MIRABEL DW: Managing Complex Energy Data in a Smart Grid

Laurynas Šikšnys(✉), Christian Thomsen, and Torben Bach Pedersen

Department of Computer Science, Aalborg University,
Aalborg, Denmark
{siksnys,chr,tbp}@cs.aau.dk

Abstract. In the MIRABEL project, a data management system for a *smart grid* is developed to enable smarter scheduling of energy consumption such that, e.g., charging of car batteries is done during night when there is an overcapacity of *green energy* from windmills etc. Energy can then be requested by means of *flex-offers* which define flexibility with respect to time, amount, and/or price. In this paper, we describe MIRABEL DW, a data warehouse (DW) for the management of the large amounts of complex energy data in MIRABEL. We present a unified schema that can manage data both at the level of the entire electricity network and the level of individual nodes, such as a single consumer node. The schema has a number of complexities compared to typical DW schemas. These include *facts about facts* and *composed non-atomic facts* and unified handling of different kinds of flex-offers and time series. We also discuss alternative data modeling strategies and how specialized variants of the generic schema can be used by different node types while we maintain compatibility and consistency between them. Finally, we present typical queries from the energy domain and a performance study.

1 Introduction

More and more *green energy* is being produced by renewable energy sources (RES) such as windmills. It is, however, not possible to store larger amounts of energy and use it later. Therefore, there often is an unused capacity, e.g., during nights when most consumers sleep, but not enough green energy during day hours when most consumers are active. The EU FP7 project MIRABEL (Micro-Request-Based Aggregation, Forecasting, Scheduling of Energy Demand. Supply and Distribution) [14] addresses this challenge by proposing a "data-driven" solution for balancing supply and demand utilizing their flexibilities. Flexible demand such as for dishwashers and charging an electric vehicle can often be shifted to a time when green energy is available. Non-flexible demand such as lights, TV, or cooking stoves must still be satisfied at demand-time. In the MIRABEL-settings, a consumer offers a so-called *flex-offer* [2,16] for every intent of flexible energy demand. The flex-offer must describe when and how much energy is needed and how flexible the demand is in time and amount. Likewise, a producer can offer a flex-offer for every intent of energy supply.

A. Hameurlain et al. (Eds.): TLDKS XXI, LNCS 9260, pp. 48–72, 2015.
DOI: 10.1007/978-3-662-47804-2_3

The different flex-offers can then be accepted (or rejected if they cannot be fulfilled) and scheduled for execution at a given time. There will be extremely large quantities of such flex-offers and they cannot be scheduled individually. Instead flex-offers are *aggregated* into larger flex-offers which become scheduled and then *disaggregated* into the smaller flex-offers again [16]. To enable this, there will be smart *nodes* at both consumer sites and producer sites in the electricity grid which we denote a *smart grid*.

There is a strong need for efficient data management in these nodes. In this paper, we present *MIRABEL DW* which is a data warehouse (DW) for the management of large amounts of complex energy data in the MIRABEL project. This paper is the first to present a DW schema for the important domain of energy data. The schema can represent different "actors" in different "roles" as defined by the "Harmonised Electricity Market Role Model" [6] as well as (individual and aggregated) flex-offers, and time series. In the future, the managed data is to be distributed over millions of nodes [2] in non-traditional ways. In the paper, we focus on a DW on a single node, but present a unified schema that can manage data both at the level of the entire electricity network and the level of individual nodes, such as a single consumer node. Compared to typical DW schemas, the schema has a number of complexities which we discuss in the paper. These include *facts about facts* and *composed non-atomic facts* and unified handling of different kinds of flex-offers and time series. We also discuss alternative data modeling strategies that use denormalization and arrays, respectively. We also discuss so-called *specializations* which allow certain variants of the generic unified schema to simplify data management in different node types which, e.g., can have limited hardware resources. Further, we present typical queries from the energy domain and a performance study that compares the described schemas with the denormalized and array-based alternatives, and the specialized schemas.

The rest of the paper is organized as follows: Our representations of flex-offers, time series and actors are presented in Sects. 2, 3, and 4, respectively. These parts together form the full schema which is presented in Sect. 5. Section 6 presents specializations of the generic schema to simplify data management at different node types. Examples of analytical queries on the schema are given in Sect. 7. A performance study is given in Sect. 8. Previous work related to this is presented in Sect. 9 before the concluding remarks and pointers to future work which are given in Sect. 10.

2 Modeling of Flex-Offers

In this and the following two sections, we first present the data model we use in MIRABEL DW. Then we discuss the non-standard and advanced techniques that are applied in the modeling.

2.1 Data Model

To represent MIRABEL's flex-offers (both aggregated and non-aggregated) is an essential task for MIRABEL DW. This is done by means of the tables shown

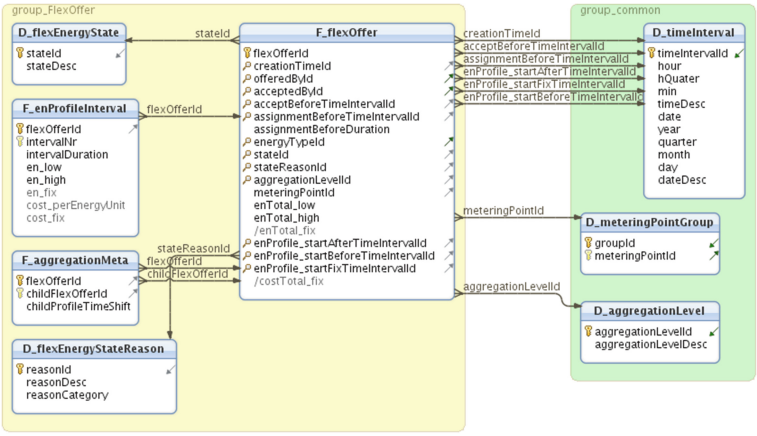

Fig. 1. Tables for representing flex-offers

in Fig. 1. We first describe the dimensions (which are recognized by the prefix D_ in their table names) and then the fact tables (recognized by the prefix F_ in their names). All dimension tables have surrogate keys with names ending with Id. The possible states for a flex-offer (such as "offered", "accepted", and "rejected") are represented in the dimension D_flexEnergyState. A flex-offer has its state for a certain reason (for example, a flex-offer becomes rejected if the offered price is too high). The possible reasons are represented in the dimension D_flex-EnergyStateReason. As we expect few generic reason categories (e.g., "Price too high") and many more specific reason descriptions (e.g. "Price (499.50 euros) too high") to exist, we have columns for both the generic categories and the specific reasons such that a hierarchy exists. In MIRABEL DW, we represent time by discretized time intervals. This is done by D_timeInterval which represents 15 min intervals (for now; other interval lengths can be chosen if needed). Flex-offers are always related to at least one metering point (at the location where the energy is to be consumed or produced), but if a flex-offer is aggregated, it will be associated with many metering points. To capture this, D_meteringPointGroup is used as bridge table [9] between the fact table and D_meteringPoint which represents the individual metering points. To represent the aggregation level of a flex-offer, D_aggregationLevel is used.

The fact table F_flexOffer holds flex-offer facts. It references all the previously described dimension tables. There are six foreign keys to D_timeInterval to represent different times such as when the flex-offer was created and when it at the latest has to be assigned etc. These foreign keys thus all represent an absolute time. There is also an attribute assignmentBeforeDuration which holds a time span telling how long before the actual execution time the assignment must take place.

Further, F_flexOffer references D_legalEntityRole (explained later) twice to represent who offered and accepted the flex-offer, respectively. Only the current information about a flex-offer is held; if a flex-offer is modified, the old fact

is overwritten. There are measures to hold the lowest and highest amount of energy required by the flex-offer as well as a measure to hold the "fixed" amount of energy that becomes accepted. Further, a measure holds the total cost of the fix. Finally, each represented flex-offer is given a unique identifier in the attribute flexOfferId which technically is a degenerate dimension.

Information about the profile intervals of flex-offers is represented in the fact table F_enProfileInterval. This fact table only has a single foreign key which references the unique flexOfferId in F_flexOffer. The imported value together with a sequential intervalNr forms the primary key for F_enProfileInterval. The reason for this design is that a single flex-offer can have many profile intervals. For each represented profile interval, there is a duration specifying how many time units the profile interval spans over, and both the lowest and highest amount of energy needed in this interval. When the flex-offer becomes fixed, the actual amount of energy in the interval and the price for this energy also becomes represented. An alternative to this design would be to represent the measures of F_enProfileInterval in *arrays* in F_flexOffer such that all data about a given flex-offer would be represented in a single fact. Yet another alternative would be to represent all attributes of F_enProfileInterval in F_flexOffer, i.e., denormalize the data and have one (wide) fact in F_flexOffer for each profile interval. (For space reasons, we do not show the alternative schemas in figures).

As flex-offers can be aggregated into larger flex-offers, we also introduce the table F_aggregationMeta which references F_flexOffer twice to point to the aggregating "parent flex-offer" and the smaller "child flex-offer" which has been aggregated, respectively. Profiles of each child flex-offer can be shifted relatively to the profile start of the parent flex-offer when aggregating child flex-offers into the parent. Therefore, for every child flex-offer, the childProfileTimeShift attribute indicates the amount of time units the profiles of the child flex-offer has been shifted in the aggregated flex-offer. This information is used in the disaggregation.

2.2 Modeling Challenges

The fact table F_flexOffer is the central fact table for representation of flex-offers. It is, however, also used as a dimension table in the sense that each fact has a unique ID such that F_enProfileInterval and F_aggregationMeta can reference F_flexOffer and in effect store *facts about facts*. Considering F_flexOffer and F_enProfileInterval, it can even be discussed *what* a fact is. An energy profile interval (in this context) always belongs to a flex-offer and any meaningful flex-offer has an energy profile interval (a flex-offer for zero consumption/production at an undefined point in time is hardly interesting). It could be argued that a single fact is represented by a single row in F_flexOffer and many rows in F_enProfileInterval. Unlike traditional DW schemas, we thus have non-atomic *composed* facts. As pointed out above, we could alternatively have modeled this by using arrays in F_flexOffer to hold the measures that currently are represented in F_enProfileInterval. This would, however, make it more cumbersome to compare different measures (e.g., en_low with the minimum energy requirement to en_fix with the assigned energy) as the interval position currently represented by intervalNr only

would be implicitly represented by the position in the array. The denormalized variant (with a fact in F_flexOffer for each profile interval) would increase redundancy dramatically.

Another interesting aspect of MIRABEL DW is how it represents facts for both non-aggregated and aggregated flex-offers in a unified way. The aggregation is unlike traditional aggregation since the parent flex-offer contains other flex-offers that can be shifted within the parent flex-offer. We call the contained flex-offers *shiftable child facts*.

3 Modeling of Time Series

3.1 Data Model

In MIRABEL DW, time series are represented by means of the tables shown in Fig. 2. It is necessary to be able to represent time series of various types, for now energy, power, and price. To represent these general classes, we use the D_typeClass dimension table. Apart from its surrogate key, it has the attribute typeClassDesc which holds a textual description of the time series type (such as "Energy") and the attribute unit which holds the unit of measurements (such as "kWh"). Instances of the general types are represented in the table D_type. For example, an instance of the "Energy" class is "Energy-Metered-Production-RES-Wind". D_type references D_typeClass to represent the hierarchy between

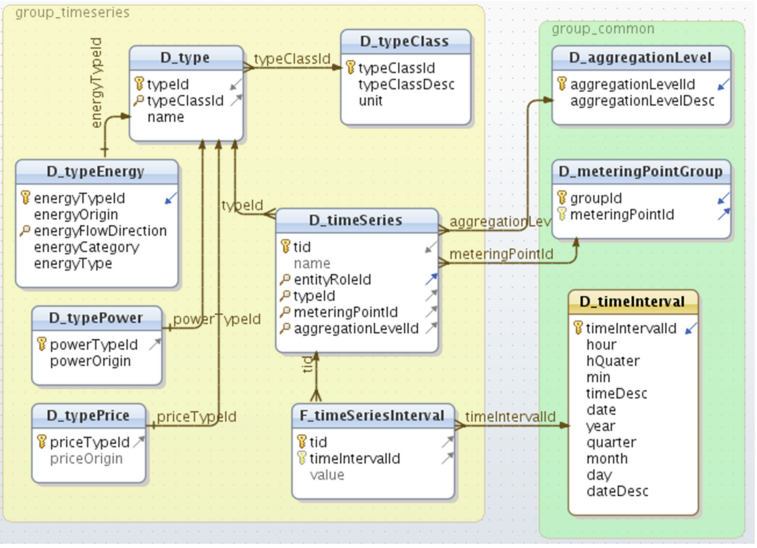

Fig. 2. Tables for representing time series

types and type classes. For different types of time series, it is, however, necessary to store different information. Therefore, we introduce the tables D_typeEnergy, D_typePower, and D_typePrice to hold the attributes that are relevant for the different types. These tables supplement, but cannot replace, D_type. The reason is that we need a single table to reference from D_timeSeries to represent the type of the time series in question. Thus D_type is referenced from D_timeSeries, but the special attributes for an energy time series are represented in D_typeEnergy. The latter table has columns to describe the origin of the time series (e.g. "Metered" or "Forecasted"), the flow direction (i.e., if it is production or consumption), the category (e.g., energy from renewable energy sources), and the type of energy (e.g. "Wind"). The design is likely to evolve in the future. For example, there is a traditional hierarchy where types roll up into categories that roll up into flow directions. A more advanced hierarchy is, however, needed to represent hybrid energy types like "At least 90 % energy from renewable energy sources and the rest produced from coal".

D_timeSeries holds a single entry for an entire time series. For each represented time series, there is a unique ID tid and a name may be given. Further, D_timeSeries references D_type (as previously described), D_aggregationLevel to represent the level of aggregation of the time series, and D_meteringPointGroup to represent which meters the time series describes. Thus, D_timeSeries is mainly used to relate different dimension values that describe the represented time series. The values of the time series are, however, represented in the fact table F_timeSeriesInterval. This table references D_timeSeries to identify the time series a value belongs to and D_timeInterval to identify the time instant when the value occured. Finally, the table holds the value itself as the measure. A fact thus exists for each value in each time series. It can, however, also be argued that a fact consists of what it represented in F_timeSeriesInterval *and* what is represented in D_timeSeries which – apart from a possible name – only points out to other dimensions.

3.2 Modeling Challenges

Similarly to the representation of flex-offers, our representation of time series also leads to compound facts where one fact can be considered to be made up of parts in different tables (D_timeSeries and F_timeSeriesInterval). Actually, an alternative design is to merge F_timeSeriesInterval into D_timeSeries such that the values instead are represented in an array, meaning that a single time interval (and all its values) only would result in one fact. Yet another alternative is to merge D_timeSeries and F_timeSeriesInterval and have a row for each value in a time series. There are thus different possible ways to represent the complex sequence-facts arising from time series. We choose the model in Fig. 2 since it both reduces complexity (compared to the first alternative where two arrays must be processed to find the value for a given time instant) and redundancy (compared to the second alternative where there is very wide fact for each value in the time series).

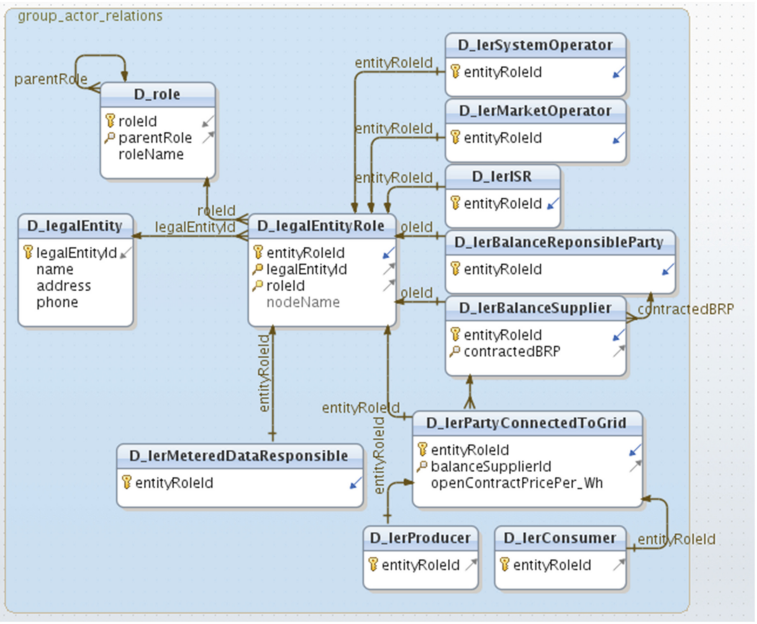

Fig. 3. Tables for representing different actors/roles

In our modeling of time series, the schema is neither a traditional star schema nor a snowflake schema. One reason for this is of course the compound facts discussed above. Another reason is the support for different types of time series for which different attributes are needed. We have different tables that reference D_type which also is the dimension table referenced from the fact table. Consider for example D_typeEnergy which represents attributes that are relevant for energy time series. An alternative design would be to join all these D_type* tables into one dimension table, but for every dimension member many attribute values would then be NULL.

4 Modeling of Different Actors and Market Areas

4.1 Data Model

Many different entities are involved in different roles in energy trading and network operation. We represent the needed actors from the "Harmonised Electricity Market Role Model" [6] by means of the tables in Fig. 3

The table D_role represents roles such as "Producer" and "Consumer". A role can belong to another parent role and this is captured by a self-reference. For example, the parent role of both "Producer" and "Consumer" is "Party Connected To Grid". Legal entities are represented by D_legalEntity. To capture when a certain legal entity plays a certain role (a single legal entity can play several

roles), we use D_legalEntityRole. This table references both D_role and D_legalEntity. Further, it has an attribute to hold a unique ID for a given legal entity playing a given role. We include this ID as it makes it easy to point to a legal entity in a certain role. We do exactly that from a number of tables as shown in Fig. 3. For each role, there is a specialized table that (directly or indirectly through another table) references D_legalEntityRole. Some of them, like D_lerSystemOperator, are simple and do only have one attribute which is a reference to this ID. The specialized table can be referenced and it is then explicit what kind of role is referenced. For example, the table D_lerSystemOperator is referenced from D_marketBalanceArea as shown in Fig. 5. A slightly more complex example is D_lerPartyConnectedToGrid which references D_legalEntityRole and also D_lerBalanceSupplier to represent that a party connected to the grid always is so through a balance supplier. Further, D_lerPartyConnectedToGrid is itself referenced from its specializations, D_lerProducer and D_lerConsumer.

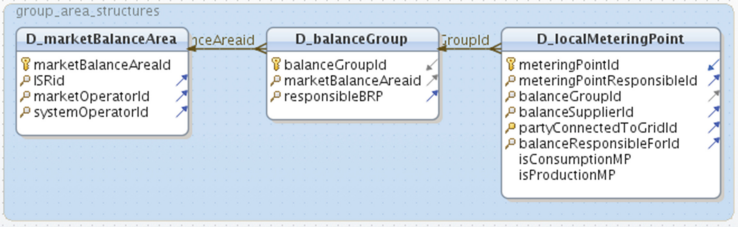

Fig. 4. Tables for representing market areas

Finally, we have tables to represent market areas as shown in Fig. 4. D_localMeteringPoint represents the meters that are connected to the grid. Such meters are installed both at the producer and consumer sites. D_localMeteringPoint references four different specializations of D_legalEntityRole. Further, it references D_balanceGroup which in turn references D_marketBalanceArea which hierarchically groups metering points.

4.2 Modeling Challenges

To the best of our knowledge, this is the first paper to describe a DW for the complex concepts of actors and roles in the "Harmonised Electricity Market Role Model" [6]. Our model captures both how legal entities can play different roles and how roles can be parts of other roles. This is captured by the tables D_legalEntity, D_role, and D_legalEntityRole. In addition to these tables, a (narrow) table has been added for each role a legal entity can play (see the D_ler* tables). It is then possible to represent attributes that are only relevant for certain roles such as done for D_lerBalanceSupplier. Further, when foreign keys reference these tables (instead of just referencing D_legalEntityRole), it is explicit what kind of role playing is referenced and it helps to avoid mistakes where, e.g., a balance supplier is referenced where a balance responsible party actually should have

been referenced. We note that if no special attributes must be stored for the different roles, then instead of storing the D_ler*'s as physical tables, they can be views selecting from D_legalEntityRole. This reduces the risk of mistakes further and makes maintenance of them automatic.

5 The Full Schema

To summarize the previous descriptions, the full schema for MIRABEL DW is shown in Fig. 5. The schema can capture the (needed) roles from the Harmonised Model [6] as well as the "actor configurations" where different actors play different roles. The schema also includes specializations of legal entities. Further, the schema can capture different kinds of time series as complex sequence facts. The schema is thus general enough to hold all the data that is needed in the MIRABEL project. It should, however, be noted that no single node is intended to hold all data. Instead, a node should only hold data that is relevant for the site where it is installed. For an end-consumer this would typically

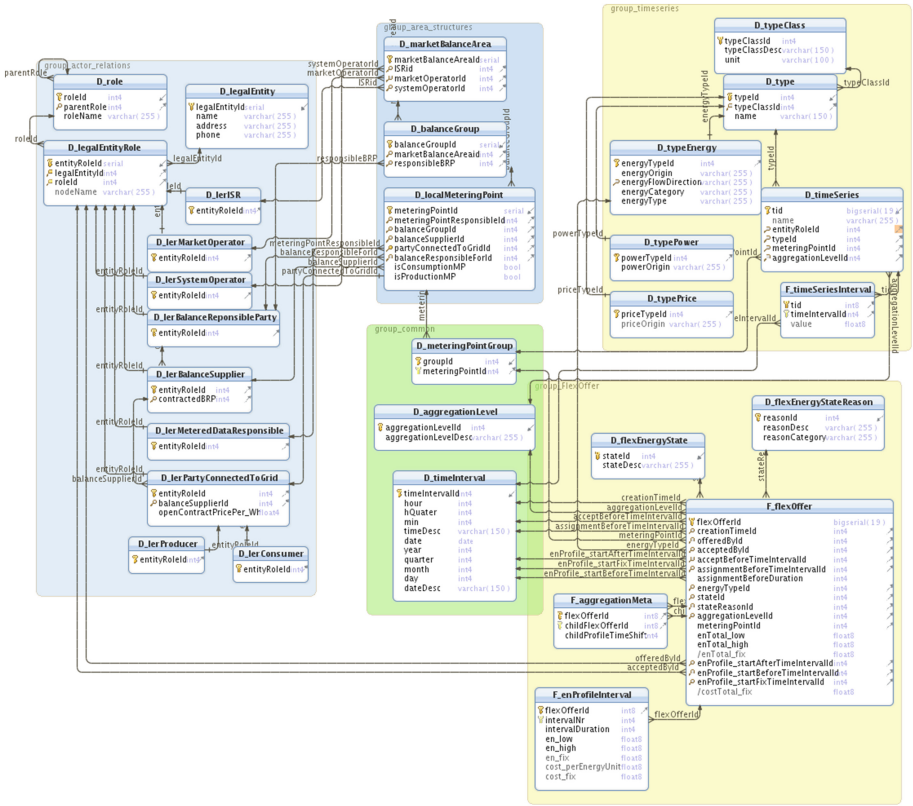

Fig. 5. The full schema for MIRABEL DW

be her own non-aggregated flex-offers and time series about metered energy. For a balance responsible party buying electricity on the market and selling it to end-consumers, it would include both aggregated and non-aggregated flex-offers, forecasted and metered time series, and market areas. The data will thus be distributed accordingly to the roles played by the owners of the nodes. The data will also be at different aggregation levels such that some nodes have detailed data while others have more aggregated data. A consumer will know the details of her flex-offers, i.e., when she has requested energy and how much. For a balance responsible party, the individual non-aggregated flex-offers and end-users generating may not be known, but the aggregated information will be known, e.g., that x MWhs must be produced in a given time interval. Note that the different nodes can use the same schema. The distribution of data is illustrated in Fig. 6 which shows different kinds of nodes. Non-aggregated flex-offers are shown as small, shaded boxes. Note that the different nodes do not represent the same flex-offers. A single node only represents the flex-offers that are relevant to its owner. Aggregated flex-offers are shown as larger, filled boxes in Fig. 6. Note also that although the nodes distribute the data and some represent non-aggregated flex-offers and others only represent aggregated flex-offers, they can use the same schema. As described in the following section, another possibility is to allow the different kinds of nodes to use specialized schemas.

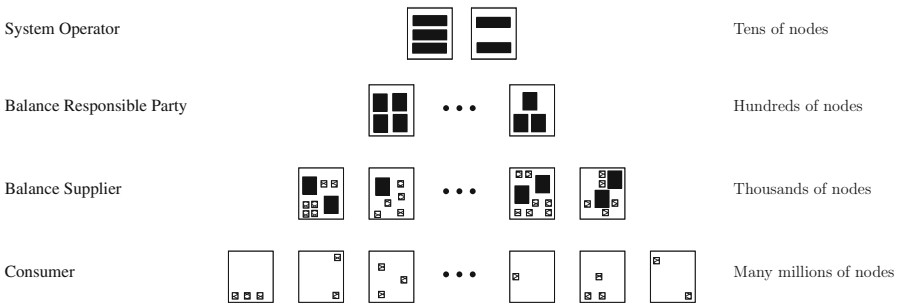

Fig. 6. Data distribution in MIRABEL DW

6 Schema Specializations

The schema in Fig. 5 is generic and can be used in all kinds of nodes in MIRABEL. It is, however, not all kinds of nodes that need to store all kinds of data. Consider, for example, a node installed at an end-consumer's site (i.e., the lowest level in the hierarchy of nodes) on limited hardware resources. Such a node does not store aggregated flex-offers; it only knows the consumer's own flex-offers. Also, it only has time series and flex-offer data for the consumer's metering point and not (groups of) other metering points. For a node at a balance responsible party, on the other hand, it is necessary to represent both

individual flex-offers from end-consumers and their aggregated flex-offers that are sent to the wholesale market. Further, it is necessary to represent information about the involved actors to know where the energy comes from, where it eventually gets consumed, who regulates the area, etc.

In a consumer node, some parts of the schema in Fig. 5 are thus not needed. For example, D_aggregationLevel is not needed and the attributes referencing it from D_timeSeries and F_flexOffer are also not needed. If they were present, they would always take the same values anyway and we thus say that they are *context-given*. Likewise representations of legal entities are context-given in a consumer node since the node only deals with a given consumer that belongs to a given market area etc. Further, a consumer node typically has limited computing resources and it can be beneficial to have a simpler database schema. At higher layers, there is typically much more computing power, but the data amounts may also be much bigger such that other schemas can be beneficial. To simplify the data management in a node, such as an end-consumer node, we employ actor-specific *specializations* of the schema. A specialized database schema S does not have to be able to represent all the data of the generic schema G, but may only be able to represent some of it and possibly in a modified form. S may have relations that are different from those in G and can, e.g., be a star schema. At an informal level, a specialized schema S can differ from the generic schema G in the following ways:

1. A new attribute a can be added to S if its values can be deterministically computed
   (a) from values of attributes in S or
   (b) from values of attributes in G *and* inverse functions that for each of these attributes in G can compute its value from the value of a are given.
2. An attribute from G can be left out from S if it in an instance of S always would take the same value if included (i.e., if it is context-given).
3. An attribute from G can be left out from S if we have a way of deterministically computing its value from the value of another attribute in S without knowing the state of G. In particular, a surrogate key is *not* enough to compute all other attributes of a relation.
4. An entire relation from G can be left out from S if all its attributes can be left out.
5. A relation in S can represent several relations from G that are equi-joined on foreign keys. A surrogate key used in a join may then be left out.

The data of an instance S must be obtainable from a number of queries on an instance of G such that the data for each relation in S is obtainable from one SPJ query. In particular, the queries may not use GROUP BY, HAVING, DISTINCT, UNION, INTERSECT, or EXCEPT from SQL. The queries can join relations on foreign keys, select an attribute once or leave it out if its values can be deduced from other included attributes or are context-given, and finally restrict the amount of tuples to those with certain values in certain attributes. The queries may not aggregate G data as this would prevent us from propagating modifications from the specialization instance back to the G instance.

6.1 Querying a Specialization

For a query q_S on a specialization S, it is possible to find a query q_G on the generic schema G that gives the same result: Since any relation in the specialization can be considered a view over one or several equi-joined relations in G, it is possible to find q_G from q_S by replacing each relation in q_S with its corresponding view definition over relations in G.

A query q_G on the generic schema G can under certain circumstances also be translated to a query q_S which gives the same result on a specialization S. Recall how a specialized schema can differ from the generic schema:

1. *A new attribute can be added if its value can be computed from values of other attributes.* Such attributes can be ignored since they obviously are not used by q_G.
2. *An attribute can be left out if it is context-given.* If q_G uses an attribute a that is context-given in S, q_S must use the appropriate constant instead of a.
3. *An attribute can be left out if it can be computed from another attribute in S.* If q_G uses an attribute b that is left out from S because it can be computed from another attribute c in S, q_S must do the necessary computation of b values by means of c, i.e., occurences of b should be replaced by $f(c)$ for a deterministic function f.
4. *A relation can be left out if it all its attributes can be left out.* If q_G uses such a relation r, all usages of attributes from r (which all necessarily are context-given or can be computed) can be replaced by appropriate constants.
5. *Several relations may have been equi-joined on foreign keys.* If a surrogate key has been left out from S, it cannot be used in queries on S. But since a surrogate key just is an integer with no special meaning, it would not make much sense to query for it anyway since it has already been used in a join to combine the right rows from two relations. We therefore assume that q_G does not query for a left-out surrogate key. Consider first the case where q_G equi-joins the relations $r_1, r_2, \ldots, r_n$ on foreign keys and S has the relation r' which holds the result of an equi-join of $r_1, r_2, \ldots, r_m$ $(m \leq n)$ on foreign keys. In that case q_S can join r' and $r'_{m+1}, \ldots, r'_n$ where r'_i $(m+1 \leq i \leq n)$ holds the corresponding data of r_i as found by applying these rules recursively. Now consider the case where S does not hold such an r', but instead holds a relation $\hat{r}$ with the result of an equi-join on foreign keys of $r_1, r_2, \ldots, r_N$ in G for an $N > n$. Then q_G in general cannot be transformed to a query on S that gives the same result since S might not represent all tuples from some r_j in G (in case no rows reference them) or represent some tuples too many times. As an example of the latter, if $n = 1$ in q_G and a relation in S holds the result of r_1 joined with r_2 (i.e., $N = 2$) on a foreign key from r_1 to r_2, tuples of r_2 might be represented several times in the resulting relation.

In other words, only item (5) can be a limitation. When one creates a specialization S, one should thus be aware that joining (i.e., denormalizing) too much makes some queries on the G schema impossible to translate to the S schema and get the same results. On the other hand this is not likely to be an issue in

realistic settings. A designer would most likely not join G tables if, e.g., one of them holds rows that do not join with any rows from the other table(s) or if SELECTs from a particular table are an important query category.

A specialization can provide a simpler schema that fits the needs of a certain node and thus can be used instead of the general G schema. As discussed above, queries can always be translated from S to G and under certain circumstances from G to S. We, however, also wish to be able to do certain modifications on relations in S and be able to translate them to corresponding modifications on relations in G. We therefore now discuss which modification operations are allowed on data in a specialization instance.

6.2 Modifications

To maintain overall compatibility and consistency among nodes, it should be possible to propagate modifications made to S data back to G. Therefore, it is not all operations that are allowed in a specialization. Instead, any allowable operation on S that brings the database from a state s (obtained by applying the specialization's defining queries – or view definitions – on G in state g) to another state s' must be mappable to a number of operations on G that brings the database from g into a state g'. As discussed above, all relations in S can be seen as views on G. The state g' must then be such that if the view definitions, denoted V, are applied on a database with the schema G and the state g', the result is a database with the schema S and state s':

$$V(G, g') = (S, s')$$

For an attribute a added to S that also can be computed from other attributes X in S, we of course require that any modification to it is consistent. In other words, the value assigned to a should correspond to what can be computed from X. We now consider the possible modifications in turn and describe how they can be supported (if so) or why they cannot be supported.

Insertion is the most needed modification type for a specialization. A node with a specialization should be allowed to insert data about its own site, e.g., data about the energy consumption at the site. For a relation R_S in S which holds data from a single relation R_G in G, insertions can be supported in the following way. Some attributes of R_G may not be available in R_S, but they are then either context-given or computable from other attributes. Thus, for a row r_S inserted into R_S, we can find a corresponding row r_G to insert into R_G to achieve the state g'. This is similar to when views are updatable in SQL-92 [5] apart from that we do not get NULL values in left-out attributes but instead find proper values. Now consider a relation R_S which is the result of an equi-join of a sequence of relations $R_{G,1}, R_{G,2}, \ldots, R_{G,n}$ from G (possibly with some attributes left out if they can be computed or are context-given) where $R_{G,a}$ can have a foreign key referencing $R_{G,b}$ only if $a < b$. If a row r_S is inserted into R_S, we can for each $R_{G,i}$ (in the order $i = n, n-1, \ldots, 1$) find the corresponding part of the row r_S and add any computable or context-given attributes. We call this corresponding row part

$r_{G,i}$. If the state g of G is such that $r_{G,i}$ is not in $R_{G,i}$, it can be inserted (if the surrogate key is not present it should be added first and afterwards also added to all row parts for $R_{G,h}$ if $R_{G,h}$ references $R_{G,i}$). If it already is in $R_{G,i}$, nothing should be done. In SQL-92, insertions into "join views" (i.e., views with data from more than one relations) are not allowed. In SQL:1999 such insertions are sometimes allowed, but each of the view's columns should be uniquely traceable back to a single column in a single table [12]. We, on the other hand, allow a natural join where a column in R_S corresponds to two columns (namely, the primary key of $R_{G,b}$ and the foreign key of $R_{G,a}$) since we can consider one of the two columns as left-out due to computability. SQL does also allow the WITH CHECK OPTION to ensure that it is not possible to insert rows that would not appear in the view anyway (but it is not all RDBMSs that support it). This functionality is not available per se in a specialization, but would have to be emulated with CHECK constraints on the relations. The described method to support insertions does, however, not guarantee that a successful insertion into R_S can be mapped to successful insertions into the $R_{G,i}$ relations. For example, a primary key violation can occur when we try to insert into $R_{G,i}$ for some i. This would not be detected when inserting into R_S.

Deletion is a modification which rarely will be done in specializations. We anyway describe how it can be supported. For a row in R_S in S, we either have all corresponding G attributes directly available or can find them as argued above (possibly apart from surrogate keys). In the simple case where a row r_S is deleted from R_S which is not the result of a join, we can find the primary key value of the corresponding row r_G in R_G and use that to delete r_G. Consider for the more complex case again an R_S which is the result of an equi-join of a sequence of relations $R_{G,1}, R_{G,2}, \ldots, R_{G,n}$ from G (possibly with some attributes left out if they can be computed or are context-given) where $R_{G,a}$ can have a foreign key referencing $R_{G,b}$ only if $a < b$. We can either find the primary key value for each corresponding $r_{G,i}$ part or find values for all its other attributes and use them to identify the row to delete. Considering each of them in the order $i = 1, \ldots, n$, if the state g is such that $r_{G,i}$ is *not* referenced by any other row in any relation in G, it can be deleted. As with insertions, we are not guaranteed that a deletion in S results in one or more deletions in G. Another issue is whether we actually want a deletion in S to possibly result in deletions from more relations in G. Only the first deletion would delete detail data while the following ones would delete from the dimension hierarchy. It depends on the concrete case whether it makes most sense to delete from all corresponding G relations or only one, but the latter would often be what is wanted. It should thus be specified during the definition of a specialization how to handle deletions for relations that hold the results of equi-joined relations from G.

Update is simple to support in the case where R_S only has data from one relation, R_G, in G. In this case, the primary key value is again known and can be used to identify the corresponding row to update in R_G. For an R_S with data from several G relations, the situation is more complicated. With join views, a general

problem is that some updates to the view cannot be mapped uniquely to a set of modifications to the base table [5]. In the case of specializations, we can, however, again benefit from knowing the primary key value for each corresponding row part and use this to do *upsertions*. Thus we can only allow updates to a relation holding the result of an equi-join of G relations if no surrogate keys have been left out. For an R_S which is the result of an equi-join of a sequence of relations $R_{G,1}, R_{G,2}, \ldots, R_{G,n}$ from G, we can for a row that is updated to r'_S consider the corresponding row parts in the order $n, \ldots, 1$. Assume that the primary key value for $R_{G,i}$ is p_i. The p_i values can be found for each corresponding row part of r'_S. If $R_{G,i}$ holds a row $r_{G,i}$ with the primary key value p_i, it should be updated to $r'_{G,i}$ (if $r_{G,i} \neq r'_{G,i}$). If $R_{G,i}$ does not hold such a row, $r'_{G,i}$ should be inserted. However, $r_{G,i}$ could be a corresponding row part of many rows in R_S, but if only one of these rows has been updated to go from the state s to the state s', we will not have that $V(G, g') = (S, s')$. To be able to map an update of R_S to G, we must therefore require that for any two rows x and y in R_S for which the primary key values of $x_{G,i}$ and $y_{G,i}$ are identical for some i, we also have that $x_{G,i} = y_{G,i}$ after the update. In other words, x and y should then have identical values for all attributes originating from R_G such that functional dependencies are maintained. This can be expensive to check and another and simpler solution is to only allow updates of the attributes in R_S originating from $R_{G,1}$.

Finally, we note that an attribute or relation that is left out from a specialization, obviously cannot be modified. If the context changes, e.g., if a consumer gets another energy supplier, a new S instance must be created for the new context.

In summary, insertions into a specialization are easy to support while deletions and updates are more complex. In particular, it is necessary to specify for a specialization how to handle deletions from a relation holding the result of a join of G relations. Also for updates, this should be specified. Updates do also require that no surrogate keys have been left out. In fact, only few modifications are expected to take place in a specialization. Typically, only insertions into one or few relations will be done; for example, meter readings from the node's location into F_timeSeriesInterval. A specialization definition can thus specify which relations may be modified and (for updates and deletions) how modifications to them should be mapped to modifications to G.

6.3 Examples

As a trivial example, it is possible to define a specialization of G with the same database schema. The mapping of modification operations is then simply the identity function. Another and more interesting specialization is for a prosumer C that has solar panels producing electricity and has agreed to buy all her remaining electricity from a given energy supplier. In this case, a specialization for a node at C's site does not need to represent the metering point (as it only has data for the single metering point) and the aggregation level (as no flex-offers are aggregated). Further, the energy type is always "solar energy" when produced by

the prosumer and "undefined" when she buys electricity (the value of energyType can thus be computed from energyFlowDirection). Further, all flex-offers are offered by the consumer herself and only accepted by the single balance supplier and thus these values are context-given. Figure 7 shows S_C. Note how F_flexOffer and (several instances of) D_timeInterval have been joined leaving out the surrogate key timeIntervalId. New computed attributes with time stamps have, however, been added and they in turn allow the attributes of D_timeInterval to be left out. This is done to avoid the possibly expensive joins with D_timeInterval on the C node which is likely to have very limited hardware resources. D_timeSeries, D_typeEnergy, and D_type have also been joined as have F_timeSerieInterval and D_timeSeriesInterval. The attribute hour has also been added to the resulting relation. This attribute can be computed from time, but has been added to allow for efficient grouping when considering the hourly energy consumption. New time series intervals and flex-offers can be created at C's site and it must be possible to represent these in the generic schema as well. Thus insertions into D_timeSeries, F_timeSeriesInterval, and F_flexOffer are mapped to insertions into relations in G. Other modification operations are not allowed. If we wanted to also support updates of facts, we would have to include the left-out surrogate keys.

7 Queries

In this section, we give examples of interesting queries on data in MIRABEL DW. We first focus on queries on flex-offers and then on time series.

7.1 Queries on Flex-Offers

The first example, `Q1`, considers the flexibility in flex-offers, both with respect to time and amount of energy.

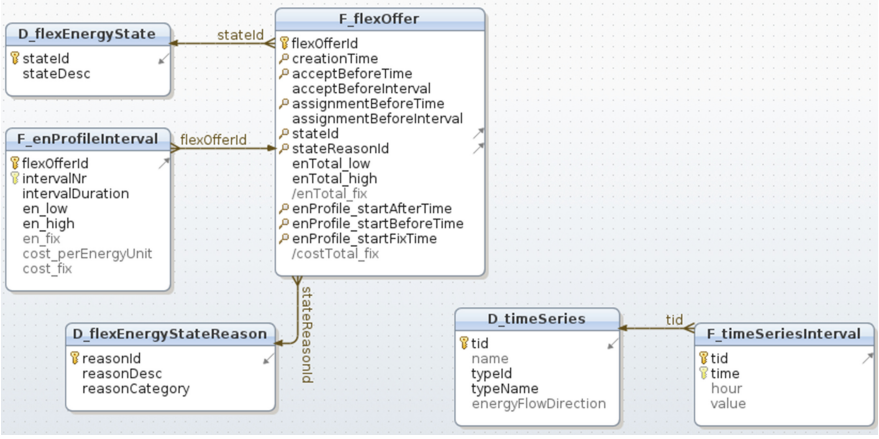

Fig. 7. The database schema S_C of a specialization

```
Q1:  SELECT AVG((enProfile_startBeforeTimeIntervalId -
                 enProfile_startAfterTimeIntervalId) *
                (SELECT SUM((en_high - en_low) * intervalDuration)
                 FROM F_enProfileInterval i
                 WHERE i.flexOfferId = f.flexOfferId)
                )
     FROM F_flexOffer f;
```

The query uses the flexibility with respect to time, i.e., the difference between when the flex-offer at the latest *has* to be executed and when it at the earliest *can* be scheduled. We assume that time interval IDs are assigned sequentially and thus use the difference between the IDs of the time intervals to find the flexibility. This flexibility is multiplied with the SUM of the energy flexibility in each profile interval. The energy flexibility in a profile interval is found as the length of the profile interval multiplied with the difference between the maximally required amount of energy and the minimally required amount of energy. Finally, the shown query considers the average of the combined flexibility for all flex-offers. The query is an example of a non-traditional kind of aggregation. If we consider a graph showing the relative start and end times for profile intervals on the X axis and the minimal and maximal energy amounts on the Y axis, the query `Q1` finds the *area of energy flexibility* for all flex-offers and multiplies these with the length of their time flexibilities before the entire average is found. This number is primarily of interest *before* the scheduling gets done and a high number indicates much freedom in the scheduling while a low number shows that the considered flex-offers are not very flexible.

The next example, `Q2`, is of interest *after* the scheduling and gives the total amount of scheduled energy. This is a simple query which, however, must read data from many rows in a realistic setting (the DBMS we use does currently not support materialized views).

```
Q2: SELECT SUM(en_fix)
    FROM F_enProfileInterval;
```

`Q3` is a more complex query to apply after scheduling has taken place. It builds a time series that, for each time interval ID, shows the amount of fixed energy.

```
Q3: SELECT timeIntervalId, SUM(en_fix_part)
    FROM (SELECT en_fix_part, ROW_NUMBER() OVER (PARTITION BY i.flexOfferId
                 ORDER BY intervalNr) - 1 + f.enProfile_startFixTimeIntervalId
                 AS timeIntervalId
          FROM (SELECT flexOfferId, intervalNr, en_fix / intervalDuration
                       AS en_fix_part, generate_series(1, intervalDuration)
                FROM F_enProfileInterval
                WHERE en_fix IS NOT NULL
               ) i, F_flexOffer f, D_flexEnergyState s
          WHERE i.flexOfferId = f.flexOfferId AND f.stateId = s.stateId
                AND s.stateDesc = 'Assigned'
         ) AS subquery
    GROUP BY timeIntervalId
    ORDER BY timeIntervalId;
```

The query computes the IDs of the time intervals where a flex-offer's profile intervals are executed. But a profile interval has a duration (in `intervalDuration`) which defines how many time intervals the profile interval spans. Therefore, it

is necessary to (evenly) distribute the profile intervals' energy amounts over one or more time intervals. To do this, one "part" row is generated for each time interval a profile interval covers by means of generate_series. This happens in the innermost SELECT. The result of this is used by the second SELECT which also uses the SQL window function ROW_NUMBER to enumerate the rows in each partition where a partition consists of the part rows for a given flex offer and is ordered by the interval numbers. Thus, the resulting row number corresponds to the number of time intervals between the assigned start time for the entire flex offer and the part represented by the row (we subtract 1 since ROW_NUMBER counts from 1). When we add enProfile_startFixTimeInterval for the flex-offer, we get the ID of the absolute time interval when the part executes. Finally, the outermost SELECT aggregates the sums of fixed energy amounts over all parts belonging to a given time interval.

7.2 Queries on Time Series

Q4 is a query that finds the balance, i.e., the difference between produced and consumed energy, for a 24 hours period.

```
Q4: SELECT date, timeDesc,
           SUM(CASE energyFlowDirection WHEN 'Production' THEN value
                                        ELSE 0 END) AS production,
           SUM(CASE energyFlowDirection WHEN 'Consumption' THEN value
                                        ELSE 0 END) AS consumption
           SUM(CASE energyFlowDirection WHEN 'Production' THEN value
                                        WHEN 'Consumption' THEN -1 * value
                                        ELSE 0 END) AS balance
    FROM F_timeSeriesInterval f, D_timeSeries ts, D_type ty,
         D_typeEnergy te, D_timeInterval ti
    WHERE f.tid = ts.tid AND ts.typeId = ty.typeId AND te.energyTypeId =
          ty.typeId  AND ti.timeIntervalId = f.timeIntervalId AND
          te.energyOrigin = 'Metered' AND ti.date = '2011-06-01'
    GROUP BY ti.timeIntervalId
    ORDER BY ti.timeIntervalId;
```

The query Q4 is an example where we use the special attributes that only apply to some time series. In this example, we consider consumed and produced energy and we thus use energyFlowDirection and energyOrigin which only exist for energy time series. The query sums the production values, consumption values, and the difference between them for each time interval that belongs to a given date.

Our last example, Q5, is a query to find those time series where the average energy usage grouped on hours exceeds the average energy usage for the hour with 25 % or more at least 10 times.

```
Q5: WITH indavguse AS (
         SELECT tid, hour, COUNT(value) AS indcnt, AVG(value) AS indavg
         FROM F_timeSeriesInterval NATURAL JOIN D_timeInterval
         GROUP BY tid, hour
    ),
    totavguse AS (
         SELECT hour, SUM(indcnt * indavg) / SUM(indcnt) AS totavg
         FROM indavguse
         GROUP BY hour
```

```
),
overuse AS (
   SELECT tid, t.hour, indavg, totavg,
          COUNT(*) OVER (PARTITION BY tid) AS cnt
   FROM totavguse t, indavguse i
   WHERE t.hour = i.hour AND indavg >= 1.25 * totavg
)
SELECT tid, cnt, hour, indavg, totavg
FROM overuse
WHERE cnt > 10
ORDER BY tid, hour;
```

The query has Common Table Expressions (CTEs) in the WITH part. In the first CTE, `indavguse`, we compute a (temporary) table with the average hourly energy usage for each time series. The result is used again to compute the second CTE, `totavguse`, where we find the average energy use per hour among all time series (we could join F_timeSeriesInterval and D_timeInterval again, but it is faster to reuse the result of the previously computed CTE). In the third CTE, `overuse`, we join the the results of the two previous CTEs to find the IDs of time series and the hours from `indavguse` where the consumption is at least 25 % higher than the general hourly average consumption found in `totavguse`. Further, we use COUNT as a window function to count how many such hours we find for a given time series. Finally, we select the ID of the time series, the count of hours with an average energy usage at least 25 % higher than the average, and the consumption in the last SELECT clause.

8 Performance Study

In this section, we consider the queries from the previous section and use them to evaluate and compare the different MIRABEL DW schema alternatives presented in Sects. 2–6. This section is split into two parts. In the first part, we focus on the generalized variants of the MIRABEL DW schema – the *original* (unmodified, called "MDW"), *denormalized*, and *array-based* variants – and use them to compare the performance of the queries Q1–5. In the second part, we compare the original (MDW) generalized MIRABEL DW schema to a specialized variant (a specialization) by evaluating performance of Q4 in a resource-limited environment.

8.1 Performance of Q1–Q5 on the Generic Schemas

We now consider the queries Q1–Q5 on the described (original) schema of MIRABEL DW and its denormalized and array-based alternatives, denoted as "MDW", "denorm", and "array" respectively. In the denormalized variant, F_flexOffer and F_enProfileInterval are joined and so are F_timeSeriesInterval and D_timeSeries (however, with the name varchar attribute replaced by an integer to make it a typical fact table). In the array variant, the same tables are joined, but now grouped on all dimension references and with measures aggregated into arrays. For the tests, we use a real life data set with consumption data from 963 customers (the data originates from the MeRegio project [13]) and we

synthetically generate flex-offers based on this data set. This gives rise to 963 (energy consumption) time series with 32.1 million time series values, and 3,1 million flex-offers. We test the performance on a Linux server with two Quad Core 1.86 GHz Intel Xeon CPUs, 16 GB RAM, 4 SATA 7200 RPM disks (with one dedicated to the DBMS). The DBMS is PostgreSQL 9.1 [15] and has the parameter shared_buffers set to 4 GB, temp_buffers to 128 MB, and work_mem to 96MB. All tables are "fully vacuumed" such that their disk representations only take up the needed space and do not occupy unused space. Further, the tables are "analyzed" such that their statistics are up-to-date. Each query is executed once in a warm-up round and then the queries are executed in a round-robin fashion such that each query gets executed five times. We report the average execution times. The results are shown in Fig. 8.

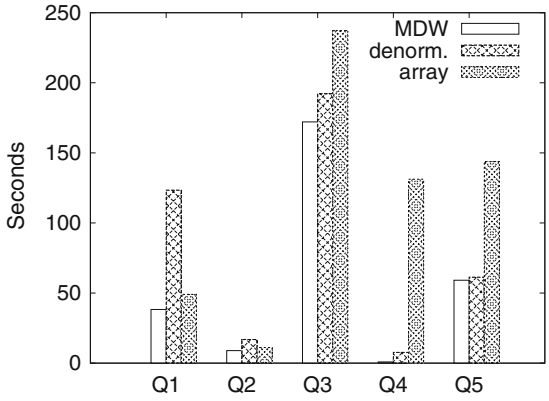

Fig. 8. Performance of Q1–Q5 on the generic schema

For `Q1`, it can be seen that the MDW variant is the fastest followed by the array variant (38.3 s and 49.1 s, respectively). These two query variants have similar plans, but with arrays there are fewer rows to process. On the other hand, these rows need to have their arrays "unnested" to produce as many values as there are rows to consider in the MDW variant. When the denormalized variant is considered, there are also many rows and these rows are wide. Further, the plan is not similar to the plans for the other variants as GROUP BY is necessary with this variant. This makes the denormalized variant the slowest (123.4 s).

For `Q2`, the MDW variant is again the fastest (8.9 s) to use. Again, the array variant is the second fastest (11.1 s). With this variant, the arrays must again be unnested to produce the values that are available in the rows in the MDW variant. The denormalized variant uses wider rows and is the slowest (16.8 s).

For `Q3`, the MDW variant remains the fastest (172.1 s) while the array variant now is the slowest (237.2 s) even though it avoids a join. On the other hand, the array variant requires a SELECT clause to unnest the array and an extra use of ROW_NUMBER to recreate the values from intervalNr which only are

implicitly available from the array positions. The denormalized variant (192.2 s) is bit slower than the MDW variant even though it avoids a join.

For Q4, the MDW variant is significantly faster (0.8 s) than the others. The denormalized variant which avoids a join, uses an order of magnitude more time (7.7 s). The array variant is by far the slowest (131.9 s) as there is no index on timeIntervalId which is an array. Thus all rows must be processed and have their rows unnested to perform a join with D_timeInterval.

For Q5, the MDW and denormalized variants perform similarly (59.1 and 61.3 s, respectively). The queries involve the same number of rows and are identical apart from that the denormalized variant uses a wider table. For the array variant, the first CTE has to unnest two arrays and the query takes longer time (143.8 s).

To summarize, the MDW variant performs the best for all queries. Another interesting thing to consider, is the disk space usage. The tables F_flexOffer, F_enProfileInterval, F_timeSeriesInterval, and D_timeSeries take up 4.1 GB in the MDW variant (not counting indexes). Their alternative representations take up 7.0 GB in the denormalized variant and 1.9 GB in the array variant, respectively. It notable how little space the array variant uses compared to the other variants due to its fewer number of rows (and thus fewer space-consuming row headers). Overall, the MDW variant is a good choice considering both its performance and space requirements.

8.2 Performance of Q4 on the Specialized and Generic Schemas

We now consider MIRABEL DW at a prosumer node (e.g., a smart-meter), which uses MIRABEL DW for storing, among other entities, electricity consumption and production measurements. As this node is expected to have limited computing and storage capabilities, we consider a MIRABEL DW specialization as opposed to the full MIRABEL DW schema for the storage of measurements. To simulate a resource-limited environment, we use three instances of the Oracle VirtualBox virtual machine (VM), each of which runs the lightweight Linux DSL 4.2.5 OS and the SQLite 3.3.10 DBMS. We deploy these instances on the machine from Sect. 8.1. The configurations of these VM instances are as follows:

VM(100,1024). The CPU clock speed/frequency is 100 % of the host machine, but the memory (RAM) is limited to 1024 MB.
VM(100,12). The CPU clock speed/frequency is 100 % of the host machine, but the memory (RAM) is limited to 12 MB.
VM(10,12). The CPU clock speed/frequency is limited to 10 % of the host machine, and the memory (RAM) is limited to 12 MB.

For the experiment, we use a dataset with consumption and production measurements collected every 15 min within an eight year time interval. These are stored as two separate time series in two databases – the first database in the generic MIRABEL DW schema G (MDW) and the second database in the specialization schema S_C from Sect. 6.3. By varying the total amount of

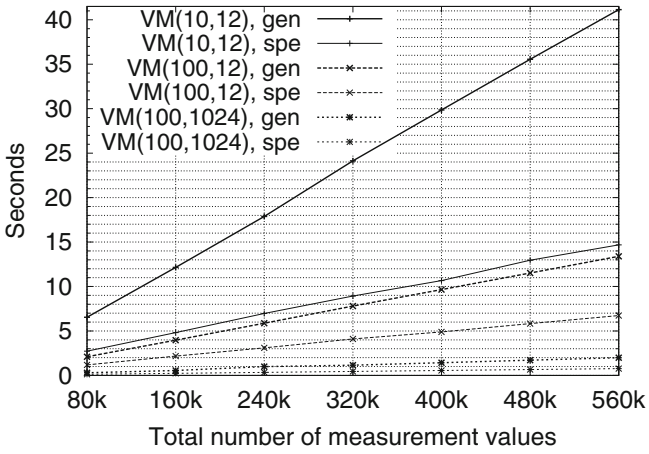

Fig. 9. Performance of Q4 on the specialized and generic schemas

measurements stored in MIRABEL DW, we submit (variants of the) query Q4 for computing the difference between production and consumption (balance) for a 24 h period for a selected day, and measure the total time required to evaluate Q4 on each of these three VM instances. The results of this experiment are shown in Fig. 9.

As seen in the figure, the query Q4 takes up to 2.5 times more time to evaluate for the generic schema in comparison with the specialized schema. The fewer resources the node has, the more it pays of to use the specialization. In summary, we can see that the use of specialized schemas has a big potential for resource-constrained devices such a smart meters.

9 Related Work

In the energy sector, there is a number of standardized data models used to represent the major objects in an electric utility enterprise [8] as well as to define administrative data internally interchanged between European electricity markets [6,7]. These models focus on various aspects of energy trading and physical electricity delivery, and specify 1) components of a power system at the electrical level, 2) actors and roles involved in the energy trading, 3) relationships and data exchange between those entities. These models are used as a basis for the MIRACLE data model [10], which further enriches them with the concept of shiftable consumption and production. All these models, however, focus on a semantic rather than the storage or the management of energy-related entities. By focusing on two most important entities in MIRABEL, i.e., time series and flex-offers, this paper, on the other hand, presents data representation models for these two types of entities offering a convenient storage and a good performance of analytical queries.

This paper is a significant extension of a previous conference paper [17], and the papers are the first to deal with the storage of flex-offers. There are previous works which focus on time series and warehousing, e.g. UML-based modeling of time-series in DWs [20], and temporal aggregation of multidimensional data [3], and temporal DWs exploiting research results from the field of temporal databases [11]. Our modeling of different time-series types has similarities with Bauer et al.'s work [1]. They discuss "locally valid dimensional attributes" whose existence depends on values of dimensional elements. This is the case, e.g., for our attribute energyType which only exists if the D_type value represents an energy time-series. The problem of representing all these attributes in a single dimension table (as in a typical star schema) is that there will be many NULLs in the held data. Bauer et al. propose to have separate tables with the specific attributes and then create views "on top" of these with common attributes as well as textual values showing the name of the relation the data comes from which can be used for hierarchical classification. In contrast, we use tables (and not views) for the common attributes of a dimension and then represent special attributes that only exist for some dimensional values in other tables that reference the table with the common attributes. This makes it possible to declare foreign keys to the dimension table with the common attributes and also declare indexes and constraints on these tables. Bauer et al. also propose to use table inheritance to represent such cases. This would also be possible in our DBMS [15], but constraints cannot be enforced on child tables then. Yu et al. [19] present an approach for storing very big time series from sensor networks using Cloud technologies such as Bigtable [4]. In contrast, we use relational storage technology and further our individual DWs cannot be merged into a single, shared DW due to privacy concerns, as the data comes from many competing companies.

In the current paper, we consider different representations of profile intervals and time series intervals which can be considered as facts with multi-valued measures. The latter case also has a many-many relationship between the time series facts and the time interval dimension. Previous work [18] has considered many-many relationships between fact tables and dimension tables. Our denormalized representation is similar to one of the methods of [18] whereas our other approaches with fact tables referencing other fact tables and measure values in arrays, respectively, are different.

Compared to the conference version [17], the current paper also provides descriptions of how to allow specializations of the generic schema. This includes how to translate queries between them and how to support modifications to the data in a specialization such that the generic schema can be modified correspondingly. The latter is related to updatable views as discussed, e.g., in [5,12], and includes a range of new experiments that compare standard and specialized schemas.

10 Conclusion

In this paper, we have presented a unified, generic DW schema for managing the complex energy data in a smart grid, including actors playing roles, flex-offers,

and different types of time series. The schema has a number of interesting complexities such as facts about facts and composed non-atomic facts. The different nodes will hold different parts of the data accordingly to the roles of the node owners and the data will be at different aggregation levels at different nodes. The same schema can, however, be used for all kinds of nodes. We have considered different alternatives for the schema modeling using denormalization and arrays, respectively, but based on the performance and space usage, the chosen design is favourable. Further, we have described how to allow specialized versions of the schema for different types of nodes, but such that queries can be formulated on the generic schema and automatically be translated to the specialized schemas (and vice versa) to make the results combinable. We also described how to support modifications on specializations.

In the near future, we are going to extend the DW schema to be able to handle other energy-specific entities such as operating schedules, parameters, and power network constraints, statuses, loads, and spatial models. Furthermore, we will perform large-scale simulations with realistic data amounts from different types of nodes. We will also perform large-scale simulations using nodes that use role-specific specializations of the general DW schema. As part of it, we will simulate the update propagation between different specializations.

References

1. Bauer, A., Hümmer, W., Lehner, W.: An alternative relational OLAP modeling approach. In: Kambayashi, Y., Mohania, M., Tjoa, A.M. (eds.) DaWaK 2000. LNCS, vol. 1874, p. 189. Springer, Heidelberg (2000)
2. Boehm, M., et al.: Data management in the MIRABEL smart grid system. In: Proceedings of EDBT/ICDT Workshops (2012)
3. Böhlen, M.H., Gamper, J., Jensen, C.S.: Multi-dimensional aggregation for temporal data. In: Ioannidis, Y., Scholl, M.H., Schmidt, J.W., Matthes, F., Hatzopoulos, M., Böhm, K., Kemper, A., Grust, T., Böhm, C. (eds.) EDBT 2006. LNCS, vol. 3896, pp. 257–275. Springer, Heidelberg (2006)
4. Chang, F., et al.: Bigtable: a distributed storage system for structured data. TOCS **26**(2), 205–218 (2008)
5. Elmasri, R., Navathe, S.B.: Fundamental of Database Systems, 4th edn. Addison Wesley, Boston (2004)
6. European Network of Transmission System Operators for Electricity. The Harmonised Electricity Market Role Model, version 2011–01. www.ebix.org/Documents/role_model_v2011_01.pdf as of 03 March 2014
7. Introduction to Business Requirements and Information Models. www.ebix.org/documents/Introduction%20to%20ebIX%20Models%200.0.D.pdf as of 03 March 2014
8. IEC61970-301 Ed. 2, Energy management system application program interface (EMS-API) - Part 301: Common information model (CIM) base, International Electrotechnical Commission (2009)
9. Jensen, C.S., Pedersen, T.B., Thomsen, C.: Multidimensional Databases and Data Warehousing. Morgan & Claypool, San Rafael (2010)

10. Konsman, M.J., Rumph, F.J.: MIRABEL Deliverable 2.3: Final data model, specification of request and negotiation messages and contracts. https://www.db.inf.tu-dresden.de/miracle/publications/D2.3.pdf as of 03 March 2014
11. Malinowski, E., Zimányi, E.: Advanced Data Warehouse Design From Conventional to Spatial and Temporal Applications. Springer, Heidelberg (2009)
12. Melton, J., Simon, A.R.: SQL:1999 Understanding Relational Language Components. Morgan Kaufmann, Boston (2001)
13. www.meregio.de/en/ as of 03 March 2014
14. As of 03 March 2014. www.mirabel-project.eu/
15. www.postgresql.org as of 03 March 2014
16. Šikšnys, L., Khalefa, M.E., Pedersen, T.B.: Aggregating and disaggregating flexibility objects. In: Ailamaki, A., Bowers, S. (eds.) SSDBM 2012. LNCS, vol. 7338, pp. 379–396. Springer, Heidelberg (2012)
17. Siksnys, L., Thomsen, C., Pedersen, T.B.: MIRABEL DW: managing complex energy data in a smart grid. In: Cuzzocrea, A., Dayal, U. (eds.) DaWaK 2012. LNCS, vol. 7448, pp. 443–457. Springer, Heidelberg (2012)
18. Song, I-Y., et al.: An analysis of many-to-many relationships between fact and dimension tables in dimensional modeling. In: Proceedings of DMDW (2001)
19. Yu, B., et al.: On managing very large senor-network data using bigtable. In: Proceedings of CCGrid (2012)
20. Zubcoff, J., Pardillo, J., Trujillo, J.: A UML profile for the conceptual modelling of data-mining with time-series in data warehouses. Inf. Softw. Technol. **51**(6), 977–992 (2008)