TU Kaiserslautern
FB Informatik, AG HIS
Prof. Dr.-Ing. Stefan Deßloch
M.Sc. Michael Hohenstein

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Worksheet 2 - DB Schema Design and Programming

"Views, Triggers, and Stored Procedures"

Introduction

Goal. In this worksheet you will learn fundamental database features such as Views, Triggers, and Stored Procedures, as well as how to implement them in DB2. The target application is a stock trading system for the Kaiserslautern Stock Exchange (KSE).

The *market* is defined as a set of stocks, each with a given *price* at which one share is traded. Each trader has a *depot*, where the amount of shares he owns for each stock is recorded. Furthermore, traders have an account *balance*, which is the amount of money they have to buy more shares.

Shares can be bought and sold by placing an *order*, in which a trader specifies the stock he wants to buy or sell, how many shares, and the price he is willing to pay/receive for each share. Buy orders are called *call orders*, while sell orders are called *put orders*. If a trader is willing to call a share for, say, 100 €, he would obviously also accept a lower price, like 98 €. The inverse holds for put orders. For this reason, the price specified in an order is called a *limit*—for calls it is an upper-bound limit and for puts it is a lower-bound limit on the accepted price.

The processing of orders and adjustment of new prices happens according to a *period*, which can vary from a day to a millisecond. At each period, the system looks at the list of pending orders—also called the *orders book*—and the limits that occur for each stock. Based on this information, a price at which the trades will actually be executed is fixed. This is where the pressure of the market speculation (the fight between bulls and bears) will decide whether the price goes up or down. The calculation of the price for the next period follows a strict set of rules, which will be given later on. Once a new price is calculated, the orders that can be fulfilled are processed and then removed from the orders book.

After each period, the *market time* of each processed stock is incremented. The *global market time* is defined as the highest market time of all stocks. If no orders arrive during a period, no trades are executed and the global market time is kept unchanged.

The database schema for this application is shown in Figure 1. It can be loaded in your DB with the files `/dbprak/share/ws2/schema.sql` and `/dbprak/share/ws2/data.sql` in Lara.

Exercises

Exercise 1: Views

The best way to control database access and ensure constraints is to hide the database tables from the applications and allow access to the data only through *views*. This also allows the DB designer to offer a more convenient representation to the application, leading to a simplified application development. This representation often introduces redundant information which is not allowed by the normalized relational schema, but simplifies the interaction with the data. Your job for the KSE is to create views to represent a summary of the trader's accounts and the order book.

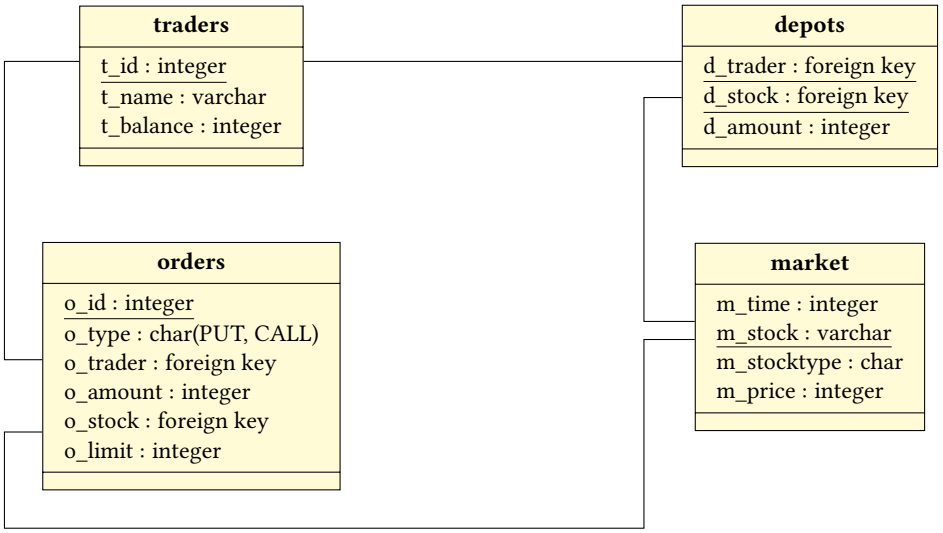

Figure 1: Trading schema

1.1. Account View

Define a view named *account_v* giving an overview of depots and all puts/calls still due. The layout of the view is shown in Figure 2.

The view shows, for each order in the system, the trader who issued the order, the target stock, the amount of shares in his depot, the amount of shares in the order, the order's limit, the current market price of the stock, and finally a market value, which is derived by multiplying price and amount. The view represents call and put orders in a unified manner, by storing order information either in *call_** or in *put_** columns. This means that call orders have all *put_** columns set to null and vice-versa. Furthermore, remember that traders may call stocks that are not yet in their depot. Ensure that they are included in the view as well.

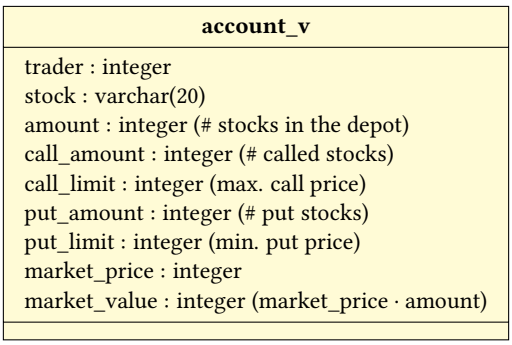

Figure 2: View *account_v*

1.2. Order view

Define a view *order_v* that summarizes the call/put situation for stocks. The view layout is shown in Figure 3.

This view summarizes the volume of shares that can be traded for each limit occurring in the orders of a stock. Therefore, the number of entries for each stock is determined by the number of distinct call and put limits. The *call_volume* (*put_volume*) column shows the number of shares that can be bought (sold) at the given limit. For example, if there is a total of 100 shares in all call orders with a limit of 150 € and another 200 in those with a limit of 151 €, the price 150 € will allow 300 shares to be bought, because those who were willing to buy for 151 € will also buy for 150 €. The limit 151 €, on the other hand, will allow at most 200 shares to be bought. The analogous situation holds inversely for put orders.

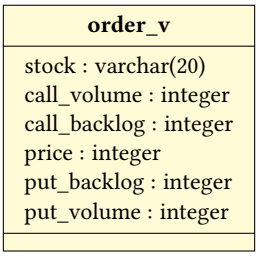

Figure 3: View *order_v*

The view also shows the put and call *backlogs*, which result from an imbalance between the number of called and put stocks for a given limit, or, in other words, from an imbalance between supply and demand. It represents the amount of shares in orders which cannot be fulfilled at a particular limit. If a limit has a call volume of 100 and a put volume of 300, for example, the put backlog is 200 and the call backlog is 0. This means that 200 shares which were meant to be sold will not be processed, because there were not enough buy orders with a compatible limit. As we shall see later on price calculation, this situation shows that supply is higher than demand, and thus the price will tend to fall.

The following example shows how this view should look like for a sample stock in the orders table:

orders

o_type	o_trader	o_amount	o_stock	o_limit
PUT	4711	100	IBM	198
PUT	4712	500	IBM	200
CALL	4713	300	IBM	199
CALL	4714	200	IBM	201

order_v

stock	call_volume	call_backlog	limit	put_backlog	put_volume
IBM	200	0	201	400	600
IBM	200	0	200	400	600
IBM	500	400	199	0	100
IBM	500	400	198	0	100

Exercise 2: Place orders

The views produced so far allow traders to visualize the status of the market without the need to access the underlying tables. In this task, you will enable the complete abstraction of the tables from the traders by enabling modifications to be carried out exclusively through the views as well. This can be achieved by defining proper *triggers* in the view *account_v*.

Your task is to define one trigger for insertions and another one for deletions. Their names should be *account_v_insert* and *account_v_delete*, respectively. Updates are not supported – they require a trader to delete the original order and insert a new one instead. Inserting into the account view allows a trader to place a new order, and similarly, deletions allow the cancellation of existing orders. Because all modifications will be controlled by the triggers, they must ensure the following business rules:

B_1: When selling stocks, the amount given in the put order must be covered by the traders depot, i.e., to sell n shares, the trader must have at least n in his depot.

B_2: When buying stocks, the account balance of the trader must cover the total price ($amount \cdot limit$) of all outstanding orders. This means that, in order to buy 100 shares at a limit of 130 €, the trader must have at least 13 000 € in his account. Once this particular order is issued, further orders must take it into consideration as well. For example, if the starting balance was 15 000 €, a second call order must not surpass 2000 €.

B_3: A trader may place at most one order per stock.

B_4: Order amounts and limits must be non-negative integers (≥ 0), but they must not be all zero (an empty order).

B_5: Orders must be either of the call or the put kind, i.e., if *put_amount* and *put_limit* are given, then *call_amount* and *call_limit* must both be zero, and vice-versa. For the chosen order kind, both the limit and the amount must be non-negative integers.

B_6: The fields *amount*, *market_price*, and *market_value* must not be manipulated directly by traders, i.e., inserts must have them set to NULL or omit it.

An insertion that results in a violation of any of the above rules (i.e., an *inconsistent* state) must produce an error and, as a consequence, cause the transaction to abort. This can be done by raising error conditions with the DB2 statement `SIGNAL SQLSTATE`. The SQLSTATE error code must indicate which business rule was violated. As a convention, the state number must be 70000 + i, where i is the number of the business rule which was violated. A violation of rule B_4, for instance, must signal the state 70004. Our evaluation will check for these codes, so make sure to return the correct value, even if your trigger works as expected.

Violations of the business rules must be raised in the order they are listed here. If an order violates B_1 and B_3, for example, the error code must indicate a violation of B_1. You are only required to check the business rules on insertions in the *account_v* view. Operations that modify any of the base tables directly can be ignored.

Exercise 3: Calculate prices

After having specified the call and put behavior, the next step for our system is to re-calculate the stock market prices based on the current orders and the market prices. The summary of orders and limits provided by the view *orders_v* is the starting point for price calculation. The new price of a stock depends on its limits and volumes occurring in the orders book. The goal is to pick a price which allows the highest amount of shares to be traded, i.e., maximizes the trading volume. There are seven possible situations that can occur for any given stock. The rules below (P_1-P_7) and the examples specify how the new price is derived in each of them. The *amount* column shows the amount of shares of the limit, in order to emphasize the behavior of the volume as the price grows and shrinks. However, it is not present in *order_v*, because only the volumes are needed to compute the new price.

P_1: There is exactly one limit that yields the highest trading volume.

Call								**Put**
	Amount	Volume	Backlog	Limit	Backlog	Volume	Amount	
Calls	200	200		202	500	700		
Calls	200	400		201	300	700		
Calls	300	700		200		700	100	Puts
		700	100	198		600	200	Puts
		700	300	197		400	400	Puts

The price is fixed to this limit, here 200 €.

P_2: Several limits yield the highest trading volume and there is a call backlog.

Call								**Put**
	Amount	Volume	Backlog	Limit	Backlog	Volume	Amount	
Calls	400	400		202	100	500		
Calls	200	600	100	201		500		
		600	100	199		500	300	Puts
		600	400	198		200	200	Puts

The price is fixed to the highest limit among those that maximize the trading volume, here 201 €. Because the backlog indicates that there are more calls than puts, it means that the market is pushing the price up, and so the highest of the considered limits is picked.

P_3: Several limits yield the highest trading volume and there is a put backlog.

Call								**Put**
	Amount	Volume	Backlog	Limit	Backlog	Volume	Amount	
Calls	300	300		202	300	600		
Calls	200	500		201	100	600		
		500		199	100	600	400	Puts
		500	300	198		200	200	Puts

The price is fixed to the lowest limit among those that maximize the trading volume, here 199 €. This situation is the inverse of P_2, and it indicates that the market is pulling the price slightly down.

P_4: Several limits yield the highest trading volume and there is both a put backlog and a call backlog.

Call								**Put**
	Amount	Volume	Backlog	Limit	Backlog	Volume	Amount	
Calls	100	100		201	100	200		
		100		200	100	200	100	Puts
Calls	100	200	100	199		100		
		200	100	198		100	100	Puts

In this situation, there is no winning force pushing the price up or down, and so the decision favors the limit which is closest to the current market price, or, in other words, the most stable limit. This rule is implemented by the function closestToMarket(m, h, l), where m is the current market price, h is the highest limit (among those that yield the highest trading volume), and l the lowest limit.

$$\text{closestToMarket}(m, h, l) = \begin{cases} h & \text{if } m \geq h \\ m & \text{if } l < m < h \\ l & \text{if } m \leq l \end{cases}$$

Note that you do not have to implement this function in SQL. It is used here solely for explanation of the price calculation.

For the example above, we would have:

- If the current market price is ≥ 201 €, the price is fixed to 201 €.
- If the current market price is > 198 € and < 201 €, the market price remains unchanged.
- If the current market price is ≤ 198 €, the price is fixed to 198 €.

P_5: Several limits yield the highest trading volume without a backlog.

Call								**Put**
	Amount	Volume	Backlog	Limit	Backlog	Volume	Amount	
Calls	300	300		202	200	500		
Calls	200	500		201		500		
		500		199		500	300	Puts
		500	300	198		200	200	Puts

This situation is similar to P_4, and so we apply closestToMarket:

- If the current market price is ≥ 201 €, the price is fixed to 201 €.
- If the current market price is 200 €, the price stays at 200 €.

– If the current market price is ≤ 199 €, the price is fixed to 199 €.

P_6: There are both put and call orders but no trades are possible.

Call								**Put**
	Amount	Volume	Backlog	Limit	Backlog	Volume	Amount	
				202	200	200	200	Puts
Calls	100	100	100	199				

This situation is the opposite of P_4, and P_5, where there were multiple limits yielding the highest trading volume. Here, all limits would produce a volume of zero, i.e., no shares will be traded. Nevertheless, closestToMarket is applied to make the price move and maybe allow more orders in future periods:

– If the current market price is ≥ 202 €, the price is fixed to 202 €.

– If the current market price is < 202 € and > 199 €, the price remains unchanged.

– If the current market price is ≤ 199 €, the price is fixed to 199 €.

P_7: There are only call or put orders.

Call							**Put**
	Amount	Volume	Backlog	Limit	Backlog	Volume	Amount
Calls	100	100	100	200			

or

Call							**Put**
Amount	Volume	Backlog	Limit	Backlog	Volume	Amount	
			202	200	200	200	Puts

This situation is inconclusive, and therefore the market price remains unchanged.

Your task is to implement an SQL stored procedure *calculate_prices* which re-calculates the stock prices in the market table based on the rule set explained above. The market time of the processed stocks is increased by one for each call of this procedure, i.e., the *m_time* of all priced stocks is set to the global market time + 1. If a stock had no orders for it in *order_v*, its market time is not touched. The procedure must return a list of all stocks and their most recent market price ordered by name.

Exercise 4: Trade stocks

Finally, after having re-calculated the market price for each stock, the call and put orders have to be processed. Remember that, in case of backlogs, not all orders can be fully processed. Ensure that these orders stay in the *orders* table. Furthermore, an order can be only partially served if the remaining volume is less than the amount of shares in it. In this case, the order must stay in the system, but its amount has to be updated to the remaining shares that could not be served. For example, if there are 3 call orders for 100 shares each, and 2 put orders, one for 200 and the other for 60 shares, then one of the call orders (the most recent) must remain in the system with an updated amount of 40.

Orders are processed in first come-first served order based on the *o_id* attribute. Thus, the orders which remain unserved or are only partially served must be those that arrived last.

Your task is to implement an SQL stored procedure *trade*, which performs the trades for the current orders book, according to the rules just described. The procedure must return a list of all traders and their balance, ordered by trader name.

Submission

Submission deadline is Sunday, June 09, 2019 at 23:59:59.
Lots of fun and success!