Feeding a temporal data warehouse from CDC (for the best time-travel experience)

Lukasz Kaminski
Dadrico BV
lukasz.kaminski@dadrico.com
2025-04-05

Abstract

CDC (Change Data Capture) solutions let us see every little change that happens on a database. It is possible to turn a CDC change register into a set of temporal tables that can be pushed through ETL to form a fully temporal data warehouse, where any change in the source is represented in the output tables (like facts or dimensions). In this paper we shortly describe our approach to temporal data warehousing and show how can the data from CDC can efficiently be extracted, transformed and loaded into the serving layer tables.

Contents

Introduction

What is a temporal data warehouse?

A temporal data warehouse is a specialized type of a data warehouse designed to manage and analyse time-varying data. Unlike traditional data warehouses, which typically store the most current or aggregated state of data, a temporal data warehouse retains historical changes, enabling users to track how data evolves over time.

In this paper I will limit the time aspect of a temporal data warehouse just to the transaction time (TT), which is the time when the data is present in the source. Following [Mahmood, 2010] I will be referring to the timeline of transaction times as an *activity timeline* with *active_from* and *active_to* as the period boundaries. An information is *active* at a given moment if it is present in the source at that moment. All other dates present in the tables will be treated as attributes, including the validity times (VT). Presence of both validity timeline and activity timeline will result in a bi-temporal table and will pose other challenges, but this is out of the scope of this paper.

> Picture a CRM system that a DWH can access through a relational database. The current state of a given area of the system – let's say customer information screen – is represented in a database table. If an employee changes the address of the customer, this will overwrite the previous address and only the new address will be visible in the source table and accessible by the DWH. The old address is no longer *active* in the source after the change has been done.
>
> Not all information behaves like this. Let's say that a customer has a default discount rate that is valid in a given year. The CRM interface will show a table of years with respective discount rates. All this information will be stored in the database and will be visible to the DWH at any time. The 2023 discount might be not valid in 2024, but from the temporal data warehouse perspective both are *active* because both are still present in the source.

A common approach in data warehousing is to keep the history of record *activity* in a persistent staging area, but use only the current records of the staging tables to feed the subsequent layers of the data warehouse. All non-staging tables in such a data warehouse contain only the current state of information.

In a temporal data warehouse as described here, the changes that are present in the staging tables are propagated to other layers and are visible in the serving layer (e.g. facts and dims). As a result, the users of the data warehouse can see how the information they care about has changed over time.

Why would you want a temporal data warehouse?

Short answer – it gives you time travel capability. In our version of a temporal data warehouse, you can travel back in time to see how the data in the facts, the dimensions or even the data marts would look like if you were to run the ETL at any moment in the past and this is simply available in all the tables in the data warehouse after the daily refresh.

A non-temporal data warehouse is good enough for most purposes. The management reporting, campaign scoring, capacity management – these are the types of use cases in which the general picture is usually sufficient and sub-permille erroneous data does not normally cause substantial harm.

On the other hand, if you use the data warehouse for auditing and compliance purposes, it might be crucial to make sure that every datapoint is correctly represented in your serving layer. You might recognize questions like "On the screen in the application XYZ I see THIS and I expect to see THIS in the FCT_XYZ table as well, but I see SOMETHING ELSE instead." It used to bother me to have to reply with "The data has changed and I can no longer see what you saw". In the temporal data warehouse, you are able to see the timeline of all the changes of the source data at any ETL stage, which makes answering these types of questions possible.

The ability to answer such questions becomes absolutely crucial when your source systems struggle with data quality issues. You can easily track down any suspicious changes in the data mart to the staging tables even if the change is already in the past.

You can also build checks to show if your data warehouse represents the source correctly to a single data point, because your data warehouse can tell you how many records were there in the database and in the serving-layer tables at any time in the past.

Another benefit of a temporal data warehouse is the ability to serve consistent data in a scenario where the data is refreshed throughout the day from different sources at different moments. Let's say that we need to refresh financial data every hour for the finance department, but the logistics needs to have it near-real time. In a regular data warehouse, all is fine until we need to present some logistic information in the reports for finance – if we just combine our serving layer tables, we will present finance with an inconsistent view. In a temporal data warehouse, we can just select the logistic data that was active at the time of the last financial data refresh.

Updating a temporal data warehouse

There is a number of ways to create temporal data in the data warehouse. One way is state processing. This approach mentioned in [Rahman, 2008] suggests running frequent updates of the data warehouse while keeping track of what has changed in the output tables. That means that the current status of the data is being loaded every time and that the timeline is being built at the end of the process, while loading the data to the output. Technically the loading process is no different from the persistent staging, it is just applied at every level of the data warehouse.

The granularity of the timeline is therefore limited to how quickly can a refresh be run. Another serious drawback of this method is that whenever there is an issue with the data warehouse refresh process, the granularity changes – due to the temporary malfunctions of ETL process, we do in fact loose data that we would otherwise have.

What is CDC (Change Data Capture)?

CDC solution is used to identify and track changes made to the data in a database or a system in order to propagate it to other systems. It is primarily used in data integration and replication.

As an example, let's use Microsoft SQL Server CDC solution. When we enable CDC on a given table in SQL Server database, a new table is created that keeps the history of changes for a given period (let's say – one week retention). Apart from all the columns that are present in the original table, the CDC change-table will have 6 extra columns (as described in the docs), out of which we will use just 3:

column name	Description	example value
__$start_lsn	log sequence number (LSN) associated with the commit transaction for the change; this value can be looked up in another CDC table - lsn_time_mapping to get the transaction begin and end times	0x0000B756003D1C480006
__$end_lsn	not used	NULL
__$seqval	sequence of the operation as represented in the transaction log	0x0000B756003D1C480003
__$operation	data manipulation language (DML) operation associated with the change 1 = delete (a record state just before deletion), 2 = insert (a record state just after insert), 3 = update (a record state just before update), 4 = update (a record state just after update)	2
__$update_mask	a bit mask based upon the column ordinals of the change-table identifying those columns that changed	0x03FFFFFFFF
__$command_id	order of operations within a transaction	1

As you can see CDC preserves all the changes of our source tables on a transaction level. All the operations performed on a record are stored and kept for a set retention period. There can be multiple changes of a given record within a day, or even within one second. It is up to us to determine the time granularity required for our data. For practical reasons we tend to use one second for granularity, which means that while reading from a CDC change table, if we find more than one change of a given record within one second (be it in multiple transactions or in one), we will only take the last record of such group, rounding the transaction end timestamp to the next full second to get our point-in-time timestamp. This way we can avoid the necessity to use milliseconds in order to show the timeline correctly and the one-second granularity is sufficient for all purposes we have stumbled upon so far.

When reading from CDC to the staging we will create a timeline of all changes that happened since the last time our data warehouse has successfully refreshed (marking the beginning of the increment window) until the start of the current run (the end of the increment window). The duration of the window can be an hour, a day or a week. No data is lost unless the window duration is longer than the retention period set for CDC. Even if the refresh frequency is low, all the changes will still be processed and stored within a staging table – this is made possible by timeline processing.

Moving all the changes from CDC to STAGING

Within this approach all the data manipulation processes operate on the activity timelines instead of just on the current states. Let's convert CDC data into a timeline. Let's take a CDC table example:

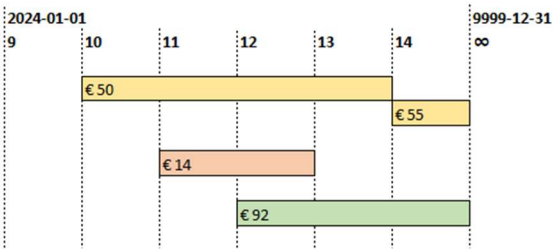
	__$lsn_transaction_end	__$operation	invoice_id	value	date	__$command_id
1	2024-01-01 10:00:00	2 (insert)	101	€ 50	2023-12-20	1
2	2024-01-01 11:00:00	3 (before update)	91	€ 12	2023-12-11	1
3	2024-01-01 11:00:00	4 (after update)	91	€ 14	2023-12-11	1
4	2024-01-01 12:00:00	2 (insert)	104	€ 90	2023-12-23	1
5	2024-01-01 12:00:00	3 (before update)	104	€ 90	2023-12-23	2
6	2024-01-01 12:00:00	4 (after update)	104	€ 92	2023-12-23	2
7	2024-01-01 13:00:00	1 (delete)	91	€ 14	2023-12-11	1
8	2024-01-01 14:00:00	3 (before update)	101	€ 50	2023-12-11	1
9	2024-01-01 14:00:00	4 (after update)	101	€ 55	2023-12-11	1

We do not need to use the type-3 records (before update). Also, when an insert and an update on the same record is done within one transaction, we can disregard all records except for the last one.

The time interval representation will be from-inclusive to-exclusive (or closed-open as described in [Kvet, 2013]), which means that the from-timestamp will be the first moment the record is active and the to-timestamp will be the first moment the record is no longer active (it is thus the same as the from-timestamp of the following activity period)

Based on that, the grayed-out rows will be disregarded and the remaining four records will result in four timeline records:

	invoice_id	value	date	active_from	active_to
1	101	€ 50	2023-12-20	2024-01-01 10:00:00	2024-01-01 14:00:00
2	101	€ 55	2023-12-20	2024-01-01 14:00:00	9999-12-31 00:00:00
3	91	€ 14	2023-12-11	2024-01-01 11:00:00	2024-01-01 13:00:00
4	104	€ 92	2023-12-23	2024-01-01 12:00:00	9999-12-31 00:00:00



row	explanation
1	the timeline begins at 10:00 (an insert CDC-row 1) and ends at 14:00, because there is an update of the same key (CDC-row 9)
2	the timeline begins at 14:00 (an update CDC-row 9) and ends in the year 9999 which is an agreed end-of-time indication. It means that the record remains active
3	the timeline begins at 11:00 (an update CDC-row 3) and ends at 13:00 (a delete CDC-row 7)
4	the timeline begins at 12:00 (an update CDC-row 6) and ends in the year 9999

Note that the invoice date (column: date) is in no way related to the activity timeline. A value of the date field is a non-technical column and is just treated as an attribute of an invoice.

Also note that the unique key of the resulting table is now composite and is composed of invoice_id and active_from columns. In order to simplify communication, we tend to speak about a "timeline key" of a table, which is just what would the PK of a given table be if not for the timeline aspect. For the rest of the paper whenever a key of a table is mentioned, we actually mean the timeline key.

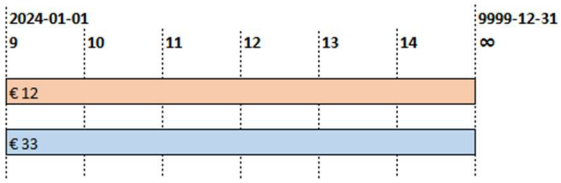
Such timeline will be applied on a staging table as an increment following this formula:

```
For each key value present in the input do:

    • If there are no records with this key value in the output,
      add all records from the input to the output.
    • If there are records with this key value in the output, overwrite the
      corresponding part of the timeline of the output with the timeline of the input
      (leaving all values prior to the first input active_from intact in the output,
      replacing all values after that moment with the input timeline).

Leave all records that are present in the output, but absent in the input, intact in
the output.
```

Example:

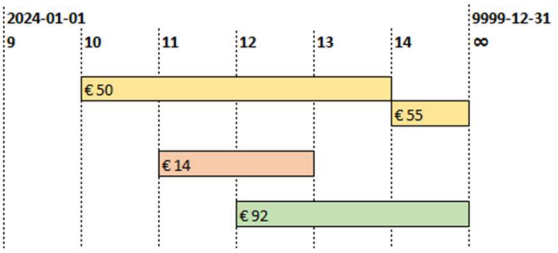
Before the staging was loaded, we had two records in the persistent staging table (OUTPUT):

	invoice_id	value	date	active_from	active_to
1	91	€ 12	2023-12-11	2023-12-18 09:00:00	9999-12-31 00:00:00
2	99	€ 33	2023-12-13	2023-12-18 09:00:00	9999-12-31 00:00:00



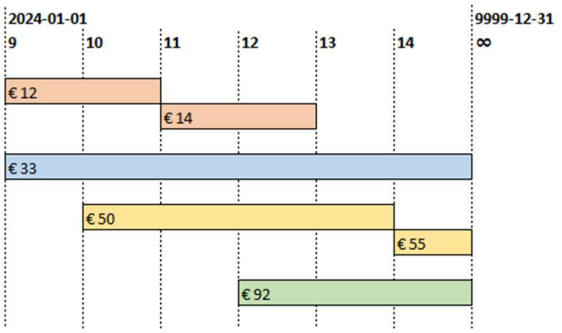
When we apply the changes from our CDC table (INPUT)...

	invoice_id	value	date	active_from	active_to
1	101	€ 50	2023-12-20	2024-01-01 10:00:00	2024-01-01 14:00:00
2	101	€ 55	2023-12-20	2024-01-01 14:00:00	9999-12-31 00:00:00
3	91	€ 14	2023-12-11	2024-01-01 11:00:00	2024-01-01 13:00:00
4	104	€ 92	2023-12-23	2024-01-01 12:00:00	9999-12-31 00:00:00



...this is what we should see in the persistent staging table (OUTPUT after load):

	invoice_id	value	date	active_from	active_to
1	91	€ 12	2023-12-11	2023-12-18 09:00:00	2024-01-01 11:00:00
2	91	€ 14	2023-12-11	2024-01-01 11:00:00	2024-01-01 13:00:00
3	99	€ 33	2023-12-13	2023-12-18 09:00:00	9999-12-31 00:00:00
4	101	€ 50	2023-12-20	2024-01-01 10:00:00	2024-01-01 14:00:00
5	101	€ 55	2023-12-20	2024-01-01 14:00:00	9999-12-31 00:00:00
6	104	€ 92	2023-12-23	2024-01-01 12:00:00	9999-12-31 00:00:00



row	explanation
1	the active_from value is unchanged compared to the previous state, but the active_to has been updated from year 9999 to 11:00, because this was the moment this key got an update
2	this is the update mentioned above and is inserted from the input
3	this key was not present in the input at all, so this record remains unchanged
4	inserted from the input
5	inserted from the input
6	inserted from the input

Timeline processing

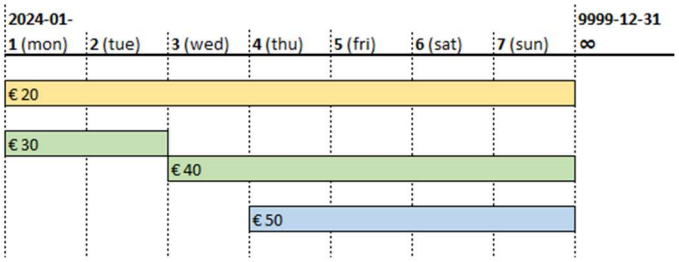
We have just seen how several changes of one record can be recorded in the persistent staging table. The same principle will be used throughout the data warehouse. At each stage of processing all the tables will have a timeline. These tables will need to be aggregated or joined with each other in order to bring the data to the format defined for the serving layer. All these operations need to process the timelines correctly.

Below we will show two examples of timeline operations. In order to make this more readable we will operate on days instead of seconds/hours, but the same principle would apply with any time granularity.

Temporal aggregation example

Let's say we want to aggregate the payments on invoices. In our input we have 3 different payments for the same invoice (invoice_id = 201).

	payment_id	invoice_id	value	active_from	active_to
	PK	FK			
1	111	201	€ 20	2024-01-01	9999-12-31
2	112	201	€ 30	2024-01-01	2024-01-03
3	112	201	€ 40	2024-01-03	9999-12-31
4	113	201	€ 50	2024-01-04	9999-12-31



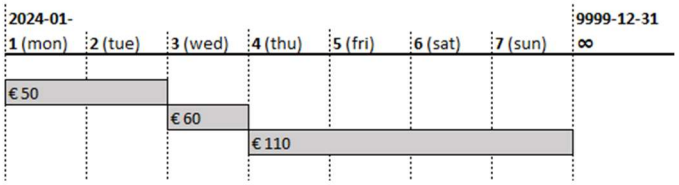
row	explanation
1	a payment inserted on Monday that never changed
2	the second payment also added on Monday, which value was later corrected on Wednesday
3	the corrected record of the second payment
4	the third payment that was inserted on Thursday

The question is: How much has already been paid for each invoice?

We are aggregating by invoices, so for the group-by column we will use the **invoice_id** and this becomes our timeline key of the output table. The payment_id won't appear in the output dataset. The column **sum_value**, will contain **sum(value)**.

And this is the output:

	invoice_id	sum_value	active_from	active_to
	PK			
1	201	€ 50	2024-01-01	2024-01-03
2	201	€ 60	2024-01-03	2024-01-04
3	201	€ 110	2024-01-04	9999-12-31

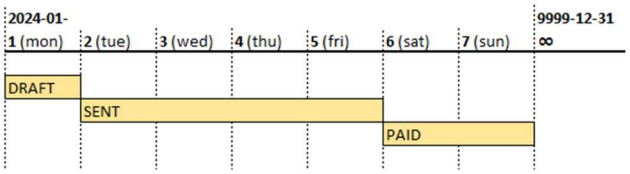


row	explanation
1	the sum of €20 + €30 = €50 – the only 2 records active in this period
2	the sum of €20 + €40 = €60 – the only 2 records active in this period
3	the sum of €20 + €40 + €50 = €110 – the 3 records active after 2024-01-04

Temporal join example

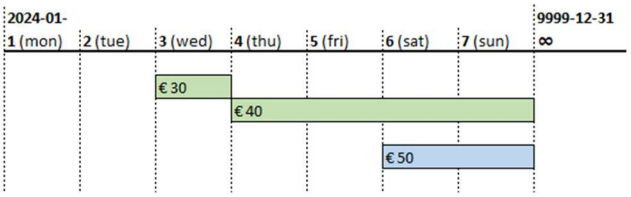
Temporal join operation combines the timelines from both tables resulting in one common timeline where the changes of the both joined tables are represented in the output. Let's take a table with invoices as our LEFT input table. It contains one invoice that changes its status over time. The key of this table is invoice_id.

	invoice_id	net_amount	status		active_from	active_to
	PK					
1	201	€ 90	DRAFT		2024-01-01	2024-01-02
2	201	€ 90	SENT		2024-01-02	2024-01-06
3	201	€ 90	PAID		2024-01-06	9999-12-31



As the RIGHT input table let's take the payments:

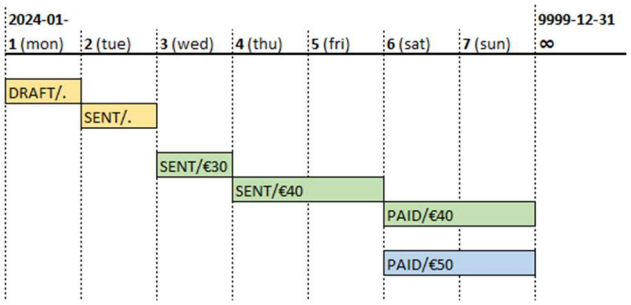
	payment_id	invoice_id	payment		active_from	active_to
	PK	FK				
1	112	201	€ 30		2024-01-03	2024-01-04
2	112	201	€ 40		2024-01-04	9999-12-31
3	113	201	€ 50		2024-01-06	9999-12-31



row	explanation
1	the first payment that was added on Wednesday and which value was later corrected on Thursday
2	the corrected record of the first payment
3	the second payment that was inserted on Saturday

The key of the LEFT table is invoice_id, the join condition is: LEFT.invoice_id = RIGHT.invoice_id.
The output is as follows:

	invoice_id	payment_id	payment	inv_status	active_from	active_to
	PK	PK				
1	201	.	.	DRAFT	2024-01-01	2024-01-02
2	201	.	.	SENT	2024-01-02	2024-01-03
3	201	112	€ 30	SENT	2024-01-03	2024-01-04
4	201	112	€ 40	SENT	2024-01-04	2024-01-06
5	201	112	€ 40	PAID	2024-01-06	9999-12-31
6	201	113	€ 50	PAID	2024-01-06	9999-12-31



row	explanation
1	in the first period the status of the invoice is DRAFT and there is no payment yet, therefore the payment columns are empty
2	the change of the status is represented in the output; still there are no payments, so the payment columns stay empty
3	the status remains SENT, but there is now a payment nr 112 of €30
4	the status is still SENT, but the value of the payment changes to €40 because of the correction; until now we still have one record per period
5	the value of the payment 112 remains €40, but now the status changes to PAID, so the previous record is closed and the new record is added
6	within the same period another payment (113) is registered; the status of the invoice in this period is PAID

Note that there are now 2 active records within the same period. The timeline key of the output is therefore composite and consists of 2 columns: **invoice_id** and **payment_id**.

These and other temporal operations can be achieved in multiple ways, but regardless of the means used, they are complex and performance-heavy. In order not to butcher the CPUs of our data warehouse system, it becomes essential to load the data incrementally.

Incremental load

Incremental load is used to limit the amount of data that needs to be processed daily. As the data that was known yesterday should be already processed by the data warehouse run of yesterday, we should – in theory – process only the data that has been altered today (since the last run).

In practice this is not possible as some data that has changed today needs to be combined with the data that has not been altered, so we cannot just take only new data and process it, as this would lead to incorrect results.

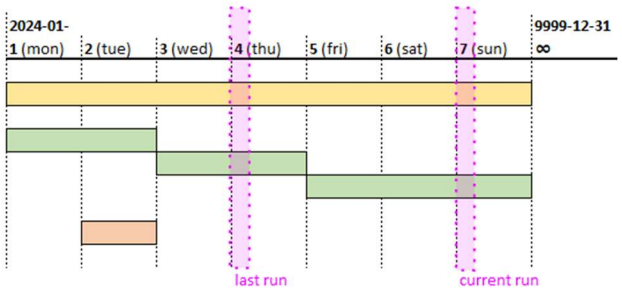
In the temporal data warehouse, we have developed two approaches to incremental load which can be applied separately or together. We call these the two dimensions of incremental load: The time dimension and the key dimension.

Incremental load – the time dimension

Full history of all changes of all tables is stored in the temporal data warehouse. The changes that happened before the last run of the ETL process were already processed, so there is no need to process them again. In order to do that, while selecting data from STAGING, we cut the timeline of processed entities so that it begins at the moment of the previous successful data warehouse run. This is effectively a selection that filters out all the records that stopped being active before that moment. In addition, for all records that were active at the time of the last successful run we adjust the active_from date so it starts at the beginning of the last successful run.

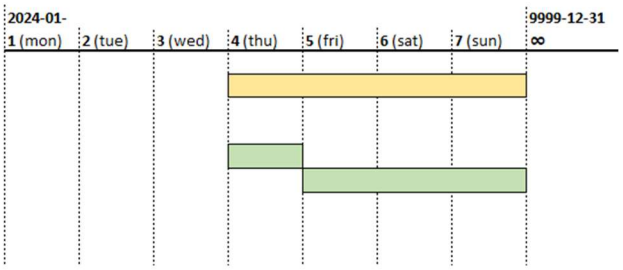
Let's say that this is our staging table (we're using days to make the example more readable, but in practice the data is refreshed daily or even more frequently than that):

	payment_id	invoice_id	value	date	active_from	active_to
	PK	FK				
1	101	201	€ 100	2023-12-30	2024-01-01	9999-12-31
2	102	202	€ 50	2023-12-30	2024-01-01	2024-01-03
3	102	202	€ 60	2023-12-30	2024-01-03	2024-01-05
4	102	212	€ 60	2023-12-30	2024-01-05	9999-12-31
5	103	203	€ 20	3024-12-01	2024-01-02	2024-01-03



After selecting the data from this table, we will have just 3 records to process:

	payment_id	invoice_id	value	date	active_from	active_to
	PK	FK				
1	101	201	€ 100	2023-12-30	2024-01-04	9999-12-31
2						
3	102	202	€ 60	2023-12-30	2024-01-04	2024-01-05
4	102	212	€ 60	2023-12-30	2024-01-05	9999-12-31
5						



row	explanation
1	in this row the active_from date has been altered – the timeline has been cut; the record will be processed further
2	this row is not selected for processing because it lies fully before the last successful run, therefore all the data that it carries should already have been processed
3	in this record the active_from date is altered (just like in record 1)
4	this one is a new row that appeared after the last successful run, so it is selected without any alterations
5	this row lies fully before the last successful run (apparently has been deleted from the source database), so it is not selected (just like record 2)

The data selected this way will be processed using timeline operations and eventually inserted in the output tables. The way it is inserted is in principle the same as the one described above for loading a STAGING table (see: *Moving all the changes from CDC to STAGING*, page 6).

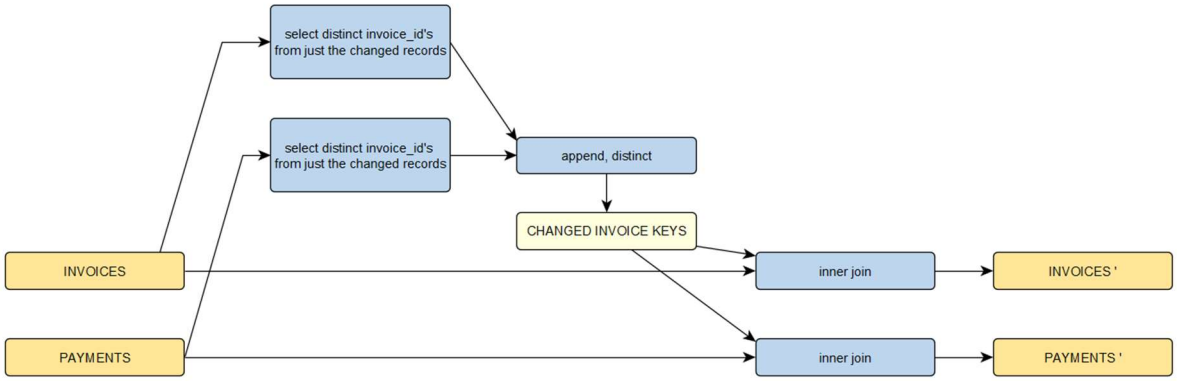
The time dimension limits the number of records slightly, but we still select all the rows that are currently active in the tables of the source system – that usually means all the entities. It can happen that a record which in the source is active, but has not changed since the last data warehouse run. In such case the data in our serving layer table is still correct and does not require an update.

Incremental load – the key dimension

The key dimension of the incremental load is a mechanism that selects only the keys that have been changed since the last successful data warehouse run.

There is one problem though – we cannot simply select just the new records from all source tables. When two such tables are joined and one of them has new records for a given key and the second one does not (but has an existing active record), then the join of these two tables filtered on only the changed records will result in missing values for all the columns coming from the second table for this given key. That is incorrect.

In order to be able to join the data from different tables, while building a pipeline we need to select a key that we will use to select on to limit the entities changed last run. Each run we need to select the same subset of keys from all the input tables of the given pipeline.



This will result in two tables INVOICES‘ (prime) and PAYMENTS‘, that contain selection of the INVOICES that have had some changes since the last run. These tables can be then joined, aggregated and eventually the result would be loaded to the serving layer table (say: FCT_INVOICE) using the mechanism described on page 6 (*Moving all the changes from CDC to STAGING*).

We found that combining these two dimensions of incremental load sufficiently limits the strain on the performance which is added by the temporality of the data warehouse.

Conclusion

We proposed a way to efficiently load changes from CDC tables through staging to the serving layer of a data warehouse without any hard restriction in the time granularity. We have illustrated the mechanism of turning the CDC registry to a table of timelines. We have proposed a data warehouse update approach based on transforming temporal tables at any stage of the ETL. We gave a couple of examples of such transformations. Finally, we have proposed the way to implement an incremental load mechanism in such a data warehouse. This paper is a description of a generic concept. In future publications we plan to dive into details of different mechanisms that make up our proposed data warehouse as well as to explore different implementation possibilities.

Resources

Code repository	Code repository with some SAS Data Integration Studio transformations for the implementation of a temporal data warehouse in SAS: **https://github.com/dadrico/public**

References

[Gao, 2005]	Dengfeng Gao, Christian S. Jensen, Richard T. Snodgrass, Michael D. Soo - Join operations in temporal databases
[Kvet, 2013]	Michal Kvet, Anton Lieskovský, Karol Matiaško - Temporal data modelling
[Mahmood, 2010]	Nadeem Mahmood, Aqil Burney, Kamran Ahsan - A Logical Temporal Relational Data Model
[Malinowski, 2006]	Elzbieta Malinowski, Esteban Zimanyi - A conceptual model for temporal data warehouses and its transformation to the ER and the object-relational models
[Rahman, 2008]	Nayem Rahman - Updating Data Warehouses with Temporal Data
[Rahman, 2010]	Nayem Rahman - Incremental Load in a Data Warehousing Environment