

Deriving Accounts Receivable In An Events-Based, Relational Database System

Alan S. Levitan, (E-mail: levitan@louisville.edu), University of Louisville

Abstract

Database designers face certain challenges in seeking to derive accounting numbers, such as accounts receivable, from an events-based relational database system. An ideal solution would record raw events uniquely in events tables, where the data are nonredundant and normalized. The actual implementation of this solution is not trivial, but it can be a rewarding exercise.

Introduction

With the continuing popularity of relational database platforms, accounting systems designers are challenged to use this tool to its full advantage. In addition, the adoption of events-based information systems requires careful data modeling to capitalize on all of the potential from the events approach.

Events-based systems collect relevant data about business events and leave these raw data in tables. The tables, in turn, can later be aggregated, queried, combined, and summarized in many different ways, to satisfy the view of any authorized information customer. The theory underlying this resource-event-agent modeling approach was first described by William McCarthy (1982).

In the Revenue, or Sales/Collection, business cycle, for example, one would record sales (shipment) events as they occur. Similarly,

Readers with comments or questions are encouraged to contact the authors via e-mail.

one would record cash collection events as they occur. Then, accounts receivable may be derived at any time by subtracting information in the cash collection table from information in the sales table.

This approach, now a fundamental construct for a textbook by Hollander, Denna, and Cherrington (1996), offers many advantages over traditional architecture. With data maintained in the tables in primitive form, the underlying facts are always available. New tables need not be created to summarize the data into journals, ledgers, trial balances, or financial statements. Yet, these traditional forms of documentation exist in virtual form. They may be created from the tables at any time with the use of queries, forms, and reports. This eliminates useless redundancy, retranscription and summarization, which were valuable for catching mistakes in manual processing days but which today have costs in excess of benefits. Storage space and human effort can be saved. And a change made in any table need be made only in that one place. Any query, form, or report

which uses that table will reflect the new data immediately. (Of course, proper controls over the recording of data are essential.)

The disaggregated data repository, then, supports multidimensional user views of the data. Unlike traditional architecture, in which aggregations are directed toward the financial view only, the raw data may be integrated in response to any information customer, financial, managerial, or other. The underlying data in the tables are stored only once, providing consistent information in support of cross-functional teams.

The system eliminates unnecessary, excessive summarization of data. More importantly, however, the data in its primitive form permit a wide variety of reporting options. For example, not only can the system produce current accounts receivable but, through selection criteria in the queries, it can also produce a value for accounts receivable at 2:15 pm last Thursday. Or, to track lost or suspicious activity, it can produce a list of cash collected between 9 am last Monday and 1 pm on Tuesday. The objective, then, of recording and storing event data once and maintaining them in raw form is worth pursuing.

Implementation Issues

While the theory is compelling, its implementation can be challenging. Using the popular relational database system Microsoft Access, the solution is not trivial. But a systematic realization of that solution is important, and is described in this article.

Laurence Paquette (1998) proposed one inventive method for solving this problem. He used a cash Receipts table, which contained a record for each receipt, and a Customer table, which contained a debit field and a credit field for each customer. This Customer table is essentially an accounts receivable subsidiary ledger. Consistent with sound design standards on user interfaces, he created a form, bound to

the Receipts table, for data entry into that table. Then, by creating Access Basic code using an After Update event procedure for the form, any addition of a record in the Receipts table via the form would trigger the update of the credit field in the Customer table. (Presumably, there would be a similar event procedure to update the debit field in the Customer table as records are added to a Sales table.)

In this article, I will propose an alternative method for reporting accounts receivable, one which is more compatible with database design principles about redundant data. Recognizing this principle, Paquette properly did not include a "balance" field in his Customer table, which would have been computed as the debit field minus the credit field, because, as he correctly asserted, "Good database design specifies that such fields be omitted from the table reducing the chances of inconsistencies from occurring in the future" (*ibid*, p. 45).

This principle is true, and can be carried further. To eliminate more thoroughly the possibility of future inconsistencies, it would have been preferable not to have the basic data on cash receipts stored in two separate tables. That is, the inclusion of the dollar value of a cash receipt stored in the Receipts table, and also summarized in the Customer table, violates the principle of normalization. Unnormalized databases may lead to the storage of redundant, inconsistent, and anomalous information within tables (Perry and Schneider, 1999, p. 119). A better solution will not have an accounts receivable subsidiary ledger. Rather, such a ledger may be derived virtually, from events tables.

Database designers must avoid potential inconsistencies, known as update anomalies, that may arise as a result of not changing every occurrence of an item of data (Romney, Steinbart, and Cushing, 1997, p. 142). A commonly quoted characteristic of the database approach is the reduction of redundancy, and the consequent decrease in inconsistencies, processing time and

storage requirements, along with the enhancement of data integrity (Wilkinson and Cerullo, 1997, p. 201). While not denying the usefulness of the skills learned in the Paquette approach, this article will propose the calculation of accounts receivable with data uniquely stored in their tables by developing a sequence of queries upon those tables.

Design of the Tables

In Microsoft Access, or any software using the relational model, tables are the funda-

mental storage entities for all data. For the accounts receivable application, five tables are proposed. Their minimal configuration is illustrated in Figure 1.

The Customer Master table contains the relatively permanent information about each customer, the external agent in this resource-event-agent model. Possible additional attributes for this table would include city, state, zip, and credit limit.

There are two tables necessary to de-

Figure 1

Customer Master			
Customer ID	Customer Name	Customer Address	
11	ABC Co.	123 First St.	
12	DEF Co.	234 Second St.	

Invoices			
Invoice ID	Customer ID	Date	Shipping Cost
1	11	8/15	\$30.00
2	11	8/20	\$45.00
3	12	8/21	\$10.00

Invoice Details			
Invoice ID	Product ID	Quantity	Price per Unit
1	200	20	\$2.00
1	300	30	\$3.00
2	50	2	\$3.00
2	100	5	\$4.00
3	10	5	\$3.00
3	399	4	\$2.00
3	499	4	\$5.00

Payments		
Cash Receipt Number	Customer ID	Date Received
1	11	9/2
2	11	9/4

Payment Details		
Cash Receipt Number	Invoice ID	Amount Applied to this Invoice
1	1	\$150.00
2	1	\$10.00
2	2	\$6.00

scribe the sales event and two for the payment event. This is required to implement the many-to-many maximum cardinalities in the model. That is, a sale could relate to many inventory items, and an inventory item could be represented on many invoices. Similarly, one payment could cover many invoices, and an invoice could be paid in many installment payments. Like the Customer Master table, these event tables could contain additional attributes, which would vary with different enterprises. The Invoice Details table would have a concatenated primary key consisting of both the Invoice ID and the Product ID, while the concatenated primary key for the Payment Details table would be the Cash Receipt Number and the Invoice ID.

A series of one-to-many relationships would be implemented in the database. The Customer ID in the Customer Master table would be related to its match in the Invoices table and in the Payments table. The Invoice ID in the Invoices table would be related to its match in the Invoice Details table and in the Payment Details table. Finally, the Cash Receipt Number in the Payments table would be dragged and dropped onto its match in the Payment Details table.

To facilitate data entry, as well as data display, forms could be created using the form-with-subform approach. For example, payments could be recorded in a form with the Payment table fields at the top as the main form, and a subform within the main form containing the Payment Details records related to that cash receipt. This would add records to both tables at once, with referential integrity enforced if properly requested when the relationships were established. In a similar manner, the form-with-subform design could be used to record sales into

the Invoices and the Invoice Details tables.

Calculations With Queries

Once the tables are in place, queries can be created, using query-by-example (QBE), to perform all necessary calculations. The first query would be Invoice Extension, which would create a dynaset from the Invoice Details table, with one row per invoice, showing the sum of the price times quantity for all items ordered on the invoice, as illustrated in Figure 2.

Invoice ID	Extension
1	\$130.00
2	\$26.00
3	\$43.00

In the QBE grid for this query, the first column would be the Invoice ID from the Invoice Details table, with "Group By" indicated on the Total line of the grid. The second column would be an expression, defined as follows:

$$\text{Sum}([\text{Quantity}] * [\text{Price per Unit}])$$

Next, an Invoice Total query could be built, based on the Invoices table and the just-created Invoice Extension query dynaset, related to each other on Invoice ID. The results of this query are shown in Figure 3.

The first three fields in this Invoice Total query (Customer ID, Invoice ID, and Ship-

Customer ID	Invoice ID	Shipping Cost	Extension	Total
11	1	\$30.00	\$130.00	\$160.00
11	2	\$45.00	\$26.00	\$71.00
12	3	\$10.00	\$43.00	\$53.00

ping Cost) are selected from the Invoices table, while the Extension attribute is selected from the Invoice Extension query dynaset. The final field, Total, is an expression built as follows:

Total: [Extension]+[Shipping Cost]

A query of total invoices by customer, showing one line per customer, could easily be created. Alternatively, a report could be used to show totals by customer. Since this example uses the report alternative, we will now direct our attention to the payments made against these invoices.

From the Payment Details table, a query will summarize Payment Totals by Invoice. The query view is shown in Figure 4.

Invoice ID	SumofAmount Applied to this Invoice
1	\$160.00
2	\$6.00

The Invoice ID field is pulled from the Payment Details table, with "Group By" on the Total line of the QBE grid. The Amount Applied to this Invoice is likewise pulled, but Sum is entered on the Total line for this column.

The final query is the Accounts Receivable one. It computes the difference between the amounts in the Invoice Total query and the Payment Totals by Invoice query and can be seen in

Figure 5.

The Invoice Total and the Payment Totals by Invoice queries are linked by Invoice ID. However, the link within this query must be changed from the default type 1 join into a type 2 join. This is accomplished by double-clicking on the line connecting the two Invoice IDs. The change is necessary because a type 1 join will show only those rows where the joined fields in both dynasets are equal. Thus, it would not show the last invoice, for which there is no matching payment. A type 2 join, on the other hand, will show all the rows in the Invoice Total dynaset, along with any matching payments.

The Still Due column is an expression built with some care:

Still Due: [Total] - Nz([SumofAmount Applied to this Invoice])

The Nz function is required to make sure that a zero is returned when a variant is null. Without it, the Still Due amount for the last invoice, for which there is no payment, would be null instead of the correct value, the full amount of the invoice.

Presenting Results in a Report

The project can be completed, with attractive output, in a report. An example is shown in Figure 6.

This report is based on the Accounts Receivable query, along with the Customer Master table for retrieving the customer names. The report groups on Customer ID. In the Cus-


Customer ID	Invoice ID	Total	Sum of Amount Applied to this Invoice	Still Due
11	1	\$160.00	\$160.00	\$0.00
11	2	\$71.00	\$6.00	\$65.00
12	3	\$53.00		\$53.00

Figure 6
Accounts Receivable Report

Customer Name	Invoice ID	Invoice Total	Amount Applied	Still Due
ABC Co.				
	1	\$160.00	\$160.00	\$0.00
	2	\$71.00	\$6.00	\$65.00
ABC Co. Total		\$231.00	\$166.00	\$65.00
DEF Co.				
	3	\$53.00		\$53.00
DEF Co. Total		\$53.00		\$53.00
Grand Total		\$284.00	\$166.00	\$118.00

customer ID group footer, text boxes for computing the three sums are created. Those same text boxes can be copied and pasted into the report footer to print the grand totals.

Summary

The calculation and presentation of accounts receivable in an events-based relational database system is neither straightforward nor trivial. The solution proposed in this paper is consistent with sound database design principles of normalization and minimized redundancy. It requires queries, queries of queries, and a report. However, once these objects have been created, the system will allow the one-time entry and storage of raw business events which may then be aggregated to satisfy the views desired by various information customers. 

References

- Hollander, Anita S., Eric L. Denna, and J. Owen Cherrington, *Accounting, Information Technology, and Business Solutions*, Richard D. Irwin, Chicago, Illinois, 1996.
- McCarthy, William E., "The REA Accounting Model: A Generalized Framework for Accounting Systems in a Shared Data Environment," *The Accounting Review*, Vol. 57, No. 3, pp. 554-577, 1982.

- Paquette, Laurence R., "Database Updates in Access," *The Review of Accounting Information Systems*, Vol. 2, No. 4, pp. 45-50, 1998.
- Perry, James T. and Gary P. Schneider, *Building Accounting Systems Using Access 97*, Third Edition, South-Western College Publishing, Cincinnati, Ohio, 1999.
- Romney, Marshall B., Paul J. Steinbart, and Barry E. Cushing, *Accounting Information Systems*, Seventh Edition, Addison-Wesley, Reading, Massachusetts, 1997.
- Wilkinson, Joseph W. and Michael J. Cerullo, *Accounting Information Systems: Essential Concepts and Applications*, Third Edition, John Wiley & Sons, New York, 1997.