AALBORG UNIVERSITY

SPECIALIZATION IN COMPUTER SCIENCE

DWStar - Automated Star Schema Generation

Authors:
Jacob B. Hansen
Søren Jensen
Michael Tarp

Supervisor:
Christian Thomsen

Group:
dpt104f17

Spring Semester 2017
Specialization in Database Technology
Computer Science 10th Semester

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg
http://www.cs.aau.dk/

Titel:
DWStar - Automated Star Schema Generation

Theme:
Specialization in Computer Science Database Technologies

Project period:
DAT10, Spring Semester 2017

Project group:
dpt104f17

Group members:
Jacob B. Hansen
Søren Jensen
Michael Tarp

Supervisor:
Christian Thomsen

No. of pages (appendix incl.):
98

Other attachments:
1 zip file of the implementation

Finished on date:
02/06-2017

Synopsis:

In this report we explore ways of automatically generating star schemas for data warehousing. The result of this is an implementation called DWStar, which is able to infer the schema of an operational source system, produce a list of star schema candidates, and finally transfer data from the operational source systems into the created star schema(s). DWStar is composed of multiple phases, each having a different responsibility. The computation in these phases is carried out by a series of modules, that determine how to find primary keys, fact tables, etc. We then perform experiments and evaluations to demonstrate that DWStar is able to automate the creation of star schemas and also do this within a reasonable time.

Preface

This master thesis is created by three computer science students at Aalborg University in the Database, Programming and Web technologies (DPW) research unit. The project took place in the spring of 2017. This project was supervised by Christian Thomsen, associate professor at the Department of Computer Science. The project ran from the 1st of February 2017 to the 2nd of June 2017, and was scheduled to 30 ECTS points. This project covers the theory and implementation of a solution, which is able to automatically produce candidate star schemas and generate relevant ETL processes based on operational source systems.

We would like to thank Xuegang "Harry" Huang for initial discussions for the project proposal as well as participating in our evaluation.

Summary

This project examined the possibility of introducing automation of tasks within the Business Intelligence (BI) field. In order to support the decision making of organizations, a Data Warehouse (DW) is usually deployed to help with reporting and data analysis in a timely manner. DWs are typically constructed by BI experts and are then populated with data from Operational Source Systems (OSSs). Building a DW is a very time-consuming process and automation can thus prove very favorable. To remedy this, we propose a solution called DWStar.

We look at methods for automatically constructing DWs and as a result try to relieve BI experts of constructing DWs from scratch. Although organizations might construct their DW using different methods, we establish a customizable set of modules based on heuristics from related work and our own experiences that employ commonly used practices within this field. DWStar also allows for the user to implement his own modules if he is not satisfied with the modules provided. The modules make use of the data (as well as metadata) available in the OSSs and uses this the data to construct star schemas for a DW.

The modules are divided by their area of responsibility, which consists of five different phases: The metadata phase, refinement phase, star phase, star refinement phase, and generation phase. Modules in the metadata phase are responsible for extracting and inferring metadata from OSSs, this is either trivially done by extracting metadata from RDBMSs or non-trivially done by inferring the metadata structure from CSV files. The refinement phase improves upon the result from the metadata phase by, for example, finding relationships between OSSs, and determining additional metadata in CSV files (like primary keys, relationships, nullability, etc.). The star phase attempts to create suitable star schema candidates based on the data from the refinement phase, which primarily involves finding fact tables and dimension tables. The star refinement phase refines the star schema by for example: Applying naming conventions, adding date and time dimensions, adding surrogate keys, and possibly adding a junk dimension. Finally, the generation phase is responsible for generating SQL scripts for the creation of the star schema and the transferral of data from the OSSs to the DW.

One evaluation and two experiments were performed to evaluate DWStar. The evaluation involved a BI practitioner who evaluated the resulting candidate star schemas of DWStar based on the Northwind database and a secondary database. The feedback from the practitioner was primarily positive and DWStar proved to be applicable in both real world cases. The first experiment involved testing of how well DWStar infers the metadata of CSV files, to which we concluded that it did to a satisfactory degree. The second experiment tested the performance of creating and population of a DW, which proved to perform within a reasonable amount of time.

Contents

Chapter 1

Introduction

Business Intelligence (BI) is commonly used to present and analyze an organization's data. The data is stored in a *Data Warehouse* (DW) which is composed of multidimensional schemas. DWs are crafted by experienced designers and are used to answer questions that aid the decision making process, like how many products are sold in each local store every day. In a DW/BI environment [1] there are four components to consider: *Operational Source Systems* (OSSs), an *Extract-Transform-Load* (ETL) system, a data presentation area (the DW), and BI applications. We introduce a solution called *DWStar* which focuses on the first three components, where the choice of a data analysis tool is left to the user.

Designing a DW typically involves: Gathering business requirements for the DW, creating *fact tables* and *dimension tables* for a multidimensional schema, and finally construction ETL processes to cleanse and transfer data from the OSSs to the DW. This can be a very time-consuming process, particularly when managing a large number of OSSs. DWStar attempts to remedy some of this by: Automatically establish a DW by using the OSSs' schemas to find and create one or more appropriate multidimensional schemas (specifically *star schemas*), and establish an ETL process that can transfer data from the OSSs to the DW. The resulting DW may either be used directly, or serve as a starting point where a significant part of the work has been done beforehand.

There exists related work that automatically or semi-automatically assist the designer in constructing multidimensional schemas. However, much of the related work have the same drawbacks such as the designer must use *entity-relationship diagrams* (ER diagrams), logical schemas, or similarly constrained inputs for the solution. This requires the user to conform to the solution and translate his OSS into the required input for the solution each time he uses it. Additionally, the customizability of the solutions are either small or non-existent. This is unfortunate since the user has no way of changing the solutions to match his requirements and must therefore hope that the solution's design happens to correlate with the user's domain. Additionally, the heuristics the related work provide are deemed sound, but there are however no way of combining them since they operate on different inputs. To improve upon these drawbacks, we provide a customizable solution that allows the user to specify the input of the solution as well as the heuristics he wants to apply. If the provided heuristics does not fulfill the user's requirements, the framework also allows users to implement their own heuristics. This allows for a fully automated solution since the user can detail exactly what DWStar should do and can thereby define what results it provides.

Various related works use ER diagrams to gather the required information about the OSSs. However, ER diagrams could either be outdated or nonexistent. Instead, we directly extract metadata information from the OSSs themselves which retrieves a up-to-date version of the tables, columns, etc. This will include semi-structured data sources such as CSV files as well, as this scenario has not been examined in depth by related work. Throughout this report when referring to OSSs this also includes CSV files.

Besides creating a schema for the DW, one also needs to populate the DW for it to be useful. A small amount of related work exists on automating the construction of ETL processes. This is reasonable, as ETL processes can be complex and require knowledge of both the OSSs and the target DW. We propose a simple approach to create ETL processes exclusively consisting of SQL statements which can transfer data from the OSSs to the DW.

1.1 Problem Statement

On the background of the introduction, to build DWStar we thus wish to investigate:

"How can a system be constructed with the following properties:

1. ***Automatic DW construction**: Automatically produce candidate star schemas on the background of OSSs.*
2. ***Customize DW construction**: Allow for customization of the process for producing one or more star schema(s).*
3. ***Metadata inference**: Automatically infer the schemas of OSSs.*
4. ***Automatic DW population**: Aid the user with populating the DW with data from OSSs."*

Chapter 2

Related Work

This chapter delves into related work regarding the points in the problem statement. Section 2.1 examines a framework for constructing star schemas through ER-diagrams. Section 2.2 looks at how snowflake schemas can be made through logical schemas. Section 2.3 presents candidate conceptual schemas on the basis of OSSs with the usage of simple heuristics. Section 2.4 discusses the existing work on automatic multidimensional schema generation and also shows how user involvement can be advantageous when constructing multidimensional schemas. Finally, Section 2.5 summarizes the points to take away from the related work in terms of how to approach DWStar.

A point to note about the *Automatic DW population* point from the problem statement is that to the authors' knowledge, no related work has been done on automatic population of a DW. There only exists related work such as Muñoz et al. [2] that does this partially by generating the executable code for ETL processes, but this generation is based on the specification of several models, which are created manually.

2.1 SAMSTAR: A Semi-Automated Lexical Method for Generating Star Schemas from an Entity-Relationship Diagram

Song et al. [3] present a method called *SAMSTAR* for generating star schemas from ER-diagrams semi-automatically. Beyond the method, they present two novel notions: *Connection Topology Value* (CTV) for identifying fact tables and dimension tables and *Annotated Dimensional Design Patterns* (A_DDP) for finding additional dimension tables.

2.1.1 Connection Topology Value

A CTV describes the likelihood of an entity (table) being a fact table. Any table's CTV above a given threshold is considered as a candidate fact table. However, to properly understand how they calculate CTVs, some background is needed on the types of relationships in an ER-diagram.

Classification of Relationships

Song et al. observe that there usually exists a many-to-one relationship between a fact table and a dimension and define the following heuristic:

- If an entity has a many-to-one relationship with another entity, then the entity at the *many*-side is a candidate fact.
- Likewise, the entity at the *one*-side of a many-to-one relationship is a candidate dimension.

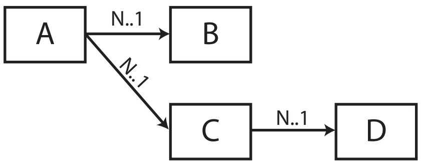


Figure 1: Figure representing three direct relationships and one indirect relationship.

With this in mind, Song et al. define two different types of relationship between entities: A *direct* many-to-one relationship and an *indirect* many-to-one relationship. A direct relationship is when an entity participates in a many-to-one relationship with another entity, whereas an indirect relationship occurs when an entity is transitivity related to another entity through a many-to-one relationship, but there are one or more intermediate entities. An example of this is seen in Figure 1, where table *A* is directly related to table *B* and *C* through a many-to-one relationship. Table *C* is also directly related to table *D*, hence table *A* is indirectly related to table *D* as the intermediate relationship is many-to-one relationship.

Calculation of Connection Topology Value

A CTV takes into account both direct and indirect relationships. Song et al. provide two user-defined values $weight_d$ and $weight_i$, which define weights for direct and indirect relationships, respectively. It is desirable to provide values such that $weight_d > weight_i$, as direct relationships are more important than indirect relationships. $CTV(e)$ for some entity e is found by summing up all of the weighted direct and indirect many-to-one relationships that e has. In the example given above with $weight_d = 1$ and $weight_i = 0.8$, the CTV for each entity is:
$CTV(D) = 0$
$CTV(B) = 0$
$CTV(C) = 1 * 1 + 0.8 * CTV(D) = 1$
$CTV(A) = 1 * 2 + 0.8 * CTV(B) + 0.8 * CTV(C) = 2.8$
If provided with a threshold $Th = 2$, SAMSTAR recognizes A as the only candidate fact table.

2.1.2 Annotated Dimensional Design Patterns

For identifying dimensions, A_DDP is used which is an annotated version of *Dimensional Design Patterns* (DDP) by Jones and Song [4]. The concept behind DDPs is that dimensions often involve some specific behavior, such as managing inventory, user accounts, etc. Jones and Song thus devised six categories of dimensions: Temporal (when), location (where), stakeholders (who), action (what is done), object (what), and qualifier (why). Song et al. then define A_DDPs to have a series of commonly occurring words for given categories. E.g., “Month”, “Date”, and “Time” are words that belong to the temporal category. If any table includes any of these words in its name, then it is a candidate for being a dimension with temporal data in this case.

2.1.3 Specification of SAMSTAR Method

SAMSTAR consists of a twelve step process which we briefly summarize below.

Firstly, as their heuristics handles one-to-many relationships, but not many-to-many, ER-diagrams are processed such that many-to-many relationships are transformed into two one-to-many relationships using a bridge table. Afterwards, the user selects the weighting factors, where the solution then calculates CTVs for all entities. The user is then presented with any entity e where $CTV(e) > Th$ and is then able to select the entities that should be considered as fact tables. Dimensions are then found by looking at

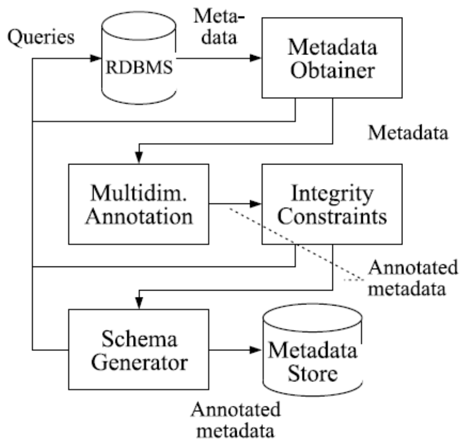


Figure 2: Architecture of the system presented by Jensen et al. Picture taken from [7].

many-to-one direct and indirect connections with the fact table and with the usage of WordNet[1] to find synonyms of entities. If either the original name or any of the synonyms returned by WordNet matches an A_DDP entry, then that entity is marked as a candidate dimension for the particular fact table. The user can then select the candidate dimensions. This is followed by a post-process step of the resulting star schemas which involves removing the time dimension if redundant, merging dimensions, and renaming fact tables and dimension tables. Finally, the process finishes by generating star schemas.

2.1.4 Evaluation

Song et al. show a way of semi-automatically generating star schemas based on ER-diagrams. They base their solution on the assumption that there always exist up-to-date ER-diagrams for all OSSs. This assumption is not supported by our talks with an experienced BI practitioner, this could prove problematic if the DW is based on outdated diagrams.

Song et al. propose two novel approaches of CTV and A_DDP for finding possible fact tables and dimension tables. However, it is questionable whether these two heuristics are enough for a satisfactory result. The SAMSTAR method does not utilize heuristics that inspect the columns of the tables and they could therefore end up with fact tables without any measures. Unless *factless fact table* as described by Kimball and Ross [6] are desired the benefit of such tables might be small.

Lastly, the set of entries for A_DDPs are predefined, but since not all systems are designed in English, the lack of customization at this point makes A_DDPs not applicable when non-English systems are used.

2.2 Discovering Multidimensional Structure in Relational Data

Jensen et al. [7] have done work on automatically constructing *snowflake schemas* on the background of relational OLTP databases. They claim to be the first to do this with only metadata from the relational source.

2.2.1 Description of Architecture

The architecture of their system is illustrated in Figure 2. From the RDBMS, the *Metadata Obtainer* retrieves metadata including: Table names, attributes (columns), properties (Primary Keys (PK), Foreign Keys (FK), etc.), and cardinalities [2].

At the *Multidimensional Annotation* component, attributes are put into three roles: (1) *Keys* (used to join tables), (2) *measures* (used for aggregating data for future cubes), and (3) *descriptive data* (used for dimensional data). An attribute is not given a definitive role, but rather a probability for being in each of these three roles. E.g.: Some attribute `time` might have a 65% probability of a *keys* role because of its name and data type. These probabilities are derived through a Bayesian network. The user is able to adjust how high the probability needs to be for an attribute to be identified as a key, according to a threshold ϑ.

The *Integrity Constraints* component is responsible for inferring information that is not explicitly given by the RDBMS. This primarily involves inferring relationships between tables where A is a foreign key that references the candidate key B if $cardinality(r_1) = cardinality(r_1 \ltimes_{r_1.A=r_2.B} r_2)$ where r_1 and r_2 are relations (tables). This is the case whether this relationship has been defined in the RDBMS or not.

The *Schema Generator* in the architecture is responsible for making the snowflake schema by using the fact tables and dimension tables found. The generated schemas are stored in the *Metadata Store*.

They argue that fact tables are typically several orders of magnitude larger than dimension tables and has to be identified first before performing any major operations such as joins. The process of finding fact tables involves the user and is semi-automatic. The user is presented with cardinalities (sizes) of tables and the possible measures they contain which the user can then choose which table(s) to consider as a fact table(s).

Dimensions are found afterwards by examining the relationships of the fact tables, in short, if a fact table has a foreign key reference to a table, that table is considered as dimension table. Lastly, hierarchies within the dimensions are discovered by checking if a dimension can be rolled-up into another.

2.2.2 Evaluation

There are several improvements in the work by Jensen et. al. compared to the previous related work. Firstly, the input information they use are taken directly from the RDBMS and thereby retrieves up-to-date metadata, this is in contrast to other related work which uses potentially out-of-date ER-diagrams. Secondly, they try to infer the relationships between tables (thus not relying entirely on the database design) and use these in their heuristics to create both fact tables and dimension tables.

Finally, while Jensen et al. claim the system is automated, they later make a contradictory claim: *"Identifying the fact table(s) is a semi-automatic process involving the user"* [7]. As the process of how fact tables are identified is only briefly mentioned, the amount of user intervention is unclear.

2.3 Automating Data Warehouse Conceptual Schema Design and Evaluation

Phipps and Davis [8] focus on automatically creating conceptual schemas for DWs based on ER schemas (or ER diagrams) of source databases.

[1] WordNet [5] is a lexical database for English.

[2] This related work refers to cardinality as the mathematical definition which is the number of elements in a set (amount of unique rows in a table), whereas outside of this related work we refer to the database design definition where cardinality details the different degrees of relationships: one-to-one, one-to-many, and many-to-many.

2.3.1 Their Examination of Related Work

Phipps and Davis investigated 16 different related work and found that there exists a large variety of models that they use. They also claim to be the first to automatically create a complete conceptual schema from an RDBMS.

In their investigation, they found that whether the related work modeled DWs conceptually or logically, they found that the star schema was the most common design model.

Their goal is to create a schema design in a user friendly manner and thereby decided to adopt a *Multidimensional Entity-Relationship* (ME/R) diagram [9] as the output. A ME/R diagram is an adaptation of an ER diagram that can describe multidimensional elements (such as time and date dimensions, fact tables, dimensions) which they also use to indicate which aspects of the diagram requires user intervention. This assists the user in knowing which aspects of the diagram requires his attention.

2.3.2 Algorithm for Schema Generation

Phipps and Davis propose a five step algorithm for schema generation that is based on two premises. The first premise assumes that the amount of numeric columns of a table correlates to the likelihood of that table being a fact table. Since most factual measures are numeric, this premise is in part useful. There does however also exist numeric values which are descriptive, e.g. a database that contains a product table which contains columns that describes the length, height, width, and weight of a product, these can all be represented by numerical values, but are not factually useful. The second premise assumes any many-to-one or many-to-many relationships are important for both potential fact tables and dimension tables. If for example the candidate fact table is generated from an `Order` table that contains an order from a customer, there would likely exist a many-to-one relationship from the `Order` table to the `Customer` table.

The algorithm takes an ER schema as an input and outputs a set of candidate ME/R schemas. The five steps of the algorithm are as follows. **Step 1** orders the entities according to the number of numeric columns and creates a corresponding fact node for each entity, which corresponds to a fact table in a DW. The entities are ordered this way as they argue that entities with larger number of numeric columns are generally better at answering queries regarding data analysis. Steps 2 to 5 iterate through each table found in step 1. **Step 2** adds each numeric column of an entity to the corresponding fact node. **Step 3** creates date and time dimensions. These are annotated differently in the schema to easily differentiate between elements that requires user intervention after the algorithm has terminated. **Step 4** consists of adding the remaining columns that are non-numeric, non-key, non-date, and non-time columns to the ME/R schema. **Step 5** recursively looks at the table's relationship and then examines the next level of relationships. If for example table A has a many-to-one relationship with B, and B has a many-to-one relationship with C, both B and C will be included in the final ME/R schema. If the relationship is not many-to-one or many-to-many, it is not included in the schema. After step 5 the first candidate schema has been created and steps 2 to 5 are repeated with another entity with the second largest number of numeric columns.

The algorithm creates multiple candidate conceptual schemas which, as Phipps and Davis argue, might not all be useful for the user. Following this, the user is required to manually select one or more of the candidate schemas and perform further refinement to those schemas. The refinement is a set of guidelines for the user, which he can use to manually process the candidate schemas and quickly determine which should be used and how to properly adjust the chosen schema(s).

2.3.3 Evaluation

As the title suggests, Phipps and Davis attempt to automate DW conceptual schema design and evaluation. The initial creation of the DW schema is done automatically with the intent that the user can choose which of the candidate ME/R schemas are useful and then finetune those schemas. However, the entire process is not fully automatic and Phipps and Davis compensate for this by providing the user with a set of guidelines for selecting the correct candidate schema and fine tuning that schema to the requirements. The solution thereby provides a more realistic approach that can ensure that the user receives the correct DW schema, even though it is not fully automated.

2.4 A Framework for Multidimensional Design of Data Warehouses from Ontologies

Romero and Abelló [10] have done work on automatically constructing fact constellations for DWs. Their work is of interest, as they both conduct a thorough analysis of the problem domain as well as propose their own method named *Automating Multidimensional Design from Ontologies* (AMDO).

2.4.1 Analysis of the Problem Domain

Romero and Abelló investigated related work that focus on how to automatically or semi-automatically construct multidimensional schemas, and they explain the following classifications:

- *Supply-driven*: This approach derives multidimensional designs with the available information in the sources (logical schemas, available ER-diagrams). Since supply-driven systems can usually extract this information themselves they are often good candidates for making multidimensional schemas automatically.
- *Demand-driven*: This approach focuses on eliciting requirements from the user to derive the multidimensional schema. As pointed out by the authors, the requirements are often not formalized, but is rather written down in natural language. This forces manual intervention to some degree, as algorithms can not handle these informally written requirements.
- *Hybrid approach*: This is a mixture of supply-driven and the demand-driven approaches, where each approach is used separately, typically with an automated supply-driven approach followed by a manual demand-driven approach.

Romero and Abelló reviewed several related work, including all the previously mentioned related work in the previous sections. In their critique they mention that for supply-driven approaches, two problems often occur: (1) With only the OSSs to consider it is necessary to generate numerous candidates schemas and usually present these to the user. While the multidimensional design has been automated (thus reducing time spent manually designing), it can be argued that the problem has now been moved to the user which now has to filter through all generated designs, which can be time-consuming. (2) When generating schemas, the common approach is to use simple heuristics (such as examining the table cardinalities or the amount of numeric columns it contains), the downside to this is that it may discover incorrect facts or miss them altogether.

A critique pointed out of demand-driven systems, is that user requirements are expected to be complete. Romero and Abelló argue that it can often be difficult for the users to have such a thorough overview of the source data that the requirements become exhaustive.

2.4.2 Proposed Method: AMDO

Beyond the survey, Romero and Abelló also propose AMDO which is a method within the hybrid category. A simplified illustration of what AMDO does is depicted in Figure 3.

AMDO is initially supply-driven in that no user input is necessary at the start and only an *ontology*[3] is required. This is followed by discovering potential dimensional concepts (dimensions, hierarchy levels,

[3]Ontology as used by Romero and Abelló is a set of entities, their data types, and relationships between entities. In this context, it is identical to a schema consisting of tables, columns, and relationships.

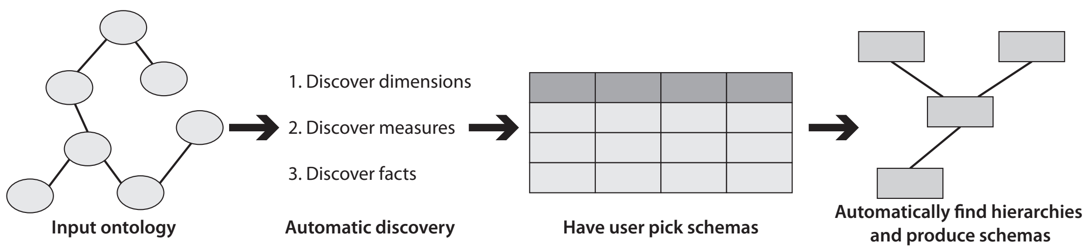


Figure 3: Simplified illustration of the steps taken by AMDO. Taken inspiration from [10].

or descriptors) and discovering potential measures. Before generating fact tables and dimensions based on these findings, they introduce two concepts: *search pattern* and *filtering function*. Essentially, a search pattern can be seen as a heuristic that can be applied to an ontology to categorize elements within the ontology. The filtering function uses search patterns to assess and rank these elements according to what the search pattern categorize them as. By this ranking, AMDO provides a ranked list of ontology elements where the higher the rank, the more likely the element is of being a fact table.

Up until now, AMDO has been entirely supply-driven. The user is now presented with the ranked list from which he picks the appropriate fact table(s) that his DW should be built around. Following this, AMDO finds dimension hierarchies that are related to the fact table and presents these to the user which he can then modify as he sees fit, e.g., deleting entire dimensions, deleting a hierarchy level, removing descriptors, etc. Finally after the user has determined the appropriate dimensions, AMDO produces a fact constellation schema that can be used in the DW.

2.4.3 Evaluation

AMDO remedies some of their perceived problems with the related work presented previously in Section 2.1, 2.2, and 2.3.

The work of Song et al. and Jensen et al. are primarily supply-driven, and AMDO remedies the problem presented in Section 2.4.1 regarding the observation of generating too many results. This was done by initially being supply-driven, and then switch to being more demand-driven when the user has a more rich conceptual layer with more information.

As presented in our introduction, a lack of customization of the current approaches was pointed out. AMDO allows for the user to customize to some degree, as it is possible to implement one's own filtering functions that can adjust how multidimensional schemas are ranked.

However, there is still room for customization. Specifically, the specification of rules about how the multidimensional model is supposed to look like, naming conventions, and what steps to take when implementing the multidimensional model into a physical schema.

Furthermore, Romero and Abelló adopt the approach of Phipps and Davis by making use of a conceptual model (in their case, ontologies) which in their case provides more information than logical schemas. While this provides additional information, Romero and Abelló do not mention the scenario of handling data that is not stored in RDBMSs. It is therefore possible that AMDO does not handle the scenario where data is stored in a semi-structured manner such as with CSV files, etc.

2.5 Summarization of Related Work

This section provides a summarization of what the related work lacks and provides a clarification of what DWStar can contribute with.

2.5.1 Degree of Automatization

The concept of creating multidimensional schemas automatically is not something new and has been done before by numerous others. The related work can be split into the three previously mentioned types of approaches: Supply-driven [11, 12, 13, 14], Demand-driven [15, 16, 17], and Hybrid [3, 7, 18, 19, 20, 21, 22]. It was only the related work that correlated most with our project that have been explained in depth in previous sections. Supply-driven approach has the most potential to be fully automated and provide the user with candidate multidimensional schemas. This can result in the user being overwhelmed by the number of candidates and this is what hybrid approached tries to rectify by combining automatic and manual steps together in one solution. We follow the idea behind a supply-driven approach and allow the user to choose whether he wants to intervene in any part of the process of DWStar. This allows the user more control over the outcome and a deeper understanding of how DWStar operates.

2.5.2 Lack of Customization

It was found that the customizability of examined related work is either low or non-existent. For instance, searching for potential fact tables, Song et al. from Section 2.1 present a method which does not examine attributes of tables, but only looks at the topology of an ER diagram, whereas Phipps and Davis in Section 2.3 focuses heavily on the number of numeric attributes of a table to detect whether the table is a candidate fact table. Only Romero and Abelló attempts to implement customizability through their filtering functions. However, more work can still be done on customizing the process for the user. DWStar does so by allowing the user to choose between multiple heuristics (some are inspired by the related work) as well as allowing implementations of the user's own heuristics. A fast and straight-forward way some related work involve the user is to allow for modification of thresholds, such as with ϑ by Jensen et al. and Th by Song et al.

Due to the nature of heuristics, it is often difficult to make any guarantees of how well a given heuristic performs according to an organization's goals. It is therefore preferable to give the user the option to try different heuristics (or combination of heuristics) to see what fits best.

DWStar rectifies the lack of customizability by allowing the user to choose between multiple heuristics (some are inspired by the related work) as well as allowing the user to implement his own heuristics.

2.5.3 Expected Input Information

There is no consensus in the related work for which input the approaches requires: Song et al. and Phipps and Davis work with ER-diagrams, Jensen et al. work with logical schemas, and Romero and Abelló work with ontologies. However, none of the related work ensures up-to-date metadata. In discussions with a BI practitioner, the only information which is reliably available and up-to-date is metadata directly from the OSSs such as information about tables, columns, and relationships from RDBMSs or inferred metadata from CSV files.

The techniques used for each related work is affected by the input domain. For instance, logical schemas specifies data types while ER-diagrams typically do not. On the other hand, ER-diagrams have explicitly stated cardinalities while logical schemas do not. E.g., the heuristics of Song et al. may rely on ER-diagrams' ability to provide cardinalities, and therefore one has to consider how this would be translated to DWStar.

We wish to examine this phase of extracting information from the input further. As logical schemas are the most reliable, we base our work on these, but also try to construct a model which is able to encompass inferred information. For instance, inferring cardinalities for a logical schema can open up the opportunity to also make use of the techniques that Song et al. presented.

Chapter 3

Concepts

This chapter delves into the fundamental structure of DWStar. Section 3.1 describes the architecture of DWStar and introduces DWStar's multiple phases and the responsibility of these phases. Additionally, Section 3.1 describes the notion of modules, which contribute with heuristics for generating the candidate star schemas. The theory that is used internally to represent the OSSs and star schema candidates is described in Section 3.2 and 3.6, respectively. Section 3.3, 3.4, 3.7, 3.8, and 3.9 describe the phases, what they do, and what contributions they bring to the generation of star schema candidates. Section 3.5 describes how the user is able to configure DWStar and how modules can be connected.

3.1 Architecture

DWStar is divided into five *phases*, where each phase contributes to the common goal of creating one or more candidate star schemas. Figure 4 displays the phases of DWStar and some of the modules they contain.

The arrows denote a transferral of data, where the gray writing above the arrows define what kind of data is transferred. There are two repeated models: The *common model* (see Section 3.2) and the *star model* (see Section 3.6). The common model is used to make an abstraction over all types of OSSs while the star model is a representation of a star schema. Phases contain different *modules*, where each implemented module is created on the background of a heuristic. Throughout this report, we use the name of a phase to refer to the modules that belongs to the phase. E.g. the *refinement phase* consists of one or more *refinement modules*. DWStar provides multiple modules for each phase, but a user can also implement his own modules if he has heuristics that better fits his needs.

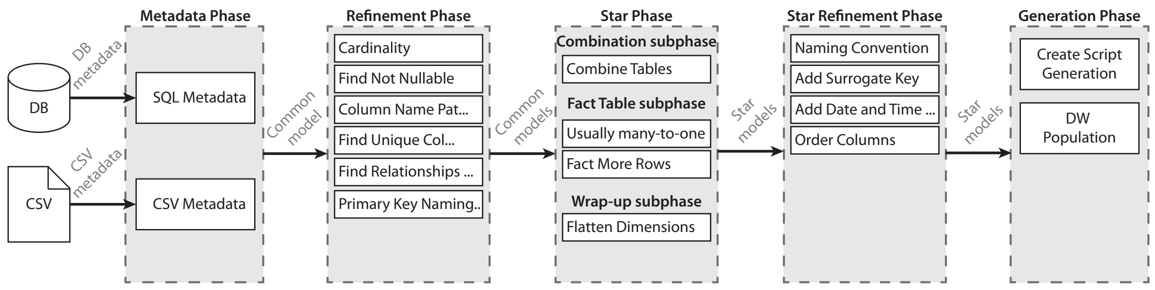


Figure 4: Architecture of DWStar.

Consider the example where the *refinement phase* consists of three different refinement modules: `Cardinality`, `Find Not Nullable`, and `Column Name Inference`. These have been selected by the user from a collection of refinement modules and saved in a *configuration* (see Section 3.5). A configuration memorizes the selections in each phase which is then put into persistent storage such that DWStar can be executed with the same configuration multiple times with different OSSs.

Additionally, it is possible for the user to intervene between the phases. This means that the user can look at the result of any phase and make any modifications to the models before continuing on to the following phase (further detail in Section 3.1).

The phases of the architecture are described from left to right in the following sections. A short description is given for each phase, but a more in-depth examination is provided later with an overview of the available modules. For a quick reference, Appendix A contains a list of all modules provided with DWStar as well as a short description of them.

Metadata Phase

Within the *metadata phase*, each type of OSS has a corresponding *metadata module*. Each metadata module takes in a specific type of OSS and output a common model. This phase ensures that regardless of whether the OSS is based on a CSV file or an RDBMS they are represented homogeneously in DWStar. If multiple OSSs exist, the metadata phase then combines the common models from each module into a single common model, which is then provided to the refinement phase.

Refinement Phase

The *refinement phase* receives a common model from the metadata phase and applies refinements to it by altering and adding information to the common model. The phase is primarily used for two reasons: Inferring missing metadata from OSSs that does not explicitly define these (such as CSV files), and finding undefined relationships between tables in the common model. Finding relationships is particularly important since we can retrieve schemas from multiple OSSs, that are implicitly related to each other.

Star Phase

Once the common models have been enriched by the refinement phase, the *star phase* begins. The star phase uses *star modules* to construct one or more star models from the common models. Some star modules are inspired by the related work and each star module is used to find a subset of the information required. For instance, some star modules identify fact tables while other star modules identity dimensions. To help separate these, this phase contains multiple subphases.

Star Refinement Phase

The *star refinement phase* modifies the star models to conform to how an organization's star schemas should look. For instance, one *star refinement module* can involve changing the names of the star schemas to comply to some naming convention, and another module extends the length of data types to allow for larger values in the future. It is recommended that the user intervenes in this step since this is the last step before the generation phase. When intervening the user can choose which of the candidate star models to provide to the generation phase, this also makes it possible for the user to change the star models such that it adheres to his requirements.

Generation Phase

Finally, the *generation phase* is responsible for taking the star models and translating them into a concrete format. For instance, creating SQL scripts for SQL Server [23], or any other implementation that is able to make use of the information contained in a star model. These scripts can also populate a DW with data from the OSSs.

User Intervention

The entire process of DWStar is automated through the implemented modules. However, even an otherwise good module may miss some small details. We therefore allow the user to intervene between each of the phases and modify the models. E.g. add missing column names, add missing relationships, and change data types. Apart from modifying the models, the user is also able to delete models that are not deemed fit.

3.2 Common Model

The *common model* is a layer of abstraction on top of the metadata obtained from OSSs. With the common model, the specific details pertaining to relational databases, CSV files, etc. is abstracted away in this model where the metadata structure is seen in a common format.

There are some considerations to keep in mind for the common model for it to be able to accommodate different types of OSSs. For instance, information such as data type of a column can often be queried directly for an RDBMS. However, for CSV files it is often the case that data types are not specified explicitly and it is therefore necessary to infer. Inferring new information can be performed based on a data sample of the OSS. Using data sampling, there is the risk of not inferring correctly if the data sample is limited in size or the data sample lacks the diversity required to correctly specify the data type. If for example we want to determine whether a column is nullable, it is required to discover a null value in the data sample. If no null values are discovered it does not mean with certainty that the column is non-nullable, as null values could occur in a later version of the data. This uncertainty is present in multiple scenarios and we therefore apply a *confidence* to inferred information. Throughout the model, multiple candidates, each with their corresponding confidence $c \in \mathbb{R}$, are shown to take into account the certainty of the found information, where $0 \leq c \leq 1$. A confidence of 1 indicates the highest confidence in a result. Confidence was chosen over probability, as probability implies a more specific definition (from probability theory), while confidence is seen as less rigid, since each module can assign a confidence in any way.

Common Model

A *commonmodel* is represented as a set of tables *Tables*, where each table represents an OSS, such as a database or a CSV file.

Tables

Each $table \in Tables$ is defined as in Equation 3.1:

$$table = (name, schema, Columns, constraints, Relationships, database) \tag{3.1}$$

Where:

- *name* is the name of the table (or equivalently, the name of a CSV file).

- *schema* is the schema that the table is contained in ($schema = \varepsilon$ for CSV files, indicating it is not applicable).
- *Columns* is the set of columns within the table (described in Equation 3.2). A column may only belong to one table.
- *constraints* describes the following constraints: primary key, unique, and not null (described in Equation 3.4).
- *Relationships* is a set of relationships that describes how this table relates to other tables (described in Equation 3.5).
- *database* is a connection string describing the connection to the origin of the data. This is applicable to both databases and CSV files.

Through this report, a *dot-notation* is used when referring to a property of a tuple, e.g. if accessing the name of a table, the following is written: $table.name$.

Columns

Each $column \in Columns$ is defined as in Equation 3.2:

$$column = (id, ordinal, Names, Datatypes) \tag{3.2}$$

Where:

- *id* is a globally unique identifier for the column. It is necessary for tuples to be unique since multiple columns across different OSSs can have the same metadata.
- $ordinal \in \mathbb{N}_1$ is the column's position in the table.
- $Names = \{(name_1, c_1), ..., (name_n, c_n)\}$ is a multiset, where for the i-th tuple, $name_i$ is a suggested name for the column, and c_i is the confidence of this suggestion being correct.
- $Datatypes = \{(datatype_1, c_1), ..., (datatype_m, c_m)\}$ is a multiset, where for the i-th tuple, $datatype_i$ is a suggested data type for the column (defined in Equation 3.3), and c_i is the confidence that this suggestion is correct.

Names and *Datatypes* are the first two examples of where the uncertainty of information is expressed with a confidence, as both of these might be lacking in a CSV file and thus has to be inferred. How these candidate names and data types are handled are detailed below with the `Highest Confidence Function`.

Data Types

Each $datatype \in Datatypes$ is defined as in Equation 3.3:

$$datatype = (type, length, scale, precision) \tag{3.3}$$

- $type \in Types$ is the data type of the column (such as integer, string, datetime), where *Types* is a set of all available data types described further in Section 4.1.3.
- $length \in \mathbb{N}_1 \cup \{\varepsilon\}$ is for a string data type the maximum number of characters that can be accepted as input, and for numeric or binary data types it is the maximum number of bytes used for storage. If the data type does not use a length, then ε is used instead.
- $scale \in \mathbb{N}_0 \cup \{\varepsilon\}$ is the number of digits to the right of the decimal point in a numerical value. For instance, 512.62 has a scale of 2. $scale = \varepsilon$ for non-applicable data types.
- $precision \in \mathbb{N}_1 \cup \{\varepsilon\}$ is the number of digits for a numeric data type. For instance, 512.62 has a precision of 5. $precision = \varepsilon$ for non-applicable data types.

Each type of OSS can have their own definition of length, scale, and precision for their specific data types, but DWStar has functions that maps an OSS's data type, length, scale, and precision to the uniform format for the common model.

Constraints

$constraints$ (from Equation 3.1) is defined as in Equation 3.4:

$$constraints = (Primarykeys, Notnulls, Uniques) \tag{3.4}$$

Where:

- $Primarykeys = \{(Columns_1, c_1), ..., (Columns_n, c_n)\}$ is a multiset, where for the i-th tuple, $Columns_i \subseteq table.Columns$ is a suggested set of columns which together form a primary key and c_i is the confidence that the columns form a primary key.
- $Notnulls = \{(column_1, c_1), ..., (column_m, c_m)\}$ is a multiset, where for the i-th tuple, $column_i \in table.Columns$ has a potential *not null* constraint and c_i is the confidence that this assertion is correct.
- $Uniques = \{(Columns_1, c_1), ..., (Columns_o, c_o)\}$ is a multiset, where for the i-th tuple, $Columns_i \subseteq table.Columns$ is a potential set of columns which together form a set without duplicate data and c_i is the confidence that the columns, collectively, are unique.

Other SQL constraints including *check*, and *default* have not been included in the common model, as no heuristics have been found which make use of these. It is also hard to infer these from other OSSs as these constraints might not be supported. Similarly to the *Names* and *Datatypes* detailed in *column*, *Primarykeys*, *Notnulls*, and *Uniques* are also described further with the `Highest Confidence Function`.

Relationships

Each $relationship \in Relationships$ (from Equation 3.1) is defined as in Equation 3.5

$$relationship = (Columns_{rel}, cardinality) \tag{3.5}$$

Where:

- $Columns_{rel} = \{(column_{anchor_1}, column_{link_1}), ..., (column_{anchor_n}, column_{link_n})\}$ where for the i-th tuple, $column_{anchor_i}$ is (part of) the primary key or (part of) a unique constraint in the table that is a being referenced, and $column_{link_i}$ is (part of) the foreign key that defines the relationship between two tables.
- $cardinality \in \{one\text{-}to\text{-}one, one\text{-}to\text{-}many, many\text{-}to\text{-}one, \varepsilon\}$ describes the relationship between two tables from the perspective of the anchor. ε is used when the cardinality is not known or inferred. *many-to-many* is not included, as this case is rarely seen in RDBMSs, and are instead represented with bridge tables.

Sample Function

The function $sample : Columns \times n \rightarrow Results$ takes as input a set of columns *Columns* and an integer n which defines the desired amount of rows to extract. The output is the multiset *Results* containing n rows for the columns chosen. If the OSS containing *Columns* has less than n rows, only the available rows are extracted.

Rowcount Function

$rowcount : table \rightarrow \mathbb{N}_0$ takes in a $table$ and outputs the number of rows in that table.

Combine Confidence Function

The function $averageconfidence$ can be used on all multisets which deals with possible candidates and confidences. This applies for $Names$ and $Datatypes$ in $column$ and $Primarykeys$, $Notnulls$, and $Uniques$ in $constraints$.

$averageconfidence$ takes in a multiset of 2-tuples, where the first element in a tuple represents a candidate and the second element represents a confidence. If a candidate is present in multiple tuples, the average confidence is found for that candidate. E.g. the definition of a column name in $column.Names$ is $(name, confidence)$. We then say that the element $name$ represents a candidate and $confidence$ represents the confidence in this candidate. As multiple candidates can be given for various constituents in the common model, we utilize $averageconfidence$ to compute the average confidence in each candidate, where candidates are grouped based on the first element in the two-tuple.

Example. *Imagine that we have constructed a multiset of column name candidates:*

$$names = \{(''ID'', 0.8), (''A'', 0.1), (''ID'', 0.5), (''City'', 0.2)\}$$

The multiset states that four candidates exist for a column: `ID`*,* `ID`*,* `A`*, and* `City`*. When we want to retrieve the list of candidate names, we then use the function $averageconfidence$ on the multiset of names and it will return the following:*

$$names = \{(''ID'', 0.65), (''A'', 0.1), (''City'', 0.2)\}$$

This result states that the overall confidence for a specific candidate is determined by the average confidence, that was given by the modules.

Highest Confidence Function

The function $highestconfidence$ can be used to retrieve the value in a multiset of 2-tuples (as described in the previous section) which has the highest average confidence. The average confidences are calculated using the function $averageconfidence$.

Example. *Assume a table consisting of two columns A and B and that there exist two modules which each provide a confidence to whether these columns can be considered as primary keys. Assume that this results in $table.Primarykeys = \{(\{A\}, 0.5), (\{B\}, 0.5), (\{A\}, 0.8), (\{B\}, 0.1)\}$. Since both modules attempts to infer the primary key, we can apply $averageconfidence$ which gives the result: $(\{A\}, 0.65), (\{B\}, 0.3)$. Following this, we can apply $highestconfidence$ which returns $\{A\}$, since it had the highest confidence in the set.*

Both $averageconfidence$ and $highestconfidence$ can be used anytime by modules, but are also used when transforming a common model into a star model. It is for example extensively used between the refinement phase and the star phase, as we want to make sure that there only exist one name and primary key for a column (using the $highestconfidence$ function) and that the remaining applicable constituents is not duplicated multiple times (using the $averageconfidence$ function).

3.3 Metadata Phase

The metadata phase is responsible for inferring the schema of OSSs through metadata modules. As mentioned in Section 3.1, there is one metadata module for each type of OSS (one for SQL server, one for CSV files, etc.) and each of them construct a common model for each OSS provided by the user. The common model produced by each metadata module is then concatenated into a single common model and forwarded to the refinement phase.

As observed in the architecture in Figure 4, the input for each metadata module depends on the format of the OSS. However, common for all of them is the goal of obtaining metadata necessary for creating a common model. The goal of a metadata module is to only retrieve the information that is directly retrievable. This is in contrast to the subsequent phases which conduct a prolonged analysis on the data. For some metadata modules, not all the information is directly available to fill out the common model. For instance as mentioned before, the column names can be missing in a CSV file.

Each of the metadata modules work independent of each other as they only have to convert the data to a common model and send it onwards to the refinement phase. It is therefore not necessary to consider dependencies or the execution order for the metadata modules.

3.3.1 Modules

The following two sections illustrate the necessity of having metadata modules that are tailored to specific OSSs. We illustrate this by explaining how our metadata modules retrieve some of the data required to build a common model.

SQL Metadata

The metadata module that generates a common model based on a relational database is one of the most straight-forward metadata modules to create. This is due to the fact that the metadata from an RDBMS naturally conforms to the schema-like structure of the common model. Many RDBMSs implement a standard called *Information Schema Views* [24] (ISVs) to some degree, which allows for querying metadata of all entities of an RDBMS. Therefore, almost all constituents of the common model (Equation 3.1 and 3.2, respectively) are easily retrieved using ISVs, or other system specific alternatives. However, some of the information such as cardinality of relationships between tables are not presented by the RDBMS and instead has to be inferred.

As was shown for the common model, multiple constituents had a confidence associated with them. For instance, both *Names* and *Datatypes* for a column has a confidence associated with them in case that there are multiple suggestions that need to be considered. For instance, a CSV file without clearly specified column names might have multiple suggestions. However, for relational databases the information is often directly retrievable using ISVs, and therefore there are most likely only a single suggestion with maximum confidence for constituents.

CSV Metadata

To fit the common model, *database* represents the folder containing the CSV file(s). Each *table* element corresponds to one CSV file, where *name* is the name of the file. *Columns* is derived from each column present in the CSV file. *Relationships* and *Constraints* are not specified in a CSV file, and thus have to be inferred in the refinement phase.

For each *column*, *ordinal* is found according to the position of the corresponding column in the CSV file. As for *Names* and *Datatypes*, these can only be derived with high confidence if specified in a schema that is brought along or columns names are specified in the first line of the CSV file. Otherwise, these need be inferred. If the data types of columns have not been specified, this module attempts to infer this metadata as described in Section 4.4.

3.4 Refinement Phase

The refinement phase is responsible for inferring any missing metadata in a common model, which includes finding relationships between tables in the common model, that are not already related to each other. After the refinement phase, we want to ensure that a common model represents a connected graph, where all tables in a common model is connected directly or indirectly. This means that if the common model is not a connected graph, we divide the model into one or more common models, where each common model represents a connected graph.

Common models from semi-structured sources usually stands to gain the most from this phase, since they do not always contain as much metadata information as their structured counterpart. This phase remedy this by inferring the missing metadata, such as data types or relationships between tables.

3.4.1 Modules

This section describes the available refinement modules. Refinement modules can choose to use both the existing values from a common model, as well as calculate these values themselves, e.g., the module for determining primary keys can reuse the not null and unique constraints that has been provided by other modules or instead use the functions *sample* and *rowcount* to identify primary keys in the data.

Cardinality

None of the currently supported OSSs explicitly supply the cardinality of relationships between tables. Therefore, a refinement module is defined to infer them.

Yeh et al. [25] provide a heuristic for discovering cardinalities, which states that if a value for a primary key only appears once for a given relationship, then the cardinality is said to be one-to-one, otherwise it is recognized as one-to-many (seen from the perspective of the anchor). It is assumed that any many-to-many relationships in a conceptual schema has been transformed into two one-to-many relationships (seen from the perspective of the original two tables) and an intermediary table (also known as a *bridge table* [26]).

For the common model, the function $card : relationship \to \{one\text{-}to\text{-}one, one\text{-}to\text{-}many\}$ in Equation 3.6 examines a relationship and returns the estimated cardinality of that relation.

$$card(relationship) = \begin{cases} one\text{-}to\text{-}one, & \text{if } \forall(column_{anchor}, column_{link}) \in \\ & relationship.Columns\ (sample(\{column_{link}\}, 1000) \\ & \text{does not contain duplicate rows} \\ one\text{-}to\text{-}many, & \text{otherwise} \end{cases} \tag{3.6}$$

card is based on a data sample obtained using *sample*. In this definition and later ones we use the value 1000, which means that we wish to retrieve 1000 rows. This limit of 1000 rows is arbitrarily chosen and could be increased to improve the accuracy of the module, however, it would also increase the time it takes to execute the module.

Example. *In this example, the sample data has been extracted both from the anchor table and the link table where there are only one anchor column and one link column in this case, shown in Table 1 and Table 2, respectively. The column of primary interest is* `Column B` *as it is the only link column in this*

Table 1: Anchor table with primary key values.

Column A (PK)
1
2
3
4

Table 2: Link table with foreign key values.

Column B (FK - refers to A)
1
2
3
3

example. However, as the data of the column contains duplicate values (3 appears twice), the cardinality between the two tables is one-to-many, as the condition for one-to-one is violated.

Column Name Inference

When the provided schema does not contain information about column names, it becomes harder to see the intent of columns. However, the data in a column at times provide adequate information to infer the column names.

With the use of *DBpedia* [27] we can apply a function $lookup : value \rightarrow Classes$ which takes in a single string *value* and outputs a set of classes that represents classifications of *value*. The function uses DBpedia to find a Wikipedia [28] article that matches *value* and returns the tags of that article as *classes*. As an example, if $value = "Copenhagen"$ then the output could be $classes = \{"settlement", "place", "populated\ place"\}$.

Since the class of a column can be used as the name of the column, we need to ensure the most recurring class for all values in a column is used as the name. However, the input of *lookup* is only one value so we introduce the function $bestclass : column \rightarrow class$ defined in Equation 3.7 which is given a column and returns the class with the highest multiplicity (number of occurrences). For Equation 3.7, we define $\uplus$ to be a union between multisets, which results in a multiset containing all the values of the supplied multisets. For example $\{1,1,2\} \uplus \{1,2\} = \{1,1,1,2,2\}$.

$$bestclass(column) = \text{Arbitrary maximal element of} \biguplus_{value \in sample(\{column\},1000)} lookup(value) \tag{3.7}$$

The result of *bestclass* is the maximal class (the one with highest multiplicity) after calling *lookup* on multiple values. The confidence is then calculated based on the ratio between the multiplicity of the returned class and the amount of supplied values (in this case, a maximum of 1000). The returned class does not necessarily have an easy readable name (e.g. "populated place") and the module therefore performs a translation from a class to a more readable name after the maximal element has been found (e.g. *"settlement"* is translated into *"town"*).

Example. *Using the lookup function on the three values: "Berlin", "London", and "Copenhagen", the following multiset of classes can be computed:*

$$Classes = \{settlement, place, populated\ place, city\} \biguplus \{settlement, place, populated\ place\}$$
$$\biguplus \{settlement, place, populated\ place\}$$

In this scenario, the maximal elements are "populated place", "place", and "settlement". In this case, name is arbitrarily chosen to be "settlement" among the maximal elements. We then perform a conversion from the class to a more readable name. The suggested column name would then become "`town`".

Table 3: Example of a table containing data in multiple formats.

Column **A**
34
95.3
"some data"

One should exercise caution when using the *lookup* function with external sources such as DBpedia to determine the classes, since the values are sent through the internet to an external server. In the case with DBpedia, organizations should use a local instance of DBpedia to avoid violating data privacy laws.

Column Names Pattern

Automatically inferring the names of columns in a CSV file is particularly difficult since the only thing available is the data of those columns. Some columns typically have a certain pattern (email address, phone number, etc.) that can be found, and this module therefore applies a name to that column if a pattern is recognized. This module is further explained in Section 4.5.1.

Detect Data Type

To detect the data type of a column, inspiration is taken from Armbrust et al. [29] which describe a heuristic for Spark SQL that infers the schema in a JSON file. Armbrust et al. finds the most specific data type that can contain the values of that column. The most generic data type is a `string` as it ought to preserve the data regardless of data type. Although this technique to find data types is applied on JSON files, it can also be used for CSV files. The function $fittingdatatype : column \rightarrow datatype$ in Equation 3.8 is given a column and from that it infers the data type of all the values in the column. We use $inferdatatype : value \rightarrow datatype$ to infer the data type of a single value. From the multiset of data types that multiple evaluations of $inferdatatype$ produces, we then find the most fitting data type, which can encompass all the sampled values.

$$fittingdatatype(column) = \text{Find most fitting data type for} \biguplus_{value \in sample(\{column\},1000)} inferdatatype(value) \tag{3.8}$$

The calculation of the confidence is based on how many occurrences the most fitting data type had. E.g. if it found hundred `integers` and one `string`, the result would be `string`, as it can encompass both `integers` and `strings`. However, the confidence should be low, as a single value changed the other hundred values into a `string`.

Example. *This heuristic can be illustrated with the simple column shown in Table 3. The CSV data is in this example shown as tabular data. This table displays Column `A` together with the data contained within this column. When initially going over the data, the first value encountered is `34` as this is a whole number, the most specific data type here is so far an integer. However, this is expanded to the more general data type of a decimal when the value `95.3` is found. Lastly, when the row containing “`some data`" is found, we will have to resort to the most generic data type (string) to contain all of the data.*

Find Not Nullable

Knowing whether a column allows null values provides useful information for the modules in the star phase. The function $notnull : column \rightarrow \{true, false\}$ in Equation 3.9 takes in a column and using the *sample* function determines if a column is nullable.

$$notnull(column) = \begin{cases} true, & \text{if } sample(\{column\}, 1000) \text{ does not contain null values} \\ false, & \text{otherwise} \end{cases} \tag{3.9}$$

The result of *notnull* is either `true` or `false`, if the result is `false`, then the module continues to the next column. In the cases where the result is `true`, the column is added as not nullable and the confidence is set using the formula $\frac{|sample(\{column\},1000)|}{1000}$, which determines a confidence based on the amount of available rows.

Example. *If we requested* 1000 *values from the sample function and retrieved* 1000 *values, the confidence would be* $\frac{1000}{1000} = 1$. *If instead sample only retrieved* 255 *values, the confidence becomes* $\frac{255}{1000} = 0.255$.

Find Unique Columns

Unique columns are discovered by using *sample* as well. The function $unique : Columns \rightarrow \{true, false\}$ in Equation 3.10 determines if a set of columns collectively contains unique values.

$$unique(Columns) = \begin{cases} true, & \text{if } sample(Columns, 1000) \text{ does not contain duplicates} \\ false, & \text{otherwise} \end{cases} \tag{3.10}$$

The result is `true` if none of the values returned are duplicated. If multiple columns are given as input, it checks if the values in the columns are collectively unique. When the result is true, the unique constraint is added to the table, from which the column(s) originated from. The formula for computing the confidence is identical to that used in `Find Not Nullable`.

Find Primary Keys

Primary keys are inferred by looking at the uniqueness and nullability of the columns. The function $primarykey : Columns \rightarrow \{true, false\}$ uses the functions *notnull* and *unique* to evaluate whether a set of columns can be a primary key or not.

$$primarykey(Columns) = \begin{cases} true, & \text{if } (\forall c \in Columns\ notnull(c)) \wedge unique(Columns) \\ false, & \text{otherwise} \end{cases} \tag{3.11}$$

Columns that are a candidate primary key returns a `true` value from the function. The confidence in this result is based on the average confidence on the not null and unique constraints, that was used to perform the check. The formula for this is shown in Equation 3.12. Here $unique_{conf}(Columns)$ represent the confidence of the columns being unique and $notnull_{conf}(c)$ represent the confidence of a column being not null.

$$confidence_{findpk}(Columns) = \frac{unique_{conf}(Columns) + \frac{\sum_{c \in Columns} notnull_{conf}(c)}{|Columns|}}{2} \tag{3.12}$$

Example. *Say we have three columns:* `A`*,* `B`*, and* `C`*, for these to collectively be a primary key, we need four conditions to be upheld:*

1. `A` *can not contain null values*

2. `B` *can not contain null values*
3. `C` *can not contain null values*
4. `A`*,* `B`*, and* `C` *must collectively be part of a unique constraint*

If all of these conditions are true, then these columns can be a primary key. We then calculate the confidence of this primary key using some made up numbers:

$$unique_{conf}(\{A,B,C\}) = 0.8$$

$$\frac{\sum_{c\in\{A,B,C\}} notnull_{conf}(c)}{|\{A,B,C\}|} = \frac{0.8+0.5+0.7}{3} = 0.\overline{66}$$

$$confidence_{findpk}(\{A,B,C\}) = \frac{0.8+0.\overline{66}}{2} = 0.7\overline{3}$$

Primary Key Naming Pattern

Since several columns can be both unique and not null, there can exist multiple candidate primary keys for a table. To ensure that a table only contains a single primary key, this module infers the primary key based on the name of the columns in a table.

This module makes two assumptions regarding primary keys of columns. The first assumption is that a primary key contains one of the following keywords: `ID`, `Key`, `No`, or `Ref`. The second assumption assumes that if the primary key consists of a single column, the name of that column correlates to the name of the table. Therefore, this module does not consider composite keys. E.g., for a table `Person` the primary key is assumed to be `PersonID`, `PersonKey`, or similar variations. The confidence is calculated with Equation 3.13.

$$findcandidate(column) = \begin{cases} 0.9, & \text{if } \exists name \in column.Names \text{ that contains a PK keyword} \\ & \text{ and contains the name of the table} \\ 0.7, & \text{if } \exists name \in column.Names \text{ that contains a PK keyword} \\ 0.0, & \text{otherwise} \end{cases} \tag{3.13}$$

Find Relationships From Names

When receiving data from multiple OSSs, there can sometimes exist relationships between them which is not defined. If for example multiple CSV files are the OSS, it is very useful to find connections between them. To infer whether two tables are connected, we examine the names of the columns of both tables, if there are a correlation between them, this would suggest there exists a relationship. The implementation of this module is described in Section 4.5.1.

Example. *If we have a column named* `CustomerID` *in the table* `Sales` *and another table* `Customers`*, we would argue that this column name strongly suggests a relationship between the tables* `Sales` *and* `Customers`*.*

3.4.2 Common Model Grouping

To describe the concept of *common model grouping*, we provide a small example, which highlights the concept. The algorithm used is detailed in Section 4.5.2.

Figure 5 illustrates an instance of a common model. The tables in `Common Model 1` originate from two OSSs, the ones with `(RDBMS)` are from a relational database and the ones with `(CSV)` come from a CSV file. We observe that `Common Model 1` contains nine tables, consisting of two sets of tables that are connected, one with four tables and another with five, thereby forming two separate groups without any relationships to each other. Using this observation, we determine that it is safe to separate each group into different common models, since no relationship exists between them. This separation is useful for the star phase so potential star models can be made for each separated part of the common model. Based on these two unrelated groups, DWStar creates a common model for each group. The result of the grouping of Figure 5 can be seen in Figure 6.

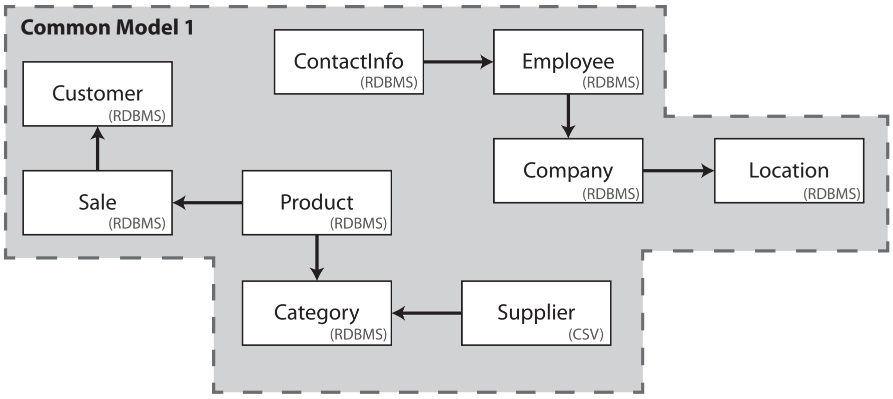


Figure 5: A common model containing a disconnected graph.

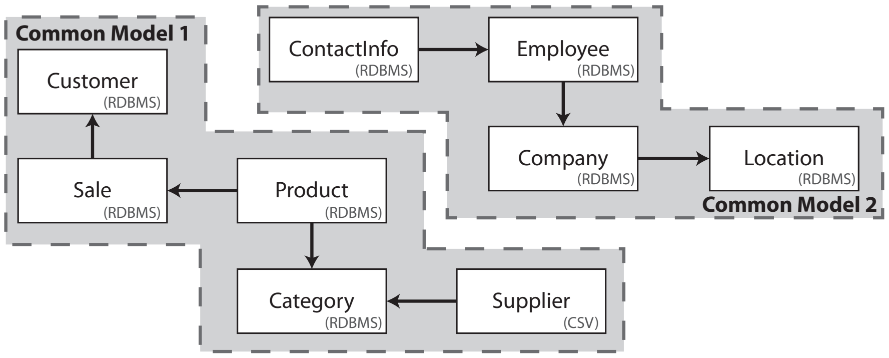


Figure 6: Grouped common models.

3.5 Configuration

Configurations help the user customize which modules that are used in each phase. So far, configurations have only been mentioned informally. This section serves to formally define a configuration and describe dependencies between modules.

3.5.1 Definition of Configuration

A configuration contains a collection of modules for each phase. Some of the phases have an execution order, which determines the order in which modules are executed. This is particularly important since some modules' input depends on the output of other modules, this concept is further explored in Section 3.5.2. The modules included in each collection are hand-picked by the user or the user can choose to use any configuration already existing in DWStar. When a configuration is constructed, it is saved for persistent storage and is thereby reusable.

Formally, a $configuration$ is defined as in Equation 3.14:

$$configuration = (name, Metadata, Refinement, star, StarRefinement, Generation) \tag{3.14}$$

Where:

- $name$ is a unique name of the configuration.
- $Metadata = \{(metadatamodule_1), ..., (metadatamodule_n)\} \subseteq Metadatamodules$ where $Metadatamodules$ is the set of all available metadata modules in the metadata phase.
- $Refinement = \{(refmodule_1, order_1), ..., (refmodule_m, order_m)\}$ where $\{refmodule_1, \ldots, refmodule_m\} \subseteq Refinementmodules$ where $Refinementmodules$ is the set of all available refinement modules of the refinement phase, and $\{order_1, \ldots, order_m\} \in \mathbb{N}_1$ defines the order in which the refinement modules are applied through a non-repeating sequence of natural numbers, where the module with the lowest number is executed first.
- $star$ is collection of all modules used in the star phase (described shortly in Equation 3.15).
- $StarRefinement = \{(starrefmodule_1, order_1), ..., (starrefmodule_o, order_o)\}$ which is similar to $Refinement$, with the exception of utilizing $\{starrefmodule_1, \ldots, starrefmodule_o\} \subseteq Starrefmodules$ instead.
- $Generation = \{(generationmodule_1, order_1), ..., (generationmodule_p, order_p)\}$ which is similar to the $Refinement$, with the exception of utilizing $\{generationmodule_1, \ldots, generationmodule_p\} \subseteq Generationmodules$ instead.

$star$ in $configuration$ is defined as in Equation 3.15

$$star = (combination, Facttable, Wrapup) \tag{3.15}$$

Where:

- $combination \in Combinationmodules$ is a combination module in the set of available combination modules. This step is optional, and it if is not included then the common models are forwarded to the next subphase.
- $Facttable = \{facttablemodule_1, \ldots, facttablemodule_n\} \subseteq Facttablemodules$ is the set of fact table modules that is selected among the set of available fact table modules $Facttablemodules$.
- $Wrapup = \{(wrapupmodule_1, order_1), \ldots, (wrapupmodule_m, order_m)\}$ is similar to $Refinement$, with the exception of utilizing $\{wrapupmodule_1, \ldots, wrapupmodule_m\} \subseteq Wrapupmodules$ instead.

3.5.2 Module Dependencies

As shown in Equation 3.14, the execution order of the modules is of significance. To explain how we handle module dependencies we examine how the modules `Find Relationships From Names` and `Cardinality` interact together. To briefly re-iterate these two modules, `Find Relationships From Names` attempts to find relationships between tables based on the names of the columns they contain, and `Cardinality` attempts to infer the cardinality of the relationships between tables. These two modules works well together when working with CSV files since no relationships exist in CSV files are defined, so the relationships has to be inferred first, followed by the cardinality of those relationships.

One could be tempted to define dependencies directly between the modules. Using the previous example, one would then define that `Cardinality` is dependent on `Find Relationships From Names`. However, using this approach one would have to also include dependencies on new modules in existing modules. Imagine a new module `Find Relationships From Data` (which uses a different method to find relationships with), this would then require `Cardinality` to also depend on `Find Relationships From Data`. Using this method for defining dependencies would require all authors of modules to continuously update their modules as new modules are created. Thus it becomes an inconvenience and a source of out-of-date dependencies if not maintained.

Instead of modules being dependent on other modules, it has rather been chosen that modules can require a set of information to be available before it is able to execute properly. This means that since module `Cardinality` requires information about `relationships`, another module has to output `relationships` before module `Cardinality` can execute properly.

A *dependency* for a module *mod* is expressed through Equation 3.16:

$$dependency(mod) = (Requires, Provides) \tag{3.16}$$

Where *Requires* is the set of *properties* that is needed before *mod* can execute properly, and *Provides* is the set of properties that *mod* adds or alters. The properties that a module uses depends on the phase that the module resides in. Currently only two phases uses dependencies: Refinement phase and the wrap-up subphase in the star phase. The rest does not rely on properties from previous modules in the same phase, so dependencies are not needed. Equation 3.17 and 3.18 define the available properties, that one can define a dependency on.

$$\begin{aligned} Properties_{refinementmodule} = \{&columnnames, datatypes, notnulls, uniques, primarykeys, \\ &relationships, cardinalities\} \end{aligned} \tag{3.17}$$

$$Properties_{wrapupmodule} = \{measures, dimensions\} \tag{3.18}$$

$dependency(mod)$ is *fulfilled* if everything that *mod* requires is provided before the execution of *mod*. We define the modules *othermodules* (or *om* for short) that needs to provide properties to *mod* as:

$$othermodules = \{om | om.Provides \subseteq mod.Requires \wedge mod \text{ and } om \text{ are modules from the same (sub)phase}\}$$

Formally, *mod* is defined to be fulfilled if Equation 3.19 is true:

$$\begin{gathered} mod.Requires \subseteq \bigcup_{om \in othermodules} (om.Provides) \text{ and} \\ \forall om \in othermodules\ mod.order > om.order \end{gathered} \tag{3.19}$$

Fulfilled Module Dependencies

As an example, consider Figure 7 which shows modules and the relationship between them in terms of dependencies.

Within this example, the following dependencies exist:

- $dependency(Find\ Relationships\ From\ Names) = (\emptyset, \{relationships\})$
- $dependency(Cardinality) = (\{relationships\}, \{cardinalities\})$

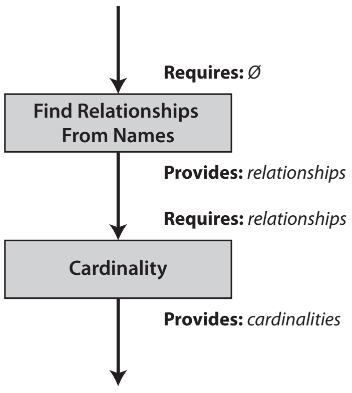


Figure 7: Example of modules and their dependencies.

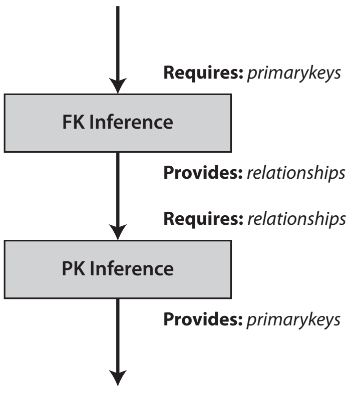


Figure 8: Example of a cyclic dependency between modules.

In this scenario `Find Relationships From Names` needs to be run before `Cardinality`, as `Find Relationships From Names` provides `relationships` while `Cardinality` requires it.

For some scenarios, it is possible to reorder the modules such that all modules are fulfilled. While the user was able to determine the order of execution for the modules, if the chosen order does not fulfill all modules then a reordering is suggested.

Unfulfilled Module Dependencies

However, there are also scenarios where it is impossible to resolve the unfulfilled dependencies. This can occur when module dependencies form a cycle. Figure 8 provides a simple fictitious example of such a scenario.

In this scenario the following dependencies exist:

- $dependency(FK\ Inference) = (\{primarykeys\}, \{relationships\})$
- $dependency(PK\ Inference) = (\{relationships\}, \{primarykeys\})$

Regardless of how these two modules are ordered, they cannot both be fulfilled simultaneously: `FK Inference` as depicted in Figure 8 is not fulfilled, as it requires `primarykeys`, but is not provided it. While this would be resolved for `FK Inference` by swapping the two modules such that `PK Inference` is now first, this would result in another module dependency not being fulfilled, as `PK Inference` is now not being provided with `relationships`.

To create a correct ordering of modules, there cannot exist cyclic dependencies. If cyclic dependencies exists, we attempt to fulfill as many modules as possible. This is further described in Section 4.3.2.

Handling Unfulfilled Dependencies

One design aspect to consider is how to handle unfulfilled dependencies. As mentioned before, some cases can be resolved by reordering the modules, however, when this is not a possibility, a warning is given to the user that problems can occur here, but otherwise continue on. If a value is missing (e.g. no column names are found), then the user can be involved to provide the missing value(s).

3.6 Star Model

The *star model* is first encountered during the star phase. The star phase generates a list of candidate star models which each represents a star schema.

The star model is defined as in Equation 3.20:

$$starmodel = (commonmodel, facttable, Dimensions) \tag{3.20}$$

Where *commonmodel* is the common model that the star model was created from. $facttable$ and each $dimension \in Dimensions$ are both represented as a $startable$, defined in Equation 3.21.

$$startable = (name, Starcolumns, table, confidence, Relationships, constraints, type) \tag{3.21}$$

Where:

- *name* is the name of the table.
- *Starcolumns* is the set of columns in the table and each $starcolumn \in Starcolumns$ is defined in Equation 3.22.
- *table* is the table in the common model that the star table originates from.
- $confidence$ is the confidence that the table is a fact table. The higher the confidence, the more likely it is a being a fact table.
- *Relationships* and $constraints$ is identical to those defined in Equation 3.5 and 3.4, respectively. $constraints$ also inherits the definitions of *Primarykeys*, *Notnulls*, and $Uniques$. The confidence is not used by anything in the star model and is therefore always set to the value of 1.
- $type \in \{fact\,table, dimension, date\,dimension, time\,dimension, junk\,dimension\}$ describes the type a star table represents. A fact table in $startable$ must have the $type\ fact\ table$, likewise a dimension can not have the $type\ fact\ table$.

As a way to easen the work in the generation phase, we introduce some variations of a *dimension*, which help to distinguish different types of dimensions from each other. The ones that we use are: *date dimension*, *time dimension*, and *junk dimension*. The definition for each of these is identical to that of *dimension*, however, when implementing DWStar, it is easy to check for dimensions that matches the type of these variations and apply additional processing to these.

Both facts tables and dimensions can originate from either a single table or a set of tables, which is not visible in the star model. This is because our process consists of a combination subphase in the Star Phase which can combine tables from a common model and create an alternative common model. This essentially tries to improve upon the model and Section 3.7.1 describes this further and provides reasoning for why this is useful is some scenarios.

$starcolumn \in Starcolumns$ is defined as in Equation 3.22.

$$starcolumn = (ordinal, name, datatype, column, measuretype, issurrogate) \tag{3.22}$$

Where:

- $ordinal \in \mathbb{N}_1$ is the star column's position in the star table.
- *name* is the name of the star column.
- *datatype* is the data type (as defined in Equation 3.3) of the star column.
- *column* is the column from the common model, that the star column is based on. If the star column is not based anything, $column = \varepsilon$.
- $measuretype \in \{numeric, descriptive, \varepsilon\}$ is used to categorize the columns. ε indicates that it is not a measure.
- $issurrogate \in \{true, false\}$ describes if the column is a surrogate column, and therefore does not originate from a column in the common model. The values for this is generated by the resulting ETL processes.

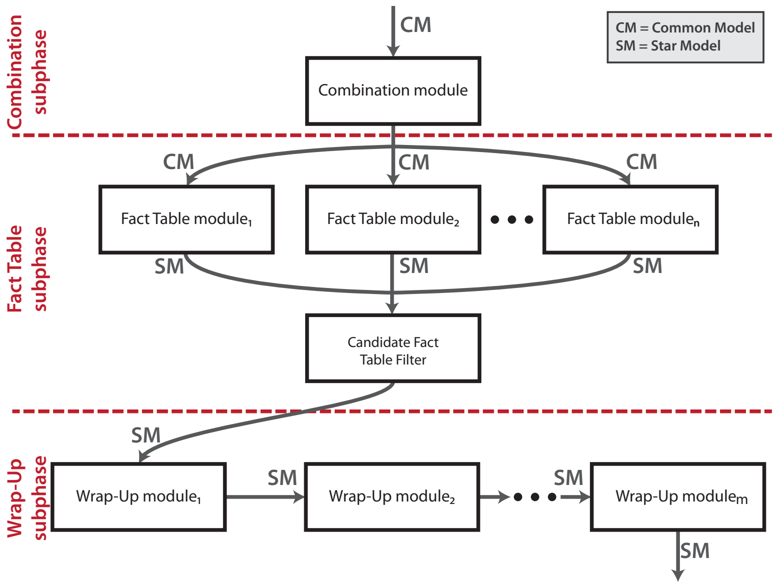


Figure 9: Depiction of the three subphases in the star phase.

3.7 Star Phase

The star phase is a central phase which is responsible for taking in common models and then generate a set of candidate star schemas (star models). This section first describes each subphase of the star phase. Secondly, a concrete scenario of how to make a star schema is given. This is followed by a description of the modules used to derive the star models.

3.7.1 Generation of Star Models

The star phase is separated up into three subphases which are illustrated in Figure 9. The star phase starts at the top of Figure 9 and follows the arrows down.

- *Combination subphase:* Combines two or more tables in the attempt to create a better fact table.
- *Fact Table subphase:* Each module receives a common model and provides a confidence on each table in the common model, detailing how likely the table is to be a fact table. Finally, the Candidate Fact Table Filter combines the confidences of all star models.
- *Wrap-Up subphase:* Finds the dimensions appropriate for a given fact table.

This process is repeated until all common models has been through these subphases.

Combination Subphase

This first subphase is an optional one in which the user can specify if he would like to attempt creating alternative common models which include combinations of tables. This phase receives a common model and can output multiple common models, based on the result of the module in this subphase. If the combination module results in multiple common models, it would return them one by one, instead of delivering a list of common models to the fact table modules. If the user has not picked a module for this subphase, this subphase is ignored and the common model from the previous phase is sent directly to the next subphase (the fact table subphase).

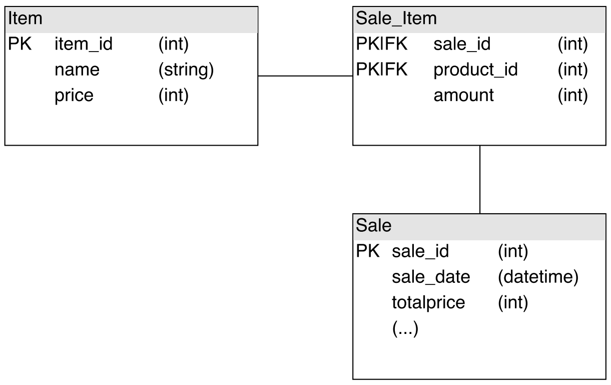


Figure 10: Illustration of a common model that contains three tables.

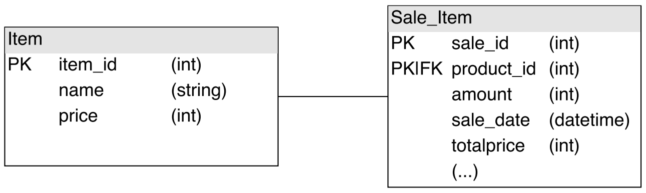


Figure 11: Illustration of a common model where two tables have been combined into one.

The combination subphase is included, as we have seen scenarios where the star schema has been improved by joining together at least two tables. Only a single module can be chosen to do the combinations on common models. If the user needs to select multiple combination modules, he is then encouraged to implement a new combination module, that fits his requirements.

The goal of this subphase is to combine tables in the common model and thereby generate combinations of tables that are potentially more likely to be fact tables than the non-combined tables in the common model. To illustrate this point, Figure 10 shows a snippet of a common model that is improved by combining two tables. Figure 10 contains a `Sale` table with a `many-to-one` relationship to a `Sale_Item` table, this relationship is very common since a single sale can involve multiple sale items, the `Sale_Item` table contains foreign key references to both `Sale` and `Item`, along with a numeric column, `amount`, describing how many of that specific item is purchased. We argue that the DW benefits most from combining `Sale` and `Sale_Item` and thereby detailing how many of a product is in a sale, this also allows `Sale` to have a `many-to-one` connection with `Item` which can prove beneficial when recognizing dimension tables.

The output of this phase is the original common model along with the newly created common model with the combined `Sale` and `Sale_Item` tables, as shown in Figure 11. These can then be evaluated by the fact table subphase.

Fact Table Subphase

The next subphase consists of star modules that attempts to find potential fact tables. As shown in Figure 9, each of the chosen star modules run in parallel, as they each work independently on every table in the common model. Each star module takes in a common model and for each table in the common model creates a star table with an applied confidence, which describes how confident that module is in that the given table is the fact table. Each module therefore returns a star model containing a single fact

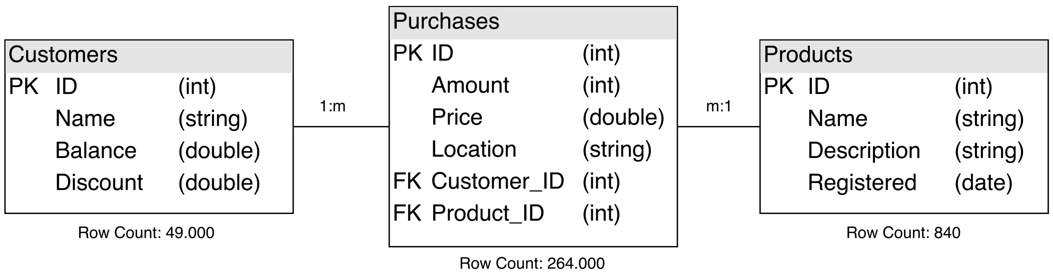


Figure 12: Example of the tables which together makes a common model.

table for each table in the common model. The star models of each star module are then forwarded to the *Candidate Fact Table Filter* (or *Candidate Filter* for short) which is responsible for calculating the average confidence of each fact table. The Candidate Filter is always included and the algorithm can therefore not be changed, as it has the specific goal of creating a list of candidate star models that will be forwarded to the next phase. The Candidate Filter has a user-defined threshold θ which define the minimum confidence a candidate fact table must have to be included in the following subphase. Having θ reduces the number of star models to consider, and thereby does not overwhelm the user when choosing between the finished star models.

The reasoning behind the Candidate Filter is that the heuristics provided to identify fact tables are often too simplistic on their own, but in combination they are more likely to find fact tables correctly. By allowing the user to choose multiple star modules whose confidences are combined, it gives the opportunity for the user to get a more thoroughly considered candidate for a fact table instead of only relying on a single heuristic. The user still has the possibility of only choosing a single star module for finding fact table candidates. In that case, the average confidence is trivially found by the Candidate Filter.

Example 1. *Figure 12 shows an example containing three tables in a common model. In our configuration we select two star modules:* `Fact More Columns` *and* `Fact More Rows` *where the first assigns confidence by looking at the ratio between the number of numeric and descriptive columns and the second looks at the number of rows in the table.*

In this example, `Fact More Columns` *gives table* `Customers`*,* `Purchases`*, and* `Products` *a confidence of* 0.66*,* 0.66*, and* 0*, respectively.* `Fact More Rows` *gives confidences of* 0.16*,* 0.84*, and* 0.003 *(based on the ratio between total amount of rows and rows in table) to these tables. The confidences provided are not normalized since each table is evaluated separately.*

The Candidate Filter finds the average confidence for each table to be: 0.41 *for* `Customers`*,* 0.75 *for* `Purchases`*, and* 0.0 *for* `Products`*. If $\theta = 0.4$ in this case, then only* `Customers` *and* `Purchases` *have a confidence that is over θ, and only these two tables are forwarded to the last subphase.*

Wrap-Up Subphase

The final subphase of the star phase is the *Wrap-Up Subphase* which continues on with all fact tables (star models) that passed the threshold θ. This subphase finds the remaining constituents of the star model such as dimensions and measures.

The user chooses the execution order of the modules in this subphase. As can be observed in Figure 9 for this subphase, the modules are executed serially. This is because each wrap-up module can provide properties to the star model that is then required for the next star module as with the process in the refinement phase. The dependency mechanism described in Section 3.5 is applied to this subphase, which helps describe the order of the modules. Here we can also recognize which of the wrap-up modules that can not be fulfilled, and whether a reordering of the star modules can remedy this. After applying more information to the star models, each star model if then forwarded to the star refinement phase.

After all wrap-up modules have been executed, each star model is forwarded to the star refinement phase.

Example 2. *In Example 1, both table `Customers` and `Purchases` was above the threshold and recognized as candidate fact tables. For each of these two star models, only the fact tables have been found, and the wrap-up subphase now finds the remaining constituents. This is done by executing the subphase using the following wrap-up module: `Flatten Dimensions`. The module `Flatten Dimensions` finds dimensions based on the assumption that if a many-to-one relationship exists between a fact table and another table, then that other table is a dimension. In this scenario, no dimensions are found for `Customers` whereas for `Purchases`, `Flatten Dimensions` finds two dimensions: `Customers` and `Products`.*

3.7.2 Modules

The star phase consists of three different types of modules, one type for each subphase. Each module is described with the following:

- A name of the module and the subphase it belongs to
- A description of the observations that lead to defining the module
- Sources and citations supporting the heuristic
- The action performed by the module

Combine Tables - Combination Subphase

Description: The reason behind the combining tables is already detailed in the Combination Subphase above.
Heuristic: This module looks explicitly for bridge tables which might contain important relationships and measures which are useful for the fact table. It is therefore deemed preferable to combine the bridge table with the fact table to encompass the required information.

This module has no sources that it drives inspiration from and is merely the product of a pattern found in the OSSs that were examined during the project. To recognize a bridge table we look at the relationships to other tables, as a bridge table typically consists of two many-to-one relationships with other tables, contains no descriptive columns, and has very few (or zero) numeric columns.

Imagine that we have a common model containing three tables: A, B, and C. Where B is a bridge table with two many-to-one relationships, one to A and one to C. The output of this module is three common models one for each relationship with B, and the original one. For the first common model, the module combines A and B which results in a common model containing the tables AB and C where AB has a many-to-one relationship with C. For the second common model, the module combines B and C which results in a common model containing the tables A and BC where BC has a many-to-one relationship with A. Finally the last common model is identical to the common model that was given to the module. When the combination module is finished, the fact table modules each retrieve (in this example) three common models: The original containing all three tables, the first common model created by this module, and the second common model created by this module.

Fact More Rows - Fact Table Subphase

Description: A table is said to be more likely to be a fact table if it contains more data than the other tables. This observation is stated by both Kimball and Ross, M. Jensen et al., and C. Jensen et al.
Sources:

- *"Since dimension tables typically are geometrically smaller than fact tables [...]"* [6].
- *"[...] due to the size of the fact table (which is often several orders of magnitude larger than the other tables) [...]"* [7].
- *"[...] dimensions typically take up only 1–5% of the total storage required for a star schema [...]"* [26].

Heuristic: The confidence of a table being a fact table directly relates to the ratio between the total amount of rows in the common model and the amount of rows in the table. Equation 3.23 shows how a confidence is calculated for each table.

$$confidence_{morerows}(table) = \frac{rowcount(table)}{\sum_{t \in commonmodel} rowcount(t)} \tag{3.23}$$

The function $confidence_{morerows}$ takes in a table and accesses the related common model *commonmodel* in order to calculate the total number of rows in all the tables in the common model. The confidence indicates how large of a portion of rows that $table$ contributes with compared to the total amount of rows in the common model.

Example. *If we again look at the example in Figure 12 the total row count for all tables is* 313.840. *Using the* $confidence_{morerows}$ *function we see that* `Customers` *with* 49.000 *rows has a confidence of* 0.16*,* `Purchases` *with* 264.000 *rows and a confidence of* 0.84*, and finally* `Products` *with* 840 *rows and a confidence of* 0.003*. When comparing the confidences of each table, it is clear that Purchases is the best fact table candidate in this scenario.*

Fact More Columns - Fact Table subphase

Description: A large collection of numeric columns can indicate that a table is a fact table. However, given that most tables contains numeric attributes, it is possible to look into the ratio between the number of numeric columns contra the number of descriptive columns. This is in part supported by Phipps and Davis, and Kimball and Ross.

Sources:

- *"[...] numeric fields represent measures of potential interest to a business and the more numeric fields in an entity the more likely it is that the entity is an event or fact."* [8].
- *"Almost all text in a data warehouse is descriptive text in dimension tables."*[30]

Heuristic: For a table $t \in commonmodel$ the number of numeric and descriptive columns can be calculated as in Equation 3.24 and 3.25.

$$\begin{aligned} numeric(table) = |\{c | c \in table.Columns \wedge HighestConfidence(c.Datatypes) \text{ is numeric} \\ \wedge c \text{ is not primary key nor foreign key}\}| \end{aligned} \tag{3.24}$$

$$\begin{aligned} descriptive(table) = |\{c | c \in table.Columns \wedge HighestConfidence(c.Datatypes) \text{ is not numeric} \\ \wedge c \text{ is not primary key nor foreign key}\}| \end{aligned} \tag{3.25}$$

Both *numeric* and *descriptive* use the data type of a column to categorize whether it is numeric or not. We currently identify a column as numeric if the data type represents any number. Any other data type is categorized as descriptive besides a primary key or a foreign key which is neither considered a numeric nor a descriptive column. Using these two functions, we can calculate the ratio between the numeric columns and the total amount of numeric and descriptive columns using Equation 3.26. In cases where the sum of $numeric(table)$ and $descriptive(table)$ equals zero, we will return a ratio of zero to avoid zero division.

$$confidence_{ratio}(table) = \frac{numeric(table)}{numeric(table) + descriptive(table)} \tag{3.26}$$

The ratio between numeric columns and descriptive columns are indicative of whether the table is considered a fact table or dimension table. If the ratio favors numeric columns, the table is more likely a fact table.

Example. *In Figure 12, `Customers` and `Purchases` each has two numeric columns and one descriptive, providing them each with a confidence of* $0.\overline{66}$*. `Products` has zero numeric and three descriptive, providing it with a confidence of* 0*. One might argue that `Customers` typically contains many more columns that are descriptive, but for this example, both `Customers` and `Purchases` are equally likely a fact table.*

Usually many-to-one - Fact Table subphase

Description: The relationship between a fact table and a dimension table has been observed to commonly be many-to-one, and can therefore be used to identify fact tables. This has been mentioned by both Song et al. as well as Romero and Abelló.
Sources:

- *"It has been observed that there usually exists a many-to-one (M:1) relationship between a fact and a dimension."* [3].
- *"[...] a dimensional concept is related to a fact by a one-to-many relationship."* [10].

Heuristic: Song et al. proposed an approach for calculating the confidence of table based on both direct and indirect relationships. Our heuristic is more simplistic and currently only identifies fact tables by looking at their direct relationships. The resulting confidence is statically provided:

$$confidence_{Manyone}(table) = \begin{cases} 0.9, & \text{if } table \text{ has more than two many-to-one outgoing relationships} \\ 0.7, & \text{if } table \text{ has two many-to-one outgoing relationships} \\ 0, & \text{if } table \text{ has less than two many-to-one outgoing relationships} \end{cases} \tag{3.27}$$

Example. *In Figure 12, `Customers` and `Products` each have zero many-to-one relationships, thereby providing them with a confidence of* 0*. `Purchases` on the other hand has two many-to-one relationships, providing it with a confidence of* 0.7*, and recognizing it as the most likely fact table.*

Flatten Dimensions - Wrap-Up subphase

Description: As the star model is defined, it only allows for dimensions to have a direct reference to the fact table. However, scenarios exist where a dimensional table is part of a larger dimension, which is also known as a dimension hierarchy. For instance, a `Product` table is part of a `Product Category` or a `Location` is part of a `Country`. These kinds of hierarchies can be combined to form a flattened dimension. This is mentioned by Zhengxin.
Sources:

- *"Classification entities are entities related to component entities by a chain of one-to-many relationships; they are functionally dependent on a compone[n]t entity (directly or transitively). Classification entities represent hierarchies embedded in the data model, which may be collapsed into component entities to form dimension tables in a star schema."* [31]

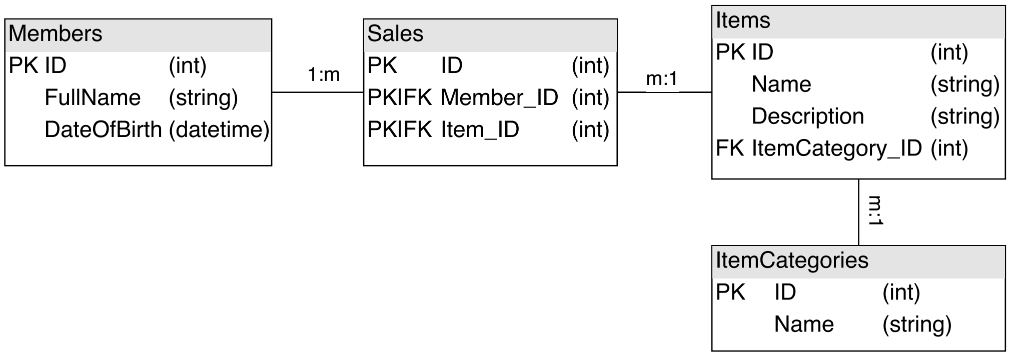


Figure 13: A (simplified) OSS before flattening potential dimensions.

Heuristic: The heuristic makes it possible to both: Identity relevant dimension tables to a given fact table, and find and flatten the hierarchies of those dimensions.

immdims as defined in Equation 3.28 finds immediate dimensions, which are the dimension which refers directly to the given table t.

$$\begin{aligned} immdims(t) = \{&t_{dim} \in commonmodel | t \text{ refers to } t_{dim} \\ &\wedge \text{ the cardinality of the relationship is } \textit{many-to-one}\} \end{aligned} \tag{3.28}$$

For each table found with *immdims* we apply the $flatten$ function which is defined in Equation 3.29.

$$flatten(t) = \begin{cases} t, & \text{if } immdims(t) = \emptyset \\ \text{combine } t \text{ with } (\bigcup_{o \in immdims(t)} flatten(o)), & \text{otherwise} \end{cases} \tag{3.29}$$

The central part of this module is the $flatten$ function which recursively calls itself until it can no longer find any dimension table in the hierarchy. The first case is the base case in which *immdims* can no longer find any dimension tables. The 2nd case contains the recursive call, which combines reachable tables from t through *immdims*.

Given that t_{fact} is the table found by the fact table subphase, all of the immediate dimensions can be found by simply using *immDims* as in Equation 3.30.

$$startdims(t_{fact}) = immdims(t_{fact}) \tag{3.30}$$

The immediate dimension can then be expanded upon by Equation 3.31 which creates the flattened dimensions.

$$newdims(t_{fact}) = \bigcup_{startdim \in startdims(t_{fact})} flatten(startdim) \tag{3.31}$$

The result of the function in Equation 3.31 can now be used as dimensions in the star model.

Example. *An example scenario can be seen in Figure 13. Figure 14 shows the result of flattening potential dimensions. Both `Members`, and `Items` were first identified as immediate dimensions. Afterwards, `Items` were then flattened to the `Items_ItemCategories` dimension table seen in Figure 14.*

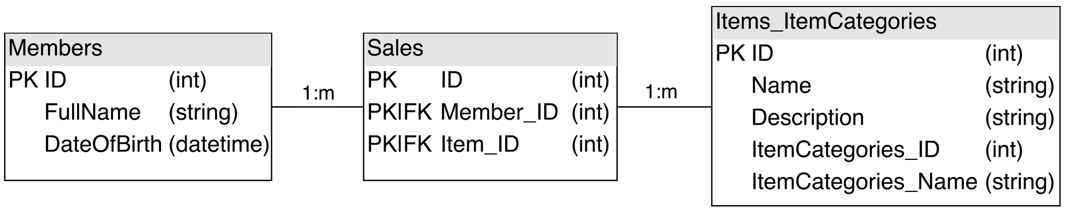


Figure 14: Schema after potential dimensions have been discovered and flattened from Figure 13.

3.8 Star Refinement Phase

The star refinement phase is responsible for the fine-tuning of the star models before they are translated to a physical format. These alterations are placed in the star refinement phase, as they operate on an existing star model, and creating a separate phase for this ensures that they are executed after a star model has a fact table and dimension tables. First, Section 3.8.1 examines the star refinement modules available.

3.8.1 Modules

The following describes the star refinement modules that are implemented.

Adhering to Naming Convention

Companies might define a naming convention for various parts of their star schema. Some conventions could for example be:

- **Fact table name:** Pascal case and prepend name with `Fact_`. E.g. `Fact_Sale`.
- **Dimension name:** Pascal case and prepend name with `Dim_`. E.g. `Dim_Customer`.
- **Column name:** Lower case. E.g. `productkey`.

For simplicity's sake, we only feature three basic operations: Change case of names, prepend text, and append text. These operations can be performed on three different elements: Fact table names, dimension names, and column names. If the user then wishes to define a specific naming convention for things such as primary keys, and foreign keys, he would then, as with the other phases, have to implement his own module that fulfills his objectives.

The user is able to specify the prefered naming convention in a settings file, which is loaded by the module when it executes. The details of how this settings file is specified is shown in Section 4.2.5.

Extending Data Type Length

During discussions with a BI practitioner, it was found that it is common practice to extend the length (including precision and scale) of the data type to make the DW long lasting if larger values are to be inserted later on. The data type of the columns is altered to extend the length, which is specified by assigning $length = length + extendLength$ where $extendLength$ is a value that is specified by the user in his settings, that describes how much the length should be extended with, similar values exists for precision and scale. This is applied to textual and numerical data types.

Add Surrogate Keys

It is common practice to add a surrogate key to dimensions. This module therefore iterate through all available dimensions in a star model, adds a new column called `SurKey` to the table in that dimension, and

Table 4: Example of values in the date dimension.

SurKey	**FullDate**	**Day**	**NameOfDay**	**Week**	**Month**	**NameOfMonth**	**Year**	**IsHoliday**	**Holiday**
5272	02/06/2017	2	Friday	22	6	June	2017	false	NULL
5842	24/12/2017	24	Monday	51	12	December	2017	true	Christmas

Table 5: Example of values in the time dimension.

Surkey	**Hour**	**Minute**	**Second**
48600	13	30	00
43200	12	00	00

adds a constraint specifying that the new column is a surrogate key by assigning *column.isSurrogate = true*. The generation module `Create Script Generation` (described in Section 4.6.1) handles the creation of new surrogate key columns.

Add Date And Time Dimensions

It is often seen, that date and time dimensions are associated with a fact table. Examples include: When an item was purchased, when it was shipped, and when it was delivered. However, for a DW, it is often more convenient to analyze data, by having dimensions dedicated to the temporal aspect of a fact.

According to Kimball and Ross [6], when creating a dimension for storing information about date and time, it is better to create two dimensions, one for the date and one for the time. The reasoning behind is that *"Date and time are almost completely independent."* [6]. We also see that each day of the year would fill 365 rows (except when leap year), and all the minutes in a day would fill 1,440 rows. If we only use one dimension for both date and time, that would become 525,600 rows per year. That number would further increase if we worked with a granularity of seconds instead of minutes. We therefore construct two separate dimensions to contain the temporal values.

What this module includes in the date and time dimensions as well as some sample data can be seen in Table 4 and 5, respectively. The rows in the tables shows what values that the timestamps `02/06/2017 13:30:00` and `24/12/2017 12:00:00` would create a reference to. The date dimension contains various representations of the timestamps, which helps an analyst extract the data that he needs. E.g. if he want to extract sales data per week, he would simply use the `Week` column. If that column was not available, he would have to find an SQL function that could transform a timestamp into the week it represents, which would not be desirable. Kimball and Rose [30] suggests a date dimension with twenty eight column, we have chosen a smaller variant here for simplicity's sake.

The construction of these dimension tables in the star model will happen in this module, by adding two tables to *Dimensions* in *starModel*. However, creating the two tables in a database and populating them is deferred to the generation phase.

Add Junk Dimension

As was mentioned previously, a fact table is considerably larger than dimensions. As a result of this, one would want to ensure that each row in the fact table is as small as possible. A fact table should therefore not contain descriptive values or values that are repeated multiple times. Kimball and Ross state that: *"[...] you don't want to store bulky descriptors on the fact row, which would cause the table to swell alarmingly."* [30]. They therefore encourage the use of *junk dimensions* for the storage of bulky descriptors and duplicated values. We construct a junk dimension when a column in the fact table:

1. Is not a foreign key

2. Is not a primary key
3. Has a data type that indicates a descriptive value. E.g. `string` and `boolean`.

These columns are moved to the new junk dimension and a relationship between the fact table and junk dimension is created.

Order Columns

The order of columns does normally not affect a table, however, it can become hard to get an overview when the columns are randomly ordered. To help the user find a column in a table, we firstly place all primary key columns first in the table, and then place the remaining columns in alphabetical order. The effect of a reordering is visible in *column.ordinal*, as this value describes the position of the column in the table.

3.9 Generation Phase

Up until now, all phases have only been concerned with generating common models or star models, however these theoretical models does not allow users to query data or analyze it. The generation phase is the final phase and is responsible for translating star model(s) from the previous phases into physical formats. We currently implement modules that generates:

- SQL scripts for creating a star schema in a relational database.
- ETL processes for populating the data warehouse.

Each generation module is provided with a star model and a location in the file system where it can store files containing the result, e.g. a text file with SQL scripts, or a file containing an ETL process. The user can then review the results and manually execute the SQL scripts on the appropriate database. Alternatively, the user could implement his own generation module, which automatically connects to the database and executes the scripts.

Multiple modules can exist for creating physical representations of the same star model. For example, one module could generate SQL scripts for SQL Server [23], and another for PostgreSQL [32]. The user can therefore select the generation module, which best suits his setup.

In this phase, multiple generation modules can be chosen to simultaneously generate different outputs. The currently available generation modules are: The `Create Script Generation`, which translates a star model into a schema for SQL Server, and the `DW Population`, which creates a series of SQL scripts for SQL Server, that populates the dimensions and fact table. These modules contains several implementation details and are therefore further described in Section 4.6.1 and 4.6.2, respectively.

Chapter 4

Implementation

This chapter goes into the implementation details of DWStar which are not covered in Chapter 3. As such, this chapter is closely related to the concepts presented in Chapter 3. Notably, both the star phase and star refinement phase are not included in this chapter since their implementation closely resembles the description given in the previous chapter.

DWStar is realized through an implementation in the programming language C#. The program is spread across 149 classes, which collectively contains roughly 7500 lines of code. Beyond implementing the concepts presented previously, a GUI has been constructed to ease user interaction. The GUI is implemented using Windows Presentation Foundation (WPF) [33] and the GUI library Material Design In XAML [34].

Section 4.1 describes how the common model definitions of *sample*, *rowcount*, and *datatype* are implemented. Section 4.2 shows how modules are implemented, loaded, and configured. Section 4.3 examines configurations further by seeing how dependency resolution is implemented. Section 4.4 describes how the `CSV Metadata` metadata module retrieve information, such as data types, headers, etc. Section 4.5 describes refinement modules which have implementation details, but also more details on how common models are grouped. Section 4.6 describes the two generation modules in more detail. Finally, Section 4.7 gives an introduction to the user interface, and what the user is capable of in DWStar.

4.1 Common Model

While most details have been described for the common model in Section 3.2, there are some implementation details which have not been described yet. Specifically, the functions *sample* and *rowcount* are explained further in this section as these differ between RDBMSs and CSV files, and the implementation of the data types used in the common model are explained further.

4.1.1 Sample Function

Recall that $sample : Columns \times n \rightarrow Results$ finds n rows of the columns in $Columns$. A sample function is implemented for each type of OSS. For a database, a sample is extracted by establishing a connection to the RDBMS and selecting the top n rows. If the OSS is a CSV file, the first n lines of that file is extracted. The connection string used to establish a connection to a data source is stored in $table.database$ and can be used by the sample functions. The *sample* function is implemented as part of a metadata module, and is therefore always available for any supported type of OSS.

We choose to return the first n rows as it less efficient to select n rows at random from the OSSs. However, selecting rows at random could prove beneficial to broaden the variety of data. Selecting n

rows at random can trivially be achieved for most RDBMS, e.g., in SQL Server this can be accomplished by using the T-SQL query in Listing 4.1, which uses `NEWID()` to assign a random order to the rows. In this listing, *tablename* is the name of the table containing *Columns*. This is also trivially accomplished with CSV files.

Listing 4.1: SQL Statement for a random sample of data.

```
SELECT TOP 1000 Columns FROM tablename ORDER BY NEWID()
```

4.1.2 Row Count Function

Recall that $rowcount : t \to \mathbb{N}_0$ takes in a table t and returns the number of rows in t. As with *sample*, the *rowcount* function is defined for each type of OSS.

Working with databases (in this case SQL Server), row count for table t is trivially found with the T-SQL statement in Listing 4.2.

Listing 4.2: T-SQL Statement for finding row count in relational database.

```
SELECT COUNT(*) FROM [t.schema].[t.name];
```

Where $t.schema$ is the table schema name and $t.name$ is the table name.

For finding amount of rows in table t for a CSV file, Algorithm 1 is used.

Algorithm 1: *rowcount* implementation for CSV files.

```
   Data: $file$ that contains $t$
   Result: $rowcount$
 1 $rowcount = 0$
 2 if $size\ of\ file \leq 100\ MB$ then
 3     while $file\ is\ not\ at\ EOF$ do
 4         getNextLine($file$)
 5         $rowcount = rowcount + 1$
 6     end
 7 else
 8     $rows$ = sample($t.Columns$, 1000)
 9     $sizePerLine = \frac{getSize(rows)}{|rows|}$
10     $rowcount = \lfloor \frac{size\ of\ file}{sizePerLine} \rfloor$
11 end
12 return $rowcount$
```

The input of the algorithm is the CSV file $file$ that contains the table t. The output will either be the number of rows of that file, or a approximation if the file is considered too large, this limit is arbitrarily set to be 100 `MB`.

Line 1 initializes the number of currently found rows to 0. Line 2 finds out whether the CSV file is less than or equal to 100 `MB` (the size of $file$ is determined by the file system). If that is the case, the precise number of rows is extracted as seen in Line 3 which iterates until the end of the file is reached. Lines 4-5 increment *rowcount* every time a line is extracted.

Line 7 is the start of the other case where there is more than 100 `MB` of data. Due to interest of making DWStar execute in a timely manner, only an approximate row count is given. This choice seems reasonable as neither our modules nor any heuristic of the related work requires a precise row count. Line 8 uses the *sample* function to get a list of rows out. Line 9 starts the approximation by finding out how many bytes are required to store the value of 1000 rows, by using the function getSize which is then divided by $|rows|$ to get the average size of a line. In Line 10 the size of the entire CSV file is found which is then divided by $fileSizePerLine$ to get an approximate number of rows for the entire file. Finally, Line 12 returns *rowcount* for both cases.

4.1.3 Specification of Data Types

Data types were previously introduced in the common model with $type \in Types$ in Section 3.2, but the specification of *Types* was delegated to the implementation.

We have chosen to use the data types in OLE DB [35]. OLE DB data types were chosen primarily because multiple database vendors provide a list of how their specific data types can be mapped to OLE DB data types. The work of determining which data type map to a specific OLE DB data type is therefore already done, as with SQL Server mappings [36] and ANSI SQL mappings [37]. A metadata module just has to map the specific data type to the relevant OLE DB data type for it to be able to translate the data type of the specific system to the common model.

4.2 Modules

This section describes how modules from the different phases are implemented, and how the user is able to implement his own modules.

Figure 15 shows an UML class diagram of the components that are used to implement modules from the various phases. The classes in the middle represents types of modules that each contribute to different phases in DWStar, where IModule and IDependent are interfaces. The following sections examines the components in the UML diagram.

4.2.1 Module Interface

As previously described, DWStar is customizable through modules, that can either be implemented by the user or delivered together with the implementation. Common for modules in the implementation is, that they all implement the interface IModule. Through this interface, each module expose the two properties Name and Description, which ought to shortly describe the functionality of a module.

4.2.2 Dependent Interface

Classes that implements IDependent declares dependencies (as defined in Section 3.5.2) through the properties Requires and Provides.

Requires and Provides are each represented as the data type short. However, we do not use the numerical value directly, but use each bit to represent a property that it requires or provides. An example of this can be seen in Table 6, where the first six bits of the sixteen bit short is displayed. A value of 1 shows that the class requires the property where 0 represents that it does not. Here we see that a module mod_1 requires both the column name and data type values. If we then have another module mod_2, that does not require anything and provides the properties shown in Table 7, we can see that mod_2 provides some of the properties that mod_1 requires. It would therefore be beneficial to execute mod_2 before mod_1. The benefit of using the binary representation of the short instead of the actual value is, that the bitwise operators can easily illustrate if any module provides some of the dependencies that another module requires. E.g. when applying the bitwise AND operator on the values 110000 and 010000 we would get a value that is greater than zero, meaning that the first module depends on the second. If the value is zero, we know that the first module does not depend on the second module.

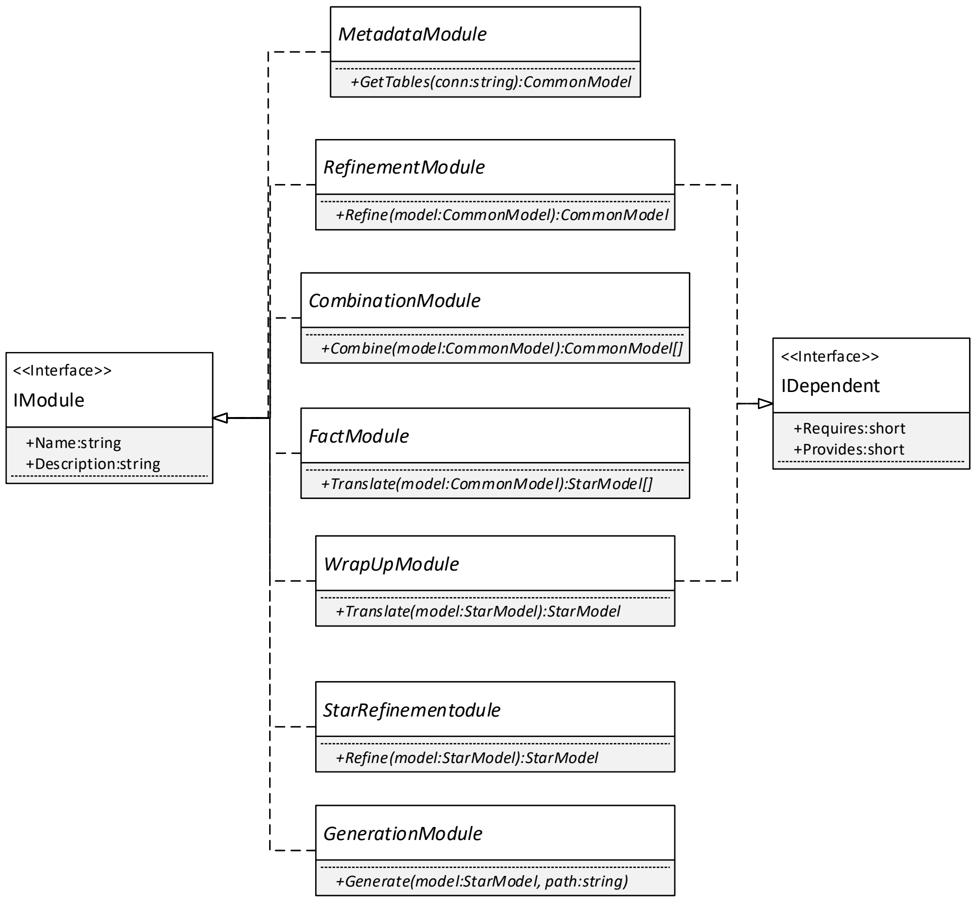

MetadataModule
+GetTables(conn:string):CommonModel
RefinementModule
+Refine(model:CommonModel):CommonModel
CombinationModule
+Combine(model:CommonModel):CommonModel[]
FactModule
+Translate(model:CommonModel):StarModel[]
WrapUpModule
+Translate(model:StarModel):StarModel
StarRefinementodule
+Refine(model:StarModel):StarModel
GenerationModule
+Generate(model:StarModel, path:string)
<<Interface>>
IModule
+Name:string
+Description:string
<<Interface>>
IDependent
+Requires:short
+Provides:short

Figure 15: UML Class diagram involving classes and interfaces relevant to modules.

Table 6: *Requires* from mod_1.

columnnames	**datatypes**	**notnulls**	**uniques**	**primarykeys**	**relationships**
1	1	0	0	0	0

Table 7: *Provides* from mod_2.

columnnames	**datatypes**	**notnulls**	**uniques**	**primarykeys**	**relationships**
0	1	0	0	0	0

4.2.3 Module Implementation Example

When implementing a module, the user has to determine which phase the module belongs to. In this example, we show the `Cardinality` module which is responsible for finding the cardinality of relationships between tables. This is implemented by creating a new C# class, which extends the abstract class `RefinementModule` and implements the properties and methods that are defined for the superclass. This implementation would then have to be compiled as a Dynamic-Link Library (DLL) and stored in a location where DWStar can locate it. If a user was to create his own module, he would have to go through the same process. Listing 4.3 shows how the `Cardinality` module could be implemented.

Listing 4.3: Shortened `Cardinality` module implementation.

```
 1  public class CardinalityModule : RefinementModule
 2  {
 3      public override string Name { get; } = "Cardinality";
 4      public override string Description { get; } = "Determines the cardinality of
            relationships.";
 5
 6      public override short Requires { get; } = CommonDependency.Relationships;
 7      public override short Provides { get; } = CommonDependency.Cardinalities;
 8
 9      public override CommonModel Refine(CommonModel commonModel)
10      {
11          foreach (Table table in commonModel.Tables)
12          {
13              foreach (Relationship relationship in table.Relationships)
14              {
15                  FindCardinality(relationship);
16              }
17          }
18
19          return commonModel;
20      }
21  }
```

Line 1 defines a new class that implements the abstract class `RefinementModule`. Lines 3 and 4 set the `Name` and `Description` properties from `IModule` interface. Lines 6 and 7 define which properties the module requires and provides from the `IDependent` interface. In this case, it requires the relationships between tables to be defined and provides the cardinality of those relationships. The method `Refine` on Line 9 is the essential part of a refinement module as it applies a series of transformations to the supplied common model and returns it. Lines 11 to 17 iterate through all relationships in the common model where Line 15 applies the method `FindCardinality` which finds the cardinality as defined in Section 3.4.1 for this `Cardinality` module. Line 1 defines a new class that implements the abstract class `RefinementModule`. Lines 3 and 4 set the `Name` and `Description` properties from `IModule` interface. Lines 6 and 7 define which properties the module requires and provides from the `IDependent` interface. In this case, it requires the relationships between tables to be defined and provides the cardinality of those relationships. The method `Refine` on Line 9 is the essential part of a refinement module as it applies a series of transformations to the supplied common model and returns it. Lines 11 to 17 iterate through all relationships in the common model where Line 15 applies the method `FindCardinality` which finds the cardinality as defined in Section 3.4.1 for this `Cardinality` module.

4.2.4 Loading Modules

With modules that can both originate from our implementation and from users, we have chosen to load user defined modules at runtime when the program is initializing. For modules to be recognized and loaded into DWStar, the following steps has to be carried out:

1. Find all DLLs in a predefined folder.

2. Load all classes from the DLL's into the currently running program.
3. Create a list of all modules, both the ones defined in the program and the external ones.
4. Create an instance of all the found modules.
5. Group them according to which phases they relate to.

When these simple steps has been performed, we have a list of modules for each phase that is ready to execute.

4.2.5 Configurability

It is undesirable to create a new module when the user wants to change simple things, such as the naming convention of columns.

For convenience, DWStar allows modules to expose *settings files*, which can then be modified by the user, thereby altering how the module behaves. These settings files are stored in JSON format. An example of such file is shown in Listing 4.4. In this listing, the settings file for the star refinement module `Naming Convention` is shown. It is for example specified, that the name of a fact table should be `Fact_%NAME%`, where `%NAME%` is a placeholder for the name of the table. This essentially prepends the table with the string `Fact_`. It is also possible to specify the casing of table and column names (in this case it is pascal casing) and to remove underscores from the names, i.e. `Customer_Orders` would become `CustomerOrders`.

Listing 4.4: Settings file for `Name Convention` module.

```
{
  "FactTableNameStructure": "Fact_%NAME%",
  "DimensionNameStructure": "Dim_%NAME%",
  "TableNameCasing": "PascalCase",
  "TableStripUnderscore": true,
  "ColumnNameStructure": "%NAME%",
  "ColumnNameCasing": "PascalCase",
  "ColumnStripUnderscore": true
}
```

The settings files is then loaded and used by the module when it is executed. The settings available to the user is defined by the module implementation.

4.3 Configuration

Section 3.5 described configurations as a concept. To restate briefly, a configuration holds all of the modules that are to be executed for each phase, including the execution order. Section 4.3.1 shows how configurations are persistently stored, and Section 4.3.2 describes the implementation behind rectifying unfulfilled dependencies.

4.3.1 Persistently Storing a Configuration

To avoid having the user repeatedly define a configuration whenever the program is restarted, DWStar serializes it and stores the result on the file system. The configuration is then loaded next time the program is started.

The definition of a configuration in Section 3.5 primarily consists of information about modules, however as we have implemented modules as classes in C#, it is problematic to save the configuration. As the modules are compiled code it is therefore difficult to store and load, and it would create duplicates of modules if multiple configurations exist. Instead we store the namespace of the class. E.g., if a user implemented the refinement module `Cardinality` he could place it in the namespace `com.company.modules.refinement` when creating a DLL file for his module. The stored value would then be the combination of the namespace and the class name (E.g. `com.company.modules.refinement.Cardinality`). The same approach is applied to all the other types of modules in the configuration.

4.3.2 Reordering Unfulfilled Dependencies

Section 3.5.2 defined dependencies and discussed them being fulfilled and reordering them if necessary. This section delves into Algorithm 2 for how unfulfilled dependencies are resolved (if possible). Essentially, it should ensure that any module that *requires* a certain property executes after another module that *provides* that property.

Algorithm 2: Constructing internal graph and reordering the modules to be fulfilled (if possible).

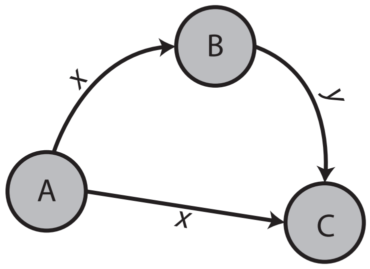
```
   Data: List of modules $Mods$
   Result: Ordering of modules
 1 $G =$
     A directed graph, where $V$ is the set of modules and $E$ is the set of dependencies between modules
 2 for $mod \in Mods$ do
 3     G.addVertex($mod$)
 4 end
 5 for $v \in G.V$ do
 6     for $req \in v.Requires$ do
 7         for $otherMod \in G.V \setminus v$ do
 8             if $req \subseteq otherMod.Provides$ then
 9                 Add a direct edge from $otherMod$ to $v$
10             end
11         end
12     end
13 end
14 vertices = DFS(G)
15 order = sort list vertices in descending order according to the finishing times given by DFS
16 return order
```

Dependencies that were presented for *dependencies* in Equation 3.16 are represented as a directed graph. Each vertex represents a module in the given phase, and each directed edge represents a dependency.

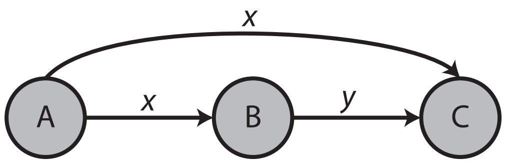
One of the essential aspects of the algorithm is the construction of the graph. Line 1 initializes the graph G. Lines 2-4 add a vertex for every module (here, $Mods$ is the set of all modules for the given phase). Lines 5-13 iterate through all of the newly created vertices. Lines 6-12 goes through the required properties of each vertex (module). Lines 7-11 then iterate through every vertex that is different from the current vertex. Lines 8-10 check each combination of dependencies to see if *req* matches with what another dependency provides. If so, a directed edge is constructed to represent the dependency.

Line 14 annotates G with finishing times through a *depth-first-search* (DFS). Line 15 then gets a list of vertices ordered according to the finishing time of the DFS. Finally, Line 16 returns the order.

Example 3. *Since DWStar currently does not have any modules that can create a cyclic dependency we introduce three fictitious modules:* `A`*,* `B`*, and* `C`*, depicted on Figure 16a. We also introduce three fictitious dependency properties called:* `x`*,* `y`*, and* `z`*. An edge between two vertices denotes that the source vertex provides a property that the target vertex requires, currently all vertices are fulfilled by what the previous vertices (modules) provides. The result of Algorithm 2 is shown in Figure 16b. When looking at the vertices, it can be seen that they are topologically sorted. When a topological sorting is possible, all dependencies can be fulfilled.*

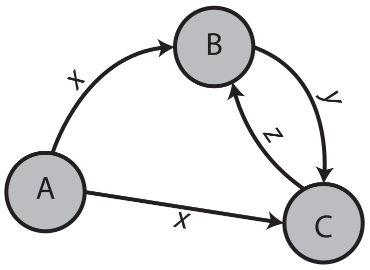


(a) Dependencies shown as a graph.

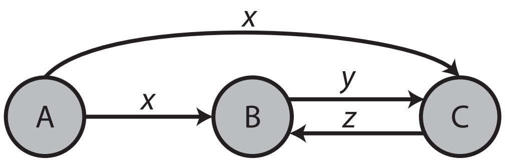


(b) Sorted dependencies.

Figure 16: Example of an acyclic graph of dependencies.



(a) Dependencies shown as a graph.



(b) Sorted dependencies.

Figure 17: Example of a cyclic graph of dependencies.

If we change `B` to also require `z`, this change creates a cycle in the graph in Figure 17a, as `B` and `C` both depend on each other. The result of Algorithm 2 is shown in Figure 17b. The difference compared to the previous graph is, that the two vertices `B` and `C` can not be properly ordered in such that all their dependencies are fulfilled. We call this result an approximate suggestion*, as the modules can not be ordered such that all dependencies are fulfilled. Instead the ordering attempts to meet as many dependencies as possible, which is trivially solved with DFS. In this particular case, we see that the `x` dependency between `A` and the two modules `B` and `C` is correctly ordered. However, no matter how `B` and `C` are ordered, they cannot both be satisfied. Approximate suggestions are permitted as a result, as it is possible for DWStar to continue on without all the dependencies being fulfilled.*

Lines 14-15 function as a topological sorting algorithm. However, the problem is that topological sorting as specified by Cormen et al. [38] requires a *directed acyclic graph* (DAG). However, we do not have the guarantee of *G* being acyclic. Instead, a DFS will suffice in this scenario, as an ordering is returned, whether linear or not.

4.4 Metadata Phase

This section briefly describes how the module `CSV Metadata` obtains metadata from CSV files. This module takes inspiration from Armbrust et al. [29], specifically their principle of finding the most fitting data type similar to the `Detect Data Type` refinement module.

Algorithm 3 shows parts of the algorithm that is used to retrieve information about a CSV file. The focus is on how to detect the data types, and it does therefore not include the details of how the common model is constructed. Line 1 initializes an empty list of tables to be used for the common model, one table for each CSV file. Lines 2-18 iterate through all CSV files in the provided folder. Line 3 defines a sample size of 1000 rows, which is deemed a reasonable estimate of rows for determining a data type. Line 4 initializes *headers* which can later contain the headers of the table if it contains them. Lines 5-8 perform a check to see if the first line in the file has headers. The function `CheckHasHeaders` (described later) determines if a CSV file contains headers. If so, we use the function `GetFields` to split the line into multiple column names, which is stored in *headers*. We then move to the next line of the file, as this should be where the data is located.

Line 9 initializes an empty array containing data types representing the data types that best fits the column in the CSV file. Lines 10-16 examine the first 1000 rows and uses these rows to determine the data types that are used for the columns. Line 11 splits the current line into a list of fields using the `GetFields` function. Line 12 then applies the function `GetDatatypes` to find the most fitting data type for the given fields (described later). The problem is now, that the function can return different data types for the same column for different rows, e.g. `Double` and `BigInt`. Line 13 uses the function `MostFittingDatatype` (described later) to determine the data type these two different data types have in common. The inferred data types in `fittingDataTypes` is incrementally refined for each iteration of the loop. Lines 14-15 move to the next row and repeats the process. When 1000 rows has been examined, Line 17 constructs the found columns and an appropriate table. Line 19 finally constructs a common model and returns it.

Algorithm 3: Algorithm for inferring the schema of CSV files and if possible retrieve headers.

```
   Data: Path p to folder
   Result: Common model
1  tables = new empty list of tables
2  foreach csv file f in p do
3      sampleSize = 1000
4      headers = new empty array of strings
5      if CheckHasHeaders(f) = true then
6          headers = GetFields(Current line in f)
7          Move to next line in f
8      end
9      fittingDataTypes = new empty data type array
10     while f is not at EOF AND sampleSize > 0 do
11         fields = GetFields(Current line in f)
12         datatypes = GetDatatypes(fields)
13         fittingDatatypes = MostFittingDatatype(fittingDatatypes, datatypes)
14         Move to next line in f
15         sampleSize = sampleSize - 1
16     end
17     tables.Add(new Table(name of f, fittingDataTypes, headers))
18 end
19 return new CommonModel(tables)
```

CheckHasHeaders

To recognize whether a CSV file contains headers, we want to ensure that we do not wrongfully recognize headers where there are none. To do so automatically, we assume that a CSV file with headers contains string values in the first row of the file. However, if all rows in a CSV file contain string values, then

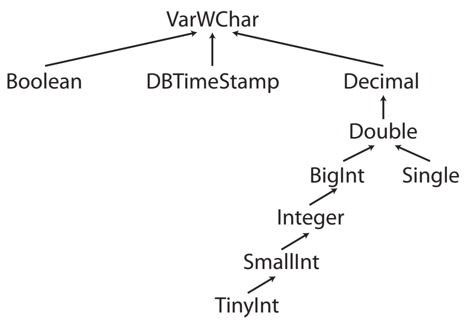


Figure 18: Specification of a data type hierarchy for the data types that we can find for the `CSV Metadata` module. At the top is the most general type `VarWChar` while the further down the hierarchy, the more specific the type is.

we can not differentiate between a CSV file with headers and one without headers. The function returns true if the first row only contains string values and any of the following ten rows' data types contains a different data type than string.

GetDatatypes

This function attempts to convert values into the most specific data types, that can accommodate the value. All values start as being of the data type `VarWChar` and are then parsed to all children types to examine whether they can encompass the value, the lowest level found is then the most fitting data type for the current value. The hierarchy is shown in Figure 18. E.g. `157.5` would first be converted from a `VarWChar` to a `Boolean` and fail, then to a `DBTimeStamp` and fail, then to a `Decimal`, then to a `Double`, and then to a `BigInt`. It would fail when converting `157.5` to a `BigInt` as the value does contain a decimal value. It would then attempt to convert to a `Single`, which it would be able to convert to successfully, thereby ending the search and returning the data type `Single`.

MostFittingDatatype

This function uses `datatypes` found with `GetDatatypes` and the `fittingDataTypes` and compares these two arrays. For the i-th entry in both arrays, the function finds the nearest vertex in the hierarchy, that the two data types have in common. This vertex (data type) is then assigned to be the most fitting data type, given the provided information up to this point. E.g. if the file contains two rows and the first is recognized as an `Integer` and the second a `Double`, the data type chosen is `Double` since it can encompass both values.

4.5 Refinement Phase

Section 3.4 introduced the refinement phase and its responsibilities. Apart from that, it also showed some refinement modules, which could be used in that phase. This section adds technical details, that were omitted in the concept section and provides descriptions on how the phase is implemented.

4.5.1 Modules

This section examines some of our refinement modules, which were briefly described in the Section 3.4, but with a focus on implementation details. These modules are not based on any external sources and are patterns we discovered through working with data from different data sets.

Column Names Pattern

In the case where the names of the columns are not known, one can examine the values stored in the column to guess the name of a column. Instead of looking directly at the values, this module looks at the pattern of the values, as some patterns can hint towards the name of a column. E.g. the values `foo@domain.com` and `bar@domain.com` would to everybody be recognized as email addresses, and thus a likely name for the column is `Email`. This module finds patterns using *Regular Expressions*, which describe how a value should be structured to be a candidate for a specific pattern. Listing 4.5 shows how we detect values formatted as email addresses, phone numbers, and addresses. The essential part of the listing is, that one can define the column name, that should be given when the values matches a pattern, and the pattern to look for.

Listing 4.5: Patterns used for detecting name of columns.

```
Email: "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$"
Phone: "^(?:\+|00)[0-9]{2,3}[ -]?[0-9]{8,10}$"
Address: "^[a-zA-Z ]+? \d+[a-zA-Z]?$"
```

These patterns are configurable, and the user is able to create his own patterns, that fit his data.

Find Relationships from Names

Relationships between tables are not always defined in OSSs. To find this, this module utilizes the observed practice where the name of a foreign key column contains the name of the table it references. E.g. a column named `MemberID` in the table `Sales` references the table `Members`. To find relationships using the name of tables and columns, we define a set of words, that a column name should include, for it to be evaluated as a foreign key. The list of words is configurable, but the predefined values are: `ID`, `Key`, `No`, and `Ref`. When a column is found containing any of these words, the substring without the keyword is extracted and matched with the names of the other tables. If a match is found, we create a relationship between the foreign key column and the primary key of the matched table. Notice, that in the example that we gave with `MemberID` and the table `Members`, the column name is singular, whereas the table name was plural. To remedy this, we use the `EnglishPluralizationService` class [39] from C#. This class is to our knowledge able to retrieve the singular form of any english word, and would also be able to convert words such as `Companies` to `Company`.

As this module only relies on the names of tables and columns, finding relationships between different kinds of OSSs (e.g.: Relational databases and CSV files) is not a problem.

4.5.2 Common Model Grouping

We introduced the concept of common models grouping in Section 3.4.2. This section details the approach we used for implementing it in DWStar. To determine how the common models should be grouped, we represent the common model as an undirected graph, where the vertices is the set of tables in the supplied common model from the metadata phase, and the edges define a relationship between tables. All tables that are connected in this graph should be represented in one common model. To find these sets of connected tables, we traverse the graph using a DFS, starting at a random non-visited vertex. When all vertices related to the randomly selected vertex have been visited, DWStar try to perform the search on another non-visited vertex. This continues until all vertices have been visited.

Listing 4.6 show how the implementation utilizes this graph and DFS. Line 3 creates an undirected graph based on the supplied common model. A relationship between two tables is represented by an edge between them. Line 4 creates a list of all tables in the common model. This list is used to track which tables have not already been found in the graph. The list is iterated in Lines 7 to 15, which contain a loop that continues to execute while the list still contains elements. Line 9 picks the first element in the list and stores that as the `curTable` variable. Line 11 then performs a DFS on the graph. The search starts at the `curTable` and visits all the tables that are directly or indirectly connected to `curTable`. The function results in a list of all these tables and Line 12 then remove these tables from the `remainingTables` list, as we do not need to visit them afterwards. Line 14 then creates a common model with these tables. When all tables in the graph has been visited the function terminates in Line 17 by returning a list of all found common models.

Listing 4.6: Group common models

```
 1 public List<CommonModel> GroupCommonModels(CommonModel model)
 2 {
 3   Graph graph = Graph.CreateUndirectedGraph(model);
 4   List<Table> remainingTables = model.Tables;
 5   List<CommonModel> groupedModels = new List<CommonModel>();
 6 
 7   while(remainingTables.Count > 0)
 8   {
 9     Table curTable = remainingTables[0];
10 
11     List<Table> visitedTables = graph.DFS(curTable);
12     remainingTables.RemoveRange(visitedTables);
13 
14     groupedModels.Add(new CommonModel(visitedTables));
15   }
16 
17   return groupedModels;
18 }
```

4.6 Generation Phase

This section describes the implementation of the two generation modules that were introduced in Section 3.9. Section 4.6.1 describes the `Create Script Generation` generation module which is able to output SQL DDL statements for creating the DW(s). Section 4.6.2 shows the `DW Population` generation module for populating one's DW with data from the OSSs.

4.6.1 Create Script Generation

The `Create Script Generation` generation module converts a star model into a series of SQL statements, which collectively creates all the tables (including their columns) in the DW from a star model. The SQL statements for this generation module targets SQL Server and are therefore T-SQL statements.

Instead of manually generating every single `CREATE` statement, we utilize a feature in SQL Server called *SQL Server Management Objects* (SMO) [40], specifically the `Scripter` class, which allows one to specify a database using C# objects and then generate a series of SQL scripts that contains: Tables, columns, primary keys, and foreign key definitions.

Listing 4.7: Generation method for creating star schema in SQL Server.

```
public void Generate(StarModel starModel, string resultPath)
{
    var server = new Server();
    var database = new Database(server, "DW");
    var scripter = new Scripter(server);

    var columns = new List<Column>();
```

```
 8     var tables = new List<Table>();
 9     var starTables = new List<StarTable>{starModel.Dimensions};
10     starTables.Add(starModel.factTable);
11 
12     foreach(st in starTables)
13     {
14         var tbl = new Table(database, st.Name);
15         foreach(clm in st.Columns)
16         {
17             var col = new Column(tbl, clm.Name, ConvertDatatype(clm.DataType));
18 
19             if(clm is surrogatekey)
20                 clm.Identity = true;
21 
22             tbl.Columns.Add(clm);
23             columns.Add(clm);
24         }
25 
26        CreatePrimaryKeyConstraint(tbl.Columns, st, tbl);
27        tables.Add(tbl);
28     }
29 
30     CreateForeignKeyConstraints(tables, columns, starTables);
31 
32     File.WriteAllLines(path, scripter.Script(tables));
33 }
```

The method in Listing 4.7 shows how a star model is converted into SQL CREATE statements and saved to a file. The `Scripter` class need a connection to a SQL Server to properly construct the queries. This connection for example retrieves the version of the SQL Server and allows the `Scripter` to generate SQL scripts targeting that specific version. Line 3 therefore creates a `Server` object which by default connects to a local SQL Server instance. Line 4 creates a temporary `Database` object, which is required when generating `Table` objects, but the database is never used in the SQL scripts. Lines 5-10 firstly defines SMO columns and tables followed by instantiation of our star tables which includes both dimensions and the fact table. Line 12-28 then iterate through all tables to convert them using SMO. For each table in the star model, Lines 15-24 create a SMO column for each star column in the star table, as well as the columns' properties (data types in Line 17, whether it auto-increments in Lines 19-20). Primary keys are marked for all columns in Line 26 using the method `CreatePrimaryKeyConstraint`. Similarly in Line 30, all foreign keys are provided to SMO using `CreateForeignKeyConstraints` which finds these by making use of the relationships contained within the star model. Finally, Line 32 writes the script that has been accumulated for SMO in a file which the user can then execute.

4.6.2 DW Population

The `DW Population` generation module creates SQL scripts for SQL Server that selects data from the OSS(s) and inserts the data into the DW.

This module relies on the aspect that all elements of a star model can be traced back to the elements of a common model. E.g. a *startable* contains the element *table*. This element shows that a specific *startable* (fact table or dimension) originates from a common model *table*. Having this connection, that tracks where the star tables originates from, makes it possible to create SQL scripts, that transfers data from the OSS(s) to the DW. The same principle is also used for *starcolumn*.

The resulting SQL scripts consists of two steps:

1. Delete data
    a Truncate the fact table
    b Delete rows in dimensions
2. Populate tables
    a Populate the dimensions

b Populate the fact table

Step 1 of the SQL script deletes the existing rows in the DW, as it otherwise would risk inserting values, that already exists in the DW. **Step 1.a** deletes all existing rows in the fact table. The statement `TRUNCATE` is used instead of `DELETE` to clear the fact table, as this statement is faster (it directly deletes the data pages) [41]. **Step 1.b** then uses `DELETE` statements to delete the rows in the dimensions. `TRUNCATE` is not used here, as the dimensions are referenced by another table (the fact table), and `TRUNCATE` can not execute when a foreign key references the affected table.

Step 2 populates the tables in the DW. Here we first populate the dimensions and afterwards the fact table. This order ensures that all the values that is referenced by the fact table should already contain data. **Step 2.a** generates a series of `INSERT INTO` statements, which inserts values from the OSS(s) into the related dimensions. If the dimension is either a `date` or `time` dimension, it inserts a hard coded script, which loops through a prespecified date range or time range and creates the necessary rows for the dimensions (these scripts can be found in Appendix B). **Step 2.b** has the purpose of populating the fact table. The statement that is generated consists of two parts: Part one retrieves data from the original source tables from the common model, which becomes values in the fact table. Part two then joins the data from part one together with the dimensions. E.g. the column `room_id` from the source table `sale` is joined with the column `id` in the room dimension. This then gathers the correct foreign key values, which is used in the fact table. Listing 4.8 provides an example of how the `Fact_Sale` fact table could be populated with a SQL query. We have in this listing replaced the lists of columns with three dots (`...`), as these would fill to much for this small example. The listing also only show one dimension `Dim_Room`, however, an `INNER JOIN` clause would exist for each dimension in the star model.

Listing 4.8: Example of how to insert data into the fact table `Fact_Sale`.

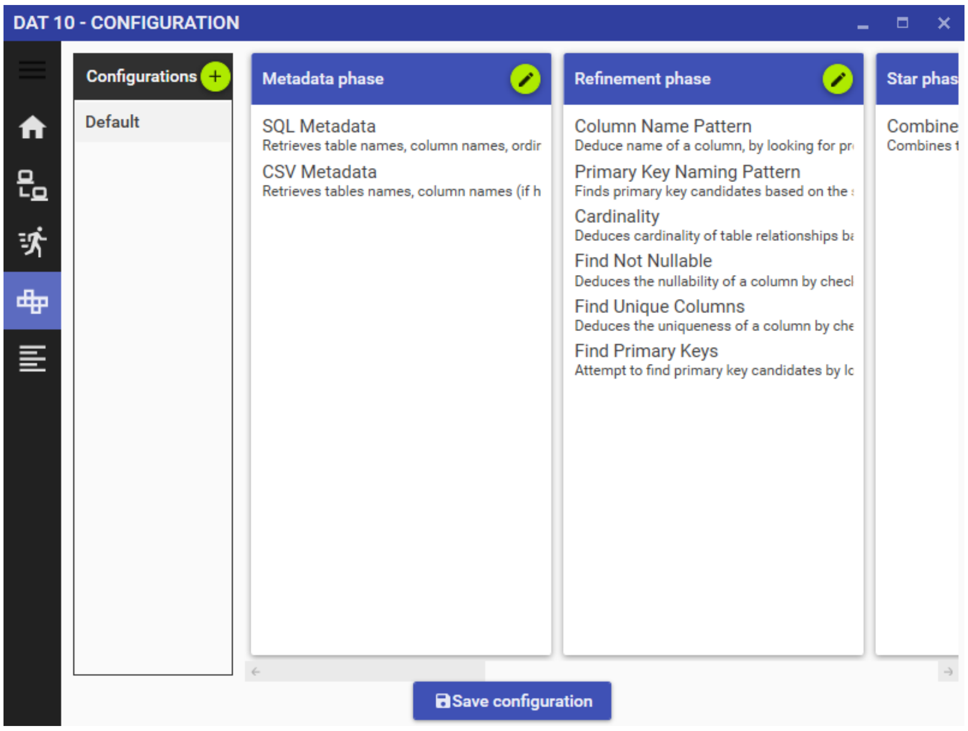
```sql
INSERT INTO [DW].[dbo].[Fact_Sale] (...)
SELECT ...
FROM [OSS].[dbo].[sale] AS s
INNER JOIN [DW].[db].[Dim_Room] AS r ON s.room_id = r.Id;
```

The join condition is different when retrieving the primary keys from the date and time dimension, as the values that we wish to compare for example are `DateTime` values from the fact table and a variety of columns in the date and time dimensions. The join conditions then have to split a `DateTime` into its components (year, month, day, hour, etc.) and compare on these values.

It can occur that a dimension or fact table is based on multiple tables. In these cases, it is necessary to join the tables together. Essentially the `FROM` clause is expanded to include a `FULL OUTER JOIN` clause, which ensures that all values from the other table(s) are included in the dimension or fact table. We join the tables together with a `FULL OUTER JOIN`, as we wish to include all columns from both tables being joined, and not only the values that can successfully be joined with the first table.

A limitation of the current implementation of this generation module (`DW Population`) is that it assumes that both the OSS(s)and the DW are located on the same server. It is assumed that this is usually not the case. One of the ways this can be remedied is by using *Linked Servers* for SQL Server [42], which allows for database instances to perform queries on other database instance, that are connected across servers. This was written about in our previous work [43], and thus will not be repeated in this report. Similarly, one can use the SQL statement `BULK INSERT` [44] for SQL Server, which allows one to import data from a file. After having imported the data, the ETL scripts can then insert it into the data warehouse.

The ETL Script Generator primarily affects the extraction and loading part of an ETL process. It is intentional that the transformation part is left mostly untouched, as it is not known how the data should be transformed. The resulting ETL scripts should therefore be seen as a foundation, from which the user can add row versioning, data cleansing, etc. or just use the module to quickly establish a DW that can be used for data analysis within minutes.



SQL Metadata
Retrieves table names, column names, ordir

CSV Metadata
Retrieves tables names, column names (if h

Column Name Pattern
Deduce name of a column, by looking for pr

Primary Key Naming Pattern
Finds primary key candidates based on the

Cardinality
Deduces cardinality of table relationships b

Find Not Nullable
Deduces the nullability of a column by checl

Find Unique Columns
Deduces the uniqueness of a column by che

Find Primary Keys
Attempt to find primary key candidates by lc

Combine
Combines t

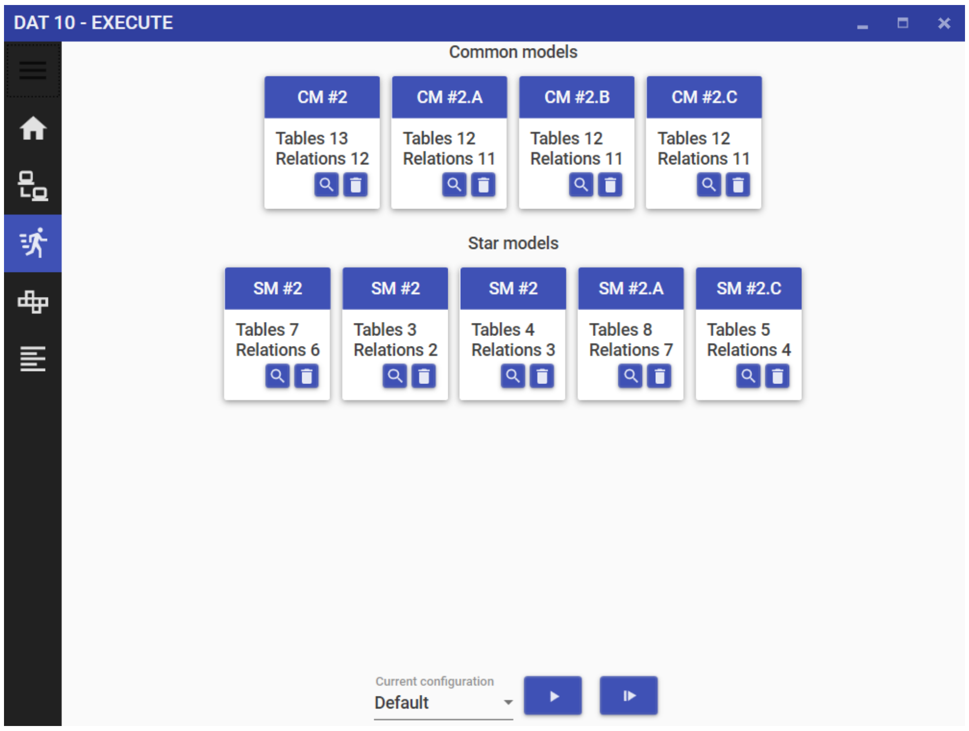
Figure 19: Screen dump of the configuration screen.

4.7 User Interface

This section gives a short introduction to the graphical user interface, shows some screen dumps of important screens, and describes how these screens interact with the rest of the implementation. This is done to further document the ways in which the user can customize the process to fit his needs.

4.7.1 Configuration

It has been stated, that the user is able to select which modules that should participate in the various phases. The `Configuration` screen in Figure 19 shows four boxes (star phase is only partially visible), where the `Configurations` box contains a list of all existing configurations (the only configuration shown in the figure, is the one named `Default`). The user is able to create or modify configurations and then switch between existing configurations. The remaining boxes represent the phases, and the content of these boxes show the modules selected in the configuration called `Default`. While not depicted fully in the figure, all phases and subphases has a box. In this screen, we see that we have selected two metadata modules (`SQL Metadata` and `CSV Metadata`). Underneath the name of the module is a short description of the module, which helps the user quickly understand what the module does. The order that the modules is shown in is also the order in which they execute, specifically from top to bottom. The modules that are included in the phase and the order they execute in can be configured by pressing the round button to the right of the phase's name.



CM #2 Tables 13 Relations 12
CM #2.A Tables 12 Relations 11
CM #2.B Tables 12 Relations 11
CM #2.C Tables 12 Relations 11

SM #2 Tables 7 Relations 6
SM #2 Tables 3 Relations 2
SM #2 Tables 4 Relations 3
SM #2.A Tables 8 Relations 7
SM #2.C Tables 5 Relations 4

Current configuration
Default

Figure 20: Screen dump of the execution screen.

4.7.2 Execution

The central screen of the user interface is the `Execution` screen in Figure 20. This screen shows the current status of the execution, as the common models and star models shown are based on which phase has been executed. This figure illustrates the final phase of the process and the resulting star models based on the common models. In the bottom of the screen we see three components (from the left): A dropdown menu, a button with a “play” button, and a “step forward” button. The dropdown menu allows the user to select the configuration he wishes to use when he executes DWStar. The “play” button runs through all (or the remaining phases) and portraits the result to the user. Finally, the “step forward” button, as the name suggest, allows the user to sequentially step through each phase and view the result of that phase.

The result of any phase is shown in the top part of the screen. From this particular execution, we have executed all phases and see four common models and five star models. At the top are the common models, denominated with `CM` followed by a number and possibly a character (`A`, `B`, `C`, etc.). The number is used to distinguish common models from each other, meaning that two common models called `CM #2` and `CM #3` would contain completely separated tables. The character value exists due to the `Combination Subphase`, which can produce alternative common models by combining two tables into one and thereby producing a new common model. Each version produced by the `Combination Subphase` is then appended with a new character value. The star models are denominated with `SM` followed by a number and possibly a character value, which correlates to the common model that the star model was derived from. Each box for a common model or star model contains two buttons, the first lets the user view the model where the second deletes the model. An example of viewing a star model is shown in Figure 21. The deletion of a model will exclude it from being processed in the following phases and thereby allows the user to preemptively filter out undesired results. We find that the user interface is both useful for config-

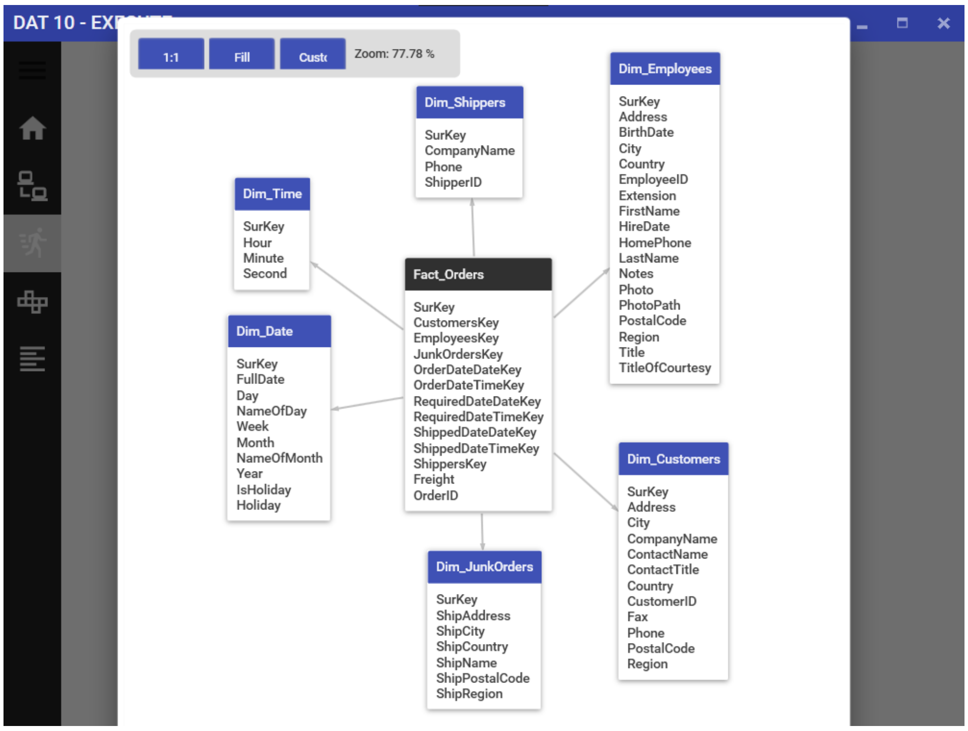


Figure 21: Screen dump of viewing a star model.

uring the process of DWStar and provides a great amount of information that helps the user understand each phase.

Chapter 5

Experiments and Evaluations

In order to evaluate whether the properties described in the problem statement have been implemented this section details two experiments and one evaluation. Section 5.1 goes through an experiment to evaluate how well missing metadata can be inferred. Section 5.2 shows an evaluation of how well star schemas are constructed using DWStar. Section 5.3 details an experiment that measures the time taken by various parts of DWStar.

One part of the problem statement details the customizability of DWStar: *"Customize DW construction"*. However, no experiments were found necessary for this point, as this has been sufficiently argued for throughout Chapter 3, which details all instances at which a user can customize the execution of DWStar. Additionally, Section 4.7 showed how these principles were implemented in the user interface.

Two points of the problem statement are to: *"Automatic DW construction"*, and *"Metadata inference"*. Both of these points are verifiable, and experiments and evaluations are therefore set up for these to verify their fulfillment.

The last point of the problem statement states: *"Automatic DW population"*. We want to test two different aspects of this requirement: Firstly, we are interested to know, whether or not the produced SQL scripts can be successfully executed, and secondly, if the scripts can be executed within a reasonable amount of time. Both of these aspects are tested in the same experiment and the results are evaluated according to our expectations. The currently implemented method for generating ETL processes is rather simplistic as it does not allow for delta loading, row versioning, or extraction from other OSSs than databases within the same database server. Various techniques for how to solve this has already been examined in our 9th semester project [43], and it is therefore already shown that it is possible to achieve this.

5.1 Experiment: Infer Missing Metadata

This experiment evaluates how well DWStar infers the schema of OSSs that does not explicitly provide them. Currently, only two metadata modules are implemented, one for SQL Server where metadata is explicitly exposed and one for CSV files, where it is not. CSV files are therefore used throughout this experiment, since inferring metadata from SQL server would be trivial. To conclude on this experiment, we examine how many deviations there exists between the original schema and the common model created on the background of CSV files.

5.1.1 Methodology

For preparation of the experiment, a OSS consisting of a database has been exported to CSV files (one CSV file for each database table). The name of the files corresponds to the table name and includes headers for all columns. Headers are included as it is assumed that this is often the case in real-life scenarios. Additionally, this is the default behavior of SQL Server 2014 Management Studio [45], which was used for exporting CSV files for this experiment. In the event that the column names were missing, one could use refinement modules (like `Column Name Inference`) to guess column names, or one could prompt the user for names. Naturally, the CSV files do not include any information regarding relationships, constraints, nor the data types of the columns.

DWStar is provided with the set of CSV files, which is parsed through the metadata phase and the refinement phase. This process produces one or more common models which are collectively compared to the source database. The comparison counts how many deviations there exist, which both includes missing constituents of the common model as well as constituents that has been inferred incorrectly.

Note that deviations for data types' length, precision, and scale are not taken into consideration, as these are seen as too minor of a difference to be of significance. As long as it is ensured that the inferred length, precision, and scale are still able to accommodate the values, the deviation is not seen as a problem, as it would not cause any problems (the `Extend Data Type Length` module could alleviate this). For instance, if the data type is `nvarchar` with a length of 50 and DWStar infers a length of 60, this should not count as a deviation, given that all textual data could be contained with the inferred length.

Recall that the common model allows for multiple suggestions to data types, constraints, etc., each with an associated confidence. In this experiment, only the suggestion with the highest confidence is used for the comparison, as it would not be straight-forward to make a comparison with multiple suggestions.

The constituents of the common model which are not compared are:

- *id* in *column* as this is only used as a unique, global identifier for a column.
- *schema* in *table* as it was described that $schema = \varepsilon$ for CSV files, so this deviation is not an inaccuracy as such, but rather the result of changing the OSS type for the experiment.
- *database* in *table* as it describes where the data is located. A change in *database* is to be expected as it will be the directory for CSV files rather than a connection string for a database.
- *cardinality* in *relationship* as the original schema does not provide this.
- The names of a *table* and *column* as these are explicitly given, so no deviations can occur.

Operational Source system

A well-known sample database called *Northwind* [46] has been chosen as the OSS. The schema diagram for this database is shown in Figure 22. A more detailed version of the schema diagram that includes data types and nullability can be found in Appendix D.

Configuration of DWStar

This section describes how DWStar is configured in terms of modules for this experiment. Any of the phases beyond the refinement phase is not relevant for this experiment, as we only use the common model of the refinement phase for the comparison.

For the metadata phase, the only metadata module used is `CSV Metadata`, as DWStar only receives CSV files as input in this experiment.

The refinement modules (Section 3.4.1) used in DWStar are shown below:

- `Detect Data Types`
- `Find Not Nullable`
- `Find Unique Columns`

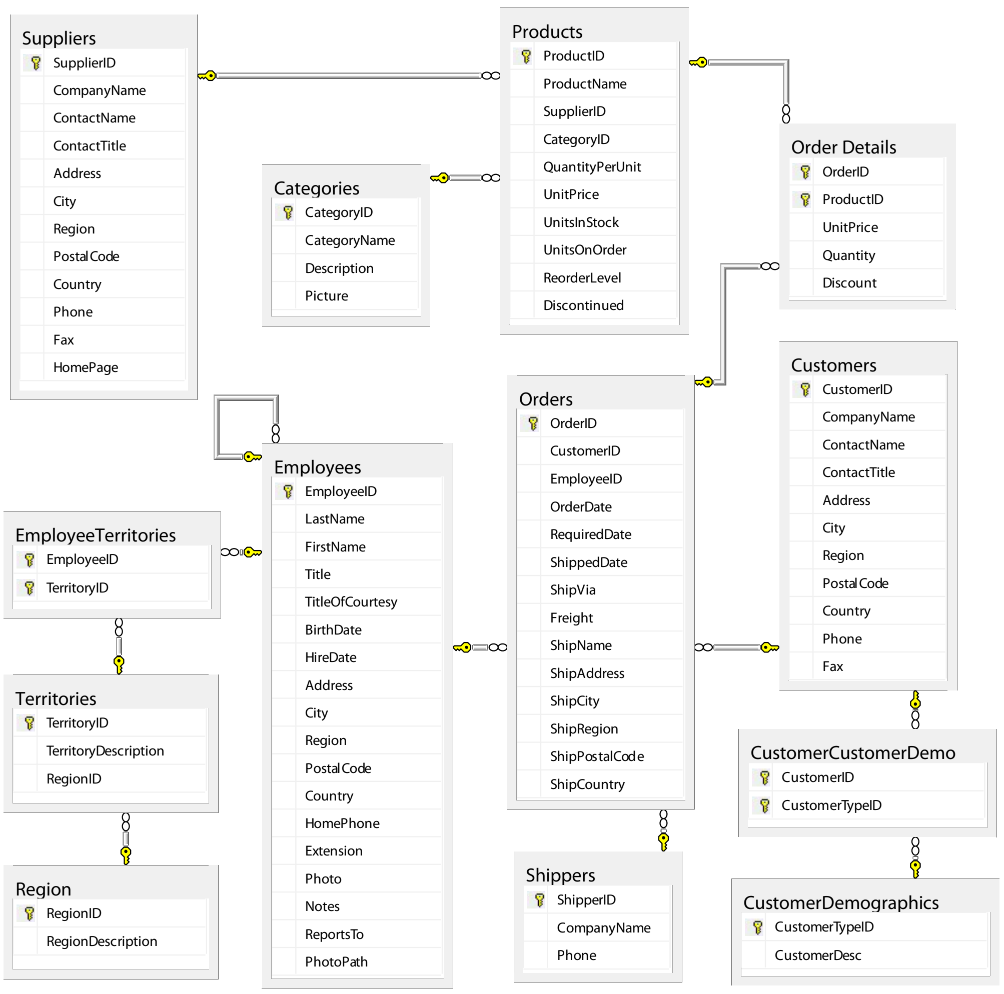


Figure 22: Schema diagram of Northwind for the experiment.

Table 8: Results showing number of deviations according to their severity.

	Data Type	**Unique**	**Nullability**	**Primary Key**	**Foreign Key**
No Deviation	85	60	40	10	10
Minor Deviation	0	28	48	0	0
Major Deviation	3	0	0	3	3

- `Find Primary Keys`
- `Find Relationships from Names`
- `Primary Key Naming Pattern`

Finally, the data types of the original schema uses SQL data types, while DWStar uses OLE DB data types. Therefore, a reasonable estimation has to be made as to whether the data types are equivalent. For instance, the SQL data type `nvarchar` is said to be equivalent to the OLE DB data type `VarWChar`. A full list of the assumptions of equivalences can be found in Appendix C.

5.1.2 Results

A summary of the results of the experiment is shown in Table 8, only the summary is shown since the results of the experiment are too large to display properly. The results are divided into three groups: *No Deviations*, *Minor Deviations*, and *Major Deviations*. Each group shows the severity of how much the inferred common models differs from the database. No deviation entails that the constituents of the common model match with the database. Minor deviation entails a mismatch between the common model and the database which has negligible effect on the result. Major deviation entails a direct error which will have a negative effect on the result. Details as to why certain constituents of the common model are categorized as minor or major deviation is detailed further in the subsections below. A detailed overview of the constructed common model can be found in Appendix D.

The OSS consisted of 13 tables, and 88 columns that included 13 are primary keys and 13 are foreign keys. To evaluate these results we provide more details to why deviations exist and possibly how these can be avoided.

Data Type

The correct data types were inferred for 85 out of the 88 columns with only 3 major deviations, these deviations occur in three different columns: `Employees.Extension`, `EmployeesTerritories.TerritoryID`, and `Territories.TerritoryID`. For `Employees.Extension`, DWStar inferred an `Integer` data type since the values found were strictly integers where in Northwind the data type was defined as `String`, which according to the data seems unnecessary. For both `EmployeesTerritories.TerritoryID`, and `Territories.TerritoryID` we also inferred an `Integer` data type where the actual type was `String`. This inference is incorrect since some of the values in these columns starts with a zero which, when converting the string value to an integer, will trim the leading (prepended) zeros, thereby providing the wrong value. To ensure this scenario does not occur in future scenarios, it is possible to check supposed `Integer` data types for leading zeros and ensure that these are included in the end result.

Unique

The results for unique does initially seem unsatisfactory since DWStar found 60 columns with no deviations and 28 with minor deviations. However, these minor deviations are attributed to the amount of data available and lacking constraints in the data source. We define a column as being unique if it contains distinct values in the column. Therefore, any columns that were a primary key in the OSS are then trivially unique. If the data sample is small enough, then any column can appear unique. E.g. the columns `Name` or `PostalCode` can appear unique, however it is often seen that person names and postal codes are duplicated across multiple rows.

Nullability

When discussing *Nullability* we refer to the property of a column being defined as either: `Not Null`, or `Null`. We inferred 40 out of the 88 columns' nullability correctly with 48 minor deviations. This large amount of minor deviations is attributed towards the problem described with `Unique`, where the amount of data is either too small to determine the nullability of a column.

Primary Key

DWStar found 10 primary keys that matched precisely with the OSS and had 3 major deviations (incorrect or no primary keys set for table). These deviations occurs since we currently have not implemented a module explicitly for identifying composite keys, such as composite keys in bridge tables. Northwind contains three different bridge tables each with a composite key consisting of two foreign keys.

Foreign Key

Our current module for detecting foreign keys are highly dependent on the naming convention of the OSS. DWStar found 10 foreign keys correctly and had 3 major deviations. The foreign keys DWStar does not detect are: `CustomerCustomerDemo.CustomerTypeID` which references `Categories.CategoryID`, `Employees.ReportsTo` which references `Employees.EmployeeID`, and `Orders.ShipVia` which references `Shippers.ShipperID`. Common for all these is that they differ from the naming convention that our current module expects, and the naming convention that is used elsewhere in the database. The module expects that a foreign key contains the name of the table it references and for example the keyword "`ID`". None of these deviations fulfills this heuristic and we therefore have not found these foreign keys.

5.1.3 Conclusion

If we set aside the deviations caused by the low amount of available data. We see that a majority of the missing metadata has been found and the minor deviations could quickly be corrected by a user. We therefore conclude, that the modules used for finding metadata in CSV files are generally functionally sound and the primary weakness of the results are reflected in the lack of available data.

5.2 Evaluation: Constructing Star Schemas

This evaluation details how well DWStar generates star schemas. Notably, Section 5.2.1 describes the approaches that were considered for the evaluation which is then followed by the evaluation. The evaluation is performed on two different OSSs where the resulting candidate star schemas are evaluated by an external BI practitioner.

5.2.1 Methodology Considerations

Multiple approaches to this experiment have been investigated. This section examines the strengths and weaknesses of the different approaches.

Four approaches were considered for this experiment:

(a) DWStar goes through the OSS and finds the (at most) three most likely candidate star schemas in terms of confidence. The candidates are then presented to a BI practitioner who would then note down the changes that would be desirable before the star schema would be considered satisfactory.

(b) Both DWStar and a BI practitioner are provided the same OSS, and will then both independently come up with star schemas. The BI practitioner's suggestion is then compared with the three most likely candidates by DWStar. The differences are then noted down.

(c) We find multiple OSSs and their DW counterparts. Based on these we then use DWStar with the OSSs and examine how closely DWStar's results matches with the designed DWs.

(d) We provide a prototype of DWStar which can then be used publicly or is actively distributed to willing participants. The participants can then provide feedback regarding how well DWStar performed.

Approach (c) requires OSSs and DWs from either a public source or from an organization. Other than AdventureWorks [47] which offers a large DW, we found no commonly known or publicly available DWs. Approach (d) also requires organization involvement which we currently do not have the connections or means to ensure, and publicly distributing a prototype of DWStar simply takes too long. We therefore conclude that both approach (c) and (d) are currently unavailable.

When creating star schemas, there are typically guidelines that are followed according to an organization's requirements. Since DWStar does not take into account an organization's requirements there cannot be provided any requirements to the BI practitioner. For approach (b), the BI practitioner is allowed to make any assumption according to OSS schema and according to what assumptions he takes, the results might differ from what DWStar provides, even if both result are equally valid.

In approach (a) where we present three candidates for a BI practitioner, it could be possible that the practitioner could be influenced by the results presented. Thus, he might have preferred a different DW design given the OSS, had he not been presented with our candidates. However, as the BI practitioner is quite experienced, we assume that this poses no complications.

Approach (a) has no significant disadvantages since only the result of DWStar is focused on. On the background of this, approach (a) was chosen for conducting the experiment.

5.2.2 Methodology

The BI practitioner is provided with two different OSSs and their candidate star schemas. He is then provided with a list of questions that are meant to evaluate the candidate star schemas. The changes that the BI practitioner writes down are then evaluated to determine the qualitative capabilities of DWStar.

Configuration Setup

The modules used in each phase are shown in Table 9. Recall that an overview of the modules are found in Appendix A. We only use a single refinement module that finds the cardinality, this is due to the OSS being an SQL server, which expose the remaining metadata directly. The threshold θ for fact tables is set to 0.8 in this experiment, as we have observed that this value produces a manageable list of high confidence candidates.

Operational Source Systems

Two different OSSs are used in this experiment. For each case, both the schema diagram and the resulting candidate star schemas of DWStar are shown.

Table 9: Configuration used for the experiment. Modules are described below for each phase.

Metadata Phase	**Refinement Phase**	**Star Phase**	**Star Refinement Phase**	**Generation Phase**
SQL Metadata	Cardinality	**Combination subphase:** Combine Tables	Add Date and Time Dimensions	ETL Script Generator
		Fact table subphase: Fact More Rows	Add Surrogate Key	
		Fact table subphase: Fact More Columns	Name Convention	
		Wrap-up subphase: Flatten Dimensions	Order Columns	

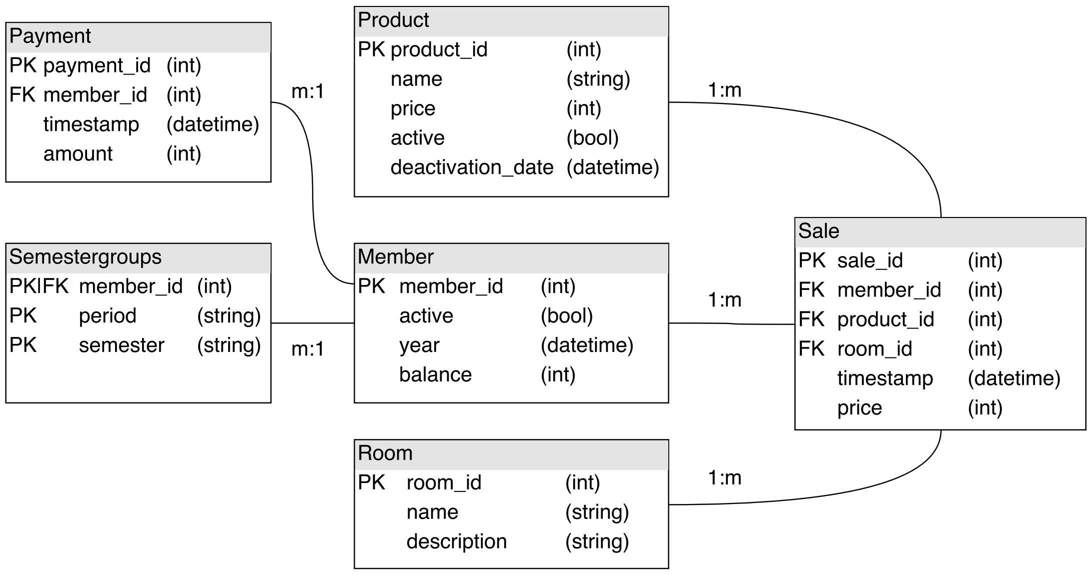


Figure 23: Schema of FKlub source database.

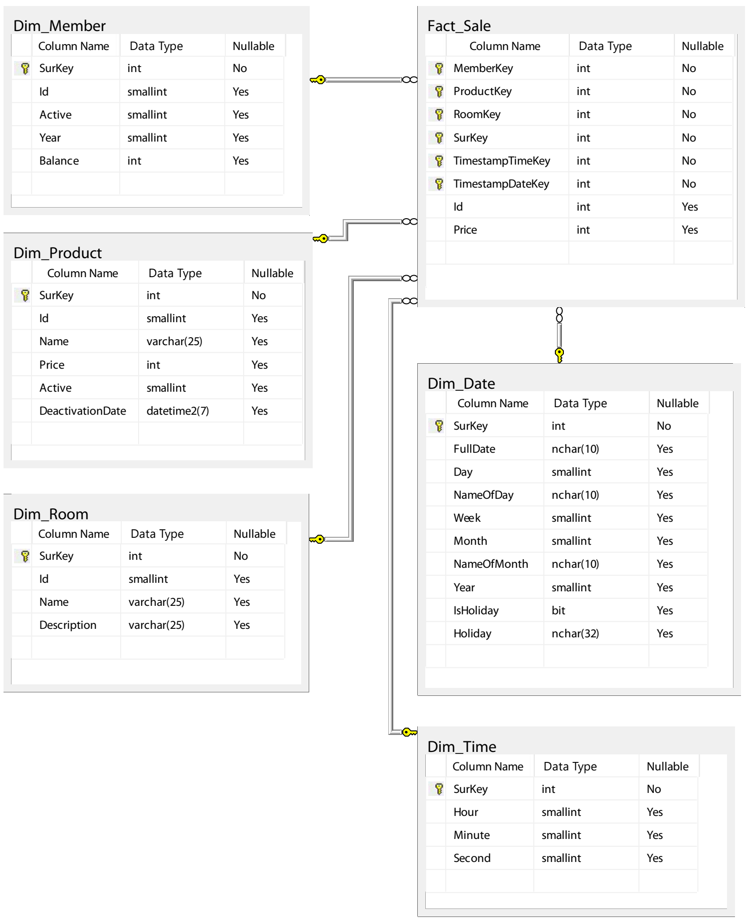
Operational Source System 1 (FKlub): The FKlub database is illustrated in Figure 23 shows a real life database from a social club at Aalborg University. This database makes it possible for students to buy various small beverages such as cola, energy drinks, and the like.

Operational Source System 2 (Northwind): This experiment also uses the Northwind database, since it is a well-known database. The schema diagram is shown in the previous experiment in Figure 22.

Candidates

This section contains a list of the candidate star schemas that DWStar created for each OSS. The illustrations in Figure 24, 25, 26, and 27 provide an overview of the candidates of DWStar and are discussed in results of this experiment. For the 1st OSS (FKlub) only one candidate was constructed as seen in Figure 24.

For the 2nd OSS (Northwind), three candidates were created by DWStar. These are shown in Figure 25, 26, and 27.



Dim_Member

Column Name	Data Type	Nullable
SurKey	int	No
Id	smallint	Yes
Active	smallint	Yes
Year	smallint	Yes
Balance	int	Yes

Fact_Sale

Column Name	Data Type	Nullable
MemberKey	int	No
ProductKey	int	No
RoomKey	int	No
SurKey	int	No
TimestampTimeKey	int	No
TimestampDateKey	int	No
Id	int	Yes
Price	int	Yes

Dim_Product

Column Name	Data Type	Nullable
SurKey	int	No
Id	smallint	Yes
Name	varchar(25)	Yes
Price	int	Yes
Active	smallint	Yes
DeactivationDate	datetime2(7)	Yes

Dim_Date

Column Name	Data Type	Nullable
SurKey	int	No
FullDate	nchar(10)	Yes
Day	smallint	Yes
NameOfDay	nchar(10)	Yes
Week	smallint	Yes
Month	smallint	Yes
NameOfMonth	nchar(10)	Yes
Year	smallint	Yes
IsHoliday	bit	Yes
Holiday	nchar(32)	Yes

Dim_Room

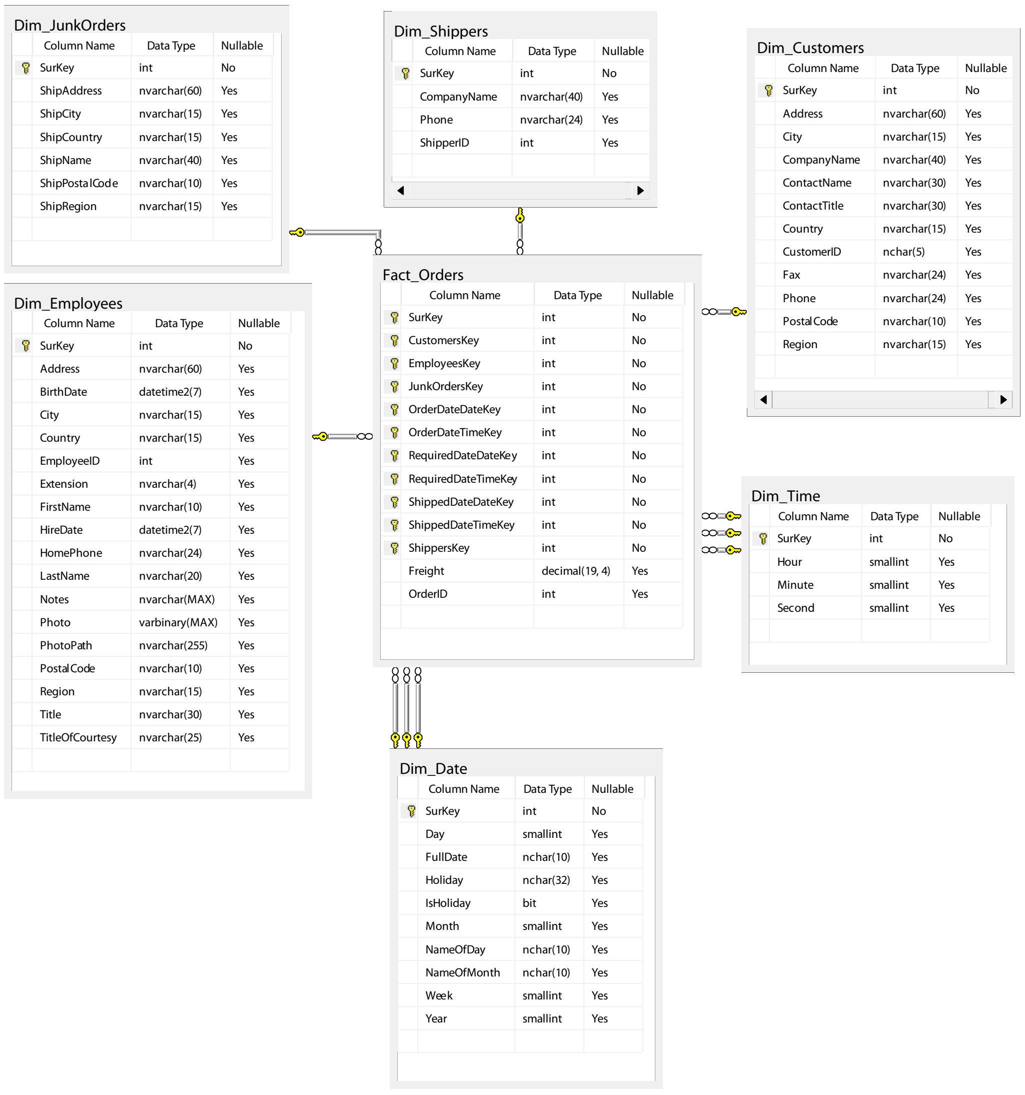
Column Name	Data Type	Nullable
SurKey	int	No
Id	smallint	Yes
Name	varchar(25)	Yes
Description	varchar(25)	Yes

Dim_Time

Column Name	Data Type	Nullable
SurKey	int	No
Hour	smallint	Yes
Minute	smallint	Yes
Second	smallint	Yes

1-1

Figure 24: First result by DWStar for the FKlub OSS. `Sales` was found to be the fact table in this case.



Dim_JunkOrders

	Column Name	Data Type	Nullable
🔑	SurKey	int	No
	ShipAddress	nvarchar(60)	Yes
	ShipCity	nvarchar(15)	Yes
	ShipCountry	nvarchar(15)	Yes
	ShipName	nvarchar(40)	Yes
	ShipPostalCode	nvarchar(10)	Yes
	ShipRegion	nvarchar(15)	Yes

Dim_Shippers

	Column Name	Data Type	Nullable
🔑	SurKey	int	No
	CompanyName	nvarchar(40)	Yes
	Phone	nvarchar(24)	Yes
	ShipperID	int	Yes

Dim_Customers

	Column Name	Data Type	Nullable
🔑	SurKey	int	No
	Address	nvarchar(60)	Yes
	City	nvarchar(15)	Yes
	CompanyName	nvarchar(40)	Yes
	ContactName	nvarchar(30)	Yes
	ContactTitle	nvarchar(30)	Yes
	Country	nvarchar(15)	Yes
	CustomerID	nchar(5)	Yes
	Fax	nvarchar(24)	Yes
	Phone	nvarchar(24)	Yes
	PostalCode	nvarchar(10)	Yes
	Region	nvarchar(15)	Yes

Dim_Employees

	Column Name	Data Type	Nullable
🔑	SurKey	int	No
	Address	nvarchar(60)	Yes
	BirthDate	datetime2(7)	Yes
	City	nvarchar(15)	Yes
	Country	nvarchar(15)	Yes
	EmployeeID	int	Yes
	Extension	nvarchar(4)	Yes
	FirstName	nvarchar(10)	Yes
	HireDate	datetime2(7)	Yes
	HomePhone	nvarchar(24)	Yes
	LastName	nvarchar(20)	Yes
	Notes	nvarchar(MAX)	Yes
	Photo	varbinary(MAX)	Yes
	PhotoPath	nvarchar(255)	Yes
	PostalCode	nvarchar(10)	Yes
	Region	nvarchar(15)	Yes
	Title	nvarchar(30)	Yes
	TitleOfCourtesy	nvarchar(25)	Yes

Fact_Orders

	Column Name	Data Type	Nullable
🔑	SurKey	int	No
🔑	CustomersKey	int	No
🔑	EmployeesKey	int	No
🔑	JunkOrdersKey	int	No
🔑	OrderDateDateKey	int	No
🔑	OrderDateTimeKey	int	No
🔑	RequiredDateDateKey	int	No
🔑	RequiredDateTimeKey	int	No
🔑	ShippedDateDateKey	int	No
🔑	ShippedDateTimeKey	int	No
🔑	ShippersKey	int	No
	Freight	decimal(19, 4)	Yes
	OrderID	int	Yes

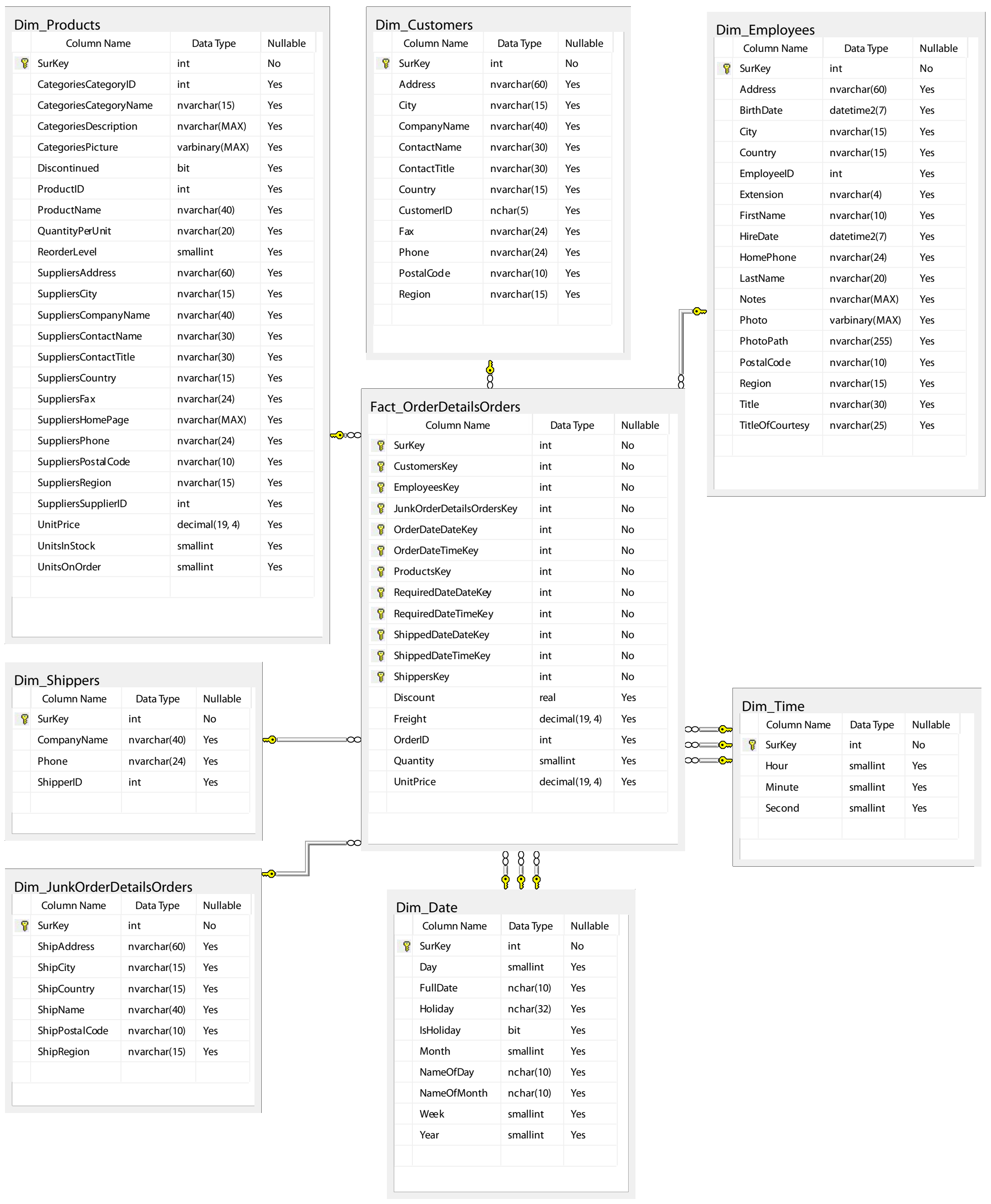
Dim_Time

	Column Name	Data Type	Nullable
🔑	SurKey	int	No
	Hour	smallint	Yes
	Minute	smallint	Yes
	Second	smallint	Yes

Dim_Date

	Column Name	Data Type	Nullable
🔑	SurKey	int	No
	Day	smallint	Yes
	FullDate	nchar(10)	Yes
	Holiday	nchar(32)	Yes
	IsHoliday	bit	Yes
	Month	smallint	Yes
	NameOfDay	nchar(10)	Yes
	NameOfMonth	nchar(10)	Yes
	Week	smallint	Yes
	Year	smallint	Yes

Figure 25: First result by DWStar for the Northwind OSS. `Orders` was found to be the fact table in this case.



Dim_Products

Column Name	Data Type	Nullable
SurKey	int	No
CategoriesCategoryID	int	Yes
CategoriesCategoryName	nvarchar(15)	Yes
CategoriesDescription	nvarchar(MAX)	Yes
CategoriesPicture	varbinary(MAX)	Yes
Discontinued	bit	Yes
ProductID	int	Yes
ProductName	nvarchar(40)	Yes
QuantityPerUnit	nvarchar(20)	Yes
ReorderLevel	smallint	Yes
SuppliersAddress	nvarchar(60)	Yes
SuppliersCity	nvarchar(15)	Yes
SuppliersCompanyName	nvarchar(40)	Yes
SuppliersContactName	nvarchar(30)	Yes
SuppliersContactTitle	nvarchar(30)	Yes
SuppliersCountry	nvarchar(15)	Yes
SuppliersFax	nvarchar(24)	Yes
SuppliersHomePage	nvarchar(MAX)	Yes
SuppliersPhone	nvarchar(24)	Yes
SuppliersPostalCode	nvarchar(10)	Yes
SuppliersRegion	nvarchar(15)	Yes
SuppliersSupplierID	int	Yes
UnitPrice	decimal(19, 4)	Yes
UnitsInStock	smallint	Yes
UnitsOnOrder	smallint	Yes

Dim_Customers

Column Name	Data Type	Nullable
SurKey	int	No
Address	nvarchar(60)	Yes
City	nvarchar(15)	Yes
CompanyName	nvarchar(40)	Yes
ContactName	nvarchar(30)	Yes
ContactTitle	nvarchar(30)	Yes
Country	nvarchar(15)	Yes
CustomerID	nchar(5)	Yes
Fax	nvarchar(24)	Yes
Phone	nvarchar(24)	Yes
PostalCode	nvarchar(10)	Yes
Region	nvarchar(15)	Yes

Dim_Employees

Column Name	Data Type	Nullable
SurKey	int	No
Address	nvarchar(60)	Yes
BirthDate	datetime2(7)	Yes
City	nvarchar(15)	Yes
Country	nvarchar(15)	Yes
EmployeeID	int	Yes
Extension	nvarchar(4)	Yes
FirstName	nvarchar(10)	Yes
HireDate	datetime2(7)	Yes
HomePhone	nvarchar(24)	Yes
LastName	nvarchar(20)	Yes
Notes	nvarchar(MAX)	Yes
Photo	varbinary(MAX)	Yes
PhotoPath	nvarchar(255)	Yes
PostalCode	nvarchar(10)	Yes
Region	nvarchar(15)	Yes
Title	nvarchar(30)	Yes
TitleOfCourtesy	nvarchar(25)	Yes

Fact_OrderDetailsOrders

Column Name	Data Type	Nullable
SurKey	int	No
CustomersKey	int	No
EmployeesKey	int	No
JunkOrderDetailsOrdersKey	int	No
OrderDateDateKey	int	No
OrderDateTimeKey	int	No
ProductsKey	int	No
RequiredDateDateKey	int	No
RequiredDateTimeKey	int	No
ShippedDateDateKey	int	No
ShippedDateTimeKey	int	No
ShippersKey	int	No
Discount	real	Yes
Freight	decimal(19, 4)	Yes
OrderID	int	Yes
Quantity	smallint	Yes
UnitPrice	decimal(19, 4)	Yes

Dim_Shippers

Column Name	Data Type	Nullable
SurKey	int	No
CompanyName	nvarchar(40)	Yes
Phone	nvarchar(24)	Yes
ShipperID	int	Yes

Dim_Time

Column Name	Data Type	Nullable
SurKey	int	No
Hour	smallint	Yes
Minute	smallint	Yes
Second	smallint	Yes

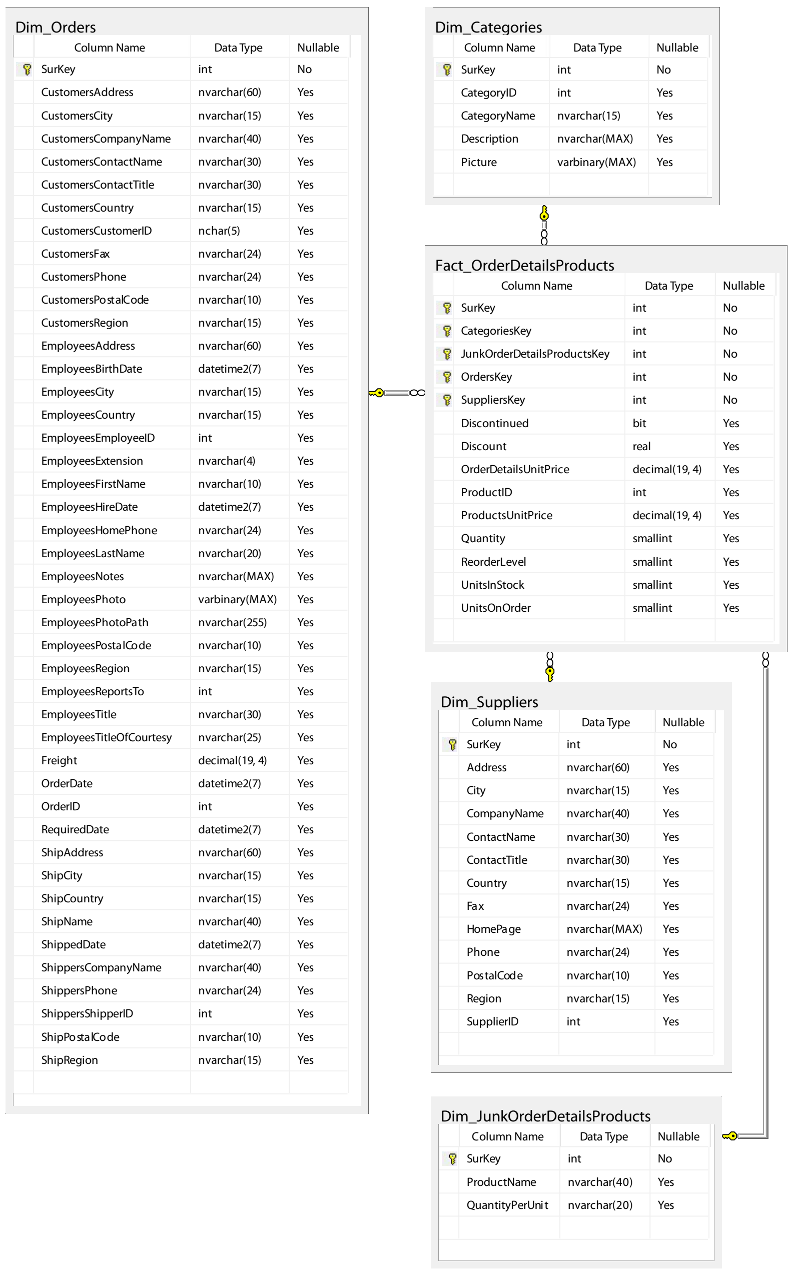
Dim_JunkOrderDetailsOrders

Column Name	Data Type	Nullable
SurKey	int	No
ShipAddress	nvarchar(60)	Yes
ShipCity	nvarchar(15)	Yes
ShipCountry	nvarchar(15)	Yes
ShipName	nvarchar(40)	Yes
ShipPostalCode	nvarchar(10)	Yes
ShipRegion	nvarchar(15)	Yes

Dim_Date

Column Name	Data Type	Nullable
SurKey	int	No
Day	smallint	Yes
FullDate	nchar(10)	Yes
Holiday	nchar(32)	Yes
IsHoliday	bit	Yes
Month	smallint	Yes
NameOfDay	nchar(10)	Yes
NameOfMonth	nchar(10)	Yes
Week	smallint	Yes
Year	smallint	Yes

Figure 26: Second result by DWStar for the Northwind OSS. A combination of `OrderDetails` and `Orders` was found to be the fact table in this case.



Dim_Orders

Column Name	Data Type	Nullable
SurKey	int	No
CustomersAddress	nvarchar(60)	Yes
CustomersCity	nvarchar(15)	Yes
CustomersCompanyName	nvarchar(40)	Yes
CustomersContactName	nvarchar(30)	Yes
CustomersContactTitle	nvarchar(30)	Yes
CustomersCountry	nvarchar(15)	Yes
CustomersCustomerID	nchar(5)	Yes
CustomersFax	nvarchar(24)	Yes
CustomersPhone	nvarchar(24)	Yes
CustomersPostalCode	nvarchar(10)	Yes
CustomersRegion	nvarchar(15)	Yes
EmployeesAddress	nvarchar(60)	Yes
EmployeesBirthDate	datetime2(7)	Yes
EmployeesCity	nvarchar(15)	Yes
EmployeesCountry	nvarchar(15)	Yes
EmployeesEmployeeID	int	Yes
EmployeesExtension	nvarchar(4)	Yes
EmployeesFirstName	nvarchar(10)	Yes
EmployeesHireDate	datetime2(7)	Yes
EmployeesHomePhone	nvarchar(24)	Yes
EmployeesLastName	nvarchar(20)	Yes
EmployeesNotes	nvarchar(MAX)	Yes
EmployeesPhoto	varbinary(MAX)	Yes
EmployeesPhotoPath	nvarchar(255)	Yes
EmployeesPostalCode	nvarchar(10)	Yes
EmployeesRegion	nvarchar(15)	Yes
EmployeesReportsTo	int	Yes
EmployeesTitle	nvarchar(30)	Yes
EmployeesTitleOfCourtesy	nvarchar(25)	Yes
Freight	decimal(19, 4)	Yes
OrderDate	datetime2(7)	Yes
OrderID	int	Yes
RequiredDate	datetime2(7)	Yes
ShipAddress	nvarchar(60)	Yes
ShipCity	nvarchar(15)	Yes
ShipCountry	nvarchar(15)	Yes
ShipName	nvarchar(40)	Yes
ShippedDate	datetime2(7)	Yes
ShippersCompanyName	nvarchar(40)	Yes
ShippersPhone	nvarchar(24)	Yes
ShippersShipperID	int	Yes
ShipPostalCode	nvarchar(10)	Yes
ShipRegion	nvarchar(15)	Yes

Dim_Categories

Column Name	Data Type	Nullable
SurKey	int	No
CategoryID	int	Yes
CategoryName	nvarchar(15)	Yes
Description	nvarchar(MAX)	Yes
Picture	varbinary(MAX)	Yes

Fact_OrderDetailsProducts

Column Name	Data Type	Nullable
SurKey	int	No
CategoriesKey	int	No
JunkOrderDetailsProductsKey	int	No
OrdersKey	int	No
SuppliersKey	int	No
Discontinued	bit	Yes
Discount	real	Yes
OrderDetailsUnitPrice	decimal(19, 4)	Yes
ProductID	int	Yes
ProductsUnitPrice	decimal(19, 4)	Yes
Quantity	smallint	Yes
ReorderLevel	smallint	Yes
UnitsInStock	smallint	Yes
UnitsOnOrder	smallint	Yes

Dim_Suppliers

Column Name	Data Type	Nullable
SurKey	int	No
Address	nvarchar(60)	Yes
City	nvarchar(15)	Yes
CompanyName	nvarchar(40)	Yes
ContactName	nvarchar(30)	Yes
ContactTitle	nvarchar(30)	Yes
Country	nvarchar(15)	Yes
Fax	nvarchar(24)	Yes
HomePage	nvarchar(MAX)	Yes
Phone	nvarchar(24)	Yes
PostalCode	nvarchar(10)	Yes
Region	nvarchar(15)	Yes
SupplierID	int	Yes

Dim_JunkOrderDetailsProducts

Column Name	Data Type	Nullable
SurKey	int	No
ProductName	nvarchar(40)	Yes
QuantityPerUnit	nvarchar(20)	Yes

Figure 27: Third result by DWStar for the Northwind OSS. A combination of `OrderDetails` and `Products` was found to be the fact table in this case.

5.2.3 Results

The results of this evaluation is a combination of the BI practitioner's feedback and our comments that reflect upon that feedback. The entirety of the feedback from the BI practitioner is available in Appendix E.

The BI practitioner provided both constructive criticism and positive feedback on the schemas. Before examining the feedback, we would like to address the following issue that was found by the BI practitioner: *"There is an issue with the design of 'Northwind-OrderDetailsProducts' schema. The fact design at this 'Northwind-OrderDetailsProducts' schema seems to be wrong."*. Currently DWStar provides a list of candidate star schemas, and this is primarily due to the fact that we do not expect that all candidate star schemas created by DWStar are useable. As an example, the third Northwind candidate that involved an `OrderDetailsProducts` fact table is a product of our module that combined the two tables: `Order Details` and `Products`. The `Products` table is, from a BI practitioner's perspective, clearly dimensional rather than factual, as the values in the columns are more or less constant and provide description of the products. The feedback from the BI practitioner reflects this aspect. He continues by saying: *"A few Product columns should be taken out and put into the junk dimension 'Dim_JunkOrderDetailsProducts'. The other schemas look quite fine."*. Here we would recommend discarding this candidate rather than repairing it since the other two candidates `OrderDetailsOrders` and `Orders` can be used in the DW instead.

There are several aspects to consider when creating a star schema, the BI practitioner reflected upon these:

Naming Convention

Question: *"Are both the tables and columns in the DW named appropriately?"*
Answer: *"Typically a DW has its own naming standard. The naming standard can be defined as a set of rules enforced during the data modeling process. Therefore it is not possible to say if the DW is named totally appropriate here. But the names seems reasonable without knowing any standard."*

Comments: The answer is justifiable given the information provided in the evaluation. We did not formally specify the applied naming convention, and the results reflects this. However, this problem is easily solved by changing the settings of the star refinement module `Naming Convention`, or implement a new module that specifies a naming convention, that is normally used by the BI practitioner, if the current settings are insufficient.

Data Type

Question: *"In general, have the data types been converted appropriately from source to DW?"*
Answer: *"Similar to the naming of column and tables, a DW also has a standard for its data types. If there is no enforced standard, then the 'source' column data types are used directly. Therefore the data types in the DW schemas seem to be appropriate."*

Comments: This is similar to the naming convention where it is possible to apply a standard through DWStar. Currently, this would be a modification of the `Create Script Generation` module in the generation phase which currently translates the OLE DB data types into corresponding SQL data types. The exact mapping between OLE DB and SQL data types is currently hard coded in the `Create Script Generation`, however, it would be beneficial to expose this mapping to the user, and allow him to redefine it.

Date and Time Dimensions

Question: *"Does the auto-generated date and time dimensions seem reasonable as a basis for customization?"*
Answer: *"The date and time dimensions should have a few extra columns in different scenarios. But in our cases they look reasonable."*
Comments: This highlights the usefulness of a customizable solution, as one might want to add extra columns to the date and time dimensions. E.g. financial calendar aligned columns, like fiscal month, fiscal week, etc.

Creating Fact Tables

Question: *"For the instances where it happens - does it seem reasonable to combine 2+ tables in the source to a single fact table?"*
Answer: *"In certain cases it has been shown ok to combine multiple tables into a single fact. However, I would always suggest that keep the 1-to-1 mapping for creating fact tables or in many cases breaking a source table into several fact tables due to certain requirements."*
Comments: As the BI practitioner mentions, there exists several ways of creating a fact table, either keeping the original source table, combining two or more source tables, or even *"breaking a source table into several fact tables"*. We have not come across a scenario where it was necessary to split a source table into multiple fact table and therefore have not made a module to find this case. We would argue that the FKlub case fits with the first scenario where we simply keep the original table, and the Northwind case fits the second scenario where we combine `Order Details` and `Orders` to provide the most appropriate fact table.

Creating Dimensions

Question: *"For the instances where it happens - does it seem reasonable to combine 2+ tables in the source to a single dimension?"*
Answer: *"Yes, in many cases, dimension tables seem to be a conformation of multiple source tables. A snowflake is only necessary if a dimension table becomes too large."*
Comments: Our current approach of flattening dimensions is common practice in a star schema, and the BI practitioner confirms that the result of this approach satisfies his expectation.

Nullability

Question: *"Everything which is not a surrogate key or references a surrogate key are not nullable - everything else is nullable. Does this seem reasonable?"*
Answer: *"In many DW design this is quite OK. It varies in different design principles. It will be cool if one can choose to 'configure' this before generating the schema."*
Comments: This matches with Kimball and Ross who writes: *"Null-valued measurements behave gracefully in fact tables. The aggregate functions (SUM, COUNT, MIN, MAX, and AVG) all do the 'right thing' with null facts. However, nulls must be avoided in the fact table's foreign keys because these nulls would automatically cause a referential integrity violation."* [30]. Currently we have no module that can configure the nullability of columns, but this could be solved by either implementing a new module that does so or improve the amount of user intervention, thereby allowing the user to modify the star model, specifically change the nullability of columns. This improved user intervention is discussed in Section 6.1.

Table 10: Specifications for the computer used in the experiments.

CPU	Intel i7-4500U CPU @ 1.80GHz
RAM	8.00 GB DDR3
Storage	256 GB SSD
Operating System	Windows 10 Home 64-bit

Candidate star schemas

Question: *"Are there any of the candidate star schemas that corresponds with your expected outcome?"*
Answer: *"The dimension tables are missing the [SCD] 'type-2' design where there are typically 'Valid_from' and 'Valid_to' dates. In addition, there can be a 'Transaction_Timestamp' column in the fact tables since the Date and Time keys are more for the business meaning. Despite of these, the basic design of the fact and dimension tables seem to be OK. [...]"*
Comments: The practitioner notices that no *Slowly Changing Dimensions* (SCD) versioning is applied to the candidate schemas, and the lack of a *"Transaction_Timestamp"*. In Section 6.4 we discuss how future work could look at including SCD *"type-2"* versioning. The *"Transaction_Timestamp"* refers to a column in the tables, that notes when the rows has been inserted into the DW. To remedy this, a module should be implemented to add this new column, and the generated SQL scripts should then insert the correct value into this column.

5.2.4 Conclusion

Although the evaluation has shown minor defects we argue that the proposed candidate star schemas still serve as a good starting point for a DW. Having said that, there is still room for improvement to modules and creation of new modules (e.g. handling of SCD type 2).

For the Northwind case we would argue that two of the three candidates can be discarded and the remaining star schema with the fact table `OrderDetailsOrders` can be chosen. `Orders` is not a desirable candidate due to the single usable measure `Freight`, which would not be able to provide much decision support, and `OrderDetailsProducts` is currently not desirable due to the faults pointed out by the BI practitioner. `OrderDetailsOrders` is the most appropriate candidate, as it is able to examine orders and has appropriate dimensions regarding: Products, product categories, suppliers, customers, employees, and more. We find that this serves as a good foundation for a DW.

5.3 Experiment: Runtime of DWStar

This experiment evaluates the runtime of DWStar and the execution time of the creation and population SQL scripts of DWStar which thereby also verify that automatic DW population is possible. This is done to verify that the execution times are within reason.

5.3.1 Methodology

The experiment is executed with the two OSSs defined in the previous experiment: The FKlub database, and the Northwind database. The specifications of the computer used for the experiment are shown in Table 10.

Both the source databases and the target databases were located on the same computer. The databases were located in an SQL Server 2014 RDBMS [23].

Table 11: Average runtimes of DWStar on the FKlub and Northwind databases.

	FKlub	**Northwind**
Phases	**Time (s:ms)**	**Time (s:ms)**
Metadata	1:058	1:417
Refinement	0:137	0:232
Star	0:063	1:645
Star refinement	0:033	0:068
Generation	0:950	1:335
Total	2:241	4:697

The experiment is separated into two parts which were deemed relevant for evaluating the runtime: (1) How long it takes DWStar to run through all phases, and (2) how long it takes to create the DW and populate it as well. Common for both parts is that the experiment is repeated three times to get a more accurate view of the average runtime. Between each experiment, both DWStar and SQL Server are restarted, as this will ensure that caches and other performance improving mechanisms does not have an impact on the results.

Part One:

DWStar is configured with the same modules as in the previous experiment (Section 5.2). The individual runtime of each phase is recorded.

Part Two:

For part two, we have a series of SQL statements that was generated in part one. These SQL statements are able to create and populate the resulting star schema candidates. The experiment is performed on the best candidate for each OSS: `Fact_Sale` from FKlub and `Fact_OrderDetailsOrders` from Northwind. We configure the date dimension to range from the `01/01/1996` up to and including the `31/12/2008`, as the data from Northwind and FKlub span from the year `1996` to `2008`. For each star schema, the amount of rows in the fact table is noted, to be able to better relate to whether the time used for population is justifiable.

5.3.2 Results

The results for both part one and part two of the experiment is given in the following sections.

Part One: Runtime of DWStar

The average runtime of each phase as well as the total runtime is illustrated in Table 11.

For the FKlub database, the total runtime was 2 second and 241 milliseconds while for the Northwind database, the total runtime was 4 seconds and 697 milliseconds. With these runtimes, it should be possible to use this program on regular basis and as a helping tool for constructing star schemas.

Part Two: Execution Time of SQL Scripts

The average runtimes for FKlub and Northwind can be found in Table 12. For each of these runtimes, $91,150$ rows were generated when populating the time and date dimensions tables.

Table 12: Average runtimes for creating and populating two star schemas.

	Creation Time (s:ms)	**Population Time (s:ms)**	**# of Rows (Fact table)**	**Total # of Rows**
FKlub (Fact_Sale)	0:041	10:285	390,239	483,814
Northwind (Fact_Order-DetailsOrders)	0:073	2:902	2082	93,502

5.3.3 Conclusion

Given the used OSSs, we have observed that no part take more than 11 seconds to execute. Even with roughly $390,000$ rows in a fact table, we see that the SQL statements are able to populate a star schema within a reasonable amount of time. Based on these results, we conclude that DWStar is able to generate and populate star schema candidates within a reasonable time span. As all the SQL scripts executed without any error, we also conclude that the generated SQL scripts were correctly structured.

Chapter 6

Discussion and Future Work

This chapter contains various discussion topics and directions for future work.

6.1 User Intervention

As mentioned throughout the report, the user is able to adjust several aspects of DWStar, whether it being establishing a configuration, applying settings to a module, implementing his own modules, or discard common models and star models between each phase.

To further allow for user intervention we would in a future version allow the user to modify both common models and star models between any of the phases. This would mean that the user can change the data type of columns, add relationships, create new tables, etc. and thereby be in full control of the outcome of DWStar. The principle behind this feature is very simplistic, but the implementation itself requires a graphical interface that allows the user to adjust the models and we simply valued this below any other feature. Additionally, we would like to implement a graphical interface, that allows the user to change settings of modules directly in the program, instead of having him edit a file in the filesystem, this was achieved with the 9th semester project [43] and could be applied here again.

6.2 ETL Tool Support

DWStar currently implements a simple ETL approach that uses SQL scripts to transfer data. This approach expects that all source data is accessible from the DW. All CSV files or data sources must be accessible, or else the approach cannot complete. This can be rectified by allowing access or importing the CSV files to the DW RDBMS. However, this restriction can be avoided by using ETL tools that can deal with OSSs that are located at different locations. The following frameworks/tools are all valid candidates, for which a generation module can be created, that enables the generation of ETL processes: SQL Server Integration Services [48], Kettle [49], pygrametl [50], or Biml [51].

6.3 Generation of OLAP Cubes

It is often seen that a DW is represented as OLAP cubes as a way to analyze the data within the DW. In order to do this, it is necessary to include the notion of hierarchies in the models and a way of detecting such hierarchies. The task of doing this depends on the format of the data. It is considerably harder to detect hierarchies in a denormalized database in relation to doing it on a normalized database as each table in a normalized could represent a different level of the hierarchy. Jensen et al. [7] provide a heuristic for discovering hierarchies and this heuristic could be used to create a relevant module. In short, if table t_1 is to be rolled-up to table t_2, these tables are said to be part of a hierarchy if every value in t_1 has a corresponding value in t_2.

6.4 Data Loading

A DW is often updated with new data from OSSs, during this process one often wish to maintain the data that is already present in the DW. The `DW Population` generation module is capable of creating the initial load of data, however, it currently simply overwrites the old data if one attempts to run the script again with new data. This behavior resembles SCD type 1, as the old values are overwritten. If one wishes for the history of rows to be retained, it is desirable to add SCD type 2 (or some other SCD types) to the `DW Population` and allow the user to choose the desired versioning technique. It could also be beneficial to add a property to the tables in a star model, which can describe how the dimensions (and possibly fact table) should be versioned. E.g. `Sales` and `Transactions` should use SCD type 1, whereas `Employees` should use SCD type 2. How to achieve SCD type 2 versioning with SQL scripts was investigated in our 9th semester project [43].

6.5 Additional Multidimensional Schemas

A star schema might not always be the proper representation of a DW. As it was said in the problem statement, we wish to create a solution that is customizable. The customizability is currently targeting the configuration and modules of DWStar. Instead we should look at introducing other DW schema designs (e.g. snowflake schemas and fact constellations). The resulting schemas could therefore be of different types, depending of which best fits the organization's requirements. However, it would complicate the creation of modules, as some modules might not function with all types of multidimensional schemas. E.g. a module that flattens a dimension hierarchy into a single table (as is the case with star schemas), should not be used when generating snowflake schemas.

6.6 Column Name Inference

For the experiment in Section 5.1 for inferring metadata, one notable point is that column names was not attempted to be inferred. While the refinement modules for column name inference are implemented, they are not accurate enough to justify the inclusion of them in the experiment. This was in part due to the results provided by DBpedia which are often so varying that it can be troublesome finding reliable results. E.g. a search for the word `Berlin` would return the top five results: `Berlin` (the capital of Germany), `1936 Summer Olympics` (held in Berlin), `Axis powers` (The "Rome-Berlin" Axis), `Humboldt University of Berlin` (university in Berlin), and `Berlin Wall`. Each of the results contain the word `Berlin` (although not all have `Berlin` in their name), however, many of these do not fit as a column name. Therefore, this refinement module was left out entirely from the experiment, and the column names were instead provided in the headers. This was advantageous, as other refinement modules depended on the column name, and the inaccuracy of the `Column Name Inference` module could have cascaded to those if it was included. We do not claim that the module is unusable, we instead delegate the work of improving the accuracy of it to future work, for example by investigating the results and tweaking the queries that are performed against DBpedia.

Inferring of column names is still relevant, whether it is done through a web service (like DBpedia), patterns, or another approach. The main goal of these kinds of modules is still to guess an appropriate column name, and thereby relieving the user of having to manually insert, if not all then some of, the names.

Chapter 7

Conclusion

This report investigated how to automatically construct and populate star schemas through a customizable process. Each property from the problem statement is now listed and evaluated as to whether DWStar contains the property.

1. *Automatic DW construction*. This property has been successfully implemented by defining five phases that each deal with their own area of concern, whether it being automatically inferring metadata from OSSs, or detecting fact tables and dimension tables. The implemented modules (and their heuristics) further support the automatic aspect and are the reason DWStar can create the appropriate star schemas without user involvement. This claim was further supported by the evaluation in Section 5.2 where the BI practitioner's feedback to the results of DWStar was generally positive.
2. *Customize DW construction*. This property has been attained since each phase of DWStar can be customized to fit the user's needs with the usage of configurations and settings.
3. *Metadata inference*. This property was implemented to a satisfactory degree through the metadata phase and refinement phase which were able to extract information from the OSSs, and infer the necessary metadata. The existence of this property was verified empirically through the experiment in Section 5.1 which showed that the CSV metadata module and refinement modules achieved a satisfactory result with the following accuracy: 97% for data types, 68% for uniqueness, 45% for nullability, 77% for primary keys, and 77% for foreign keys.
4. *Automatic DW population*. This property was attained through the `DW Population` generation module for DWStar which generates a series of SQL scripts that extracts data from the OSSs and loads it into the DW(s). In connection to the experiment in Section 5.2, it was verified that this generation module works in a real-life case. It was also verified that it ran within a reasonable amount of time, as the total elapsed time for transferring and generating data took approximately 10 seconds in the worst case with approximately 483,000 rows.

To summarize, we conclude that we have successfully described how to construct a system which implements the properties in the problem statement and is thereby capable of automatically producing star schemas in a customizable manner, and afterwards aid in populating these with existing data. DWStar can then benefit BI practitioners, as they are able to skip trivial parts of the DW design and instead focus on more important parts of building a successful DW. As we have created an automatic solution, this could also help people that are less experienced in BI practices, as they are able to merely specify an OSS and generate a set of candidate star schemas. Using these star schemas, they could transfer data into a DW and immediately use the available data.

Bibliography

[1] Ralph Kimball and Margy Ross. *The data warehouse toolkit: the complete guide to dimensional modeling*. John Wiley & Sons, 2011.

[2] Lilia Muñoz, Jose-Norberto Mazón, and Juan Trujillo. Automatic generation of etl processes from conceptual models. In *Proceedings of the ACM twelfth international workshop on Data warehousing and OLAP*, pages 33–40. ACM, 2009.

[3] Il Yeol Song, Ritu Khare, and Bing Dai. Samstar: a semi-automated lexical method for generating star schemas from an entity-relationship diagram. In *Proceedings of the ACM tenth international workshop on Data warehousing and OLAP*, pages 9–16. ACM, 2007.

[4] Mary Elizabeth Jones and Il-Yeol Song. Dimensional modeling: identifying, classifying & applying patterns. In *Proceedings of the 8th ACM international workshop on Data warehousing and OLAP*, pages 29–38. ACM, 2005.

[5] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[6] Ralph Kimball and Margy Ross. *The data warehouse toolkit: The complete guide to dimensional modeling*. John Wiley & Sons, 2002.

[7] Mikael R Jensen, Thomas Holmgren, and Torben Bach Pedersen. Discovering multidimensional structure in relational data. In *International Conference on Data Warehousing and Knowledge Discovery*, pages 138–148. Springer, 2004.

[8] Cassandra Phipps and Karen C Davis. Automating data warehouse conceptual schema design and evaluation. In *DMDW*, volume 2, pages 23–32. Citeseer, 2002.

[9] Carsten Sapia, Markus Blaschka, Gabriele Höfling, and Barbara Dinter. Extending the e/r model for the multidimensional paradigm. In *International Conference on Conceptual Modeling*, pages 105–116. Springer, 1998.

[10] Oscar Romero and Alberto Abelló. A framework for multidimensional design of data warehouses from ontologies. *Data & Knowledge Engineering*, 69(11):1138–1157, 2010.

[11] Luca Cabibbo and Riccardo Torlone. A logical approach to multidimensional databases. In *International Conference on Extending Database Technology*, pages 183–197. Springer, 1998.

[12] Andrea Carmè, Jose-Norberto Mazón, and Stefano Rizzi. A model-driven heuristic approach for detecting multidimensional facts in relational data sources. *Data Warehousing and Knowledge Discovery*, pages 13–24, 2010.

[13] Daniel L Moody and Mark AR Kortink. From enterprise models to dimensional models: a methodology for data warehouse and data mart design. In *DMDW*, page 5, 2000.

[14] Nicolas Prat, Jacky Akoka, and Isabelle Comyn-Wattiau. A uml-based data warehouse design method. *Decision support systems*, 42(3):1449–1473, 2006.

[15] Ralph Kimball. *The data warehouse lifecycle toolkit*. John Wiley & Sons, 2008.

[16] Fá Rilston Silva Paim and Jaelson Freire Brelaz de Castro. Dwarf: An approach for requirements definition and management of data warehouse systems. In *Requirements Engineering Conference, 2003. Proceedings. 11th IEEE International*, pages 75–84. IEEE, 2003.

[17] Naveen Prakash and Anjana Gosain. Requirements driven data warehouse development. In *CAiSE Short Paper Proceedings*, volume 252, 2003.

[18] Angela Bonifati, Fabiano Cattaneo, Stefano Ceri, Alfonso Fuggetta, Stefano Paraboschi, et al. Designing data marts for data warehouses. *ACM transactions on software engineering and methodology*, 10(4):452–483, 2001.

[19] Michael Boehnlein and Achim Ulbrich-vom Ende. Deriving initial data warehouse structures from the conceptual data models of the underlying operational information systems. In *Proceedings of the 2nd ACM international workshop on Data warehousing and OLAP*, pages 15–21. ACM, 1999.

[20] Jamel Feki. An automatic data warehouse conceptual design approach. In *Encyclopedia of Data Warehousing and Mining, Second Edition*, pages 110–119. IGI Global, 2009.

[21] Matteo Golfarelli, Dario Maio, and Stefano Rizzi. Conceptual design of data warehouses from e/r schemes. In *System Sciences, 1998., Proceedings of the Thirty-First Hawaii International Conference on*, volume 7, pages 334–343. IEEE, 1998.

[22] Mikael R Jensen, Thomas H Moller, and Torben Bach Pedersen. Specifying olap cubes on xml data. In *Scientific and Statistical Database Management, 2001. SSDBM 2001. Proceedings. Thirteenth International Conference on*, pages 101–112. IEEE, 2001.

[23] SQL Server 2016, apr 2017. URL `https://www.microsoft.com/en-gb/sql-server/sql-server-2016`.

[24] Information Schema Views (Transact-SQL), mar 2017. URL `https://msdn.microsoft.com/en-us/library/ms186778.aspx`.

[25] Dowming Yeh, Yuwen Li, and William Chu. Extracting entity-relationship diagram from a table-based legacy database. *Journal of Systems and Software*, 81(5):764–771, 2008.

[26] Christian S. Jensen, Torben Bach Pedersen, and Christian Thomsen. *Multidimensional Databases and Data Warehousing*. Morgan & Claypool, 2010.

[27] About | DBpedia, mar 2017. URL `http://wiki.dbpedia.org/about`.

[28] Wikipedia:About, mar 2017. URL `https://en.wikipedia.org/wiki/Wikipedia:About`.

[29] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[30] Ralph Kimball and Margy Ross. *The data warehouse toolkit: The definitive guide to dimensional modeling*. John Wiley & Sons, 2013.

[31] Zhengxin Chen. *Intelligent Data Warehousing: From data preparation to data mining*. CRC press, 2001.

[32] PostgreSQL, apr 2017. URL `https://www.postgresql.org/`.

[33] Introduction to WPF in Visual Studio 2015, may 2017. URL `https://msdn.microsoft.com/en-us/library/aa970268(v=vs.110).aspx`.

[34] Material Design In XAML, may 2017. URL `http://materialdesigninxaml.net/`.

[35] Microsoft OLE DB, mar 2017. URL `https://msdn.microsoft.com/en-us/library/ms722784(v=vs.85).aspx`.

[36] Type Indicators in a Microsoft SQL Server Provider, apr 2017. URL `https://msdn.microsoft.com/en-us/library/ms714373(v=vs.85).aspx`.

[37] Type Indicators in an ANSI SQL Provider, apr 2017. URL `https://msdn.microsoft.com/en-us/library/ms725457(v=vs.85).aspx`.

[38] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[39] EnglishPluralizationService Class, may 2017. URL `https://msdn.microsoft.com/en-us/library/system.data.entity.infrastructure.pluralization.englishpluralizationservice(v=vs.113).aspx`.

[40] Overview (SMO), apr 2017. URL `https://docs.microsoft.com/en-us/sql/relational-databases/server-management-objects-smo/overview-smo`.

[41] TRUNCATE TABLE (Transact-SQL), may 2017. URL `https://docs.microsoft.com/en-us/sql/t-sql/statements/truncate-table-transact-sql`.

[42] Linked Servers (Database Engine), apr 2017. URL https://docs.microsoft.com/en-us/sql/relational-databases/linked-servers/linked-servers-database-engine.

[43] Jacob B. Hansen, Michael Tarp, and Søren Jensen. MainArchive: Maintained Archive. Technical report, Aalborg University, 2016.

[44] BULK INSERT (Transact-SQL), apr 2017. URL https://docs.microsoft.com/en-us/sql/t-sql/statements/bulk-insert-transact-sql.

[45] SQL Server Management Studio (SSMS), may 2017. URL https://docs.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms.

[46] Northwind Sample Database, apr 2017. URL http://www.wilsonmar.com/northwind.htm.

[47] AdventureWorks Sample Databases, may 2017. URL https://msdn.microsoft.com/en-us/library/ms124501(v=sql.100).aspx.

[48] SQL Server Integration Services (SSIS), may 2017. URL https://docs.microsoft.com/en-us/sql/integration-services/sql-server-integration-services.

[49] Data Integration - Kettle, may 2017. URL http://community.pentaho.com/projects/data-integration/.

[50] ETL programming in Python, may 2017. URL http://www.pygrametl.org/.

[51] Varigence. Business Intelligence Markup Language, may 2017. URL https://varigence.com/biml.

[52] OleDbType Enumeration, may 2017. URL https://msdn.microsoft.com/en-us/library/system.data.oledb.oledbtype(v=vs.110).aspx.

[53] ntext, text, and image (Transact-SQL), may 2017. URL https://docs.microsoft.com/en-us/sql/t-sql/data-types/ntext-text-and-image-transact-sql.

Appendix A

Summarization of Modules

This appendix serves as a section to give the reader a short summary of all of the available modules across all phases. Table 13 details each module with the phase it belongs to, its name, a short description, and the page it is defined on.

Table 13: Name and description of modules and at what page they are described on.

Phase	Module Name	Module Description	Page
Metadata	SQL Meta-data	Retrieves table names, column names, ordinals, data types, nullability, uniqueness, primary keys, and foreign keys from an SQL server.	17
Metadata	CSV Meta-data	Retrieves tables names, column names (if headers are present), data types, and ordinals from CSV files.	17
Refinement	Cardinality	Infers the cardinality of table relationships based on data samples.	18
Refinement	Column Name Inference	Infers the name of a column, by looking up sampled data on DBpedia and find most commonly occurring term.	19
Refinement	Column Name Pattern	Infers the name of a column, by looking for predefined patterns in data samples.	20
Refinement	Detect Data Type	Infers the data type of a column by finding the most fitting data type for a data sample.	20
Refinement	Find Not Nullable	Infers the nullability of a column by checking for null values in a data sample.	20
Refinement	Find Unique Columns	Infers the uniqueness of a column by checking for repeating values in a data sample.	21
Refinement	Find Primary Keys	Infers primary key candidates by looking at the nullability and uniqueness of columns.	21
Refinement	Primary Key Naming Pattern	Infers primary key candidates based on the similarity between the column name and the table name, and whether the column contains any of the keywords: `ID`, `Key`, `No`, and `Ref`.	22
Refinement	Find Relationships From Names	Infers relationships between tables based on the names of the columns. If the columns contains the keywords: `ID`, `Key`, `No`, and `Ref` they are likely part of a relationship.	22

Star	Combine Tables	Combines two tables in the attempt to create a more appropriate fact table.	31
Star	Fact More Rows	Applies confidence on the likelihood of a table being a fact table based on the assumption that a table with a large amount of rows compared to other tables is more likely to be a fact table.	31
Star	Fact More Columns	Applies confidence on the likelihood of a table being a fact table based on the amount of numeric columns.	32
Star	Usually many-to-one	Applies confidence on the likelihood of a table being a fact table based on how many many-to-one connections it has. More many-to-one connections provides higher confidence.	33
Star	Flatten Dimensions	All tables with a one-to-many connection to the fact table are recognized as a dimension. Each table with a one-to-many or one-to-one connection to that dimension table is then combined with it. This is done with both direct or indirect connections.	33
Star Refinement	Naming Convention	Applies naming conventions to fact tables and dimension tables in order to fit the names to an organization's standards.	35
Star Refinement	Extend Data Type Length	Extends the length of a column's data type.	35
Star Refinement	Add Surrogate Key	Adds surrogate keys to each dimension table.	35
Star Refinement	Add Date and Time Dimensions	Adds a time dimension table and a date dimension table to a star model, if it does not already exist.	36
Star Refinement	Add Junk Dimension	Moves the descriptive columns from the fact table to a junk dimension.	36
Star Refinement	Order Columns	Changes ordinal of columns such that primary keys come first, and all other columns are alphabetically sorted.	37
Generation	DW Population	Generates SQL scripts that transfers data from OSSs to the DW, and also generates data for any new surrogate key, time dimension table, and date dimension table.	37
Generation	Create Script Generation	Generates SQL scripts that creates the DW.	37

Appendix B

Date and Time SQL Scripts

This appendix contains the scripts for generating data for the `date` dimension and the `time` dimension. The script for generating dates is shown in Listing B.1. The script for generating times is shown in Listing B.2.

Listing B.1: SQL Script for generating data for a date dimension.

```sql
BEGIN TRANSACTION;
SET LANGUAGE british;

DECLARE @StartDate DATETIME = '01-01-2017' -- Adjust to fit your data
DECLARE @EndDate DATETIME = '31-12-2017' -- Adjust to fit your data

DECLARE @CurrentDate AS DATETIME = @StartDate

DELETE FROM [dbo].[Dim_Date];

INSERT INTO [dbo].[Dim_Date] VALUES ('Unknown', 00, 'Unknown', 00, 00, 'Unknown', 0000, 0,
    NULL); -- Dummy date

-- Populate dates
WHILE @CurrentDate <= @EndDate
BEGIN
    INSERT INTO [dbo].[Dim_Date]
    SELECT
       CONVERT(char(10), @CurrentDate, 103), -- FullDate
       DATEPART(DD, @CurrentDate), -- Day
       DATENAME(DW, @CurrentDate), -- NameOfDay
       DATEPART(WW, @CurrentDate), -- Week
       DATEPART(MM, @CurrentDate), -- Month
       DATENAME(MM, @CurrentDate), -- NameOfMonth
       DATEPART(YY, @CurrentDate), -- Year
       0, -- IsHoliday
       NULL -- Holiday

    SET @CurrentDate = DATEADD(DD, 1, @CurrentDate) -- Increment date
END

-- Set holidays
UPDATE [dbo].[Dim_Date]
    SET Holiday = 'Easter' WHERE [Month] = 4 AND [Day] = 21;

UPDATE [dbo].[Dim_Date]
    SET IsHoliday = 0 WHERE Holiday IS NULL;
UPDATE [dbo].[Dim_Date]
    SET IsHoliday = 1 WHERE Holiday IS NOT NULL;
COMMIT TRANSACTION;
GO
```

Listing B.2: SQL Script for generating data for a time dimension.

```sql
BEGIN TRANSACTION;
SET LANGUAGE british;

DECLARE @StartDate DATETIME2 = '01/01/2017'
DECLARE @EndDate DATETIME2 = '01/01/2017 23:59:59'

DECLARE @CurrentDate AS DATETIME2 = @StartDate

DELETE FROM [dbo].[Dim_Time];

-- Populate times
WHILE @CurrentDate <= @EndDate
BEGIN
    INSERT INTO [dbo].[Dim_Time]
    SELECT DATEPART(HH, @CurrentDate),
          DATEPART(MI, @CurrentDate),
          DATEPART(SS, @CurrentDate)

    SET @CurrentDate = DATEADD(SS, 1, @CurrentDate) -- Increment seconds
END
COMMIT TRANSACTION;
GO
```

Appendix C

Equivalent Data Types Between SQL Server and OLE DB

This appendix goes into more detail about a part of the methodology of the experiment in Section 5.1. On the background of the descriptions that Microsoft gives of OLE DB data types [52], we have composed Table 14. This table shows the equivalences between the data types found in Northwind (SQL Server data types) and the data types in our common model (OLE DB data types).

Remarks

It is worth mentioning that the two data types `ntext` and `image` (which are used in Northwind), are no longer advised to be used in newer SQL Server databases, and should therefore be converted to `nvarchar` or `varbinary` [53].

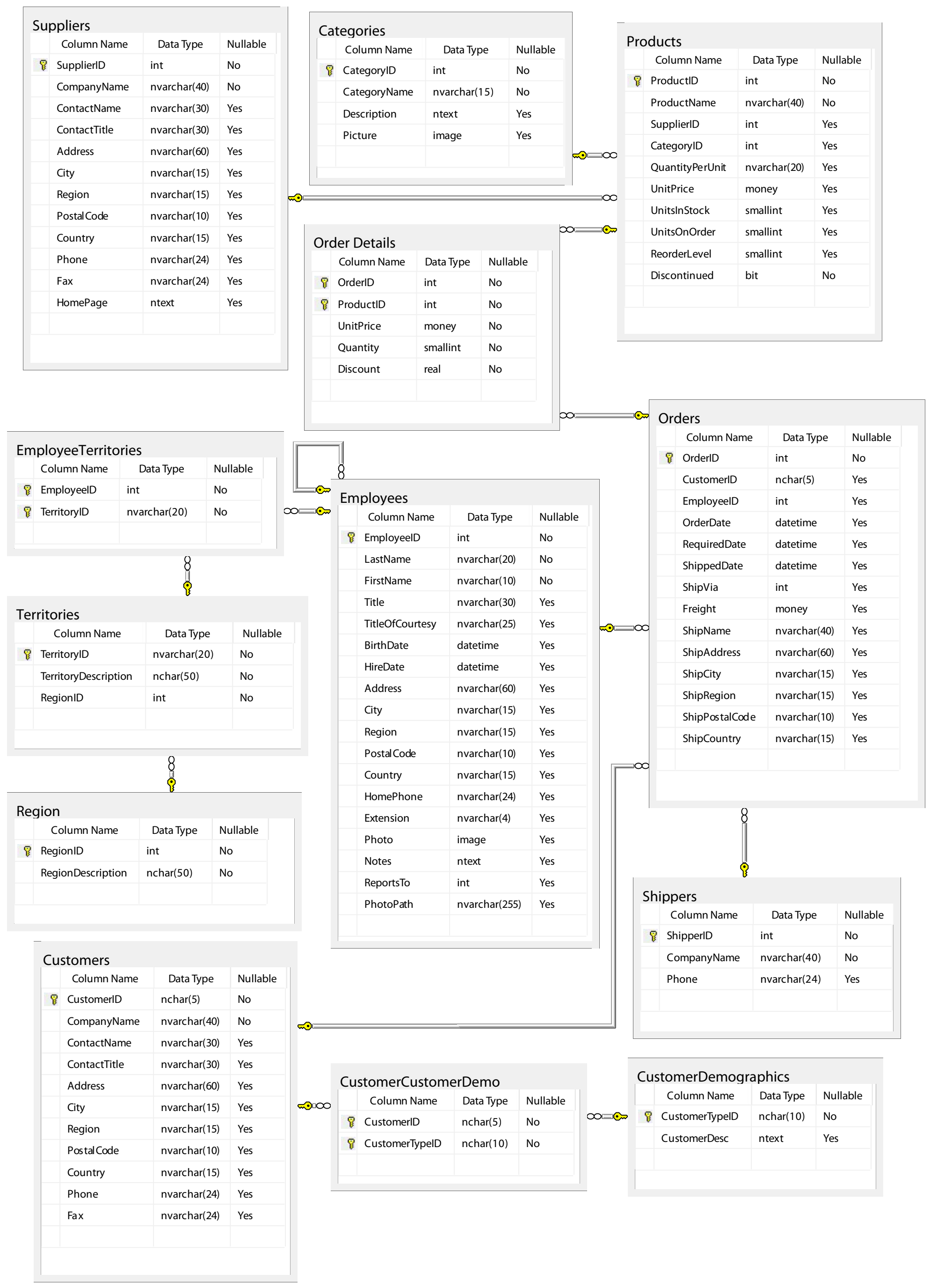
Table 14: Equivalence between SQL data types and OLE DB data types.

SQL Server data type	**OLE DB data type**
int	Integer
smallint	Integer
nvarchar	VarWChar
nchar	VarWChar
ntext	VarWChar
image	VarWChar
datetime2	DBTimeStamp
real	Double
money	Double
bit	Boolean

Appendix D

Results of Infer Missing Metadata Experiment

This section shows the full schema diagram of Northwind, which includes data types and nullability as well as the common model produced by DWStar. Northwind is shown in Figure 28. The common model exists internally within DWStar, so to retrieve a schema diagram, the common model has been parsed into SQL Server Management Studio to retrieve a schema diagram. DWStar uses OLE DB Types, but to be able to use SQL Server Management Studio, SQL types were needed instead. To remedy this, all OLE DB data types were converted to the equivalent SQL type, as defined in Appendix C. The resulting schema diagram is shown in Figure 29.



Suppliers

Column Name	Data Type	Nullable
SupplierID	int	No
CompanyName	nvarchar(40)	No
ContactName	nvarchar(30)	Yes
ContactTitle	nvarchar(30)	Yes
Address	nvarchar(60)	Yes
City	nvarchar(15)	Yes
Region	nvarchar(15)	Yes
PostalCode	nvarchar(10)	Yes
Country	nvarchar(15)	Yes
Phone	nvarchar(24)	Yes
Fax	nvarchar(24)	Yes
HomePage	ntext	Yes

Categories

Column Name	Data Type	Nullable
CategoryID	int	No
CategoryName	nvarchar(15)	No
Description	ntext	Yes
Picture	image	Yes

Products

Column Name	Data Type	Nullable
ProductID	int	No
ProductName	nvarchar(40)	No
SupplierID	int	Yes
CategoryID	int	Yes
QuantityPerUnit	nvarchar(20)	Yes
UnitPrice	money	Yes
UnitsInStock	smallint	Yes
UnitsOnOrder	smallint	Yes
ReorderLevel	smallint	Yes
Discontinued	bit	No

Order Details

Column Name	Data Type	Nullable
OrderID	int	No
ProductID	int	No
UnitPrice	money	No
Quantity	smallint	No
Discount	real	No

Orders

Column Name	Data Type	Nullable
OrderID	int	No
CustomerID	nchar(5)	Yes
EmployeeID	int	Yes
OrderDate	datetime	Yes
RequiredDate	datetime	Yes
ShippedDate	datetime	Yes
ShipVia	int	Yes
Freight	money	Yes
ShipName	nvarchar(40)	Yes
ShipAddress	nvarchar(60)	Yes
ShipCity	nvarchar(15)	Yes
ShipRegion	nvarchar(15)	Yes
ShipPostalCode	nvarchar(10)	Yes
ShipCountry	nvarchar(15)	Yes

EmployeeTerritories

Column Name	Data Type	Nullable
EmployeeID	int	No
TerritoryID	nvarchar(20)	No

Employees

Column Name	Data Type	Nullable
EmployeeID	int	No
LastName	nvarchar(20)	No
FirstName	nvarchar(10)	No
Title	nvarchar(30)	Yes
TitleOfCourtesy	nvarchar(25)	Yes
BirthDate	datetime	Yes
HireDate	datetime	Yes
Address	nvarchar(60)	Yes
City	nvarchar(15)	Yes
Region	nvarchar(15)	Yes
PostalCode	nvarchar(10)	Yes
Country	nvarchar(15)	Yes
HomePhone	nvarchar(24)	Yes
Extension	nvarchar(4)	Yes
Photo	image	Yes
Notes	ntext	Yes
ReportsTo	int	Yes
PhotoPath	nvarchar(255)	Yes

Territories

Column Name	Data Type	Nullable
TerritoryID	nvarchar(20)	No
TerritoryDescription	nchar(50)	No
RegionID	int	No

Region

Column Name	Data Type	Nullable
RegionID	int	No
RegionDescription	nchar(50)	No

Shippers

Column Name	Data Type	Nullable
ShipperID	int	No
CompanyName	nvarchar(40)	No
Phone	nvarchar(24)	Yes

Customers

Column Name	Data Type	Nullable
CustomerID	nchar(5)	No
CompanyName	nvarchar(40)	No
ContactName	nvarchar(30)	Yes
ContactTitle	nvarchar(30)	Yes
Address	nvarchar(60)	Yes
City	nvarchar(15)	Yes
Region	nvarchar(15)	Yes
PostalCode	nvarchar(10)	Yes
Country	nvarchar(15)	Yes
Phone	nvarchar(24)	Yes
Fax	nvarchar(24)	Yes

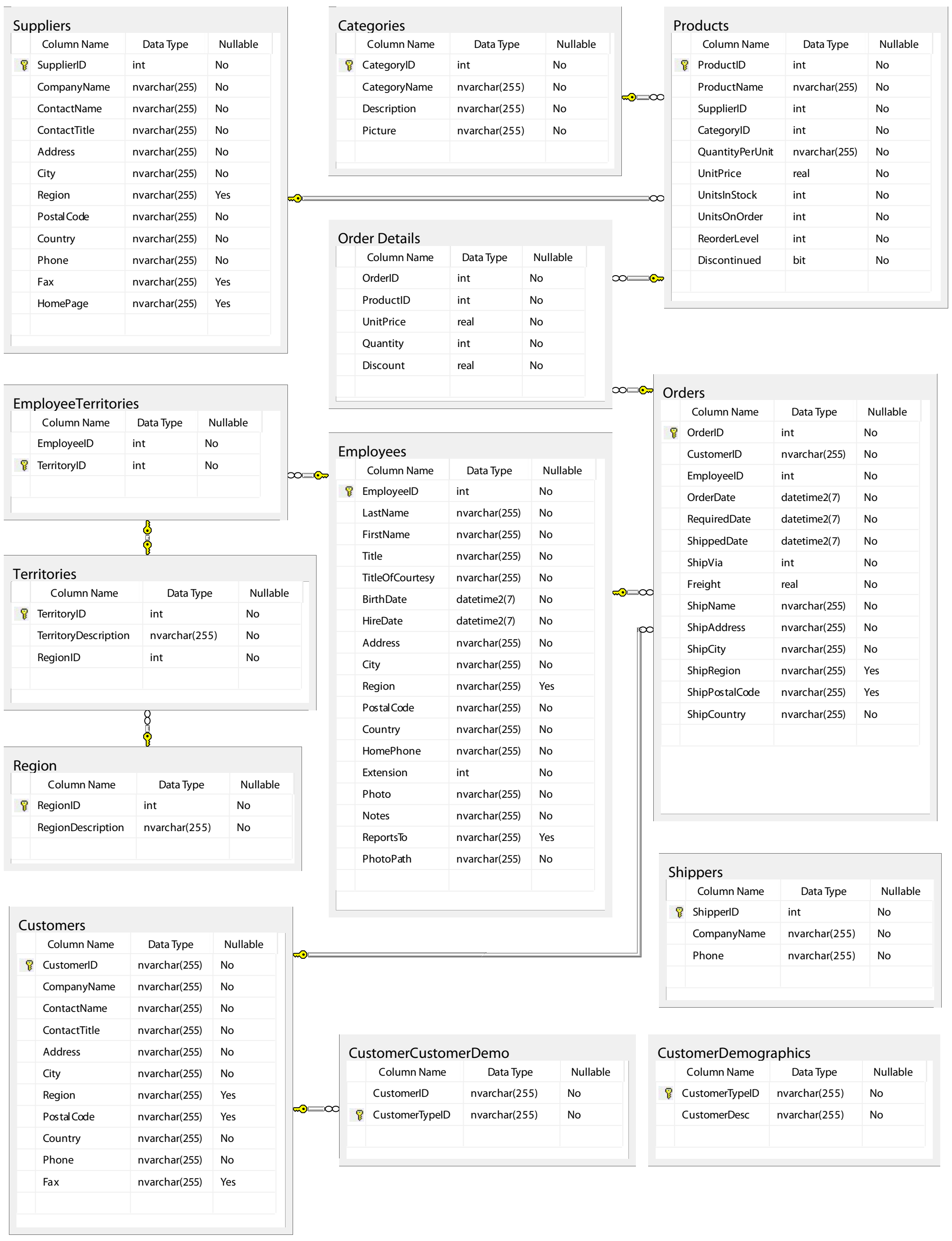
CustomerCustomerDemo

Column Name	Data Type	Nullable
CustomerID	nchar(5)	No
CustomerTypeID	nchar(10)	No

CustomerDemographics

Column Name	Data Type	Nullable
CustomerTypeID	nchar(10)	No
CustomerDesc	ntext	Yes

Figure 28: Schema diagram of Northwind including table names, columns names, relationships, primary keys, data types, and nullability.



Suppliers

Column Name	Data Type	Nullable
SupplierID	int	No
CompanyName	nvarchar(255)	No
ContactName	nvarchar(255)	No
ContactTitle	nvarchar(255)	No
Address	nvarchar(255)	No
City	nvarchar(255)	No
Region	nvarchar(255)	Yes
PostalCode	nvarchar(255)	No
Country	nvarchar(255)	No
Phone	nvarchar(255)	No
Fax	nvarchar(255)	Yes
HomePage	nvarchar(255)	Yes

Categories

Column Name	Data Type	Nullable
CategoryID	int	No
CategoryName	nvarchar(255)	No
Description	nvarchar(255)	No
Picture	nvarchar(255)	No

Products

Column Name	Data Type	Nullable
ProductID	int	No
ProductName	nvarchar(255)	No
SupplierID	int	No
CategoryID	int	No
QuantityPerUnit	nvarchar(255)	No
UnitPrice	real	No
UnitsInStock	int	No
UnitsOnOrder	int	No
ReorderLevel	int	No
Discontinued	bit	No

Order Details

Column Name	Data Type	Nullable
OrderID	int	No
ProductID	int	No
UnitPrice	real	No
Quantity	int	No
Discount	real	No

EmployeeTerritories

Column Name	Data Type	Nullable
EmployeeID	int	No
TerritoryID	int	No

Orders

Column Name	Data Type	Nullable
OrderID	int	No
CustomerID	nvarchar(255)	No
EmployeeID	int	No
OrderDate	datetime2(7)	No
RequiredDate	datetime2(7)	No
ShippedDate	datetime2(7)	No
ShipVia	int	No
Freight	real	No
ShipName	nvarchar(255)	No
ShipAddress	nvarchar(255)	No
ShipCity	nvarchar(255)	No
ShipRegion	nvarchar(255)	Yes
ShipPostalCode	nvarchar(255)	Yes
ShipCountry	nvarchar(255)	No

Employees

Column Name	Data Type	Nullable
EmployeeID	int	No
LastName	nvarchar(255)	No
FirstName	nvarchar(255)	No
Title	nvarchar(255)	No
TitleOfCourtesy	nvarchar(255)	No
BirthDate	datetime2(7)	No
HireDate	datetime2(7)	No
Address	nvarchar(255)	No
City	nvarchar(255)	No
Region	nvarchar(255)	Yes
PostalCode	nvarchar(255)	No
Country	nvarchar(255)	No
HomePhone	nvarchar(255)	No
Extension	int	No
Photo	nvarchar(255)	No
Notes	nvarchar(255)	No
ReportsTo	nvarchar(255)	Yes
PhotoPath	nvarchar(255)	No

Territories

Column Name	Data Type	Nullable
TerritoryID	int	No
TerritoryDescription	nvarchar(255)	No
RegionID	int	No

Region

Column Name	Data Type	Nullable
RegionID	int	No
RegionDescription	nvarchar(255)	No

Shippers

Column Name	Data Type	Nullable
ShipperID	int	No
CompanyName	nvarchar(255)	No
Phone	nvarchar(255)	No

Customers

Column Name	Data Type	Nullable
CustomerID	nvarchar(255)	No
CompanyName	nvarchar(255)	No
ContactName	nvarchar(255)	No
ContactTitle	nvarchar(255)	No
Address	nvarchar(255)	No
City	nvarchar(255)	No
Region	nvarchar(255)	Yes
PostalCode	nvarchar(255)	Yes
Country	nvarchar(255)	No
Phone	nvarchar(255)	No
Fax	nvarchar(255)	Yes

CustomerCustomerDemo

Column Name	Data Type	Nullable
CustomerID	nvarchar(255)	No
CustomerTypeID	nvarchar(255)	No

CustomerDemographics

Column Name	Data Type	Nullable
CustomerTypeID	nvarchar(255)	No
CustomerDesc	nvarchar(255)	No

Figure 29: Schema diagram of common model produced by DWStar.

Appendix E

BI Practitioner Feedback

This appendix contains the feedback from the BI practitioner in its entirety, which was used in the evaluation in Section 5.2. Below are the questions that was put forth as guidelines, and answers that the BI practitioner responded with.

Question: Are there any of the candidate star schemas that corresponds with your expected outcome?
Answer: The dimension tables are missing the “type-2” design where there are typically “Valid_from” and “Valid_to” dates. In addition, there can be a “Transaction_Timestamp” column in the fact tables since the Date and Time keys are more for the business meaning. Despite of these, the basic design of the fact and dimension tables seem to be OK. There is an issue with the design of “Northwind-OrderDetailsProducts” schema. The fact design at this “Northwind-OrderDetailsProducts” schema seems to be wrong. A few Product columns should be taken out and put into the junk dimension “Dim_Junk-OrderDetailsProducts”. The other schemas look quite fine.

Question: Does the choice of fact table seem reasonable?
Answer: Yes, the choice of the fact table is reasonable.

Question: Do the columns included in the fact table seem reasonable?
Answer: Yes, the columns seem reasonable.

Question: Do the choices of dimensions seem reasonable?
Answer: Most of the dimensions seem ok. The “junk” dimensions are mostly correct despite the one in “Northwind-OrderDetailsProducts”

Question: Are both the tables and columns in the DW named appropriately?
Answer: Typically a DW has its own naming standard. The naming standard can be defined as a set of rules enforced during the data modeling process. Therefor it is not possible to say if the DW is named totally appropriate here. But the names seems reasonable without knowing any standard.

Question: In general, have the data types been converted appropriately from source to DW?
Answer: Similar to the naming of column and tables, a DW also has a standard for its data types. If there is no enforced standard, then the “souce” column data types are used directly. Therefore the data types in the DW schemas seem to be appropriate.

Question: Do the auto-generated date and time dimensions seem reasonable as a basis for customization?

Answer: The date and time dimensions should have a few extra columns in different scenarios. But in our cases they look reasonable.

Question: For the instances where it happens - does it seem reasonable to combine 2+ tables in the source to a single fact table?
Answer: In certain case it has been shown ok to combine multiple tables in to a single fact. However, I would always suggest that keep the 1-to-1 mapping for creating fact tables or in many cases breaking a source table into several fact tables due to certain requirements.

Question: For the instances where it happens - does it seem reasonable to combine 2+ tables in the source to a single dimension?
Answer: Yes, in many cases, dimension tables seem to be a conformation of multiple source tables. A snowflake is only necessary if a dimension table becomes too large.

Question: Everything which is not a surrogate key or references it is not nullable - everything else is nullable. Does this seem reasonable?
Answer: In many DW design this is quite OK. It varies in different design principles. It will be cool if one can choose to "configure" this before generating the schema.

Question: Do the created junk dimensions seem reasonable?
Answer: Most of them are ok. I have an issue with the dimension "Dim_JunkOrderDetailsProducts" in the schema "Northwind-OrderDetailsProducts". I believe this should have been a "Product" dimension. But then we are missing several product columns (ProductID, CategoriesKey, SuppliersKey, Discontinued, osv.....) and these product columns are actually in the fact table.