Derived account balance vs stored account balance for a simple bank account?

Asked 8 years, 1 month ago Modified 5 months ago Viewed 24k times

70

Like our normal bank accounts we have a lot of transactions which result in inflow or outflow of money. The account balance can always be derived by simply summing up the transaction values. What would be better, storing the updated account balance in the database or re-calculating it whenever needed?

Expected transaction volume per account: <5 daily.

Expected retrieval of account balance: Whenever a transaction happens and once a day on an average otherwise.

database database-design derived-table

Share Follow

edited Dec 11, 2022 at 5:48
philipxy
14.8k 6 38 81

asked Apr 17, 2015 at 2:04

Anmol Gupta
2,729 9 26 43

2 Answers

Sorted by: Highest score (default)

203

Preface

There is an objective truth: Audit requirements. Additionally, when dealing with public funds, there is Legislature that must be complied with.

You don't have to implement the full accounting requirement, you can implement just the parts that you need.

Conversely, it would be ill-advised to implement something *other than* the standard accounting requirement (the parts thereof) because that guarantees that when the number of bugs or the load exceeds some threshold, or the system expands,you will have to re-implement. A cost that can, and therefore should, be avoided.

It also needs to be stated: do not hire an unqualified, un-accredited "auditor". There will be consequences, the same as if you hired an unqualified developer. It might be worse, if the Tax Office fines you.

Method

The Standard Accounting method in not-so-primitive countries is this. The "best practice", if you will, in others.

This method applies to any system that has similar operations; needs; historic monthly figures vs current-month requirements, such as Inventory Control, etc.

Consideration

First, the considerations.

1. Never duplicate data.
   If the `Current Balance` can be derived (and here it is simple, as you note), do not duplicate it with a summary column.
   - Such a column is a duplication of data. It breaks Normalisation rules.
   - Further, it *creates* an `Update Anomaly`, which otherwise does not exist.
2. If you do use a summary column, when a new `AccountTransaction` is inserted, the summary column `Current Balance` *value* is rendered obsolete, therefore it must be

updated all the time anyway. That is the consequence of the `Update Anomaly`. Which eliminates the value of having it.

3. External publication.
   Separate point. If the balance is published, as in a monthly Bank Statement, such documents usually have legal restrictions and implications, thus that published Current Balance value must not change after publication.
   - Any change, after the publication date, in the database, of a figure that is published externally, is evidence of dishonest conduct, fraud, etc.
     - Such an act, attempting to change published history, is the hallmark of a novice. Novices and mental patients will insist that history can be changed. But as everyone should know, ignorance of the law does not constitute a valid defence.
   - You wouldn't want your bank, in Apr 2015, to change *the* Current Balance that they published in their Bank Statement to you of Dec 2014.
   - That figure has to be viewed as an Audit figure, published and unchangeable.
4. To correct an erroroneous `AccountTransaction` that was made in the past, that is being corrected in the present, the correction or adjustment that is necessary, is made as a new `AccountTransaction` in the current month (even though it applies to some previous month or duration).
   - This is because that applicable-to month is closed; Audited; and published, because one cannot change history after it has happened and it has been recorded. The only *effective* month is the current one.
   - For interest-bearing systems, etc, in not-so-primitive countries, when an error is found, and it has an historic effect (eg. you find out in Apr 2015 that the interest calculated monthly on a security has been incorrect, since Dec 2014), the value of the corrected interest payment/deduction is calculated today, for the number of days that were in error, and the sum is inserted as a `AccountTransaction` in the current month. Again, the only effective month is the current one.

     And of course, the interest rate for the security has to be corrected as well, so that that error does not repeat.
   - The same principles apply to Inventory control systems. It maintains sanity.
5. All real accounting systems (ie. those that are accredited by the Audit Authority in the applicable country, as opposed to the mickey mouse "packages" that abound) use a **Double Entry Accounting** system for all `AccountTransactions`, precisely because it prevents a raft of errors, the most important of which is, funds do not get "lost". That requires a General Ledger and Double-Entry Accounting.
   - You have not asked for that, you do not need that, therefore I am not describing it here. But do remember it, in case money goes "missing", because that is what you will have to implement, not some band-aid solution; not yet another unaccredited

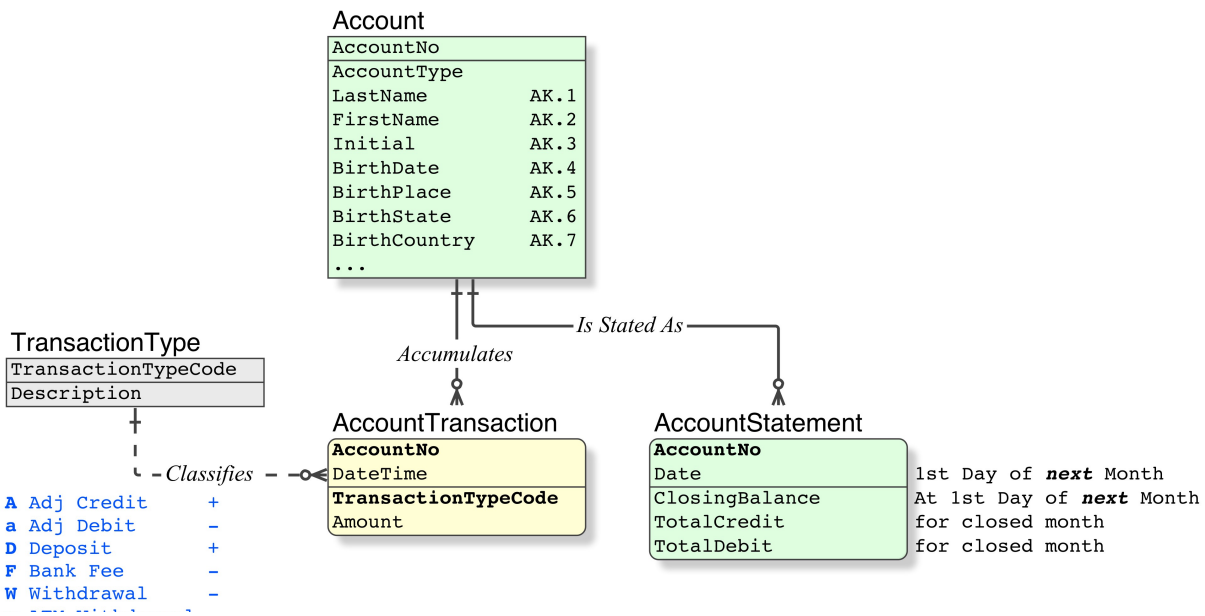
"package".

This Answer services the Question that is asked, which is **not** Double-Entry Accounting.
For a full treatment of that subject (detailed data model; examples of accounting Transactions; rows affected; and SQL code examples), refer to this Q&A: **Relational Data Model for Double-Entry Accounting**.

6. The major issues that affect performance are outside the scope of this question, but to furnish a short and determinant answer: it is dependent on:
   - Whether you implement a genuine Relational Database or not (eg. a 1960's Record Filing System, which is characterised by `Record IDs`, deployed in an SQL container for convenience).
   - whether you us a genuine SQL Platform (architected; stable; reliable; SQL-compliant; OLTP; etc) or the pretend-SQL freeware (herd of programs; ever-changing; no complaince; scales like a fish.
   - The use of genuine Relational Keys, etc, will maintain high performance, regardless of the population of the tables.
   - Conversely, an RFS will perform badly, they simply cannot perform. "Scale" when used in the context of an RFS, is a fraudulent term: it hides the cause and seeks to address everything but the cause. Most important, such systems have none of the Relational Integrity; the Relational Power; or the Relational Speed, of a Relational DBMS.

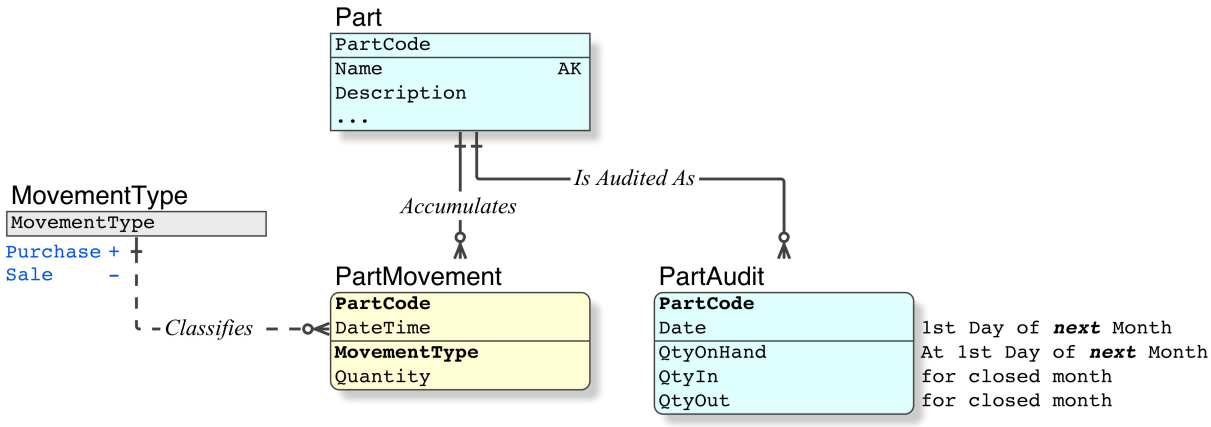
Implementation

Relational Data Model • Bank Account



```
w ATM Withdrawal -
```

Relational Data Model • Inventory



Notation

- All my data models are rendered in **IDEF1X**, the Standard for modelling Relational databases since 1993.
- My **IDEF1X Introduction** is essential reading for those who are new to the *Relational Model*, or its modelling method. Note that IDEF1X models are rich in detail and precision, showing all required details, whereas home-grown models have far less than that. Which means, the notation has to be understood.

Content

1. For each `AccountNo`, there will be one `AccountStatement` row per per month, with a `ClosingBalance`; Statement `Date` (usually the first day of the next month) and other Statement details for the closed month.
   - This is not a "duplicate" or a derivable value that is stored because (a) the value applies to just one `Date`, (b) it is demanded for Audit and sanity purposes, and (c) provides a substantial performance benefit (elimination of the SUM(all transactions)).

     For Inventory, for each `PartCode`, there will be one `PartAudit` row per month, with a `QtyOnHand` column.
   - It has an additional value, in that it constrains the scope of the Transaction rows required to be queried *to* the current month
     - Again, if your table is Relational and you have an SQL Platform, the Primary Key for `AccountTransaction` will be (`AccountNo`, Transaction `DateTime`) which will retrieve the Transactions at millisecond speeds.
     - Whereas for a Record Filing System, the "primary key" will be

`AccountTransactionID` , and you will be retrieving the current month by Transaction Date, which may or may not be indexed correctly, and the rows required will be spread across the file. In any case at far less than ClusteredIndex speeds, and due to the spread, it will incur a tablescan.

2. The `AccountTransaction` table remains simple (the real world notion of a bank account Transaction is simple). It has a single positive `Amount` column.
3. For each `Account` , the `CurrentBalance` is:
   - the `AccountStatement.ClosingBalance` of the previous month, dated the first of the next month for convenience
     - for inventory, the `PartAudit.QtyOnHand`
   - plus the `SUM( Transaction.Amounts )` in the current month, where the `AccountTransactionType` indicates a deposit
     - for inventory, the `PartMovement.Quantity`
   - minus the `SUM( Transaction.Amount )` in the current month, where the `AccountTransactionType` indicates a withdrawal
   - (code provided below).
4. In this Method, the `AccountTransactions` in the current month, only, are in a state of flux, thus they **must be retrieved**. All previous months are published and closed, thus the Audit figure `AccountStatement.ClosingBalance` **must be used**.
5. The older rows in the `AccountTransaction` table can be purged. Older than ten years for public money, five years otherwise, one year for hobby club systems.
6. Of course, it is essential that any code relating to accounting systems uses genuine OLTP Standards and genuine SQL ACID Transactions (not possible in the pretend-SQL freeware).
7. This design incorporates all scope-level performance considerations (if this is not obvious, please ask for expansion). Scaling inside the database is a non-issue, any scaling issues that remain are actually outside database.

Corrective Advice

These items need to be stated only because incorrect advice has been provided in many SO Answers (and up-voted by the masses, democratically, of course), and the internet is chock-full of incorrect advice (amateurs love to publish their subjective "truths"):

1. Evidently, some people do not understand that I have given a Method in technical terms, to operate against a clear data model. As such, it is not pseudo-code for a specific application in a specific country. The Method is for capable developers, it is not detailed enough for those who need to be lead by the hand.

- They also do not understand that the cut-off period of a month is an **example**: if your cut-off for Tax Office purposes is quarterly, then by all means, use a quarterly cut-off; if the only legal requirement you have is annual, use annual.
- Even if your cut-off is quarterly for external or compliance purposes, the company may well choose a monthly cut-off, for internal Audit and sanity purposes (ie. to keep the length of the period of the state of flux to a minimum).

  Eg. in Australia, the Tax Office cut-off for businesses is quarterly, but larger companies cut-off their inventory control monthly (this saves having to chase errors over a long period).

  Eg. banks have legal compliance requirements monthly, therefore they perform an internal Audit on the figures, and close the books, monthly.
- In primitive countries and rogue states, banks keep their state-of-flux period at the maximum, for obvious nefarious purposes. Some of them only make their compliance reports annually. That is one reason why the banks in Australia do not fail.

2. In the `AccountTransaction` table, do not use negative/positive in the Amount column. Money always has a positive value, there is no such thing as negative twenty dollars (or that *you owe me minus fifty dollars*), and then working out that the double negatives mean something else.
3. The movement direction, or what you are going to do with the funds, is a separate and discrete fact (to the `AccountTransaction.Amount`). Which requires a separate column (two facts in one datum breaks Normalisation rules, with the consequence that it introduces complexity into the code).
   - Implement a `AccountTransactionType` reference table, the Primary Key of which is (`D, W`) for Deposit/Withdrawal as your starting point. As the system grows, simply add (`A, a, F, w`) for Adjustment Credit; Adjustment Debit; Bank Fee; ATM_Withdrawal; etc.
   - No code changes required.
4. In some primitive countries, litigation requirements state that in any report that lists Transactions, a running total must be shown on every line. (Note, this is not an Audit requirement because those are superior [(refer Method above) to the court requirement; Auditors are somewhat less stupid than lawyers; etc.)

   Obviously, I would not argue with a court requirement. The problem is that primitive coders translate that into: *oh, oh, we must implement a* `AccountTransaction.CurrentBalance` *column*. They fail to understand that:
   - the requirement to print a column on a report is not a dictate to store a value in the database
   - a running total of any kind is a derived value, and it is easily coded (post a question if it isn't easy for you). Just implement the required code in the report.

- implementing the running total eg. `AccountTransaction.CurrentBalance` as a column causes horrendous problems:
  - introduces a duplicated column, because it is derivable. Breaks Normalisation. Introduces an Update Anomaly.
  - the Update Anomaly: whenever a Transaction is inserted historically, or a `AccountTransaction.Amount` is changed, all the `AccountTransaction.CurrentBalances` **from that date to the present** have to be re-computed and updated.
- in the above case, the report that was filed for court use, is now obsolete (every report of online data is obsolete the moment it is printed). Ie. print; review; change the Transaction; re-print; re-review, until you are happy. It is meaningless in any case.
- which is why, in less-primitive countries, the courts do not accept any old printed paper, they accept only published figures, eg. Bank Statements, which are already subject to Audit requirements (refer the Method above), and which cannot be recalled or changed and re-printed.

Comments

> Alex:
> yes code would be nice to look at, thank you. Even maybe a sample "bucket shop" so people could see the starting schema once and forever, would make world much better.

For the data model above.

Code • Report Current Balance

```
SELECT  AccountNo,
        ClosingDate = DATEADD( DD, -1 Date ), -- show last day of previous
        ClosingBalance,
        CurrentBalance = ClosingBalance + (
            SELECT SUM( Amount )
                FROM AccountTransaction
                WHERE AccountNo = @AccountNo
                    AND TransactionTypeCode IN ( "A", "D" )
                    AND DateTime >= CONVERT( CHAR(6), GETDATE(), 2 ) + "01"
                ) - (
            SELECT SUM( Amount )
                FROM AccountTransaction
                WHERE AccountNo = @AccountNo
                    AND TransactionTypeCode NOT IN ( "A", "D" )
                    AND DateTime >= CONVERT( CHAR(6), GETDATE(), 2 ) + "01"
                )
    FROM AccountStatement
    WHERE AccountNo = @AccountNo
```

```
    AND Date = CONVERT( CHAR(6), GETDATE(), 2 ) + "01"
```

> By denormalising that transactions log I trade normal form for more convenient queries and less changes in views/materialised views when I add more tx types

God help me.

1. When you go against Standards, you place yourself in a third-world position, where things that are not supposed to break, that never break in first-world countries, break.

   It is probably not a good idea to seek the right answer from an authority, and then argue against it, or argue for your sub-standard method.

2. Denormalising (here) causes an Update Anomaly, the duplicated column, that can be derived from TransactionTypeCode. You want ease of coding, but you are willing to code it in two places, rather than one. That is exactly the kind of code that is prone to errors.

   A database that is fully Normalised according to Dr E F Codd's *Relational Model* provides for the easiest, the most logical, straight-forward code. (In my work, I contractually guarantee every report can be serviced by a single `SELECT` .)

3. `ENUM` is not SQL. (The freeware NONsql suites have no SQL compliance, but they do have extras which are not required in SQL.) If ever your app graduates to a commercial SQL platform, you will have to re-write all those `ENUMs` as ordinary LookUp tables. With a `CHAR(1)` or a `INT` as the PK. Then you will appreciate that it is actually a table with a PK.

4. An error has a value of zero (it also has negative consequences). A truth has a value of one. I would not trade a one for a zero. Therefore it is not a trade-off. It is just your development decision.

Share Follow

edited Dec 10, 2022 at 9:41

answered Apr 18, 2015 at 5:18

PerformanceDBA
31.7k 10 64 90

1 @Alex. 1) A Credit_Debit_Type plus a TransactionType would be non-Logical, as well as it breaks Normalisation. Because TransactionType *absolutely* determines Credit_Debit_Type. 2) Do you not know how to use a `CASE` statement ? There is no additional code that is called for. In fact checking two columns requires more code than checking one column. – PerformanceDBA Dec 18, 2019 at 13:07

1 @Alex. The two INSERTS will be `D` . The data model does not show the GeneralLedger (the internal bank accounts), such as `BankCash` . Not the same table, it would have more columns. Such accounts are dependent on the report: if you were counting cash, it would be that one. `Asset/Liability/etc` defines the treatment, along with its external `AccountNo` . Your concept of SUM() does not apply. The operators apply. – PerformanceDBA Dec 19, 2019 at 11:57

1 I think without missing entities that stripped schema is very confusing. There is always a debit and credit, if you deposit something to account you must take it from some other account. Inserting two

1

This is fairly subjective. The things I'd suggest taking into account are:

1. How many accounts are there, currently?
2. How many accounts do you expect to have, in the future?
3. How much value do you place upon scalability?
4. How difficult is it to update your database and code to track the balance as its own field?
5. Are there more immediate development concerns that must be attended to?

In terms of the merits of the two approaches proposed, summing the transaction values on-demand is likely to be the easier/quicker to implement approach.

However, it won't scale as well as maintaining the current account balance as a field in the database and updating it as you go. And it increases your overall transaction processing time somewhat, as each transaction needs to run a query to compute the current account balance before it can proceed. In practice those may be small concerns unless you have a very large number of accounts/transactions or expect to in the very near future.

The downside of the second approach is that it's probably going to take more development time/effort to set up initially, and may require that you give some thought to how you synchronize transactions within an account to ensure that each one sees and updates the balance accurately at all times.

So it mostly comes down to what the project's needs are, where development time is best spent at the moment, and whether it's worth future-proofing the solution now as opposed to implementing the second approach later on, when performance and scalability become real, rather than theoretical, problems.

Share Follow

answered Apr 17, 2015 at 2:16

aroth
53.8k 20 135 176

Hey Aroth, thanks for your inputs! So, 1) Currently the app is under development, hence only test users are there 2) and 3) It will need to scale but not in near future (say we need not worry for about at least a year) 4) It's pretty straight to update the database and code to track, just another addition in the code and another update in the database. 5) Yes, there are other concerns, this accounting is a small part of the project. I too feel that second approach would be required when it actually scales up..
– Anmol Gupta Apr 17, 2015 at 2:40