Database for Financial Accounting Application II: Infrastructure

Niemand25

9 Aug 2019 CPOL

Rate this: ★★★★★ 4.92 (26 votes)

Designing simple yet functional database for financial accounting application

Download database create script for MySQL

Introduction

In the previous article, we discussed general financial accounting application database design concepts and defined a very basic roadmap for the whole database like: defining the business domain, basic requirements to be met, primary key usage policy, naming conventions. We also set up the infrastructure for extensibility and basic lookup codes.

This article will be dedicated to the accounting infrastructure design: general ledger, chart of accounts, (source) documents and financial statements structure.

At first, I thought of designing database schema (tables, entities) in the order of the table dependency. However, when writing this article, I realized that it’s just not possible to explain, e.g., financial statements functionality, without implementing general ledger first.

At first, I also planned to cover more functionality in this article, like person profile, costs centre and company’s profile. However, the article got already too long after finishing description of financial statements.

Accounting Fundamentals And General Ledger

Before we start designing accounting entities, we need to understand financial accounting fundamentals – how financial accounting works at its core.

There are two fundamental financial accounting methods – single entry and double entry. The single entry accounting method is rarely allowed by law. E.g., in Lithuania, it can only be used by individual entrepreneurs (natural persons) whose annual income is below 45.000 EUR; in Germany, it can only be used by individual entrepreneurs (natural persons) whose annual income is below 500.000 EUR. The single entry accounting method is rarely used even in countries that allow flexible choice of the accounting methods (e.g., UK). Therefore, we are going to implement double accounting method.

Double entry accounting method is based on fundamental accounting equation:

ASSETS = LIABILITIES + EQUITY + REVENUES - EXPENSES

In the equation:

- Assets are everything that the company is an owner of: cash, accounts receivable, supplies, equipment, shares of other companies, etc.
- Liabilities are everything that the company owns to other persons: notes payable, accounts payable, wages payable, taxes, etc.
- Equity is everything that belongs to the company's shareholders (owners, investors): company's shares (not shares of other companies), investments into the company, retained earnings, etc.

The fundamental accounting equation shall hold for every recorded financial transaction, which means that the equation can also be written as:

ΔASSETS = ΔLIABILITIES + ΔEQUITY + ΔREVENUES - ΔEXPENSES

The equation means that every financial transaction could change the amount of company's assets, liabilities, equity, revenues or expenses (either one of the categories or few of them). Financial accounting splits a transaction into components that correspond to the variables in the equation. So that every transaction is represented as two (or more) changes of the company's financial state. The equation is easiest to understand in action (full explanation):

	Δ Assets =	**Δ Liabilities**	**+ Δ Equity**	**+ Δ Revenues**	**- Δ Expenses**
Company issued shares (10,000 shares at $3 each) of common stock for $30,000 cash. The $30,000 cash was deposited in the new business account.	+ $30,000	-	+ $30,000	-	-
Company paid $ 5,500 cash for equipment (two computers).	- $ 5,500 + $ 5,500	-	-	-	-
Company purchased supplies on account for $500.	+ $500	+ $500	-	-	-
Company paid $500 for the supplies purchased	- $500	- $500	-	-	-
Company earned a total of $50,000 in revenue from clients who paid cash.	+ $50,000	-	-	+ $50,000	-
Company paid a total of $900 for office salaries.	- $900	-	-	-	+ $900
Total Balance:	**$79,100 =**	**$0**	**+ $30,000**	**+ $50,000**	**- $900**

As you can see from the example, every transaction changes one or more variables of the equation and the fundamental accounting equation remains true for every change.

The database table schema to keep the data for the example is pretty obvious:

The problem with this (over) simplified solution is that it only allows filtering by a very generalised type of the variable affected. E.g., an accountant would only see an increase and a decrease of category assets for purchase by cash transaction. An accountant would certainly want to distinguish some more details, e.g., not only the fact that a transaction increased assets value, but also the (exact) type of the asset (or assets) which value increased. The concept of subtypes of the fundamental accounting equation variables (categories) is referred as "account" and all of the accounts used by a company *in corpore* are referred as "chart of accounts". A simplified chart of accounts looks like that:

Id	Account Name	Category
1	Fixed assets	Assets
122	Equipment	Assets
2	Current assets	Assets
201	Supplies	Assets
241	Accounts receivable	Assets
271	Cash in a bank account	Assets
3	Equity	Equity
301	Equity capital	Equity
4	Liabilities	Liabilities
443	Accounts payable	Liabilities
4492	VAT Payable	Liabilities
5	Revenues	Revenues
500	Sales revenues	Revenues
6	Expenses	Expenses
6304	Salary expenses	Expenses

A classical chart of accounts is organized according to a numerical system. Thus, each major category will begin with a certain number, and then the sub-categories within that major category will all begin with the same number. If current assets are classified by numbers starting with the digit 2, then cash accounts might be labelled 271, accounts receivable might be labelled 241, supplies might be labelled 201, and so on. Whereas, if liabilities accounts are classified by numbers starting with the digit 4, then accounts payable might be labelled 443, short-term debt might be labelled 44, and so on. Such labelling conventions allows for grouping of different resources under the same category. E.g., if bank accounts are classified by numbers starting with the digit 271 and the company has multiple bank accounts classified by numbers 27101, 27102, 27103, etc., an accountant could get total cash turnover in all bank accounts by querying all accounts that start with digits 271. Which translates to SQL as `WHERE account_id LIKE '271%'` and eliminates need for child – parent account reference in the accounts table. However, it doesn't necessarily mean that a transaction could only use (modify) accounts without child accounts. Sometimes child accounts are used to distinguish some transactions from common ones. In this case, common transactions use (modify) parent account and some specific transactions – child account. Which allows for exclusion of type "all but specific".

As always in the wild, there are exceptions. I've seen some charts of accounts that use letters in account numbers, two part numbers ("code & number") or even plainly ignores classical numbering convention. However, those are (rare) exceptions and, to my mind, could hardly be considered as good practice. Therefore, I will only implement classical account numbering convention and use `BIGINT` type for account id. (`BIGINT` supports 18 digits, while `INT` supports only 9). It is also a natural key in the database design terms. As discussed in the previous article, I will use it as a primary key.

If the double entry accounting method had been invented a couple of hundred years later, than it was, the accounting method would be that simple – plainly record a transaction as a collection of changes (deltas) in various accounts making sure that the fundamental accounting equation is observed. However, when the double entry accounting method was invented (around X century A.D., maybe earlier), the concept of negative numbers was known a little. Therefore, negative numbers are not used in accounting. Instead, there is a debit - credit concept. The concept is relatively simple – instead of subtracting some amount from an account, we do exactly the same, but refer to that by different name – either debit or credit. Moreover, to reflect the sides of the fundamental equation, we switch the naming convention subject to the base type of the account:

- For the accounts on the left side of the equation (i.e., assets), we refer to an increase as "debit" and to a decrease as "credit";
- For the accounts on the right side of the equation (i.e., liabilities, equity and revenues), we refer to an increase as "credit" and to a decrease as "debit";
- For the expenses accounts, we refer to an increase as "debit" and to a decrease as "credit"; because the expenses go with a minus sign in the equation even though they are on the right side.

If the total debit amount in an account is greater than the total credit amount in the same account, we say that the account has a debit balance, which is (no surprise) equal to total debit amount minus total credit amount.

If the total credit amount in an account is greater than the total debit amount in the same account, we say that the account has a credit balance, which is (again no surprise) equal to total credit amount minus total debit amount.

Because the usage of debit/credit notation reflects the fundamental accounting equation, a total amount of debit entries within a transaction shall always be equal to the amount of credit entries within the same transaction. It is easy to demonstrate by transforming the previous example to use debit-credit notation (pluses for assets and expenses are replaced by debits “D”; minuses for assets and expenses are replaced by credits “C”; pluses for liabilities, equity and revenues are replaced by credits “C”; minuses for liabilities, equity and revenues are replaced by debits “D”):

	Δ Assets =	**Δ Liabilities**	**+ Δ Equity**	**+ Δ Revenues**	**- Δ Expenses**
Company issued shares (10,000 shares at $3 each) of common stock for $30,000 cash. The $30,000 cash was deposited in the new business account.	D $30,000	-	C $30,000	-	-
Company paid $ 5,500 cash for equipment (two computers).	C $ 5,500 D $ 5,500	-	-	-	-
Company purchased supplies on account for $500.	D $500	C $500	-	-	-
Company paid $500 for the supplies purchased	C $500	D $500	-	-	-
Company earned a total of $50,000 in revenue from clients who paid cash.	D $50,000	-	-	C $50,000	-
Company paid a total of $900 for office salaries.	C $900	-	-	-	D $900
Total Balance:	**D $79,100**	**$0**	**C $30,000**	**C $50,000**	**D $900**

The database schema to keep the data for the modified example is very similar to the previous – just add a table for chart of accounts instead of variable type enumeration and extra column for entry type:

As you can see, I also made some extra minor changes that are worth to discuss:

- We added column `transaction_date` for the table transactions, well, for obvious reasons. The column is indexed, as it will always be used as search parameter.
- We use table name `accounts` instead of `chart_of_accounts` because of naming conventions we adopted in the previous article. (plural for entity stored)
- We use `account_type` field of type `INT` instead of `ENUM('assets', 'liabilities', 'equity', 'revenues', 'expenses')` to allow the application to provide for more fine-grained enumeration. The enumeration might change as the application is upgraded. Therefore, it cannot not be defined as `ENUM` in the database. For the data integrity purpose (to prevent illegal/non-existent types), a technical lookup table might be used. But I am undecided about it yet.
- We use `is_archived` field to accounts in order to allow the application to hide not currently used accounts in lookup controls.
- We add standard audit trail fields to `accounts` as the accounts are obviously parent entities.
- We use table name `ledger_entries` instead of `deltas` because (general) ledger is a traditional name of the registry (ledger) that contains all the entries made by transaction to all of the accounts.
- We use column name `amount` instead of `delta` because it is no longer a delta value.
- We add index on `entry_type`, as it will be extensively used as filter parameter.

This is it. We implemented core financial accounting functionality for double entry method. Actually, it is almost ready to use model. We only need a few changes to make it fully functional, as an accountant would expect to:

- Add analytics by a person and a cost centre
- Add source document concept
- Add support for tax reporting
- Add support for financial statements

Analytics by a Person and a Cost Centre

It is obvious that an accountant will want to filter transactions and ledger entries (accounts turnover) by a person or a costs centre. We will discuss “person” and “cost centre” entities in detail in the next article. For now, the important aspect is the place (table) where these foreign keys are added. It’s tempting to add them at transaction level. Actually, you would see that in multiple examples found on web. However, it’s a wrong approach due to the real life use cases, e.g.:

- Simple invoice made by a VAT payer has three ledger entries: credits a revenues account by invoice amount (total price), credits a VAT payable account by VAT amount and debits accounts receivable by the sum of invoice amount and VAT amount. It’s natural that the accounts receivable entry is related to the person (client). However, it would be at least ambiguous to relate the same person with VAT payable account (VAT is payable to the state not the client). It’s natural, that the expenses account entry is related to some costs centre. However, it would be at least ambiguous to relate the same costs centre with VAT payable account (it’s not costs at all).
- Multilateral offset has two (or more) ledger entries: debits an accountants payable for one person (party, supplier) and credits an accounts receivable for another person (party, client). Obviously, all the entries are related to different persons. The same goes for wage sheet and other documents that are naturally related to multiple persons.

Therefore, the right approach is to add analytics in the `ledger_entries` table.

Source Document Concept

You could also wonder how the model could be almost complete if the transactions table only has two columns – `date` and `description` – while documents obviously are described by more data fields. The reason behind it is the difference between concepts “transaction” and “(source) document”. Though every transaction is related to some (source) document (e.g., invoice), a (source) document might have no associated transaction (e.g., labour contract, time sheet, etc.) or have multiple transactions (e.g., unearned revenue, accrued expenses). Therefore:

- The `transactions` table is a child table of the `documents` table with the relationship type 1 -> 0…n
- A `transaction` naturally has no specific fields except for `date` and (rarely used) `description`
- A `transaction` is completely generic, i.e., independent of a particular document type

As discussed in the previous article, the (source) document implementation should be extendable, i.e., extensions shall be able to define their own document types while using the base document implementation as common access point. This requirement also implies that the base document implementation should be generic – the table should only contain fields that are definitely common for all of the (source) document types. These requirements leads us to a very simple table schema:

Fields	**Description**

Fields	Description
id	A document entity does not have any natural key. Any of the fields can be changed without affecting the identity of the document. Therefore, we use synthetic primary key.
document_date	A date of the document. As previously discussed, could be different from the transaction date. As "date" is a reserved word, we use prefix.
document_no, description, document_comments, internal_comments	A number and a description of the document, as well as comments regarding the document. Document comments are displayed on the document itself while internal comments are only meant for internal use. (an accountant's comments for himself) All of those fields could be empty, but there is no semantical difference between those fields being empty and null. Therefore, we use not null constraint to avoid unnecessary null values. We set max length for those fields following the guidelines in the previous article. Though I'm unsure about document_comments field. It's very uncommon for the descriptive fields in the accounting domain to be longer than 500 characters. Maybe, it would be more reasonable to use shorter field and only add long field in the custom operations tables that actually require that (e.g. depreciation, accounting note, maybe a few more).
document_type	An application defined enumeration of the document types that a handled by the base application functionality. It might change. Therefore, ENUM type cannot be used. For the data integrity purpose (to prevent illegal/non-existent types), a technical lookup table might be used. But I am undecided about it yet.
extended_document_type	A document type defined by an extension of the application. (if the document was created by an application extension) Discussed in the previous article.
external_id	An id of the document assigned by external application, e.g., REST service. Used to sync documents with external sources.
inserted_at, inserted_by, updated_at, updated_by	Standard audit trail fields as defined in the previous article.

Support for Tax Reporting

It's not much that you can do with tax accounting at the general ledger level. At least I never found a way to map general ledger accounts balances to tax reports. However, there is one simple tax reporting requirement that emerged recently. The requirement is the ability to export general ledger accounts using a state provided account classificatory, i.e., each account in the chart of accounts should have a state defined classification code (see SAF-T). There is no requirement to support multiple code versions (for different SAF-T versions). Therefore, implementation of the requirement is as simple as adding a single field to the table accounts – official_code. The code does not affect any real taxes and only used by the tax inspectorate for full-scale audit. Which means, it is not sensitive to input errors and simple text field is sufficient.

After implementing all the changes, the (almost) final schema (relevant portion) will be the following:

Alternative General Ledger Implementations

It's worth mentioning that there are other possible schemas that provide exactly the same general ledger functionality (for discussion, see: Need help with Double Entry DB Design, Database schema design for a double entry accounting system, Double Entry Accounting in a Relational Database). The only option proposed, that I would disagree with, is one row per two entries schema, that uses two account fields and one amount. The credit equals debit rule is a business rule, not data integrity rule. Therefore, it belongs to the application model and shall not be duplicated in database. Moreover, it does not correspond to the real world ledger entries that are not always paired. (e.g., simple invoice made creates the following ledger entries: D241 – 1.210,00 EUR; C500 – 1.000,00 EUR; C4492 – 210,00 EUR)

For the rest of the alternative ledger schemas, you can as well use:

- bit flag `is_debit` instead of enumeration debit/credit
- sign of amount as a debit/credit indicator, i.e., positive numbers as debit, negative – as credit
- columns `amount_debit` and `amount_credit` instead of single amount with a debit/credit indicator

As all of those options are semantically equivalent, I've benchmarked all of them for performance. For that purpose, I created test databases for each of the schemas with 1 million transactions and 2,5 million ledger entries. For the bit flag version, I also added an index on the bit flag field. For the "amount sign" implementation, I also added index on the amount field.

Benchmarking was done on my Dell Inspiron 15 Series 5000 laptop:

- **Processor**: Intel Core i5-5200U, 2,20GHz;
- **RAM**: 4 GB;
- **OS**: Win7 Pro;
- **MySQL Server**: v. 5.7 with innodb_buffer_pool_size = 1024MB.

The following queries (adapted per schema type) were used for benchmarking:

Classical General Journal

Hide Copy Code

```
SELECT SQL_NO_CACHE t.transaction_date AS Date, t.description AS
DescriptionOrAccountTitle,
null as AmountDebit, null AS AmountCredit, t.id AS Reference, null AS IsLine
FROM transactions t
LEFT JOIN ledger_entries e ON e.transaction_id = t.id
LEFT JOIN accounts a ON a.id = e.account_id
WHERE t.transaction_date BETWEEN '2018-01-01' AND '2018-06-30'
UNION
SELECT null AS Date, (CASE WHEN e.entry_type = 'D' THEN
a.account_name ELSE CONCAT('-  ', a.account_name) END) AS DescriptionOrAccountTitle,
(CASE WHEN e.entry_type = 'D' THEN e.amount ELSE null END) AS AmountDebit,
(CASE WHEN e.entry_type = 'C' THEN e.amount ELSE null END) AS AmountDebit,
t.id AS Reference, (CASE WHEN e.entry_type = 'D' THEN 1 ELSE 2 END) AS IsLine
FROM transactions t
LEFT JOIN ledger_entries e ON e.transaction_id = t.id
LEFT JOIN accounts a ON a.id = e.account_id
WHERE t.transaction_date BETWEEN '2018-01-01' AND '2018-06-30'
ORDER BY Reference, IsLine;
```

Example Output

Date	DescriptionOrAccountTitle	AmountDebit	AmountCredit	Reference	IsLine
2018-01-01	Payment received description seq 120464			884835	
	Bank account	3413		884835	1
	- Accounts receivable		3413	884835	2
2018-01-01	Payment made description seq 120464			884837	
	Accounts payable	3413		884837	1
	- Bank account		3413	884837	2
2018-01-01	Payment received description seq 120465			884839	
	Bank account	3413		884839	1
	- Accounts receivable		3413	884839	2

Ledger Account Turnover

Hide Copy Code

```
SELECT SQL_NO_CACHE t.transaction_date, t.description,
(CASE WHEN e.entry_type='D' THEN e.amount ELSE 0 END) AS DebitAmount,
(CASE WHEN e.entry_type='C' THEN e.amount ELSE 0 END) AS CreditAmount
FROM transactions t
LEFT JOIN ledger_entries e ON e.transaction_id = t.id
WHERE t.transaction_date >= '2018-01-01' AND t.transaction_date <='2018-06-30'
AND e.account_id = 271
ORDER BY t.transaction_date;
```

Example Output

transaction_date	description	DebitAmount	CreditAmount
2018-01-01	Payment received description seq 120464	3413	
2018-01-01	Payment made description seq 120464		3413
2018-01-01	Payment received description seq 120465	3413	
2018-01-01	Payment made description seq 120465		3413
2018-01-01	Payment received description seq 120466	3413	

Simple General Ledger

Hide Copy Code

```
SELECT SQL_NO_CACHE t.transaction_date, t.description,
SUM(CASE WHEN e.entry_type='D' THEN e.amount ELSE 0.0 END) AS Amount,
GROUP_CONCAT(DISTINCT CONCAT(e.entry_type, e.account_id)
SEPARATOR ', ') AS Entries
FROM transactions t
LEFT JOIN ledger_entries e ON e.transaction_id = t.id
WHERE t.transaction_date >= '2018-01-01' AND t.transaction_date <='2018-06-30'
GROUP BY t.id ORDER BY t.transaction_date;
```

Example Output

transaction_date	description	Amount	Entries
2018-01-01	Payment made description seq 120540	996	C271, D410
2018-01-01	Payment received description seq 121221	5705	C240, D271
2018-01-01	Payment made description seq 121221	5705	C271, D410
2018-01-01	Payment received description seq 121331	3288	C240, D271

Ledger Accounts Balances

Hide Copy Code

```
SELECT SQL_NO_CACHE e.account_id, a.account_name,
SUM(CASE WHEN e.entry_type='D' THEN e.amount ELSE -e.amount END) AS Balance
FROM transactions t
LEFT JOIN ledger_entries e ON e.transaction_id = t.id
LEFT JOIN accounts a ON a.id = e.account_id
WHERE t.transaction_date <='2018-06-30' GROUP BY e.account_id
ORDER BY CAST(e.account_id AS CHAR);
```

Example Output

account_id	account_name	Balance
220	VAT receivable	124583351
240	Accounts receivable	55682607
271	Bank account	329570550
410	Accounts payable	-36592865
445	VAT payable	-185094735
505	Revenus	-881399566

account_id	account_name	Balance
601	Expenses	593250359

Trial Balance At Account Level

Hide Copy Code

```
SELECT SQL_NO_CACHE e.account_id, a.account_name,
SUM(CASE WHEN t.transaction_date < '2017-01-01' AND e.entry_type='D' _
         THEN e.amount ELSE 0.0 END) AS TotalDebitBefore,
SUM(CASE WHEN t.transaction_date < '2017-01-01' AND e.entry_type='C' _
         THEN e.amount ELSE 0.0 END) AS TotalCreditBefore,
SUM(CASE WHEN t.transaction_date >= '2017-01-01' AND _
    t.transaction_date < '2018-01-01' AND e.entry_type='D' _
    THEN e.amount ELSE 0.0 END) AS DebitFirstPeriod,
SUM(CASE WHEN t.transaction_date >= '2017-01-01' AND t.transaction_date < '2018-01-01'
_
         AND e.entry_type='C' THEN e.amount ELSE 0.0 END) AS CreditFirstPeriod,
SUM(CASE WHEN t.transaction_date >= '2018-01-01' AND e.entry_type='D' _
         THEN e.amount ELSE 0.0 END) AS DebitSecondPeriod,
SUM(CASE WHEN t.transaction_date >= '2018-01-01' AND e.entry_type='C' _
         THEN e.amount ELSE 0.0 END) AS CreditSecondPeriod
FROM transactions t
LEFT JOIN ledger_entries e ON e.transaction_id = t.id
LEFT JOIN accounts a ON a.id = e.account_id
WHERE t.transaction_date <='2018-12-31' GROUP BY e.account_id _
       ORDER BY CAST(e.account_id AS CHAR);
```

Example Output

account _id	account _name	Total DebitBefore	Total CreditBefore	Debit FirstPeriod	Credit FirstPeriod	Debit SecondPeriod	Credit SecondPeriod
220	VAT receivable	46331739	0	52297183	0	52359806	0
240	Accounts receivable	398054598	343264938	446540435	445596168	448165737	448829030
271	Bank account	343264938	230975372	445596168	300814740	448829030	302109106
410	Accounts payable	230975372	266957141	300814740	301329700	302109106	301690229
445	VAT payable	0	69084102	0	77499185	0	77781242
505	Revenues	0	328970810	0	369041711	0	370385116
601	Expenses	220625870	0	249032630	0	249330805	0

Benchmarking results (seconds on average per 10 query runs):

	Account turnover	Classical general journal	Simple general journal	Balance for all accounts	Trial balance
ENUM flag	0,156	14,071	2,247	14,274	13,963
TINYINT flag	0,140	14,961	2,309	12,215	13,587
Amount sign as flag	0,141	18,252	2,324	13,962	13,057
Separate columns for debit and credit amounts	0,327	13,119	2,199	14,009	13,915

Conclusion – all of those schema variations have identical performance stats. Therefore, the choice is a matter of personal taste.

As a side result, we can see that the general ledger schema is viable for a company with 100.000 documents per year for 10 years of operations. Of course, such a company wouldn’t use my laptop as a server. With appropriate hardware and SQL server configuration, the performance will likely increase by a factor. While a couple of seconds per query is quite acceptable.

Just out of curiosity, I have also tested using `BIGINT` type for amount field. Rumours that it's faster than DECIMAL turned out to be `false`. Performance is the same.

Financial Statements Overview

As discussed in the previous article, financial accounting is the field of accounting concerned with the summary, analysis and reporting of financial transactions related to a business. This involves the preparation of financial statements available for public use.

Financial statements (*aka financial reports*) are formal records of the financial activities and position of a business, person, or other entity. Relevant financial information is presented in a structured manner and in a form, which is easy to understand. They typically include four basic financial statements accompanied by a management discussion and analysis:

- **A balance sheet** (*aka statement of financial position*), reports on a company's assets, liabilities, and owners' equity at a given point in time. (sample for US, sample for UK, sample for France, sample for Lithuania)
- **An income statement** (*aka profit and loss report, P&L report, statement of comprehensive income, statement of revenue & expense*) reports on a company's income, expenses, and profits over a stated period of time. A profit and loss statement provides information on the operation of the enterprise. These include sales and the various expenses incurred during the stated period. (sample for US, sample for UK, sample for France, sample for Lithuania)
- **An equity statement** (*aka statement of changes in equity, statement of retained earnings*) reports on the changes in equity of the company over a stated period of time. (sample for US, sample for UK, sample for France, sample for Lithuania)
- **A cash flow statement** reports on a company's cash flow activities, particularly its operating, investing and financing activities over a stated period of time. (sample for US, sample for UK, sample for France, sample for Lithuania)
- **Notes to financial statements** include an extensive set of footnotes to the financial statements and management discussion and analysis. The notes typically describe each item on the balance sheet, income statement and cash flow statement in further detail.

All of those reports (except for the notes) have a certain forms (structures) that are set by law or accounting standards. The forms (structures) are different for different company types (by size and legal forms, not to mention different jurisdictions). The forms (structures) are subject to change when the respective laws or accounting standards change (and they do change). Therefore, all of those report forms (structures) should be implemented in a way that allows a user (an accountant) to set their structure up.

In the following sections, we will implement the financial statements one by one. We will skip the notes to financial statements (for now), because they require analysis of specific accounting operation types (fixed-assets, inventory and possibly some other operations) in order to prepare them.

Balance Sheet And Income Statement

Let's begin designing financial statements functionality with the balance sheet and income statement. They have the following (basic) structure:

It should look familiar by now. And yes, that's a hierarchical view of the charts of accounts. Some would argue that it actually is the chart of accounts, but there are some slight differences. As discussed before, in the charts of accounts a parent account and a child account could both be used (debited or credited) at the same time. Income statement and balance sheet, on the other hand, are strictly hierarchical: parent line shall always contain a sum of all the child lines. Another difference is the authority that sets chart of accounts and financial statements. The company itself sets chart of accounts. Of course, there are some guidelines and even model accounts, that should be respected, but the final decision is always up to the company. Financial statements (structure, forms), on the other hand, are set by the state. The differences means that the income statement and balance sheet structures should be stored separately and related to the accounts subject to the following rules:

- An account could only be related with one line of either balance sheet of income statement. Otherwise, we would get duplicate value of assets, equity or net income.
- All of the accounts should be related to some lines of either balance sheet of income statement. Otherwise, we would not respect the fundamental accounting equation. On the other hand, there are situations when some or all of the accounts are temporally not related to any lines of either balance sheet of income statement. E.g., state (government) changes the mandatory financial statements structure, an accountant has to rebuild it in the application. After rebuilding the structure, some lines are deleted. Therefore, corresponding accounts lose the relations. An accountant shall move to the chart of accounts and reassign new lines to all of the accounts. For that reason, we cannot use `NOT NULL` constraint on the accounts relation (foreign key). Observance of this rule falls on the accountant. In this case, it's not critical, because the relationship is only used when fetching financial statements, which makes it easy for the application to check this rule at the request.
- An account could only be assigned to a line of either balance sheet or income statement if the line does not have any child lines. This requirement is hard to safeguard in database. At first glance, you could add a check restraint on foreign key. However, that wouldn't help if an accountant modifies the structure of balance sheet or income statement and adds some child lines on the line that did not have any children previously. Even if you add check restraint on the balance sheet or income statement line, the best that it could do is to set the related accounts foreign key to `null`, which brings us to the previously discussed rule and the same solution. As the relationship is only used when fetching financial statements, the application shall check this rule before doing the actual fetch.

The next thing to consider is the method how balance sheet and income statement values are fetched:

- Balance sheet line value equals total sum of balances of all the accounts related to the line on a requested period end date, i.e., taking into account all the previous periods.
- Income statement line equals total sum of balances of all the accounts related to the line for a requested income statement period, i.e., NOT taking into account any previous period.

All those values are available from account level trial balance query that we used for general ledger performance benchmarking. However, to calculate and assign appropriate value, the application needs to know whether the line belongs to a balance sheet or to an income statement, i.e., we need to store line type. One more thing that we need to know is how to display debit/credit balance. Neither balance sheet nor income statement has a notion of debit/credit, only a plain number. Therefore, we need to store some indicator, whether to display debit balance as positive number or *vice versa*.

With all these fields in place, we have enough data to fetch both balance sheet and income statement and can proceed to the actual SQL schema.

The balance sheet and income statement structure is hierarchic and, obviously, sequence of items is important. One balance sheet or income statement line can only have one parent line. The structures of balance sheet and income statement change rarely, but they are queried relatively frequently. Moreover, all of the balance sheet and income statement structure is always edited together, as a whole. Perfectly suitable SQL model for this kind of tree is nested set model.

As you will see shortly, the main requirement for the tree model is ability to fetch child nodes in the most simple way possible. In the nested set model, it's as simple as checking whether line left index is between parent line left and right indexes (`SELECT * FROM nested_set_model AS ParentNode LEFT JOIN nested_set_model AS ChildNode ON ChildNode.left_index BETWEEN ParentNode.left_index AND ParentNode.right_index`). Closure table model and Joe Celko's modified adjacency list model require extra table and respectively extra join, which is a more complex solution obviously. Adjacency list model requires either recursive queries or custom functions, which is not an option due to the simple technology requirement as discussed in the previous article. Not to mention that it's a way more complex than the nested set implementation. Path enumeration (materialized path) model can do child nodes select (join) in similar way as the nested model does. However, Path enumeration model requires complex inserts, which makes it more complex. Therefore, I'll stick to the nested set model as best suited for the task.

To sum it all up, the resulting SQL schema (relevant part) for balance sheet and income statement functionality is:

Field	Description
`id`	Neither balance sheet nor income statement line has any natural key. Except of the `line_type` field, all the fields can change without changing identity of the line. Therefore, we use synthetic primary key.

Field	Description
line_type	We store both balance sheet and income statement lines in one nested set table, which is ok having in mind that each account can be related to either balance sheet or income statement line but not to both of them. However, for this reason we need a "superstructure". There will be: 1. one base header item, that all the other items are children of 2. one balance sheet header item, that all the balance items (lines) are children of; and 3. one income statement header item, that all the income items (lines) are children of In this case, the line types are metadata that describe the way we store actual data. Therefore, we can safely use ENUM type.
visible_index	Even though nested set model supports exact item sequence of the items within the hierarchy, bureaucrats manage to invent such statement structures that even nested set cannot handle. For such special cases, we need visible index that defines how all the lines are arranged on report.
printed_no	A number that is printed in the report next to the item line. At least in Lithuania, bureaucrats use a mixture of Roman and Arabic numerals.
line_text	A text of the item line, e.g., "Accounts Receivable".
line_type	A type of the balance (debit/credit) to display ("print") as a positive number. Could also use a bit flag instead, e.g. is_credit_positive, but the ENUM is more developer friendly.
left_index, right_index	Technical fields of the nested set model.
inserted_at, inserted_by, updated_at, updated_by	Standard audit trail fields as defined in the previous article.

Having this schema in place, we can now fetch balance sheet and income statement structural data in a way that is convenient for common tree object structure (e.g., the current balance sheet and income statement implementation in my current accounting solution):

Hide Copy Code

```
SELECT node.id, node.line_type, (COUNT(parent.id) - 1) AS depth, _
node.visible_index, node.printed_no, node.line_text, node.line_type, node.left_index, _
node.right_index,
(SELECT COUNT(*) FROM accounts a WHERE a.balance_and_income_line_id = node.id) AS
AccountsCount
FROM balance_and_income_lines AS parent
LEFT JOIN balance_and_income_lines AS node ON node.left_index _
        BETWEEN parent.left_index AND parent.right_index
GROUP BY node.id ORDER BY node.left_index;
```

Count of the ledger accounts assigned is added only to improve user experience – warn the user that the line he's going to delete has some accounts related.

Next having this schema in place, we can now fetch actual balance sheet and income statement for the periods requested by a user, e.g., let the first period be from 2017-01-01 to 2017-12-31 and the second period from 2018-01-01 to 2018-12-31:

Hide Shrink ▲ Copy Code

```
SELECT line.printed_no as RowNo, line.line_text AS RowText,
(SELECT GROUP_CONCAT(CAST(a.id AS CHAR) SEPARATOR ', ') _
FROM accounts a WHERE a.balance_and_income_line_id = line.ID) AS RelatedAccounts,
SUM(CASE WHEN line.line_type <> 'balance_line' _
AND line.line_type <> 'income_line' THEN null ELSE
(CASE WHEN line.line_type = 'balance_line' _
THEN val.TotalDebitBefore - val.TotalCreditBefore + val.DebitFirstPeriod - _
val.CreditFirstPeriod ELSE (val.DebitFirstPeriod - val.ClosingDebitFirstPeriod)
- (val.CreditFirstPeriod - val.ClosingCreditFirstPeriod) END) END)
* (CASE WHEN line.value_type='D' THEN 1 ELSE -1 END) AS ValueFirstPeriod,
SUM(CASE WHEN line.line_type <> 'balance_line' _
AND line.line_type <> 'income_line' THEN null ELSE
(CASE WHEN line.line_type = 'balance_line' _
THEN val.TotalDebitBefore - val.TotalCreditBefore + val.DebitFirstPeriod - _
val.CreditFirstPeriod + val.DebitSecondPeriod - val.CreditSecondPeriod _
ELSE (val.DebitSecondPeriod - val.ClosingDebitSecondPeriod)
- (val.CreditSecondPeriod - val.ClosingCreditSecondPeriod) END) END)
* (CASE WHEN line.value_type='D' THEN 1 ELSE -1 END) AS ValueSecondPeriod
FROM balance_and_income_lines AS line
LEFT JOIN balance_and_income_lines AS child_line ON child_line.left_index _
BETWEEN line.left_index AND line.right_index
LEFT JOIN (
SELECT a.balance_and_income_line_id AS LineId, l.line_type AS LineType,
SUM(CASE WHEN t.transaction_date < '2017-01-01' _
AND e.entry_type='D' THEN e.amount ELSE 0.0 END) AS TotalDebitBefore,
SUM(CASE WHEN t.transaction_date < '2017-01-01' _
AND e.entry_type='C' THEN e.amount ELSE 0.0 END) AS TotalCreditBefore,
SUM(CASE WHEN t.transaction_date >= '2017-01-01' _
AND t.transaction_date < '2018-01-01' AND e.entry_type='D' _
THEN e.amount ELSE 0.0 END) AS DebitFirstPeriod,
SUM(CASE WHEN t.transaction_date >= '2017-01-01' _
AND t.transaction_date < '2018-01-01' AND e.entry_type='C' _
THEN e.amount ELSE 0.0 END) AS CreditFirstPeriod,
SUM(CASE WHEN d.document_type = 13 AND t.transaction_date >= '2017-01-01' _
AND t.transaction_date < '2018-01-01' AND e.entry_type='D' _
THEN e.amount ELSE 0.0 END) AS ClosingDebitFirstPeriod,
SUM(CASE WHEN d.document_type = 13 AND t.transaction_date >= '2017-01-01' _
AND t.transaction_date < '2018-01-01' AND e.entry_type='C' _
THEN e.amount ELSE 0.0 END) AS ClosingCreditFirstPeriod,
SUM(CASE WHEN t.transaction_date >= '2018-01-01' AND e.entry_type='D' _
THEN e.amount ELSE 0.0 END) AS DebitSecondPeriod,
SUM(CASE WHEN t.transaction_date >= '2018-01-01' AND e.entry_type='C' _
THEN e.amount ELSE 0.0 END) AS CreditSecondPeriod,
SUM(CASE WHEN d.document_type = 13 AND t.transaction_date >= '2018-01-01' _
AND e.entry_type='D' THEN e.amount ELSE 0.0 END) AS ClosingDebitSecondPeriod,
SUM(CASE WHEN d.document_type = 13 AND t.transaction_date >= '2018-01-01' _
AND e.entry_type='C' THEN e.amount ELSE 0.0 END) AS ClosingCreditSecondPeriod
FROM documents d
LEFT JOIN transactions t ON t.document_id = d.id
LEFT JOIN ledger_entries e ON e.transaction_id = t.id
LEFT JOIN accounts a ON a.id = e.account_id
LEFT JOIN balance_and_income_lines l ON l.id = a.balance_and_income_line_id
WHERE t.transaction_date <='2018-12-31' GROUP BY a.balance_and_income_line_id
) AS val ON val.LineId = child_line.id OR (child_line.id=14 AND val.LineType =
'income_line')
GROUP BY line.id ORDER BY line.visible_index;
```

The query seems a bit monstrous but actually, it is very simple and fast (only took 20 seconds on a test database with one million transactions):

- The heavy lifting is done by the subquery in the last `join`. The subquery gets total debit and credit ledger turnovers for the periods requested by a user and groups it by balance sheet or income statement line. The result of the query is a small virtual table with a few hundred lines at max (because at max, there are a few hundred lines in both balance sheet and income statement). The result of the query contains all the information required to calculate balance sheet or income

statement values for the lines that are directly assigned to ledger accounts. As discussed previously, balance sheet line value at the end of a period equals to the total sum of debit and credit entries at the end of a period, while income statement line value at the end of a period equals to the total sum of debit and credit entries during period. Therefore, balance sheet line value for the first period equals `TotalDebitBefore– TotalCreditBefore + DebitFirstPeriod - CreditFirstPeriod`; Income statement line value for the first period equals `DebitFirstPeriod - CreditFirstPeriod`. However, income statement lines actually have some correction components. The corrections come from a special accounting operation named accounts closure. The operation is performed once per accounting period (usually calendar year) and effectively nullifies all of the income statement accounts (revenues and expenses), i.e., the operation debits or credits accounts balances so that the resulting balance is zero. Of course, an accountant wouldn't like to see all null income statement. Therefore, we need to reverse entries made by the operation. You should note that the resulting values are debit balances, while in some lines credit balance should be displayed as positive numbers (e.g., revenues line). To fix it, we multiply the value by minus one if the credit balance is expected in the line. The same method applies for the second period.
- Next, we need to deal with the aggregate lines that should sum the values of their child lines. For that purpose, we use self-join – for each balance sheet or income statement line, we join all the child lines including the line itself. In the nested set model, it's a trivial task – if a line's left index falls between another line's left and right indexes then the first line is a child of the second one. And here, we join our previous subquery result set that contains actual values per balance sheet or income statement line. Group the result by the parent lines and you get actual balance sheet and income statement.
- The last thing to discuss is the strange join condition for subquery result:
`ON val.LineId = child_line.id OR (child_line.id=14 AND val.LineType = 'income_line')`
The first part is simple – we join balances for the child line. The second part is meant to handle simulation of the accounts closure. As discussed previously in this article, debit/credit balance only holds for the entirety of accounts: assets, equity, liabilities, revenue and expenses. However, when dealing with balance sheet and income statement, we split accounts in two parts: assets, equity and liabilities fall into the balance sheet, while revenue and expenses fall into income statement. Which in turn means that the balance sheet will imminently be "unbalanced", i.e., total amount of assets will not be equal to the total amount of equity and liabilities. In some cases, it is a desired result as it discloses the fact that there is no closing entry If you wish to fetch the balance "as is", just remove the second part of the join condition. Nevertheless, in the most practical cases, an accountant would like to see a valid balance sheet as it would look like after a closing entry is made. In order to meet expectations, we need to simulate accounts closure, i.e., move the total balance of the revenues and expenses to the equity account, which is typically labelled as retained earnings account. In the query terms, it means that we need to identify the balance line, which holds retained earnings account value, and add the total balance of the revenues and expenses. As all of the revenues and expenses accounts imminently fall into income statement lines we can easily distinguish the balances required by `line_type` field. That's it – we just simulated closure of accounts. For now, the balance line, which holds retained earnings account value, is hardcoded into the query. Later retained earnings account will be stored in the company's profile, which will allow us to fetch an id of the relevant balance line.

In real life scenarios, balance sheet and income statement are usually fetched together with the account level balance, which used the same subquery but grouped by account instead of balance sheet or income statement line. As namely the subquery is doing heavy lifting, executing separate queries for account level balance and balance sheet and income statement essentially doubles the database workload for the very similar data. For that reason, at the application level, I opted for only fetching account level balance and doing the transforms for balance sheet and income statement within the application. This approach is almost twice as fast as using two heavy queries.

Cash Flow Statement

The statement of cash flows is one of the financial statements issued by a business, and describes the cash flows into and out of the organization. It acts as a bridge between the income statement and balance sheet by showing how money moved in and out of the business.

Cash flows in the statement are divided into the following three areas:

- **Operating activities**. These constitute the revenue-generating activities of a business. Examples of operating activities are cash received and disbursed for product sales, royalties, commissions, fines, lawsuits, supplier and lender invoices, and payroll.
- **Investing activities**. These constitute payments made to acquire long-term assets, as well as cash received from their sale. Examples of investing activities are the purchase of fixed assets and the purchase or sale of securities issued by other entities.
- **Financing activities**. These constitute activities that will alter the equity or borrowings of a business. Examples are the sale of company shares, the repurchase of shares, and dividend payments.

There are two methods to present a cash flow statement – direct and indirect.

Direct method is essentially a simple classification of individual cash operations. Therefore, it is not derivable from ledger accounts. You could add cash operation classification for cash flow statement in the ledger itself. However, that would impose considerable amount of work on an accountant – he would have to classify every cash transaction. As a side effect, that would render impossible automatic cash operation import from e-bank, POS and other similar systems. Clearly, that would be an overkill for small to medium business. Therefore, I'm not going to implement cash flow statement using direct method directly. And yes, according to generally accepted accounting standards, it is possible to present cash flow statement by direct method indirectly. However, going indirect direct way requires that the chart of accounts be structured in specific order to collect different types of information, e.g., all payable accounts (VAT payable, accounts payable, etc.) should be structured the same way as income accounts, which is rarely the case in real life. In our case, we cannot require a company to draw a specific chart of accounts that could support presenting cash flow statement by direct method indirectly. Therefore, we will also not implement cash flow statement using direct method indirectly.

It is notable that even though the standard-setting bodies encourage the use of the direct method, it is rarely used, for the excellent reason that the information in it is difficult to assemble; companies simply do not collect and store information in the manner required for this format. Therefore, our application will not stand out against other accounting applications by not implementing cash flow statement using direct method.

Indirect method is based on modifications of net income, which is the result of income statement, using the following (approximate) rules:

- When an asset (other than cash) increases, the Cash account decreases.
- When an asset (other than cash) decreases, the Cash account increases.
- When a liability increases, the Cash account increases.
- When a liability decreases, the Cash account decreases.
- When owner's equity increases, the Cash account increases.
- When owner's equity decreases, the Cash account decreases.

Cash flow for the purpose of cash flow statement equals:

(NET INCOME + NON-CASH EXPENSES – NON-CASH REVENUES)

– (Δ ASSETS – Δ CASH + Δ DEPRECIATION&AMORTIZATION – Δ REVALUATION – Δ ACCRUED REVENUE)

+ (Δ LIABILITIES – Δ ACCRUED EXPENSES)

+ (Δ OWNER'S EQUITY - Δ REVALUATION)

The fundamental accounting equation tells us that:

ΔLIABILITIES + ΔEQUITY + ΔREVENUES – ΔEXPENSES – ΔASSETS = 0

Therefore, we can eliminate those from the cash flow formula. The resulting formula is:

(NON-CASH EXPENSES – NON-CASH REVENUES)

– (– Δ CASH + Δ DEPRECIATION&AMORTIZATION – Δ REVALUATION – Δ ACCRUED REVENUE)

+ (– Δ ACCRUED EXPENSES)

+ (- Δ REVALUATION)

=

NON-CASH EXPENSES – NON-CASH REVENUES + Δ CASH - Δ DEPRECIATION&AMORTIZATION + Δ ACCRUED REVENUE – Δ ACCRUED EXPENSES

As non-cash expenses in fact are depreciation & amortization plus accrued expenses, and non-cash revenues in fact are accrued revenue, the equation becomes trivial:

CASH FLOW = Δ CASH

Therefore, if the only thing we needed was the final line, we could just check aggregate cash accounts balance change. However, the devil is in the details. The cash flow statement is not about the final line, which is obvious, but about showing the way that we take from the balance sheet and income statement to the actual cash, i.e., a cash flow statement represents the equation in the most expanded way.

Let's see how to calculate cash flow statement line by line (for detailed examples, see Cash Flow Statement (explanation), How to Prepare a Statement of Cash Flows Using the Indirect Method):

- Net income can be calculated by taking aggregate credit balance of all revenues and expenses accounts. We can get that from the balance sheet and income statement query by summing all the lines of type income_line.

- Non-cash expenses and revenues can be calculated by relating the relevant revenue and expenses accounts to the lines of cash flow statement that represent such expenses or revenues, e.g., depreciation expenses account. In this case, debit balance change should be added to the net income.
- Assets value change can be calculated by relating the assets costs accounts to the lines of cash flow statement that represent assets value change; assets depreciation, amortization and revaluation as well as accrued revenues are accounted for in the special accounts, that could be plainly ignored (not assigned to any cash flow statement line) effectively removing their values from the assets value change. Debit balance change for assets means increase of assets and to acquire some more assets you need to spend some cash. Therefore, debit balance change should be subtracted from the net income.
- Liabilities value change can be calculated by relating the liability accounts to the lines of cash flow statement that represent liability change; accrued expenses are accounted for in the special accounts, that could be plainly ignored (not assigned to any cash flow statement line) effectively removing their values from the liabilities value change. Debit balance change for liabilities means decrease of liabilities and to decrease liabilities you need to spend some cash. Therefore, debit balance change should be subtracted from the net income.
- Equity's value change can be calculated by relating the equity accounts to the lines of cash flow statement that represent equity value change; revaluation is accounted for in the special equity account, that could be plainly ignored (not assigned to any cash flow statement line) effectively removing its value from the equity value change. When calculating equity's value change accounts closure operation should be ignored (just like for income statement), because it adds net income to equity as retained earnings, which does not affect cash. Debit balance change for equity means decrease of equity, which in turn means that a company either decreased its nominal capital or paid some dividends, i.e., paid some money to its owners. Therefore, debit balance change should be subtracted from the net income.

There are few more tricky aspects of cash flow statement presentation due to the required classification specifics:

- Grouping of revenues and expenses into operating activities, investing activities and financing activities. E.g., a company has received some dividends from other company; if such revenues are classified as investing activities, then the amount of revenues should be subtracted from the operating activities (one line in the cash flow statement) and added to investing activities (another line the cash flow statement). Which brings us to requirement (possibility) to assign two cash flow statement lines for one account. In this case, the debit balance change in the operational activities line will increase the income; and the debit balance change in the operational activities line will decrease the income.
- Separate cash flow statement lines for fixed assets acquisition and sale. Which brings us to the same requirement of (possibility) to assign two cash flow statement lines for one account. However, in this case, one cash flow statement line should only take into account debit balance change (and ignore credit change) and the other cash flow statement line should only take into account credit balance change (and ignore debit change). Which brings us to the requirement to specify required balance type, i.e., partial balance.
- Cash and cash equivalents total amount at the beginning and at the end of the period should be presented. Which brings us to one more requirement to specify one more required balance type, i.e., final balance, not a balance per period as used in other statement lines.

Simple, yet too good to be true. Unfortunately (for developers), generally accepted accounting standards require excluding numerous non-cash transactions from the cash flow statement. E.g., a company acquires some fixed assets in exchange for some unsettled debt to the company. If we present cash flow statement as described above, the credit change to the debt will be effectively cancelled by the debit change in the fixed assets, i.e., wouldn't affect the bottom-line result. However, such operation does not involve any cash and, therefore, shall be excluded from the cash flow statement according to the requirements of generally accepted accounting standards. In some cases, such non-cash transactions can even render the cash flow statement invalid. E.g., we do not take into account balance changes in the fixed assets revaluation account; however, the revaluation result could be used to increase company's nominal capital. If we handle the equity's increase as described above, we will invalidate the statement by showing equity's increase (a) “out of nowhere” and (b) having nothing to do with cash. To further complicate the subject matter, you cannot even distinguish non-cash transactions by a document type. E.g., fixed assets sale (or even swap) is formalized as an invoice, which is typically a cash document, not to speak that a single invoice might actually contain some cash items and some non-cash items.

The bottom line – we cannot present a correct cash flow statement using only general ledger functionality. Functionality of individual document types cannot help with that either; the same document (type) at different times could be cash, non-cash or a mixture of the two. To present a correct cash flow statement, we need a deep knowledge of business transactions, their interrelationship and background. However, there is one good thing in this complicated situation: non-cash transactions that require special adjustments to the cash flow statement are relatively rare. Therefore, it wouldn't require unreasonable efforts to do manual adjustments. Actually, that way we follow the method that accountants use to draw a cash flow statement by hand: first create a preliminary statement using general ledger data as is (in other words – balance sheet and income statement), then add adjustments that cannot be derived from the ledger data.

In our data model, we present a preliminary cash flow statement using methods described above, i.e., set cash flow statement line values using ledger accounts balances subject to the account type specific rules. Hence, adjustments shall somehow adjust ledger accounts balances. Common method to change a balance of an account is either to debit or to credit it. E.g., if a transaction made undesirable debit entry, we add a credit entry with the same amount *et vice versa*. However, we cannot use general ledger for that purpose, because in this case, the adjustments shall only affect cash flow statement, i.e., the adjustments are not financial transactions (does not change company's financial state). Hence, the solution is to add a new data entity (database table) – `cash_flow_adjustments` – that has pretty much the same fields as `ledger_entries`: `id, account id, entry type` (debit/credit) and `amount`. In order to identify the (candidate) parent of cash flow adjustments, let's consider the following example (use case):

> An accountant formalizes sale of fixed-assets – building – by registering an invoice. The invoice creates the following ledger entries:
>
> - Debit Accounts receivable 121.000 EUR (total amount that the buyer has to pay);
> - Credit Assets costs 100.000 EUR (book value of building assuming it has never been used by the company and never been depreciated, i.e. acquisition value);
> - Credit VAT payable 21.000 EUR.
>
> An accountant knows from business background that the buyer is not going to pay the (full) price for a long time (couple of months or more). Therefore, for the cash flow statement presentation purposes, the following ledger entries shall not be included:
>
> - Credit Assets costs 100.000 EUR – because the buyer hasn't paid the price yet and the transaction at the moment is non-cash;
> - Debit Accounts receivable 100.000 EUR – amount that the buyer has to pay namely for the building (not tax) because it's a part of non-cash transaction and does not actually decrease cash flow while normally debit balance change for Accounts receivable decreases cash flow;
>
> In order to cancel the entries, an accountant adds the cash flow adjustments (!!! not ledger entries !!!): debits Assets costs and credits Accounts receivable with the same amounts.
>
> When an accountant registers a bank operation, which is a payment received from the buyer, he adds cash flow adjustments – credit Assets costs and debits Accounts receivable for the building value – because now the building is finally paid (cash received) and the transaction turned into a cash transaction.

As you can see, the adjustments made allow us to present a correct cash flow statement for any period. If the period jumps in the middle of full sale cycle, it will not show cash flow increase due to the sale. If the period encompass all the sale cycle, it will show cash flow increase due to the sale. If there had been partial payments, the cash flow statement would remain correct for any intermediate period.

It is notable that the cash flow adjustments always adjusts some ledger transactions. The adjustments never happens "out of nowhere", because in order to adjust something you should have what to adjust. It follows from the fundamental accounting equation that the cash can never change if other parts of equation remains the same. Therefore, although a bit counterintuitive, the cash flow adjustments parent entity is a transaction (that changes some parts/variables of the fundamental accounting equation but happens not to affect cash part due to some business background).

To sum it up, we have the following requirement for cash flow statement model:

- Cash flow statement lines shall have the following value types: debit and credit. It defines which type of balance is considered as positive value in the statement line.
- Cash flow statement lines shall have the following balance types: full, per period, debit only and credit only. It defines which types of ledger account turnovers are included when calculating the balance.
- Cash flow statement lines shall have flag `is_net_income`. Obviously, only one line within the statement can have this flag set to true ("`1`"). It defines that a completely different algorithm (subquery) should be used to fetch values.
- Cash flow statement is a hierarchic document similar to income statement. Therefore, we use nested set model, i.e., add technical fields – `left_index, right_index, visible_index` and `line_type` – that serve the same purpose as for the income statement.
- Every account within the chart of accounts shall be able to relate up to two cash flow statement lines (0…2). Which brings us to many-to-many relation and a technical table `cash_flow_line_assignments`
- A special type of entity – `cash_flow_adjustments` – is required that would allow an accountant to add adjustments for a non-cash transaction.

And the resulting schema (relevant fragment) is:

Having this schema in place, we can now fetch cash flow statement structural data in the same way we did for balance sheet and income statement.

Next, we can now fetch actual cash flow statement for the periods requested by a user, e.g., let the first period be from 2017-01-01 to 2017-12-31 and the second period from 2018-01-01 to 2018-12-31:

Hide Shrink ▲ Copy Code

```
SELECT line.printed_no as RowNo, line.line_text AS RowText,
SUM(
CASE WHEN line.line_type<>'line' THEN NULL
WHEN line.is_net_income > 0 THEN
(SELECT SUM(CASE WHEN e.entry_type='D' THEN -e.amount ELSE e.amount END)
FROM documents d
LEFT JOIN transactions t ON t.document_id = d.id
LEFT JOIN ledger_entries e ON e.transaction_id = t.id
LEFT JOIN accounts a ON a.id = e.account_id
LEFT JOIN balance_and_income_lines l ON l.id = a.balance_and_income_line_id
WHERE t.transaction_date >= '2017-01-01' AND t.transaction_date < '2018-01-01'
AND d.document_type<>13 AND l.line_type='income_line')
ELSE
(CASE WHEN child_line.id IS NULL THEN
(CASE line.balance_type
WHEN 'total' THEN val.TotalDebitBefore - val.TotalCreditBefore + _
                  val.DebitFirstPeriod - val.CreditFirstPeriod
WHEN 'per_period' THEN val.DebitFirstPeriod - val.CreditFirstPeriod
WHEN 'D' THEN val.DebitFirstPeriod
ELSE - val.CreditFirstPeriod
END) * (CASE WHEN line.value_type='D' THEN 1 ELSE -1 END)
ELSE
(CASE child_line.balance_type
WHEN 'total' THEN val.TotalDebitBefore - val.TotalCreditBefore + _
                  val.DebitFirstPeriod - val.CreditFirstPeriod
WHEN 'per_period' THEN val.DebitFirstPeriod - val.CreditFirstPeriod
WHEN 'D' THEN val.DebitFirstPeriod
ELSE - val.CreditFirstPeriod
END) * (CASE WHEN child_line.value_type='D' THEN 1 ELSE -1 END)
END)
END) AS FirstPeriodValue,
SUM(
CASE WHEN line.line_type<>'line' THEN NULL
WHEN line.is_net_income > 0 THEN
(SELECT SUM(CASE WHEN e.entry_type='D' THEN -e.amount ELSE e.amount END)
FROM documents d
LEFT JOIN transactions t ON t.document_id = d.id
LEFT JOIN ledger_entries e ON e.transaction_id = t.id
LEFT JOIN accounts a ON a.id = e.account_id
LEFT JOIN balance_and_income_lines l ON l.id = a.balance_and_income_line_id
WHERE t.transaction_date >= '2018-01-01' AND t.transaction_date <= '2018-12-31'
AND d.document_type<>13 AND l.line_type='income_line')
ELSE
(CASE WHEN child_line.id IS NULL THEN
(CASE line.balance_type
WHEN 'total' THEN val.TotalDebitBefore - val.TotalCreditBefore + val.DebitFirstPeriod
- val.CreditFirstPeriod + val.DebitSecondPeriod - CreditSecondPeriod
WHEN 'per_period' THEN val.DebitSecondPeriod - val.CreditSecondPeriod
WHEN 'D' THEN val.DebitSecondPeriod
ELSE - val.CreditSecondPeriod
END) * (CASE WHEN line.value_type='D' THEN 1 ELSE -1 END)
ELSE
(CASE child_line.balance_type
WHEN 'total' THEN val.TotalDebitBefore - val.TotalCreditBefore + val.DebitFirstPeriod
- val.CreditFirstPeriod + val.DebitSecondPeriod - CreditSecondPeriod
WHEN 'per_period' THEN val.DebitSecondPeriod - val.CreditSecondPeriod
WHEN 'D' THEN val.DebitSecondPeriod
ELSE - val.CreditSecondPeriod
END) * (CASE WHEN child_line.value_type='D' THEN 1 ELSE -1 END)
END)
END) AS SecondPeriodValue
FROM cash_flow_lines AS line
LEFT JOIN cash_flow_lines AS child_line ON child_line.left_index _
                          BETWEEN line.left_index AND line.right_index
LEFT JOIN (
SELECT c.cash_flow_line_id AS LineId,
SUM(CASE WHEN t.transaction_date < '2017-01-01' AND e.entry_type='D' _
         THEN e.amount ELSE 0.0 END) AS TotalDebitBefore,
SUM(CASE WHEN t.transaction_date < '2017-01-01' AND e.entry_type='C' _
         THEN e.amount ELSE 0.0 END) AS TotalCreditBefore,
SUM(CASE WHEN t.transaction_date >= '2017-01-01' AND t.transaction_date < '2018-01-01'
AND e.entry_type='D' AND d.document_type <> 13 THEN e.amount ELSE 0.0 END) AS
DebitFirstPeriod,
SUM(CASE WHEN t.transaction_date >= '2017-01-01' AND t.transaction_date < '2018-01-01'
AND e.entry_type='C' AND d.document_type <> 13 _
                     THEN e.amount ELSE 0.0 END) AS CreditFirstPeriod,
SUM(CASE WHEN t.transaction_date >= '2018-01-01' AND e.entry_type='D'
AND d.document_type <> 13 THEN e.amount ELSE 0.0 END) AS DebitSecondPeriod,
SUM(CASE WHEN t.transaction_date >= '2018-01-01' AND e.entry_type='C'
AND d.document_type <> 13 THEN e.amount ELSE 0.0 END) AS CreditSecondPeriod
FROM documents d
LEFT JOIN transactions t ON t.document_id = d.id
LEFT JOIN ledger_entries e ON e.transaction_id = t.id
LEFT JOIN cash_flow_line_assignments c ON c.account_id = e.account_id
WHERE t.transaction_date <='2018-12-31' AND NOT c.id IS NULL GROUP BY
c.cash_flow_line_id
UNION ALL
SELECT c.cash_flow_line_id AS LineId,
SUM(CASE WHEN t.transaction_date < '2017-01-01' AND e.entry_type='D' _
         THEN e.amount ELSE 0.0 END) AS TotalDebitBefore,
SUM(CASE WHEN t.transaction_date < '2017-01-01' AND e.entry_type='C' _
         THEN e.amount ELSE 0.0 END) AS TotalCreditBefore,
SUM(CASE WHEN t.transaction_date >= '2017-01-01' AND t.transaction_date < '2018-01-01'
AND e.entry_type='D' THEN e.amount ELSE 0.0 END) AS DebitFirstPeriod,
SUM(CASE WHEN t.transaction_date >= '2017-01-01' AND t.transaction_date < '2018-01-01'
AND e.entry_type='C' THEN e.amount ELSE 0.0 END) AS CreditFirstPeriod,
SUM(CASE WHEN t.transaction_date >= '2018-01-01' AND e.entry_type='D'
THEN e.amount ELSE 0.0 END) AS DebitSecondPeriod,
SUM(CASE WHEN t.transaction_date >= '2018-01-01' AND e.entry_type='C'
THEN e.amount ELSE 0.0 END) AS CreditSecondPeriod
FROM transactions t
LEFT JOIN cash_flow_adjustments e ON e.transaction_id = t.id
LEFT JOIN cash_flow_line_assignments c ON c.account_id = e.account_id
WHERE t.transaction_date <='2018-12-31' AND NOT c.id IS NULL GROUP BY
c.cash_flow_line_id
) AS val ON val.LineId = child_line.id
GROUP BY line.id ORDER BY line.visible_index;
```

The query is more complex than the one, we used for balance sheet and income statement. It is also slower almost twice as we use two heavy load subqueries instead of one. The query has the following stages:

- First, we get ledger account turnovers for the periods required and group it by cash flow statement line. It's pretty much the same as for balance sheet and income statement except for the different grouping criteria. The only significant difference is the way we treat accounts closing operation. As for the cash flow presentation purposes, the accounts closing operations shall always be excluded, we do not select their values in separate fields; instead, we directly exclude them from the ledger account turnovers. It's a heavy load (sub)query, because of the large ledger table size.
- Next, we get cash flow adjustments turnovers for the periods required and group it by cash flow statement line. It's relatively low load (sub)query, because the cash flow adjustments are relatively rare, hence, small table size.
- Next, we do `UNION ALL` because both types of turnovers have the same meaning for the cash flow statement.
- Next, we do the parent query – fetch cash flow statement tree the same way as we did for balance sheet and income statement. The query uses very small source sets fetched by the previous query. Therefore, it is very fast.
- Finally, we assemble actual cash flow statement line values using the aggregated values from the previous subquery. The cash flow statement business logic is much more complex than the balance sheet. Therefore, we have a much more complex CASE. The first case chooses an algorithm to use for the statement line type:

  1. if it's a header line, no calculations are required;
  2. If it's a net income line, we use special subquery to fetch net income as previously discussed; It's a heavy load subquery, because of the large ledger table size;
  3. If it's any other statement line, we use the aggregated values from the previous subquery. The case for common statement line chooses an algorithm to use for a parent and child lines. In contrast with a balance sheet, cash flow statement parent lines do not have a particular balance type. E.g., operational cash flow line takes net income (credit balance) and adds depreciation (debit balance), which increases the total group value. Therefore, for parent lines, we need to use value calculated by child irrespective of the value balance type:

     - if it's a child line (that has no child lines), we calculate values using the line settings (value type and balance type);
     - If it's a parent line (that has a child line), we calculate values using the child line settings.

In real life scenarios, financial statements (including cash flow statement) are usually fetched together with the account level balance, which uses the same subquery but grouped by account instead of balance sheet, income statement or cash flow statement line. As namely the subquery is doing heavy lifting, executing separate queries for account level balance and balance sheet, income statement and cash flow statement essentially triples the database workload for the very similar data. For that reason, at the application level I opted for only fetching account level balance and doing the transforms for balance sheet, income statement and cash flow statements within the application. This approach is at least three times faster than using three heavy queries.

Equity Statement

The last financial statement to implement within this article is the equity statement. Equity statement, often referred to as Statement of Retained Earnings in U.S. GAAP, details the change in owners' equity over an accounting period by presenting the movement in reserves comprising the shareholders' equity.

Shareholders' equity is comprised of the following components:

- Shares nominal value
- Shares par value (when the shares issued were subscribed/bought for greater than nominal value)
- Treasury shares (shares owned by the company itself)
- Retained earnings
- Revaluation reserve
- Other reserves

Those components (more detailed if needed) are always reflected in the chart of accounts, i.e., each equity component has a dedicated account.

Movement in shareholders' equity over an accounting period comprises the following elements (see Stockholders' Equity (Explanation) for details):

- Net profit or loss during the accounting period
- Increase or decrease in shares amount or/and nominal value
- Increase or decrease in reserves
- Dividend payments to shareholders
- Gains and losses recognized directly in equity
- Effect of changes in accounting policies
- Effect of correction of prior period error

The purpose of the equity statement is to present changes made by various operations to the equity components. Therefore, the structure of the equity statement is fundamentally different from the balance sheet, income statement and cash flow statement. Instead of the hierarchical structure, equity statement has a matrix structure where the columns represent equity components and the rows represent types of equity movements (operations). Simplified example of equity statement looks like this:

	Share Capital	**Retained Earnings**	**Revaluation Surplus**	**Total Equity**
Balance at the beginning of the first period	**100.000**	**30.000**	**-**	**130.000**
Changes in accounting policy	-	-	-	-
Correction of prior period error	-	-	-	-
Restated balance at the beginning of the first period	**100.000**	**30.000**	**-**	**130.000**
Changes in equity for the first period				
Issue of share capital	-	-	-	-
Income for the year	-	25.000	-	25.000
Revaluation gain	-	-	10.000	10.000
Dividends	-	(15.000)	-	(15.000)
Balance at the end of the first period	**100.000**	**40.000**	**10.000**	**150.000**
Changes in equity for the second period				
Issue of share capital	-	-	-	-
Income for the year	-	30.000	-	30.000
Revaluation gain	-	-	5.000	5.000
Dividends	-	(20.000)	-	(20.000)
Balance at the end of the second period	**100.000**	**50.000**	**15.000**	**165.000**

As you can see, the rows contain some classification of transactions by transaction (document) type (e.g. dividends) and equity type/account (e.g. retained earnings) that are further grouped by period thus creating sort of third dimension of the matrix. There is no way that we can do classification of transactions using only ledger data. We also cannot directly bind transactions (documents) with the equity statement rows, as the change in the statement structure would render impossible presenting the statement in consistent way. Cannot compare two periods using single statement structure if the transactions are bound to multiple different structures. Therefore, the rows can only be bound to the (source) document types. Which in turn will force us to implement a fine-grained equity related document types. It is possible (especially taking into account extensions) that the (source) document type detalization is even to detailed for the equity statement. Hence, it is possible that a few document types will be assigned to a single equity statement row.

Each equity statement column represent a type of the equity. As mentioned before, equity structure is revealed in the chart of accounts. However, the chart of accounts could provide more detailed structure than required by the equity statement. Therefore, each equity statement column could be assigned to one or more accounts in the chart of accounts but not *vice versa*.

To sum it up, in order to implement equity statement, we need:

- A list of equity statement columns that are bound to one or more accounts in the chart of accounts. Equity balance is always of credit type. Therefore, we do not need to define balance type for this statement.
- A list of equity statement rows (lines) that are bound to one or more (source) document type. Which brings us to a technical equity_line_assignments
- As you can see in the example, there are seven equity statement line types that define overall structure of the statement: `initial_balance`, `cumulative_delta`, `zero_balance`, `first_delta`, `first_balance`, `second_delta`, `second_balance`.

And the resulting schema (relevant fragment) is:

Having this schema in place, we can now fetch actual equity statement for the periods requested by a user, e.g., let the first period be from 2017-01-01 to 2017-12-31 and the second period from 2018-01-01 to 2018-12-31:

Hide Shrink ▲ Copy Code

```
SELECT l.printed_no AS LineNo, l.line_text AS LineText, _
            l.line_type AS LineType, v.ColumnText, v.ColumnIndex,
SUM(CASE line_type
WHEN 'initial_balance' THEN v.BalanceBefore
WHEN 'cumulative_delta' THEN v.CumulativeDelta
WHEN 'zero_balance' THEN v.BalanceBefore + v.CumulativeDelta
WHEN 'first_delta' THEN v.FirstPeriodDelta
WHEN 'first_balance' THEN v.BalanceBefore + v.CumulativeDelta + v.FirstPeriodDelta
WHEN 'second_delta' THEN v.SecondPeriodDelta
ELSE v.BalanceBefore + v.CumulativeDelta + v.FirstPeriodDelta + v.SecondPeriodDelta END)
_
                        AS Value
FROM equity_lines l
LEFT JOIN (
SELECT c.id AS ColumnId, c.column_text AS ColumnText, _
               c.visible_index AS ColumnIndex, l.id AS LineId,
SUM(CASE WHEN t.transaction_date < '2017-01-01' _
AND e.entry_type='C' THEN e.amount ELSE 0.0 END)
- SUM(CASE WHEN t.transaction_date < '2017-01-01' _
AND e.entry_type='D' THEN e.amount ELSE 0.0 END) AS BalanceBefore,
SUM(CASE WHEN t.transaction_date >= '2017-01-01' _
AND t.transaction_date < '2018-01-01' AND e.entry_type='C' _
AND l.line_type = 'first_delta' THEN e.amount ELSE 0.0 END)
- SUM(CASE WHEN t.transaction_date >= '2017-01-01' _
AND t.transaction_date < '2018-01-01' AND e.entry_type='D' _
AND l.line_type = 'first_delta' THEN e.amount ELSE 0.0 END) AS FirstPeriodDelta,
SUM(CASE WHEN t.transaction_date >= '2018-01-01' AND e.entry_type='C' _
AND l.line_type = 'second_delta' THEN e.amount ELSE 0.0 END)
- SUM(CASE WHEN t.transaction_date >= '2018-01-01' AND e.entry_type='D' _
AND l.line_type = 'second_delta' THEN e.amount ELSE 0.0 END) AS SecondPeriodDelta,
SUM(CASE WHEN t.transaction_date >= '2017-01-01' AND e.entry_type='C' _
AND l.line_type = 'cumulative_delta' THEN e.amount ELSE 0.0 END)
- SUM(CASE WHEN t.transaction_date >= '2017-01-01' AND e.entry_type='D' _
AND l.line_type = 'cumulative_delta' THEN e.amount ELSE 0.0 END) AS CumulativeDelta
FROM documents d
LEFT JOIN equity_line_assignments s ON d.document_type=s.document_type
AND (d.extended_document_type_id = s.extended_document_type_id
OR (d.extended_document_type_id IS NULL AND s.extended_document_type_id IS NULL))
LEFT JOIN equity_lines l ON l.id = s.equity_line_id
LEFT JOIN transactions t ON t.document_id = d.id
LEFT JOIN ledger_entries e ON e.transaction_id = t.id
LEFT JOIN accounts a ON a.id = e.account_id
LEFT JOIN equity_columns c ON c.id = a.equity_column_id
WHERE t.transaction_date <='2018-12-31' _
AND NOT l.id IS NULL AND NOT c.id IS NULL GROUP BY l.id, c.id
) AS v ON v.LineId=l.id OR l.line_type IN('initial_balance', _
'zero_balance', 'first_balance', 'second_balance')
GROUP BY l.id, v.ColumnId ORDER BY l.line_type, l.visible_index, v.ColumnIndex;
```

The query is less complex than the ones we used for previous financial statements. It is also fast one as we filter out significant part of (source) documents and ledger accounts. The query has the following two stages:

- First, we get ledger account turnovers for the periods required and group it by equity statement column and line. As the document type is always bound to 0…1 equity statement line, the grouping by document type and by equity statement line are functionally equivalent.
- Next, we calculate the actual value for each line and column subject to the type of equity statement line.

The query *inter alia* uses grouping by document type, which cannot be deduced from ledger account level balances. Therefore, we will not be able to reuse ledger account level balances and will have to use the query specifically for equity statement.

Conclusion

In this article, we have developed database schema for the core accounting infrastructure: general ledger, chart of accounts, (source) documents and financial statements.

We also proved that the application schema is capable of handling one million documents with an acceptable performance. Which means that the application will be fit for companies with up to 100.000 operations per year.

The next article will be dedicated to the finalization of the accounting infrastructure design: company profile, person profiles, costs centres, bank and cash fund accounts and all the other entities that are used across all of the specialized accounting documents.

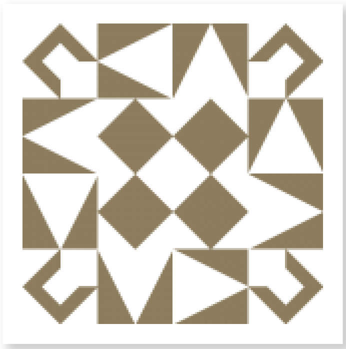
History

- 28th July, 2019: Initial version
- 9th August, 2019: Switched to `VARCHAR` type where appropriate

License

Share

About the Author



Niemand25
Business Analyst Linden
Lithuania

I'm a lawyer in a law firm. Programing is my hobby.

Comments and Discussions

Thank you for saving me hundreds of hours
Mike Thomson 11-Apr-20 23:48

The new article
Member 14739127 8-Feb-20 19:45

Re: The new article
Member 14721959 9-Feb-20 4:17

Re: The new article
Niemand25 14-Feb-20 11:01

inserts
Member 14721959 8-Feb-20 17:11

Re: inserts
Member 14843424 25-May-20 11:48

Great article!!
Member 14592783 13-Sep-19 19:30

Re: Great article!!
Niemand25 22-Sep-19 1:01

valuable
Member 2831101 16-Aug-19 5:27

Great Help
Member 13377491 12-Aug-19 21:35

Interesting but biased
Eric Lapouge 11-Aug-19 4:56

Re: Interesting but biased
Niemand25 11-Aug-19 5:22

Re: Interesting but biased
Eric Lapouge 11-Aug-19 5:36

Re: Interesting but biased
Niemand25 11-Aug-19 7:45

Re: Interesting but biased
Eric Lapouge 11-Aug-19 8:10

Re: Interesting but biased
Niemand25 11-Aug-19 8:34

Re: Interesting but biased
Eric Lapouge 11-Aug-19 10:19

کوپلینگ هیدرولیکی
teknosanat 10-Aug-19 2:39

Brilliant article - voted 5
arzulfi 9-Aug-19 20:19

My vote of 5
NelsonCosta 6-Aug-19 9:52

My vote of 5

arroway 30-Jul-19 0:53

Thanks
aldo hexosa 29-Jul-19 5:37

Re: Thanks
Niemand25 29-Jul-19 8:13

Refresh 1

General News Suggestion Question Bug Answer Joke Praise Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.