

COMPARISON OF METHODS FOR VALIDATION OF EXCEEDED LIMIT VALUES IN BATCH DATA LOADING

Siniša Ilić

Faculty of technical sciences of University of Pristina, K. Mitrovica

Slobodan Obradović

Megatrend University, Belgrade

Abstract

In the large information systems there is often a need to copy data from other information systems via interface files. During the data files loading, data values need to be validated (checked) against constraints defined in the database that imports data. If the validation is performed on each record, the duration of batch loading is slower. At some DBMS there is a trigger function that enables validation of the whole batch at once, but in other DBMS that don't have this functionality we have used trigger on the last record to perform the same. In this paper the results of the measurements of the time needed for five DBMS to load and validate data by using table trigger - for every loaded record and after the last record in "for each row" and for every loaded record and whole batch in "for each statement" trigger.

Keywords: validation, batch data loading, databases, trigger, measurement of loading speed

INTRODUCTION

In majority of information system implementations, there is a need to enter data that are already stored in other information system not by re-typing, but via file interface. Interface enables exchanging of data in electronic manner. The process is likely simple: the data should be firstly exported from the source information system to the interface file of predefined format and secondly imported to the target information system. Interface files can be defined in various formats such as: ASCII (or plain text), CSV (comma separated values), XML and others. It is not enough to define only the fields, sequence of fields, widths of fields, separators or number of columns – one should be very carefully in defining the formats of fields, character sets, appearance of some values etc. For example, for date field it is important to define the separator (if it is a dot, slash or minus sign), date parts sequence (ddmmyyyy, mmddyyyy, ...), the allowed values (not dates of weekends, national holidays etc.).

When the data from interface data file is to be loaded into target information system, firstly the validation of data file format is to be exe-

cuted and afterward the validation of data values. The validation of interface file format is usually executed in client application that reads the interface file, but the validation of data values is executed in database system [1]. Sometimes it is very important and when (in which moment) data is loaded, because there are some limits in database that are changed in time as the impact of entering of other data that decrease these limits. As the consequence of decreased limits, data from interface files might be rejected.

BATCH LOADING SCENARIO

Let's assume that there are two tables in database: LIMITS and PAYMENT_REQUESTS (Figure 1).

The tables presented here are just simple examples of more complicate tables that are really used in payment system. Usually there are pretty much columns for different payment types, reconciliation, swift messages etc. Our goal was just to present problem and to offer the solution for the problem, so we have created very simple tables that we have used for testing.

limits							
payment_type	period_id	amount_limit	amount_committed				
1	1	100000.00	50000.00				
1	2	100000.00	15000.00				
1	3	100000.00	0.00				
1	4	100000.00	0.00				
1	5	100000.00	0.00				
1	6	100000.00	0.00				
1	7	100000.00	0.00				

payment_req							
payment_req_id	payment_type	due_date	amount	status	batch_id	flag_last	
5	1	1/25/2010	10.00	1	20	1	1
6	2	1/25/2010	5.00	1	20	0	0
7	3	1/25/2010	10.00	1	20	0	0
8	1	2/25/2010	5.00	1	20	0	0
9	1	1/25/2010	10.00	1	20	0	0
10	1	1/25/2010	10.00	1	20	0	0
11	2	1/25/2010	5.00	1	20	0	0
12	3	1/25/2010	10.00	1	20	0	0

Figure 1. Metadata of tables *limits* and *payment_requests*

The table LIMITS from Figure 1 consists of „budget lines” or limits per payment type and per period. Payment type in this case means if payment is for salary, for consumables, for investment, for re-payment of debt etc. Period might be the month in the year, the quarter, the week etc. The table PAYMENT_REQUEST consists of payments that are waiting to be executed at due date. The column “status” is for evidence if the payment is waiting in the cue to be paid (1), sent to the payment system (2), paid (3), rejected from the payment system (4) etc. The flag_last column is the column that we added in order to distinct if the record from the batch is first (1 or 3), intermediate (0) or the last (2 or 3). If the batch consists of only one record, the record is the first and the last in the same time (3).

In order to be understandable, suppose that payment_type 1 is code for consumables, and period_id = 1 means first month – January. We can see in the first record of table LIMITS that there is allocated \$100,000 for consumables in January, and that that \$50,000 is already committed. If we insert the payment request with payment_req_id = 5 from table PAYMENT_REQUESTS (from Figure 1), to buy a cartridge for ink jet, with amount \$10 and due date 25.01.2010, the database system has to check if there is enough un-committed money in January for payment type 1. We can see that we can insert that payment request, because there is $100,000 - 50,000 = 50,000$ available for spending.

The previous example shows a typical scenario when payment request is entered into database in one-by-one order by operator through a client application. But, when the several thousands or more payment requests are prepared for loading, the validation of these requests (records in the database) need a time for each request to be checked against limits and afterwards to update committed amounts. That time can be very long in the case when several hundreds or thousands of concurrent users are connected to information system and when several of them are loading their interface data files.

If more than one users access the same budget line in table LIMITS, it is very important which of them access the table first. It might happen that one of them inserts the last request (because the available amount was enough for it) and other is rejected because the committed amount is equal to limited and that no new request can be loaded.

Usually the data from interface data file is to be loaded according to the all/but nothing principle. If one of the records is rejected in validation process, all already loaded records from the same batch are purged through the rollback process.

In order to speed up the loading process, there is the alternate scenario to load all data from interface file to temporary tables without validation, and afterwards to validate aggregate amounts against budget lines and copy data from temporary tables to PAYMENT_REQUESTS.

If one takes a look on table PAYMENT_REQUESTS on figure 1, he can notice that 8 lines may be gathered to 4 lines grouped by payment type and month of due dates. Then in the validation process these 4 lines can be checked against appropriate 4 lines in table LIMITS in one command.

DATABASE TRIGGER TYPES

The best place for execution of data validation in multi-user environment is database trigger [1]. The program code written in the trigger on some table becomes active (is executed) when some change in the table records is occurring: when the new record is inserted, updated or deleted. Unlike in other objects in database (procedures, packages, functions etc.) that can be executed simultaneously by more users, trigger is executed serially – more than one user cannot change the contents of table at

once in the same time, only one after another.

There are two types of trigger behavior dependent on number of records that can be accessed within the body of trigger: “for each row” and “for each statement” [2, 3, 4, 5, and 6]. In “for each row” trigger one can access to the new values of all columns for new record (for operations insert and update) and old values of deleted record (for operations update and delete) in trigger body. This behavior is standard when the records in a table are inserted, updated or deleted as the result of execution of one single SQL command. But, when the result of execution of one SQL command is insertion, update or deletion of several records, system enters in the trigger as many times as many records are tackled by that command. Each time the values in the “new” and “old” record match the values of appropriate records that are changed in the table. The more practical solution is given in “for each statement” trigger where one can access to all records tackled by one SQL command in the body of trigger. Two tables are then generated in the body of trigger: new – with new (inserted) records and old – with old (deleted) records. In order to be clear, the update process consists of two sub-processes: deletion of old values followed by insertion of new values.

Today there are many RDBMS that support triggers as database objects, but not all of them have the same properties. Some DBMS support only one trigger type (“for each row”: MySQL, PostgreSQL, Oracle, “for each statement”: MS SQL) and some of them support both types (DB2). Some DBMS don’t allow access to other table records in the body of trigger except for the record that is currently accessible (Oracle).

DEFINITION OF TASK

Taking into account all possible scenarios for data loading and validation, and trigger behavior and types, we tried to compare times needed for data loading and validation: 1) when data records are inserted and validated one by one and 2) when all data records are loaded firstly without validation and afterwards the validation on all inserted records is executed at once.

Also, we have tried to compare the time needed to load and validate data in batch loading process for 5 chosen DBMS: MySQL, PostgreSQL, Oracle, DB2 and MS SQL [7].

ENVIRONMENT OF EXPERIMENT EXECUTION

Data needed for batch loading into table PAYMENT_REQUEST is prepared in four interface files that consist of: 100, 1000, 10000 and 100000 records. The values in these files are distributed in the way to charge only 5 budget lines in table LIMITS uniformly by number of records. The LIMITS table is already filled with 128 different payment types and with 12 different periods per each type – that means 1536 records.

All before-mentioned DBMS are installed on 64 bit operating systems. Oracle 11gR1, MySQL 5.1, PostgreSQL 8.4 and DB2 Express-C 9.7 are installed on Ubuntu Linux 10.4 LTS and MS SQL 2008 Express is installed on MS Windows Vista on computer with AMD Athlon 64 X2 dual core processor 5000+, with 8 GB of RAM Memory and 500 GB HDD. During the installation process of all DBMS default settings are used. One stored procedure and one validation trigger are created for every DBMS using the appropriate DBMS SQL syntax. The computer was connected to 100Mb LAN.

The data from interface file was read by MS Windows application on another computer created in VB .NET 2008 64-bit environment (hereinafter - the application). The connections to DBMS are made through appropriate ADO .NET drivers. For Oracle and DB2 we had to use client connection components too: Oracle instant client and IBM data server client – db2copy1.

Both computers are connected to a single network switch with only two active ports, so the bitrate between these computers was guaranteed.

On all DBMS three tables are created: LIMITS, PAYMENT_REQUESTS and TMP_PAYMENT_REQUESTS. The table TMP_PAYMENT_REQUESTS has the same number of fields with the same data types. The purpose for creating of this table is to store loaded data from interface data file without data validation. Hereinafter we will consider this table as temp table. The validation process is implemented in a trigger body of PAYMENT_REQUESTS table.

The above-mentioned scenarios for batch data loading are realized by three different physical scenarios:

1) Application sends to DBMS command for begin of transaction and as many INSERT INTO PAYMENT_REQUESTS commands as there are records in the interface file (assume

the number is N). For each INSERT command, DBMS executes validation and updates amount_committed value in appropriate budget line in LIMITS table. After successful validation of all records, application sends commit command to DBMS, but if any of records fails in validation process, the application generates rollback command. This physical scenario corresponds to first before-mentioned scenario and is applied to all DBMS;

2) Application sends to DBMS INSERT INTO TEMP_PAYMENT_REQUESTS (temp table) records. DBMS does not execute validation on data in tmp table. After all records from file is loaded into tmp table, application calls stored procedure sp_insert_into_pr which copies all records from temp table to PAYMENT_REQUESTS table by one SQL command. DBMS that support only “for each row” trigger ignores N-1 entrance into trigger body, and when process enters into trigger body for Nth record, the values from temp table are aggregated and validated against the appropriate lines in LIMITS table. We can see that validation is executed once for all records in one entrance in trigger body. If validation is successful, the procedure executes commit command, and if validation fails the procedure executes rollback command. This physical scenario corresponds to second before-mentioned scenario and is applied to Oracle, MySQL and PostgreSQL;

3) This physical scenario begins as previous scenario and is executed in the same manner up to the call of procedure sp_insert_into_pr. DBMS that support “for each statement” trigger enters into body of trigger only once and can access to all batch records that are copied from temp table. These records are stored in NEW table in trigger body. After aggregation of values from NEW table the comparison against appropriate budget lines in table LIMITS is executed. We can see that validation is executed once for all records in one entrance in trigger body. If validation is successful, the procedure executes commit command, and if validation fails the procedure executes rollback command. This physical scenario corresponds again to second before-mentioned scenario and is applied to MS SQL and DB2;

RESULTS AND DISCUSSION

The times needed for batch loading at five chosen DBMS are presented at the tables 1, 2,

3, 4 and 5. In the application we have created, in scenario 1 we have started the stopwatch just before “begin transaction” command (moment *start*) and have stopped immediately after the “commit” command (moment *stop*). But, in scenarios 2 and 3 the clock has been started before first INSERT command has been sent (moment *start*), the *pass* moment has been captured after the last INSERT command and the stopwatch has been stopped (moment *stop*) after stored procedure has returned success as return value.

The column “initial number (#) of records” presents the number of records that were initially in the table of payment requests before the batch is loaded. The “number (#) of records” denotes the actual number of records that were loaded in the batch. As we stated before, we have physically three types of scenarios: one-by one validation (Scenario 1), group validation on last record through the “for each row trigger” (Scenario 2/2) and group validation through the „for each statement trigger“ (Scenario 2/3). In the last two scenarios we have measured and the time needed for transferring data from client application to temp table without validation (the column “load to temp” calculated as *time=pass-start*) and it has helped us to determine the time needed just for validation of data (as the difference between two last columns, calculated as *valid_time=stop-pass*). By using scenario 1 it was impossible to measure just validation time, because it was incorporated in the total time (*tot_time=stop-start*).

The comparisons of times needed for batch loading and validations is performed on data files of 100, 1000, 10000 i 100000 records. Two cases are studied separately: 1) when the database was empty and 2) when there was already some previous data in database (100000 records).

Table 1 – the results of batch loading times for MySQL DBMS

Initial # of records	# of records	Scenario 1	Scenario 2/2	
			Load to temp	Completed
0	100	0.178	0.081	0.166
0	1,000	1.285	0.514	0.673
0	10,000	12.914	5.202	5.718
0	100,000	130.652	49.660	54.790
100,000	100	0.166	0.091	0.283
100,000	1,000	1.699	0.536	0.819
100,000	10,000	13.041	4.833	5.443
100,000	100,000	133.088	50.350	56.150

Table 2 – the results of batch loading times for PostgreSQL DBMS

Initial # of records	# of records	Scenario 1	Scenario 2/2	
			Load to temp	Completed
0	100	0.217	0.098	0.163
0	1,000	1.273	0.525	0.588
0	10,000	21.177	5.167	5.444
0	100,000	698.96	51.763	55.203
100,000	100	0.142	0.052	0.141
100,000	1,000	1.290	0.518	0.628
100,000	10,000	21.490	5.071	5.774
100,000	100,000	699.10	51.800	56.677

Table 3 – the results of batch loading times for Oracle DBMS

Initial # of records	# of records	Scenario 1	Scenario 2/2	
			Load to temp	Completed
0	100	0.186	0.108	0.142
0	1,000	3.421	1.037	1.177
0	10,000	36.144	22.596	23.109
0	100,000	324.286	266.404	275.480
100,000	100	0.184	0.107	0.142
100,000	1,000	2.109	1.088	1.198
100,000	10,000	36.887	28.979	29.923
100,000	100,000	358.272	285.958	297.368

As we have assumed at the beginning, the time needed for batch loading with one-by-one validation was longer than the time needed for loading with group validation. The difference in time was negligible for loading of 100 records, and obvious for 1000 records and more. The ratios vary from 1.007 for loading of 100 records after initializing table with 100,000 records for DBMS PostgreSQL to 12.661 for loading of 100,000 records in empty table for the same DBMS. The smallest variances in ratios are noticed for MS SQL Server (1.073 to 1.458).

We have noticed that for DB2 the ratio was between 7.5 and 8.6 and that the times needed for all scenarios were higher than to other DBMS. We have tried to obtain more results than it was shown at the table 4, but the DBMS has reported deadlock problems.

For batch loading with the highest number of records, the shortest time is achieved by MySQL DBMS (54.79 s with group validation), and the longest measured time is achieved by PostgreSQL (11 min, 39s with one-by-one validation). As we have explained we haven't

performed measurement with 100,000 records on DB2 DBMS.

At MySQL DBMS, the ratio was more than 2 for 100000 records (133 seconds compared to 56 seconds).

The process of loading and validation by using one call to database trigger was faster, according to the expectations, and for commercial DBMS.

According to the results achieved, it can be seen that the data that are already present in database does not have almost any influence on loading speed. As it is shown in tables, the difference of appropriate time of loading when database is empty compared to time when there are some data in database is very small.

Table 4 – the results of batch loading times for DB2 DBMS

Initial # of records	# of records	Scenario 1	Scenario 2/3	
			Load to temp	Completed
0	100	3.699	0.412	0.487
0	1,000	34.740	3.882	4.037
0	10,000	350.499	45.491	45.911

We have also noticed that the time used for validation of aggregate values (one call to database trigger) is very small compared to the time needed for the transfer of data to the database for all DBMS. The validation time was about 10% compared to the transfer time at MySQL and PostgreSQL, 5% at Oracle DBMS and just 1% at DB2 and MS SQL when number of records was large (10000 and 100000 records).

Table 5 – the results of batch loading times for MS SQL DBMS

Initial # of records	# of records	Scenario 1	Scenario 2/3	
			Load to temp	Completed
0	100	0.274	0.172	0.323
0	1,000	1.607	1.380	1.428
0	10,000	15.313	13.707	13.875
0	100,000	192.425	136.769	140.129
100,000	100	0.245	0.134	0.168
100,000	1,000	1.714	1.468	1.596
100,000	10,000	15.474	13.685	13.857
100,000	100,000	154.939	139.128	140.487

The differences in times from DBMS to DBMS are not the topic of this paper, so we haven't commented these differences.

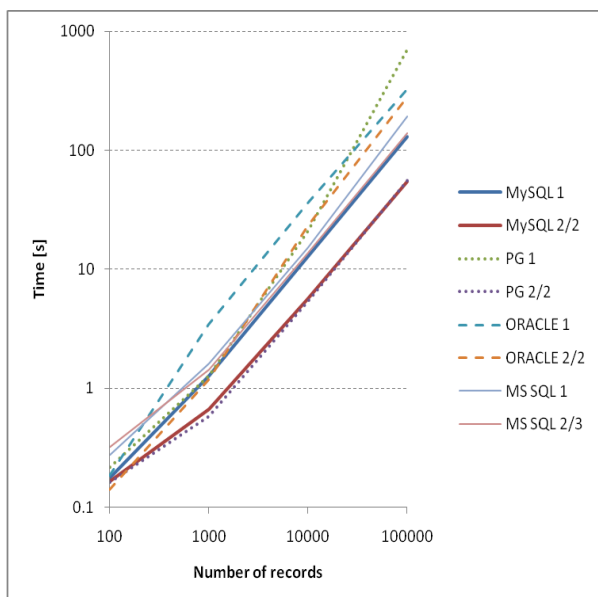


Figure 2. Comparison of time needed for batch loading

CONCLUSION

When transferring data between two information systems, it is necessary to check the accuracy of the data. This means checking the data formats and verification of data values. Both verifications are usually performed by client application. But when the data to be entered in database depend on some values that are set in a database (as it is in budgeting systems), the verification is performed at DBMS. The data packet with correct values might overhead the budget and becomes faulty. Transaction like that must be rejected and information on budget violation must be transferred to users.

In general, the verification of aggregate values (sums) against limits can be performed in two ways: after insertion of each record and after insertion of whole group of records that belong to same batch.

Some DBMS (MS SQL Server, DB2) have the ability to execute aggregated checking of data group in database trigger and others don't. At DBMS that don't have ability like this the verification is executed on each record and it slows down the performance of database.

In order to overcome lack of mentioned ability at second group of DBMS, we have created the procedure that activates the database trigger only after insertion of last record from batch and performs verification on aggregate values. We have tested the time savings obtained by using of verification on group of data.

Achieved results confirm the assumption that aggregate verification of group of records against the limits is considerably faster than

verification of each record separately. The testing results show at all DBMS that the time needed for aggregate verification is smaller. The ratio differs and depends on number of records in a group, on DBMS and on number of records that are already stored in database. For large number of records in a group we have got at postgresQL 13 times faster validation on aggregated data against the validation on each record separately. The benefit obtained on Oracle is not so large (only 20%), but it exists.

This suggests that the creation of procedures for the group validation is necessary and that the effort pays off.

We have simplified scenarios for batch data loading to be understandable, by using two very simple tables with only two columns whose values need to be matched between them (joins of period_id and payment_type) and one column to be verified (amount against the amount_available). In practical case there are more tables involved in batch loading validation with large number of columns (general ledger account, bank account, date of commitment, expected and executed date of payment, date of reconciliation, textual description of payment etc.). The time needed for transferring data from source to database is proportional to the size (in bytes) of data, but the validation time is proportional only on number of columns to be compared and number of columns to be matched (in joins with an assumption that join columns are indexed).

The real behaviour on how variations of data types and number of columns to be inserted and verified impact on transfer and validation time is a subject to be experimentally studied in the future work of authors.

REFERENCES

- [1] Siniša Ilić and Slobodan Obradović „Efficient validation of exceeded limit values in batch data loading by database trigger“, International Scientific Conference Unitech 09 Gabrovo, Bulgaria
- [2] Jason Price, Oracle Database 10g SQL, McGraw-Hill/Osborne 2004.
- [3] Paul Nielsen, SQL Server 2005, Wiley Publishing, 2007.
- [4] Steven Feuerstein, Guy Harrison, MySQL Stored Procedure Programming, O'Reilly, 2006.
- [5] Neil Matthew and Richard Stones, Beginning Databases with PostgreSQL, Apress, 2005.
- [6] Grant Allen, Beginning DB2: From novice to professional, Apress, 2008.
- [7] Slobodan Obradovic, Sasa Slavnic and Anton Donchev: Comparison of the Free DBMS Alternatives for MS SQL, International Scientific Conference Unitech 08 Gabrovo, Bulgaria.

