps growing and eventually leads to an overflow. Finally, the *response time* is the average time between placing an order and getting a fill. We give the dependency of these measurements on the control variables in Figures 12–16; the scales of all graphs are logarithmic.

In Figure 12, we show how the performance changes with the number of old orders. The main-loop and response times are approximately linear in the number of orders. The throughput in small markets grows with the number of orders; it reaches a maximum when the market grows to about two hundred orders, and slightly decreases with further increase in the market size.

In Figure 13, we show that both the main-loop time and response time are linear in the number of new orders. We do not show the dependency of the throughput on the number of new orders, because the number of new orders directly depends on the throughput, and we cannot treat it as an independent control variable.

In Figure 14, we give the dependency of the performance on the number of attributes. The main-loop and response times are super-linear in the number of attributes, whereas the throughput is in inverse proportion to the same super-linear function.

In Figure 15, we show that the main-loop and response times grow sub-linearly with the number of values per attribute, and the throughput slightly decreases with an increase in the number of values.

In Figure 16, we show that the main-loop and response times grow linearly with the matching density. On the other hand, we have not found any monotonic dependency between the matching density and throughput.

5.2 Real markets

We have applied the system to an extended used-car market and a commercial-paper market; the results are similar to those of artificial tests.

Used cars. We have considered a used-car market that includes all models available through AutoNation (www.autonation.com), described by eight attributes: transmission (2 values), number of doors (3 values), interior color (7 values), exterior color (52 values), year (103 values), model (257 values), option package (1,024 values), and mileage (500,000 values). We have controlled the number of old orders, number of new orders, and matching density; we show the results in Figures 17–19. The system scales to markets with 300,000 orders, and processes 100 to 1,000 new orders per second. The main-loop and response times are linear in the number of orders; the throughput first increases and then slightly decreases with the number of orders.

Commercial paper. When a large company needs a loan, it may issue *commercial paper,* which is a fixed-interest "promissory note" similar to a bond. The company sells commercial paper to investors, and later returns their money with interest; the payment day is called the *maturity date.* The main difference from bonds is duration of the loan; commercial paper is issued for a short term, from one week to nine months. We have described commercial paper by two attributes: issuing company (5,000 values) and maturity date (2,550 values). We plot the dependency of the system's performance on the control variables in Figures 20–22. The system scales to markets with 300,000 orders, and processes 500 to 5,000 new orders per second.

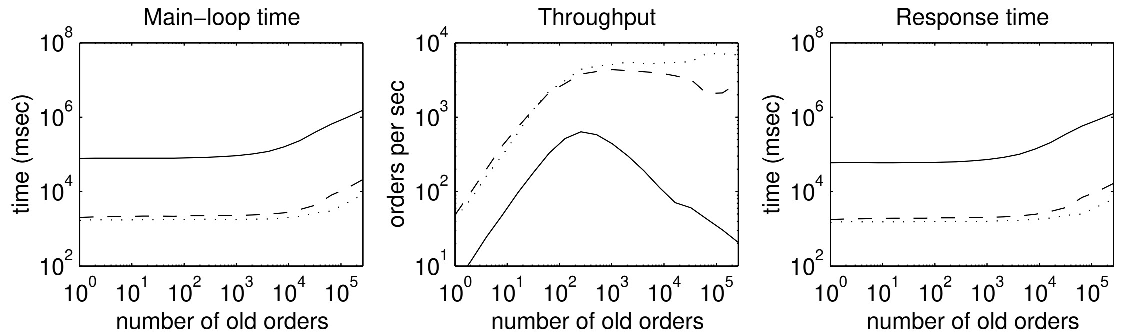


Figure 12: Dependency of the system's performance on the *number of old orders* for the artificial markets. The dotted lines show experiments with one attribute, two values per attribute, 300 new orders, and matching density of 0.0001. The dashed lines are for three attributes, sixteen values per attribute, 10,000 new orders, and matching density of 0.001. The solid lines are for ten attributes, 1,000 values per attribute, 10,000 new orders, and matching density of 0.01.

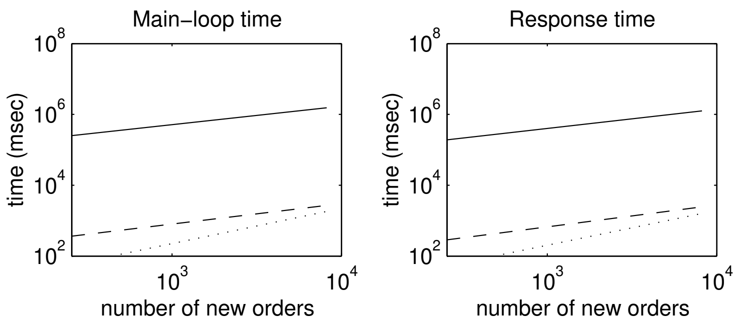


Figure 13: Dependency of the performance on the *number of new orders* for the artificial markets. The dotted lines show experiments with one attribute, two values per attribute, 300 old orders, and matching density of 0.0001. The dashed lines are for three attributes, sixteen values per attribute, 10,000 old orders, and matching density of 0.001. The solid lines are for ten attributes, 1,000 values per attribute, 300,000 old orders, and matching density of 0.01.

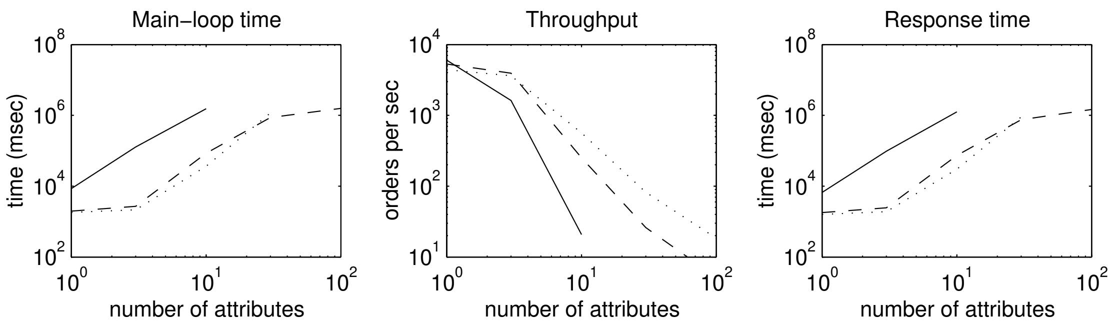


Figure 14: Dependency of the performance on the *number of market attributes* for the artificial markets. The dotted lines show experiments with two values per attribute, 300 old orders, 300 new orders, and matching density of 0.0001. The dashed lines are for sixteen values per attribute, 10,000 old orders, 10,000 new orders, and matching density of 0.001. The solid lines are for 1,000 values per attribute, 300,000 old orders, 10,000 new orders, and matching density of 0.01. We do not plot the solid lines for 30 and 100 attributes, because we have been unable to run the corresponding experiments, which would require more than one-gigabyte main memory.

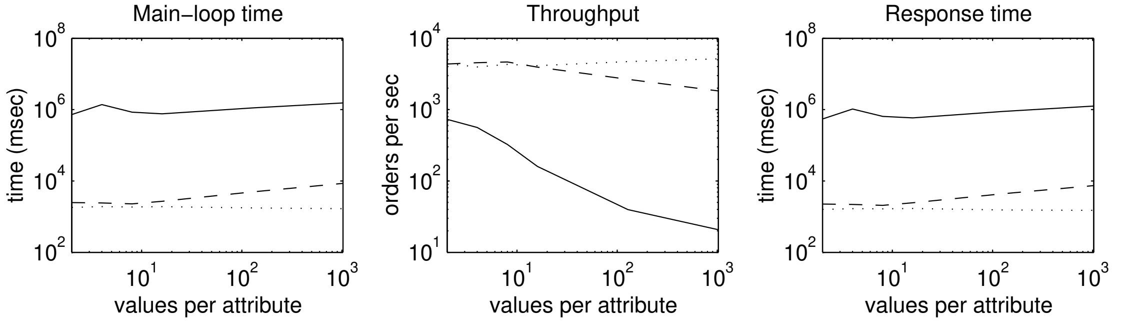


Figure 15: Dependency of the performance on the *number of values per attribute* for the artificial markets. The dotted lines show experiments with one attribute, 300 old orders, 300 new orders, and matching density of 0.0001. The dashed lines are for three attributes, 10,000 old orders, 10,000 new orders, and matching density of 0.001. The solid lines are for ten attributes, 300,000 old orders, 10,000 new orders, and matching density of 0.01.

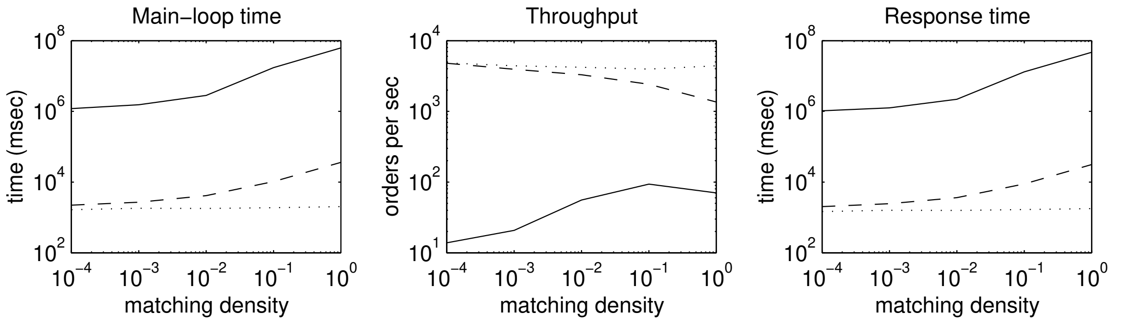


Figure 16: Dependency of the performance on the *matching density* for the artificial markets. The dotted lines show experiments with one attribute, two values per attribute, 300 old orders, and 300 new orders. The dashed lines are for three attributes, sixteen values per attribute, 10,000 old orders, and 10,000 new orders. The solid lines are for ten attributes, 1,000 values per attribute, 300,000 old orders, and 10,000 new orders.

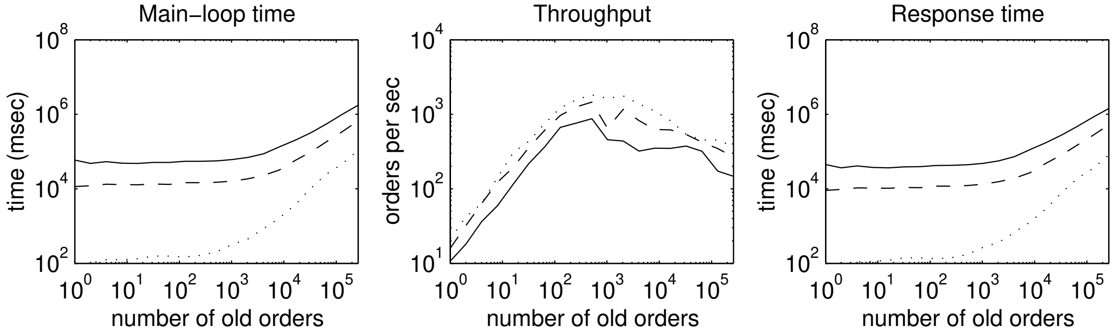


Figure 17: Dependency of the performance on the *number of old orders* for the used-car market. The dotted lines show experiments with 300 new orders and matching density of 0.0001. The dashed lines are for 10,000 new orders and matching density of 0.001. The solid lines are for 10,000 new orders and matching density of 0.01.

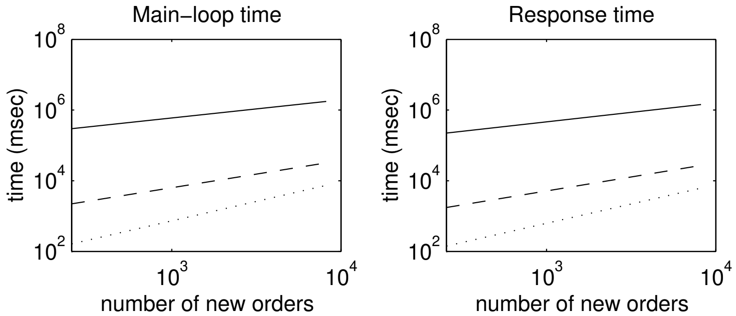


Figure 18: Dependency of the performance on the *number of new orders* for the used-car market. The dotted lines show experiments with 300 old orders and matching density of 0.0001. The dashed lines are for 10,000 old orders and matching density of 0.001. The solid lines are for 300,000 old orders and matching density of 0.01.

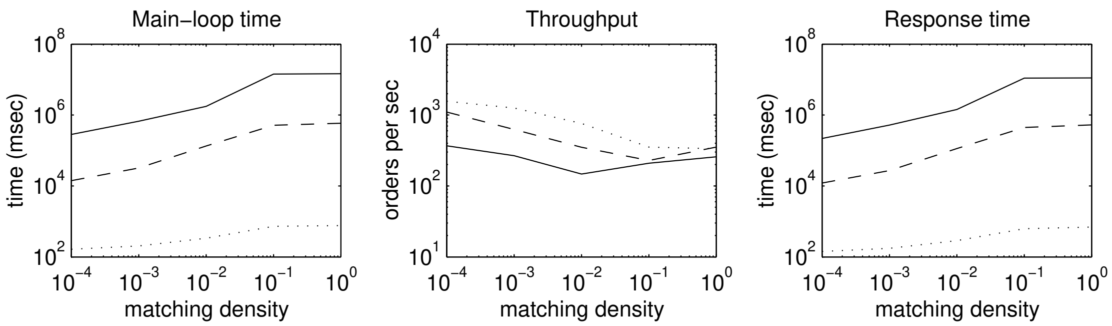


Figure 19: Dependency of the performance on the *matching density* for the used-car market. The dotted lines show experiments with 300 old orders and 300 new orders. The dashed lines are for 10,000 old orders and 10,000 new orders. The solid lines are for 300,000 old orders and 10,000 new orders.

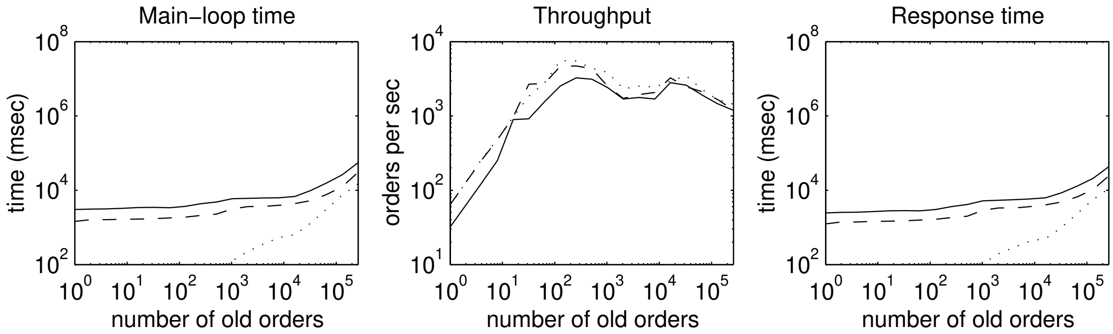


Figure 20: Dependency of the performance on the *number of old orders* for the commercial-paper market. The dotted lines show experiments with 300 new orders and matching density of 0.0001. The dashed lines are for 10,000 new orders and matching density of 0.001. The solid lines are for 10,000 new orders and matching density of 0.01.

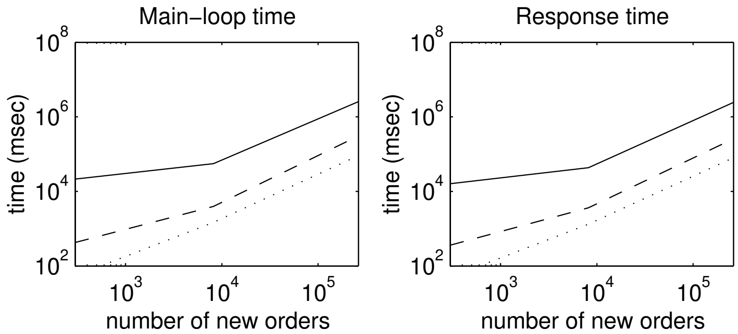


Figure 21: Dependency of the performance on the *number of new orders* for the commercial-paper market. The dotted lines show experiments with 300 old orders and matching density of 0.0001. The dashed lines are for 10,000 old orders and matching density of 0.001. The solid lines are for 300,000 old orders and matching density of 0.01.

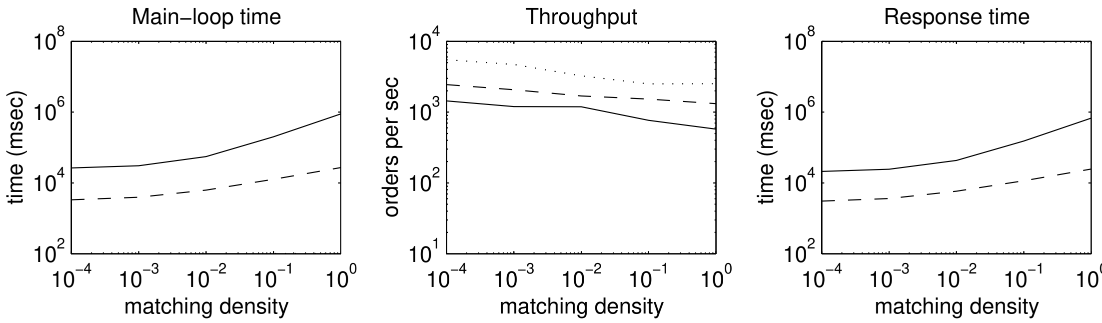


Figure 22: Dependency of the performance on the *matching density* for the commercial-paper market. The dotted lines show experiments with 300 old orders and 300 new orders. The dashed lines are for 10,000 old orders and 10,000 new orders. The solid lines are for 300,000 old orders and 10,000 new orders.

6 Concluding remarks

The modern economy includes a variety of marketplaces with millions of participants, and the Internet has opened opportunities for efficient on-line trading. Computer scientists have studied algorithms for auctions and standardized exchanges, but they have done little work on exchange markets for multi-attribute goods.

The reported work is a step toward the development of automated exchanges for nonstandard goods. We have proposed a formal model for trading such goods, and built an exchange that allows constraints and preference functions in the description of orders. It supports markets with up to 300,000 orders and processes hundreds of new orders per second on a 400-MHz computer with one-gigabyte memory. The system keeps all orders in the main memory, and its scalability is limited by the available memory. We are presently working on a distributed system that includes a central matcher and multiple preprocessing modules, whose role is similar to that of stock brokers.

References

[Bichler *et al.*, 1999] Martin Bichler, Marion Kaukal, and Arie Segev. Multi-attribute auctions for electronic procurement. In *Proceedings of the First* IBM IAC *Workshop on Internet-Based Negotiation Technologies*, 1999.

[Bichler, 2000] Martin Bichler. An experimental analysis of multi-attribute auctions. *Decision Support Systems*, 29(3):249–268, 2000.

[Che, 1993] Yeon-Koo Che. Design competition through multidimensional auctions. RAND *Journal of Economics*, 24(4):668–680, 1993.

[Cripps and Ireland, 1994] Martin Cripps and Norman Ireland. The design of auctions and tenders with quality thresholds: The symmetric case. *Economic Journal*, 104(423):316–326, 1994.

[Gonen and Lehmann, 2000] Rica Gonen and Daniel Lehmann. Optimal solutions for multi-unit combinatorial auctions: Branch and bound heuristics. In *Proceedings of the Second* ACM *Conference on Electronic Commerce*, pages 13–20, 2000.

[Gonen and Lehmann, 2001] Rica Gonen and Daniel Lehmann. Linear programming helps solving large multi-unit combinatorial auctions. In *Proceedings of the Electronic Market Design Workshop*, 2001.

[Gong, 2002] Jianli Gong. Exchanges for complex commodities: Search for optimal matches. Master's thesis, Department of Computer Science and Engineering, University of South Florida, 2002.

[Hershberger, 2003] Jonathan Wade Hershberger. Exchanges for complex commodities: Toward a general-purpose system for on-line trading. Master's thesis, Department of Computer Science and Engineering, University of South Florida, 2003.

[Hu, 2002] Jenny Ying Hu. Exchanges for complex commodities: Representation and indexing of orders. Master's thesis, Department of Computer Science and Engineering, University of South Florida, 2002.

[Johnson, 2001] Joshua Marc Johnson. Exchanges for complex commodities: Theory and experiments. Master's thesis, Department of Computer Science and Engineering, University of South Florida, 2001.

[Kalagnanam *et al.*, 2000] Jayant R. Kalagnanam, Andrew J. Davenport, and Ho S. Lee. Computational aspects of clearing continuous call double auctions with assignment constraints and indivisible demand. Technical Report RC21660(97613), IBM, 2000.

[Lavi and Nisan, 2000] Ran Lavi and Noam Nisan. Competitive analysis of incentive compatible on-line auctions. In *Proceedings of the Second* ACM *Conference on Electronic Commerce*, pages 233–241, 2000.

[Lehmann *et al.*, 2001] Benny Lehmann, Daniel Lehmann, and Noam Nisan. Combinatorial auctions with decreasing marginal utilities. In *Proceedings of the Third* ACM *Conference on Electronic Commerce*, pages 18–28, 2001.

[Nisan, 2000] Noam Nisan. Bidding and allocation in combinatorial auctions. In *Proceedings of the Second* ACM *Conference on Electronic Commerce*, pages 1–12, 2000.

[Rothkopf *et al.*, 1998] Michael H. Rothkopf, Aleksandar Pekeč, and Ronald M. Harstad. Computationally manageable combinatorial auctions. *Management Science*, 44(8):1131–1147, 1998.

[Sandholm and Suri, 2000] Tuomas W. Sandholm and Subhash Suri. Improved algorithms for optimal winner determination in combinatorial auctions and generalizations. In *Proceesings of the Seventeenth National Conference on Artificial Intelligence*, pages 90–97, 2000.

[Sandholm and Suri, 2001] Tuomas W. Sandholm and Subhash Suri. Market clearability. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 1145–1151, 2001.

[Sandholm *et al.*, 2001a] Tuomas W. Sandholm, Subhash Suri, Andrew Gilpin, and David Levine. CABOB: A fast optimal algorithm for combinatorial auctions. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 1102–1108, 2001.

[Sandholm *et al.*, 2001b] Tuomas W. Sandholm, Subhash Suri, Andrew Gilpin, and David Levine. Winner determination in combinatorial auction generalizations. In *Proceedings of the International Conference on Autonomous Agents, Workshop on Agent-Based Approaches to* B2B, pages 35–41, 2001.

[Sandholm, 2000] Tuomas W. Sandholm. Approach to winner determination in combinatorial auctions. *Decision Support Systems*, 28(1–2):165–176, 2000.

[Wurman *et al.*, 1998] Peter R. Wurman, William E. Walsh, and Michael P. Wellman. Flexible double auctions for electronic commerce: Theory and implementation. *Decision Support Systems*, 24(1):17–27, 1998.