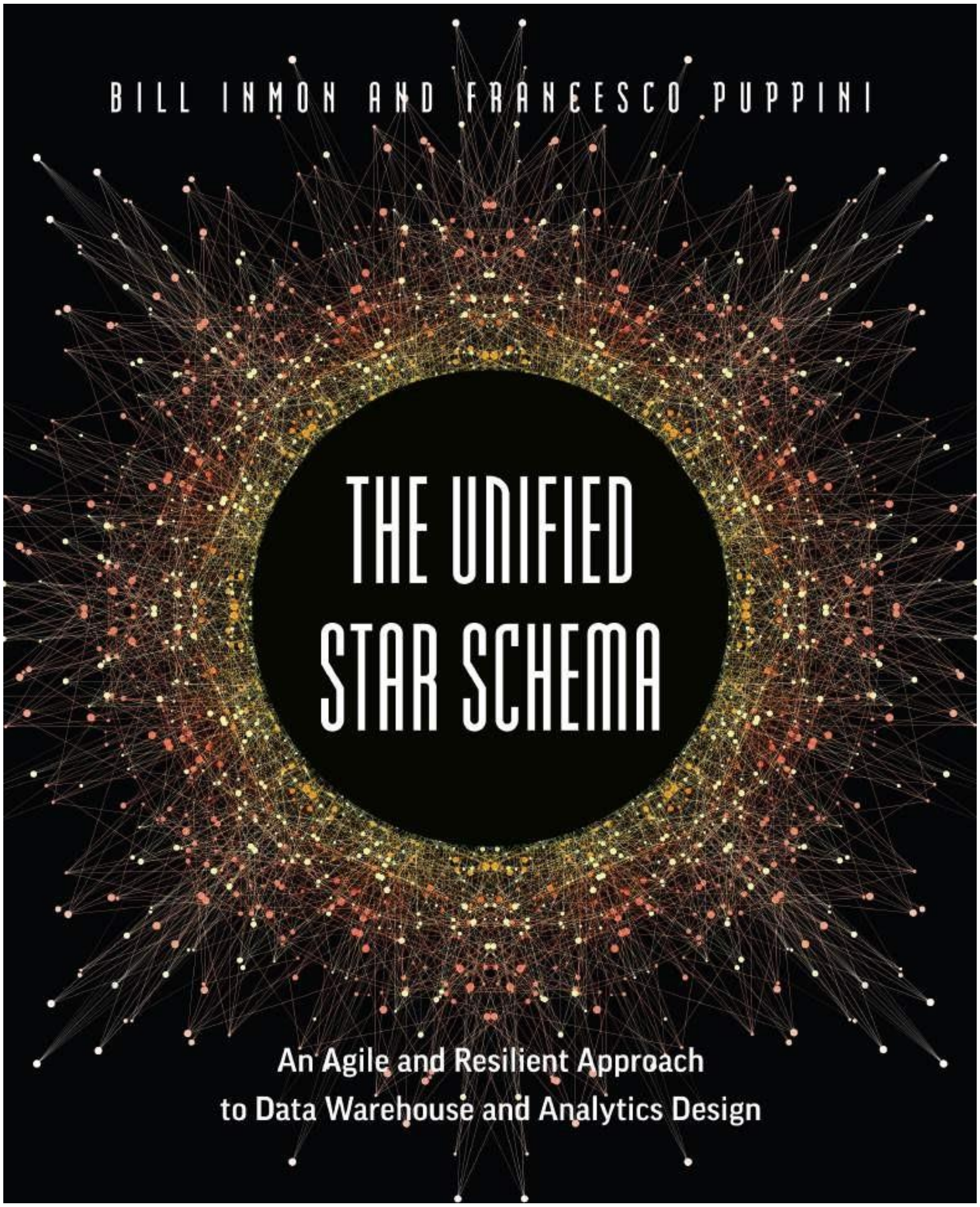


BILL INMON AND FRANCESCO PUPPINI

# THE UNIFIED STAR SCHEMA

An Agile and Resilient Approach to Data Warehouse and Analytics Design

# The Unified Star Schema

## An Agile and Resilient Approach to Data Warehouse and Analytics Design

Bill Inmon

Francesco Puppini

Technics Publications

Published by:

TECHNICS PUBLICATIONS
TECHNOLOGY / LEADERSHIP

2 Lindsley Road
Basking Ridge, NJ 07920 USA
https://www.TechnicsPub.com

Edited by Sadie Hoberman
Cover design by Lorena Molinari

First Printing 2020

Copyright © 2020 by Bill Inmon and Francesco Puppini

| | |
|---|---|
| ISBN, print ed. | 9781634628877 |
| ISBN, Kindle ed. | 9781634628884 |
| ISBN, ePub ed. | 9781634628891 |
| ISBN, PDF ed. | 9781634628907 |

Library of Congress Control Number: 2020946176

CHAPTER 9

# Introduction to the Unified StAr SchemA

In Part I, we reviewed the history of the data warehouse from the early days until now. Across the chapters, we have learned that integrated data marts are the most recommended solution for an enterprise. However, we have also learned that enterprises today tend to accumulate too many data marts, which leads to confusion. This problem needs to be solved.

The Unified Star Schema (USS) solves this problem.

We introduce the USS in this chapter. Learn about its architecture and use cases, as well as how the USS approach differs from the traditional approach. Analogies will reinforce key USS concepts, such as hunter and prey and houses connected with telephone lines. Also, this chapter will show you the dangers of denormalization.

The Unified Star Schema is a star schema centered on a table called the “Bridge”. See Figure 9.1.

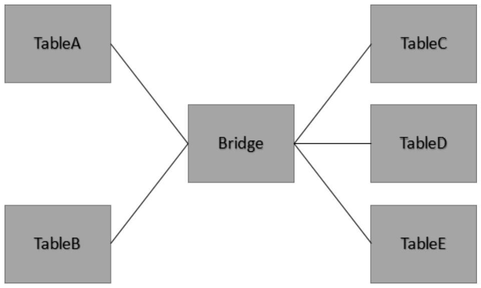


**Figure 9.1: The Unified Star Schema centers on the “Bridge”**

The Bridge handles the connections among all the tables, such as Sales, Products, Clients, Shipments, Invoices, Purchases, Suppliers, Targets, and Stock. If you create a normal join involving all of these tables, you will end up having duplicates. Instead, with the Bridge, you will have no duplicates at all. This book will show you how this is possible.

Please note that, with traditional dimensional modeling, every star schema (or snowflake schema) centers on a single fact table at a time. For example, if you have six fact tables, you will need to build at least six star schemas. In some cases, the same fact tables may need to be used in multiple ways, at different levels of granularity. For this reason, a project of business intelligence normally has a very large number of star schemas (or snowflake schemas). With this approach, instead, there is one single star schema. Even if you have many facts and dimensions at different levels of granularity and a complex set of business requirements, you will always have one single star schema that serves as a foundation for every possible business requirement.

Although, during the next few chapters, we will frequently use the terms "fact" and "dimension", the methodology to build the Unified Star Schema does not need to make this distinction. The only relevant distinction is whether or not a table contains "measures".

Measures in literature have been defined in many different ways, and we like to give our own definition. Imagine that a business intelligence report contains a bar chart: this will have an "X" axis and a "Y" axis. Measures are those numeric columns that can be displayed in the "Y" axis of a bar chart, and they are usually aggregated by sum or by average. On the "X" axis, instead, you will have the dimensions: these can be columns containing textual descriptions, dates, or numbers that should not be aggregated. In the next chapters, you will find many examples of measures.

According to traditional dimensional modeling, measures belong to the fact tables. But in reality, measures can be anywhere. In the chapter of Northwind, for example, we will see that the table “Products” contains information about the units in stock and the units in order. Therefore, we can say that Products contains measures, but it is certainly not a fact table. The Unified Star Schema goes beyond the assumption that measures belong to fact tables, and it treats all the tables the same way.

The ultimate goal of the USS approach is to reduce the data transformations drastically. With traditional dimensional modeling, each data mart is created “ad-hoc” for a particular set of business requirements. The USS approach is different.

*With the USS approach, one single data mart serves as a foundation for every possible business requirement.*

When the business requirements are very complex, you may need to build additional transformations and views on top of the foundation. In all the other cases, there will be no need for transformations at all: the Unified Star Schema will be your ready-to-use data source.

## The Architecture

The Unified Star Schema is a data mart, and it is positioned in the presentation layer of a data warehouse. See Figure 9.2. The traditional data marts typically consist of multiple star schemas or snowflake schemas. The USS, instead, consists of one single star schema.

The design process of the Unified Star Schema is extremely simple.

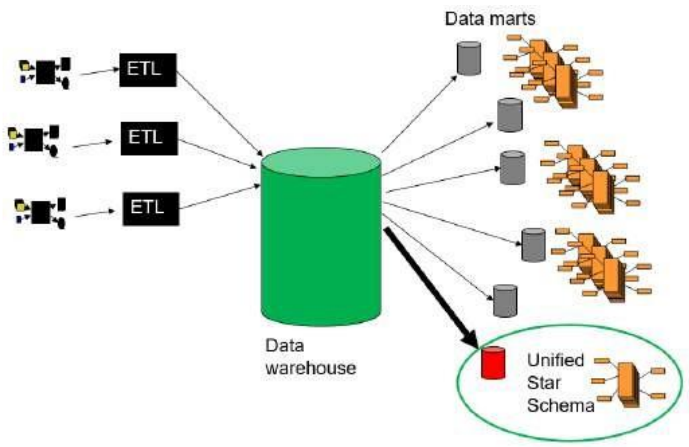


**Figure 9.2: The USS is a new data mart to be added to the existing ones**

*For the creation of the USS, we do not need to go through the classic exercise of conceptual, logical, and physical data modeling. The Unified Star Schema is built entirely based on data, and it does not depend on the business requirements.*

The business requirements play an important role in the design of a data warehouse, because they define first of all the "scope" of a project. An organization may have thousands of tables, but only a few of them need to be loaded into the data warehouse. The "core layer" of the data warehouse contains all the data that is "in scope". This data is called "raw data" because it is not yet prepared for the end-users. Only later, in the presentation layer, the data marts will implement the detail of what the end-users want to see in the final reports and dashboards.

The Unified Star Schema introduces a change to this paradigm: the presentation layer of the data warehouse is also "unbiased", like the core layer. It is prepared to be consumed by the end-users, but it is built independently from the business requirements. The business requirements are pushed outside of the data warehouse: they will be implemented in the BI tools because that's the place where they belong.

*The business requirements belong to the BI tools.*

The BI tools are the right place to implement the business requirements, and we say this for a few good reasons. First of all, BI tools are very powerful: their libraries of functions are usually much richer and smarter than the SQL language. Second, BI tools are much easier to use: most have a graphical interface, and they have been designed to be used by an audience of end-users with no data expertise. Last but not least, Key Performance Indicators (KPIs) often contain ratios of numbers that typically need to be implemented in BI tools, because in the data marts they would not be "additive". This deserves an explanation.

Imagine that you need to implement a KPI that calculates the ratio of sales versus target, in percentage. If you calculate this ratio row by row in a data mart, you will not be able to perform aggregations because the sum of ratios does not make any sense. Conversely, if you do it inside a BI tool, all the numbers will be aggregated first, and then the ratio will happen later. The calculation of a KPI inside a BI tool executes the steps in the correct order.

Pushing the business requirements to the BI tool sounds like "postponing the problem", but it is not. In today's reality, business requirements are implemented both in the data mart and in the BI tool. They are happening "a bit here, and a bit there". This is not a good way to implement a solution. Shifting the whole business logic to the BI tool makes the maintenance much easier, because it needs to be done in one place instead of two.

It has to be said, however, that every project has its own peculiarities, and the ultimate decision needs to be made by a data architect, based on a set of specific needs and challenges. In some cases, a part of the business logic may need to be implemented in the data warehouse for performance reasons, or for reasons of “re-usability”. Our recommendation is to push the business requirements to the BI tool as much as possible, unless you have a good reason to do otherwise.

It is not easy to build a data mart that serves as a foundation for every possible business requirement. The next chapters will explain how the Unified Star Schema is able to achieve this result.

## The USS approach

The Unified Star Schema can be implemented whenever the data sources are in a tabular format, such as in database tables, Excel files, and CSV files.

But the USS can also be built in the cases when data is not organized in tables—as long as it’s possible to convert this data into tables. This is the case of XML files, JSON files, Avro format, Parquet data storage, and any other data format that can be converted into a tabular format. This makes the Unified Star Schema easy to integrate also with the cloud technologies and with the APIs.

The guiding principle of the USS approach is that two tables are never connected directly to each other—they are always connected through the Bridge.

*With the USS approach, all the tables are always connected through the Bridge.*

Figure 9.3 shows the main difference between the traditional approach and the USS approach.

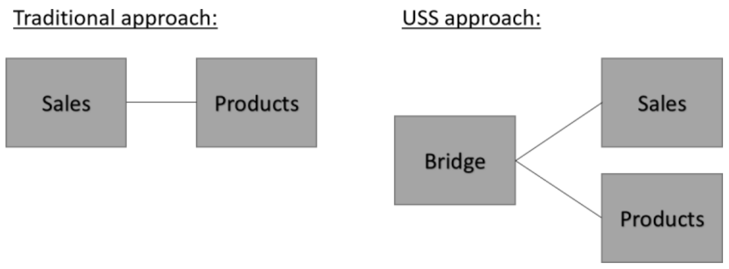
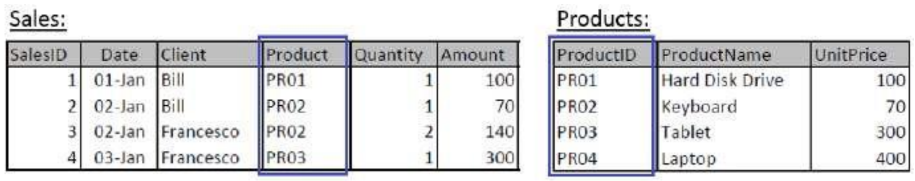


**Figure 9.3: Sales and Products with the USS approach are connected through the Bridge**

It is worth clarifying this concept with an example. Please note that all the examples in the next chapters will have very few rows. There is no need to work on millions of rows: ten or fifteen rows are more than enough to illustrate the concepts that we want to share. Figure 9.4 shows Sales and Products.



Sales:

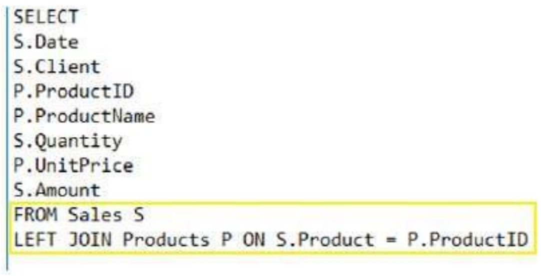
| SalesID | Date | Client | Product | Quantity | Amount |
|---|---|---|---|---|---|
| 1 | 01-Jan | Bill | PR01 | 1 | 100 |
| 2 | 02-Jan | Bill | PR02 | 1 | 70 |
| 3 | 02-Jan | Francesco | PR02 | 2 | 140 |
| 4 | 03-Jan | Francesco | PR03 | 1 | 300 |

Products:

| ProductID | ProductName | UnitPrice |
|---|---|---|
| PR01 | Hard Disk Drive | 100 |
| PR02 | Keyboard | 70 |
| PR03 | Tablet | 300 |
| PR04 | Laptop | 400 |

**Figure 9.4: Sales and Products**

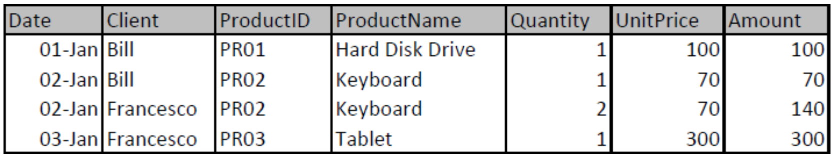
These two tables are connected through the product ID, which is called “Product” in Sales and “ProductID” in Products. Figure 9.5 contains an example of an SQL query, based on the traditional approach.



```
SELECT
S.Date
S.Client
P.ProductID
P.ProductName
S.Quantity
P.UnitPrice
S.Amount
FROM Sales S
LEFT JOIN Products P ON S.Product = P.ProductID
```

**Figure 9.5: An SQL query based on the traditional approach**

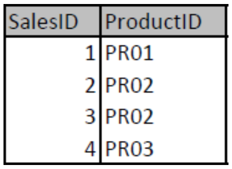
When we create an SQL query, we typically read from multiple tables. With the traditional approach, we need to choose what table to mention first (in this case, the query starts with Sales), and then we need to add all the other tables, one by one. In a traditional SQL query, the order that we choose for the tables is particularly relevant: starting with a different table or swapping the order of two tables may produce different results. Figure 9.6 contains the results of this query.



| Date | Client | ProductID | ProductName | Quantity | UnitPrice | Amount |
|---|---|---|---|---|---|---|
| 01-Jan | Bill | PR01 | Hard Disk Drive | 1 | 100 | 100 |
| 02-Jan | Bill | PR02 | Keyboard | 1 | 70 | 70 |
| 02-Jan | Francesco | PR02 | Keyboard | 2 | 70 | 140 |
| 03-Jan | Francesco | PR03 | Tablet | 1 | 300 | 300 |

**Figure 9.6: Result of the query, based on the traditional approach**

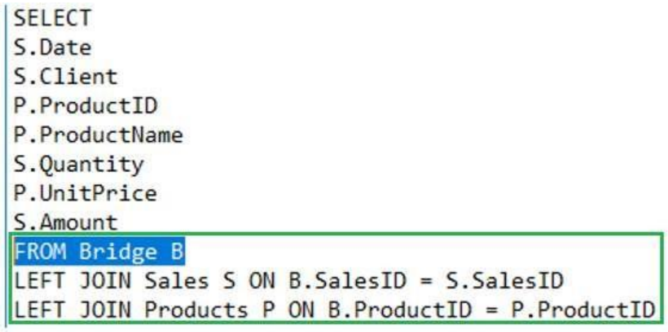
Now let's introduce a simple example of the Bridge table, as shown in Figure 9.7. The Bridge is a table that only contains IDs. This is a very simplified version. In the next chapters, we will give a proper definition of the Bridge and a set of rules and conventions to build it.



| SalesID | ProductID |
|---|---|
| 1 | PR01 |
| 2 | PR02 |
| 3 | PR02 |
| 4 | PR03 |

**Figure 9.7: The Bridge in its simplest form**

Figure 9.8 contains an example of an SQL query, based on the USS approach.



```
SELECT
S.Date
S.Client
P.ProductID
P.ProductName
S.Quantity
P.UnitPrice
S.Amount
FROM Bridge B
LEFT JOIN Sales S ON B.SalesID = S.SalesID
LEFT JOIN Products P ON B.ProductID = P.ProductID
```

**Figure 9.8: An SQL query based on the USS approach. The Bridge must always be mentioned first**

An SQL query based on the USS approach must always start with the Bridge, and then all of the other tables must be joined with a "left join". With the USS approach, the order that we choose for the other tables in the query will not affect the results.

The result of the query in Figure 9.8 will look identical to the results from the previous query, shown earlier in Figure 9.6. The advantages of using this approach will be explained throughout the next chapters.

The USS approach requires every table to have a "unique identifier": a column that uniquely defines each row of that table. When the unique identifier consists of multiple columns, it is recommended to concatenate (or hash) them into one single column. This unique identifier will be used as a connection between each table and the Bridge. If such an identifier is not available, it is always possible to create a "surrogate key": a "system-generated unique identifier" with no business meaning.

Throughout the book, we will use the term "Primary Key" (PK) to indicate the unique identifier for a table. A table, in general, may have multiple unique identifiers, but only one must be chosen as the "primary".

The term "Primary Key" may generate a bit of confusion because, in the world of relational databases, it may have a specific meaning of "enforced Primary Key": a mechanism that produces an error when the uniqueness of the key is violated. But this is not what we mean: by PK we simply mean the unique identifier for a table. Other authors call it a "Business Key" or "Natural Key".

Likewise, throughout the book, we will use the term "Foreign Key" (FK) to indicate a column referencing the PK of another table. This term may also generate a bit of confusion because, in the world of relational databases, the term "Foreign Key" may have a specific meaning of "enforced Foreign Key": a mechanism that produces an error when a particular value is unreferenced. But this is not what we mean: by FK we simply mean a column that references a PK.

In our example based on Sales and Products, shown in Figure 9.4, the column "Product ID" is a PK for Products. The column "Product" in the Sales table is a FK that references the Products table.

Now that we know what we mean by PK and FK, we can give a preliminary definition of the Bridge.

*The Bridge is a table that absorbs all the PKs and FKs from all the tables.*

## Hunter And prey

The tables of a database need to be connected to each other.

It is important to understand that this connection is always "oriented": one table points to the other, and not vice versa. For this reason, it is represented as an arrow, as seen in Figure 9.9.

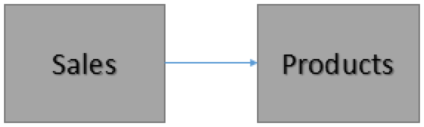


**Figure 9.9: Sales points to Products, and not vice versa**

*The connection between two tables is always oriented.*

The concept of "oriented connection" is the foundation of the USS approach. Starting from this concept, it will be possible to introduce our own concept of an "Oriented Data Model" (ODM), which is a graphical convention for drawing data models. Based on the ODM convention, we will be able to give some straightforward and simplified definitions of “fan trap” and “chasm trap”, which are two scenarios that occur very often in business intelligence. These traps generate duplicates, and duplicates are a problem. Based on our simplified definitions, we will be able to detect these problems and prevent them.

*Solving problems is great. Preventing them is even better.*

To better explain the concept of oriented connection, let’s make an analogy with the world of wild animals, as in Figure 9.10. Let’s compare the tables of a database with hunter and prey, like the lion and the gazelle. The Sales table contains a Foreign Key (FK) that references the Primary Key (PK) of Products. By this key, Sales “grabs” some information from Products. So, we can say that Sales is the hunter, and Products is the prey. Sales is the lion, and Products is the gazelle. In the spoken language, we often say that “Sales points to Products”.

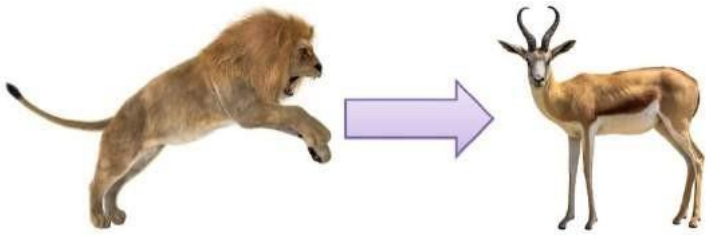


**Figure 9.10: Sales points to Products the same way as the lion points to the gazelle**

The analogy with hunter and prey helps to fix in mind that a connection between two tables is always oriented: one table points to the other, and not vice versa. The gazelle will never hunt the lion. Likewise, Products will never have a FK pointing to Sales.

This first example is easy because it deals with only two subjects, and it is very easy to determine which one is the hunter and which one is the prey. But the real world is much more complex than this. Sometimes more hunters may aim at the same prey. The lion hunts the gazelle, but the leopard also hunts the gazelle. And the lion may hunt the leopard. See Figure 9.11.

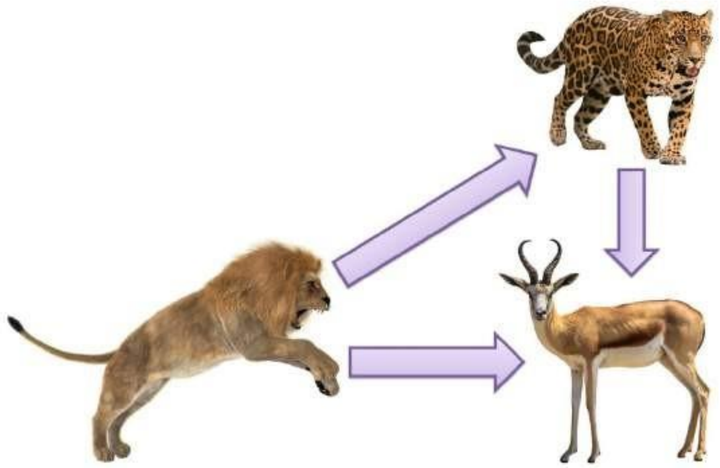


**Figure 9.11: The food chain is actually more complex than a simple chain**

*The food chain, despite its name, does not look like a chain at all: it is actually a complex tangle of relations.*

The tables of a database are even more complex than the food chain of wild animals. Traditional dimensional modeling is based on a simplified assumption: facts point to dimensions. Or, in the case of the snowflake schema, dimensions point to other dimensions, within the same hierarchy. But these two scenarios only represent a partial picture of reality. Sometimes a fact points to another fact. Sometimes a dimension points to a dimension that is not within the same hierarchy. Sometimes multiple facts point to the same dimension but on a different level of granularity. Sometimes it is not easy to determine whether a table is a fact or a dimension. Sometimes tables form a closed circle, also known as a “circular reference” or “loop”. Sometimes we need information from multiple tables, but we cannot find an easy way to connect them together.

Sometimes two tables point to each other. Sometimes a table points to itself. And the list goes on.

We need a data structure that is able to handle all these scenarios in an easy way. And that's where the Unified Star Schema comes into play.

## Loops

Loops are another problem that occurs very often in business intelligence. For this reason, they deserve our attention in this chapter.

Let's start from the previous example based on Sales and Products, and add a new table, "Shipments". Shipments is a fact table.

In our daily life, when we make a large purchase, it may happen that the vendor does not have enough units in stock. Let's imagine that a client wants to buy 40 hard disks, and the shop only has ten of them in stock. A first shipment should be made immediately with the ten available hard disks, and then a second shipment should be made as soon as possible with the remaining 30.

In the underlying database, the Shipments table contains a FK that points to the PK of Sales. In this case, we can say that Shipments is the hunter, and Sales is the prey. This is an example of a fact pointing to another fact. See Figure 9.12 for how we represent the oriented connection between Sales and Shipments.

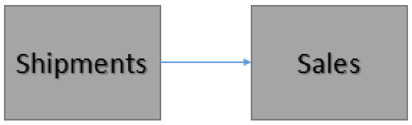
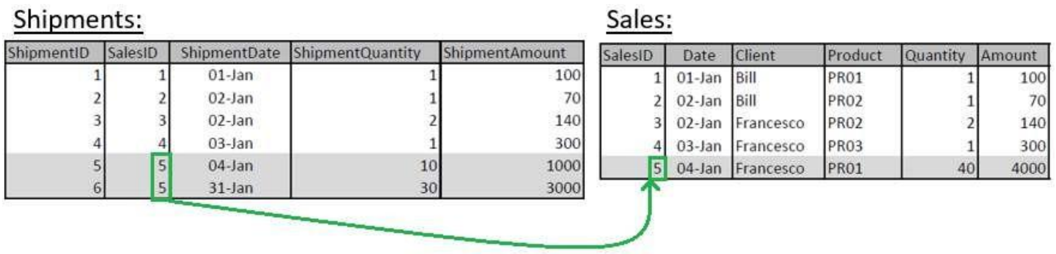


**Figure 9.12: One fact (Shipments) pointing to another fact (Sales)**

Thanks to this connection, we are able to identify, for each row of Shipments, the "one and only" row of Sales that it references. This may be useful, for example, if we want to retrieve, for each shipment, the information of the sales date. Based on this, we can calculate the "fulfilment latency", row by row. This allows us to build a KPI that shows the average latency per country, per product category, per month, and so on.

This is a "one-to-many" relationship. One row of Sales is potentially associated with multiple rows of Shipments, but each row of Shipments is associated with "one and only one" row of Sales.

Figure 9.13 shows that two rows of Shipments reference the same row of Sales.



Shipments:

| ShipmentID | SalesID | ShipmentDate | ShipmentQuantity | ShipmentAmount |
|---|---|---|---|---|
| 1 | 1 | 01-Jan | 1 | 100 |
| 2 | 2 | 02-Jan | 1 | 70 |
| 3 | 3 | 02-Jan | 2 | 140 |
| 4 | 4 | 03-Jan | 1 | 300 |
| 5 | 5 | 04-Jan | 10 | 1000 |
| 6 | 5 | 31-Jan | 30 | 3000 |

Sales:

| SalesID | Date | Client | Product | Quantity | Amount |
|---|---|---|---|---|---|
| 1 | 01-Jan | Bill | PR01 | 1 | 100 |
| 2 | 02-Jan | Bill | PR02 | 1 | 70 |
| 3 | 02-Jan | Francesco | PR02 | 2 | 140 |
| 4 | 03-Jan | Francesco | PR03 | 1 | 300 |
| 5 | 04-Jan | Francesco | PR01 | 40 | 4000 |

**Figure 9.13: There are two rows of Shipments referencing one row of Sales**

But Sales, in turn, points to Products. This forms a chain. Figure 9.14 shows that Shipments, Sales, and Products are connected as a chain.

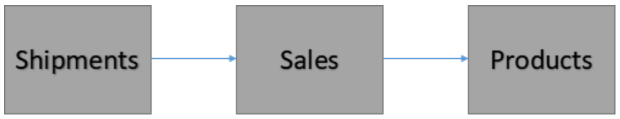


**Figure 9.14: Shipments, Sales, and Products form a chain**

In the above schema, we can see that Sales is in the middle. Sales is, at the same time, hunter and prey.

But what happens now if Shipments contains a Product ID that points to Products directly? By adding one more connection, we produce a "loop", as shown in Figure 9.15.

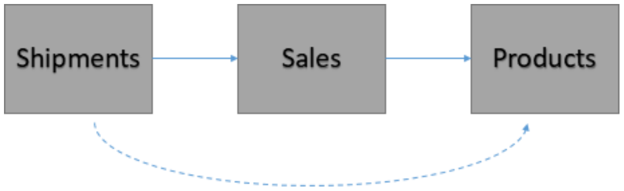


**Figure 9.15: If we add one more connection between Shipments and Products, we produce a "loop"**

A loop is a topology of entities where we have multiple possible paths to go from one entity to another. See a loop in Figure 9.16. To go from A to C, you can go either directly, or via B. This is a loop.

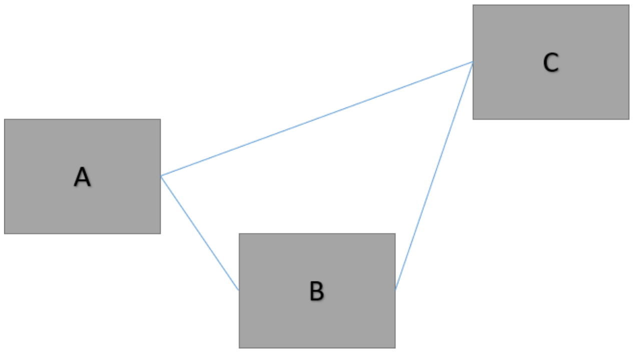


**Figure 9.16: Schema of a loop**

In the particular loop that we have seen earlier in Figure 9.15, the problem could be solved by "disregarding" the direct connection from Shipments to Products, assuming that the same information is available also in Sales. This assumption is supposed to be true when everything in the process goes fine, but this is not always the case. The direct connection from Shipments to Products is a piece of information that may be helpful at some point, for example, for troubleshooting. In general, disregarding Foreign Keys should be avoided.

Loops are a big problem in SQL because they introduce "ambiguity", and this ambiguity produces an error in the execution of the SQL query. Loops also produce an error in business intelligence tools.

To better explain loops, let's make another analogy—this time with a village of houses connected with telephone lines. See Figure 9.17.

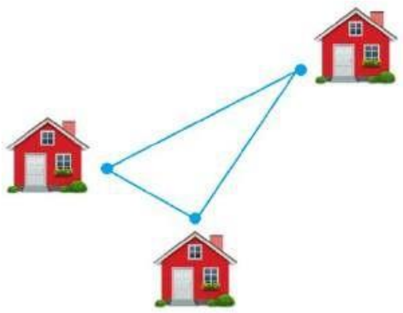


**Figure 9.17: Tables and connections can be compared with houses and telephone lines**

We can compare the tables of a database with houses, and we can see the connection between two tables like the telephone line that connects two houses.

If we have three houses and we want to avoid a loop, we can build at most two lines. The third line will inevitably create a loop. See Figure 9.18.

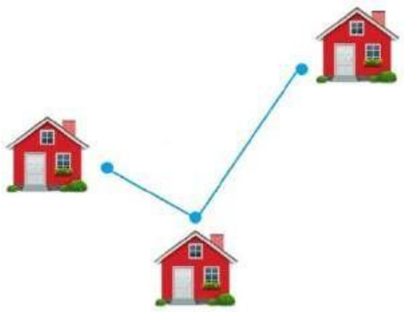


**Figure 9.18: With three houses we can build at most two lines. The third line will inevitably create a loop**

If we have five houses and we want to avoid loops, we can build at most four lines. The fifth line will inevitably create a loop. You can easily verify this with pen and paper: you will never manage to build more than four lines without creating a loop. See Figure 9.19.

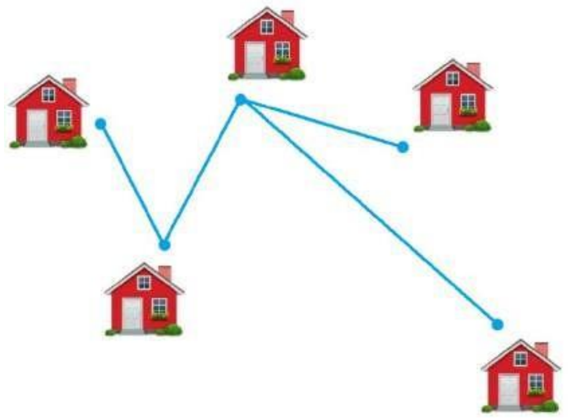


**Figure 9.19: With five houses we can build at most four lines. The fifth line will inevitably create a loop**

We can now generalize:

*If we have (n) houses and we want to avoid loops, we can build at most (n-1) lines. Likewise, if we have (n) tables and we want to avoid loops, we can build at most (n-1) connections.*

In a typical database, when we have (n) tables, we usually have more than (n-1) connections. This means that, if we want to avoid loops, we need to "disregard" some of the connections. In other words, we have to disregard some of the FK columns. But disregarding columns means losing information, and we don't want that.

There is one more question that we should ask ourselves at this point: "If we were OK with creating loops, how many lines could we possibly build?" In the example of five houses, we could build ten lines, as you can see in Figure 9.20. Where does this number 10 come from? It comes from the formula [n * (n-1) / 2]. We have five houses (n = 5): the possible connections are [5 * 4 / 2] = 10. If we had 100 houses (n = 100), then the possible connections would be [100 * 99 / 2] = 4950.

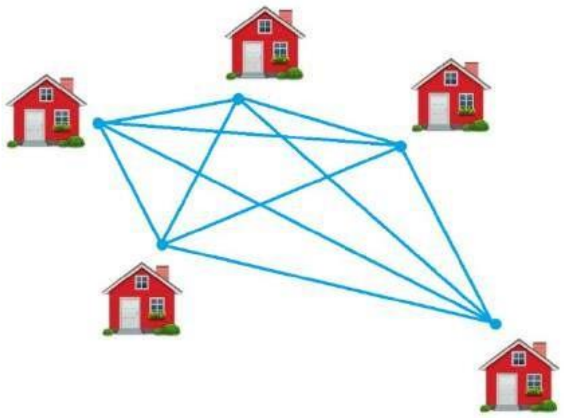


**Figure 9.20: With five houses, the possible connections are ten**

These lines are clearly creating multiple loops, but they are all needed. How can we solve this problem?

## The centrAL TABle

If we want to have all the houses connected and we want to avoid loops, we can add one central switchboard, as shown in Figure 9.21.

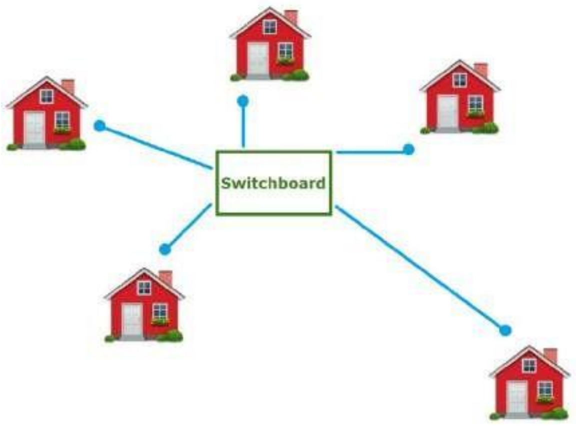


**Figure 9.21: A central switchboard handles all the connections**

A switchboard is a device that handles all the possible connections. Figure 9.22 shows an old switchboard.

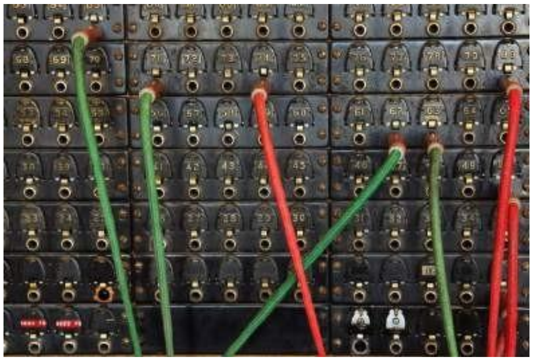


**Figure 9.22: An old switchboard**

The Bridge that we use in the Unified Star Schema is very similar to a telephone switchboard.

If we have a database with 100 tables, they will all connect to the Bridge. The drawing of the data model will show 100 connections. But in reality, the Bridge will be able to handle all the possible connections, which are 4950. See Figure 9.23.

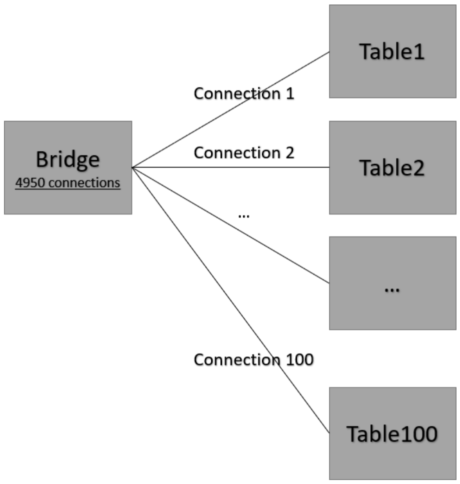


**Figure 9.23: The Bridge handles many more connections than the drawing shows**

In reality, with 100 tables, the Bridge can handle even more than 4950 connections, using the technique of "multiple stages per table". But this is an advanced topic that goes beyond the scope of this book.

Earlier in this chapter, in Figure 9.15, we saw a loop based on Shipments, Sales, and Products. Instead of discarding one of the three connections, thanks to the Bridge, we can keep them all. Figure 9.24 shows the USS approach applied to that example.

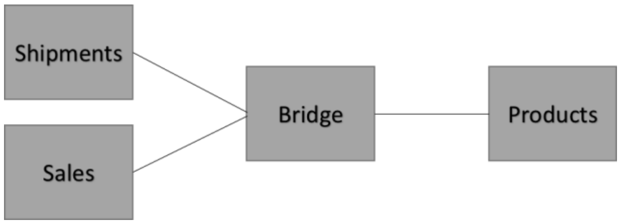


**Figure 9.24: The Bridge solves the loop**

Although the model in Figure 9.24 appears simplistic, it is actually magical! The three lines that you see in this figure are just the connections of the tables to the Bridge. We can call them the "trivial connections". But the real information about "who points to what" is embedded "inside" the Bridge.

*All the connections are now handled inside the "magic box".*

The next chapters will clarify in detail how this works.

## The DANgers of denormALizATion

Before proceeding with the next chapters, let's go through some thoughts about denormalization, also known as the process of "flattening" tables.

Data is usually created in the applications. The basic structuring of data in the applications is the relational model. This has been considered for decades the best way to handle operations, and today it is still the most common and recommended industry standard.

The data warehouse is also structured as a relational model. However, to make the analysis easier, the standard structuring for the data marts is the dimensional model. This allows data to be accessed and analyzed easily because the data relationships in the data mart have been minimized. The dimensional model is organized in a way that the developer does not need to create complex joins to access the data. In some cases, the data can be denormalized into one single table (or view), which means that the developer does not need to create any join at all.

But then, let's go a bit beyond now. If denormalization is so good, why don't we build a fully denormalized table that joins all the tables together? Why don't we build a "big table with everything inside"?

The most common answers to this question would mention "size", "performance", and "security". Someone else would also say that too many columns in the same table would confuse the end-users.

But these are not the real reasons. The real reason is that a fully denormalized table would create two families of problems:

1. Losses
2. Duplicates

Let's take the example of "Northwind", a well-known sample database from Microsoft. It contains only 13 tables, and it has a total of 3308 rows and 86 columns. We can definitely say that size, performance, and security cannot be a problem with such a small database containing sample data. So, does this mean that we can join all 13 tables together, denormalizing them all into one single table?

No, we cannot do it. If we do it, we will have losses and duplicates.

*Denormalization causes losses and duplicates*

Losing data is not a good thing, and duplicating data is even worse because it can produce incorrect numbers.

By "incorrect numbers", we mean numbers that are different from the ones that we see in the data source.

The next chapters will describe in detail those scenarios that produce losses and duplicates. You will understand why these phenomena happen, and you will learn how to prevent them.