Int. J. Appl. Math. Comput. Sci., 2019, Vol. 29, No. 1, 31–49
DOI: 10.2478/amcs-2019-0003

MODELING AND QUERYING FACTS WITH PERIOD TIMESTAMPS IN DATA WAREHOUSES

GIOVANNI MAHLKNECHT [a], ANTON DIGNÖS [a,*], NATALIJA KOZMINA [b]

[a]Faculty of Computer Science
Free University of Bozen-Bolzano, Dominikanerplatz 3, 39100 Bozen, Italy
e-mali: {giovanni.mahlknecht,dignoes}@inf.unibz.it

[b]Faculty of Computing
University of Latvia, Raiņa bulvāris 19, Riga, LV-1586, Latvia
e-mail: natalija.kozmina@lu.lv

In this paper, we study various ways of representing and querying fact data that are time-stamped with a time period in a data warehouse. The main focus is on how to represent the time periods that are associated with the facts in order to support convenient and efficient aggregations over time. We propose three distinct logical models that represent time periods as sets of all time points in a period (instant model), as pairs of start and end time points of a period (period model), and as atomic units that are explicitly stored in a new period dimension (period* model). The period dimension is enriched with information about the days of each period, thereby combining the former two models. We use four different classes of aggregation queries to analyze query formulation, query execution, and query performance over the three models. An extensive empirical evaluation on synthetic and real-world datasets and the analysis of the query execution plans reveal that the period model is the best choice in terms of runtime and space for all four query classes.

Keywords: data warehouse, time periods, logical models.

1. Introduction

Time plays a central role in data warehouses with an omnipresent time dimension that is used to timestamp facts in order to capture the historical evolution of the data over time. Typically, facts capture single events that happened at a certain time point, or a cumulative situation at a certain point in time. Such facts are timestamped with a time point at some base granularity, e.g., days.

In many application domains, however, fact data that describe a situation over a time period (or time interval) are important. As a motivating example, consider hotel bookings in a tourism domain, which are described by a time period with a start date and an end date. For an analysis of the tourism business, the time periods of the bookings capture valuable information. For instance, the average duration of hotel bookings or the number of bookings that are longer than a week are typical queries that tourism organization are interested in. Such events with associated time periods can be represented by facts that store the start point and the end point of each event, or by facts that store an (instant) event for each time point between the start time and the end time.

There has been extensive research on dealing with various aspects of temporal information in data warehouses, often referred to as temporal data warehousing; see the works of Golfarelli and Rizzii (2009b; 2011) for an overview. Most of the past research on temporal data warehousing concentrates on changes in the dimension tables, often referred to as slowly changing dimensions (Jensen *et al.*, 2010; Kimball and Ross, 2013; Faisal and Sarwar, 2014) and on the evolution of data warehouse schemas (Blaschka *et al.*, 1999; Wrembel and Bebel, 2007; Ahmed *et al.*, 2014). How to model and represent changes in the fact data has been less studied, with a few exceptions (e.g., Bliujute *et al.*, 1998; Goller and Berger, 2015), but none of them investigates data warehouse scenarios with aggregation queries over time. Many of the temporal concepts have been borrowed and adapted

[*]Corresponding author

from the temporal database community, which for several decades studied different ways on how to model, represent and query temporal data, with point-based, period-based and parametric models being the most prominent representatives (Jensen *et al.*, 1994; Böhlen *et al.*, 2009; Jensen and Snodgrass, 2009). Each model favors certain query types, and stumbles with other query types.

In this paper, we investigate different ways regarding how to model, represent and query *facts with period timestamps* in a data warehouse, i.e., facts that hold over a time period. The central question is how to model time periods in order to make querying simple and efficient for different types of aggregation queries. We propose three different logical models. First, the *instant model* represents time periods as the set of all time points in the period, that is, a fact is represented by a set of point events. Second, the *period model* represents time periods as a pair of a start and an end point, that is, a fact is stored by one row with two time points representing respectively the start and end time of the period. Finally, the *period** *model* is a combination of the former two models. Each fact is represented by one row that is timestamped with a time period, which is stored in a new dimension table. In addition, a bridge table is used to connect the period dimension to the date dimension in order to explicitly represent the individual days of each time period.

The three models produce fact tables of different size and are expected to favor different types of queries. For queries that involve each time point, one might expect the instant model to be superior over the period model. In contrast, if the queries involve only the endpoints of time periods, the period model should be the preferred solution. The period* model aims to combine the advantages of the other two models. The results of an extensive empirical evaluation and an analysis of the query plans show, however, that for all aggregation queries the period model is superior in terms of query time. Moreover, it is the model that generally has the lowest storage costs.

The key contributions of this paper can be summarized as follows:

- We describe three different logical models for the representation of fact data with period timestamps in data warehouses: the instant model, the period model and the period* model.
- We discuss and analyze the formulation of four different types of aggregation queries as well as their rollup variants over the three models.
- We conduct an extensive experimental evaluation with synthetic and real-world datasets to show storage costs, extract-transform-load performance, and query time of the proposed three models.

The rest of the paper is structured as follows. Section 2 provides an overview of related work, followed by a case study in Section 3. In Section 4, we present three different models to represent period timestamped facts. In Section 5, we show the formulation of temporal aggregation queries in these models, and in Section 6 the formulation of rollup queries. In Section 7, query execution plans are analyzed, followed by experimental results in Section 8. Section 9 concludes the paper.

2. Related work

Several decades of intensive research activities regarding temporal databases studied various aspects of representing and querying temporal data in database management systems. The research work concentrated on various data models and query languages (Jensen *et al.*, 1994; Jensen and Snodgrass, 2009; Böhlen *et al.*, 2009; Dignös *et al.*, 2012; 2016) as well as evaluation algorithms for selected operators, such as temporal aggregation (e.g., Kline and Snodgrass, 1995; Zhang *et al.*, 2001; Moon *et al.*, 2003; Yang and Widom, 2003; Böhlen *et al.*, 2006b; Piatov and Helmer, 2017) and temporal joins (e.g., Zhang *et al.*, 2002; Gao *et al.*, 2005; Piatov *et al.*, 2016; Bouros and Mamoulis, 2017; Cafagna and Böhlen, 2017); for an overview, see the work of Böhlen *et al.* (2018). Fundamental concepts that emerged in this research are the distinction between different time dimensions, e.g., valid time, when a fact is true in the modeled reality, and transaction time, when a fact has been stored in the database. For the representation of temporal data, instant-based models timestamp each fact with a time point, whereas period-based models timestamp each fact with a time period.

It is not surprising that there has been a lot of research on dealing with temporal information in data warehouses, often referred to as temporal data warehousing. Many concepts and solutions have been adopted from temporal database research and then further developed, such as changes in dimension data, changes in factual data, schema changes, querying temporal data, and designing temporal data warehouses. A comprehensive survey is given by Golfarelli and Rizzi (2009b). Table 1 provides a summary of previous contributions in temporal data warehousing that are most related to our work, classified along the following criteria: *temporal support* (schema, dimensions, or fact table), *support for validity periods* (e.g., supported, implicitly derivable), *support for aggregation*, and use of *DBMS and standard SQL* or *an extension*. N/A indicates the lack of precise information. A more detailed discussion follows.

Most of the past research on temporal data warehousing concentrates on changes in the dimension tables, often referred to as slowly changing dimensions (Kimball and Ross, 2013; Faisal and

Table 1. Summary of the related research studies on temporal data warehousing.

Approach	Temporal support	Support for validity periods	Aggregation	DBMS / standard SQL or extension
Ahmed *et al.*, 2014	schema	N/A	supported	N/A
Blaschka *et al.*, 1999	schema	implicitly derivable	basic aggregation functions	Oracle 11g / Seq-SQL
Blaschka *et al.*, 1999	schema	implicitly derivable	supported (formally)	N/A
Malinowski and Zimányi, 2008	schema	supported	supported (with semantic assumptions)	N/A
Wrembel and Bebel, 2007	schema	supported (for DW schema versions)	supported	Oracle 10g - Oracle SQL
Eder *et al.*, 2002	schema	supported (for DW schema versions)	supported	Oracle 8.1 - Oracle SQL
Kimball and Ross, 2013	dimension	supported (e.g., in Type 2, 6, 7)	N/A	N/A
Koncilia, 2003	dimension	supported (formally)	supported (formally)	N/A
Faisal and Sarwar, 2014	dimension	supported	supported	N/A
Bliujute *et al.*, 1998	fact table	supported (in state model)	N/A	Oracle 7.2 / Oracle SQL
Garani *et al.*, 2016	fact table	supported (as time periods)	N/A	N/A / BTN-SQL
Goller and Berger, 2013	fact table	implicitly derivable	limited	N/A
Goller and Berger, 2015	fact table	implicitly derivable	N/A over members of the version dimension	SQLite / most of the standard SQL
Koncilia *et al.*, 2014	fact table	implicitly derivable for non-overlapping periods	basic aggregation functions	PostgreSQL / SQL-like language
This work	**fact table**	**supported**	**supported**	**standard SQL**

Sarwar, 2014), and on the evolution of data warehouse schemas (Blaschka *et al.*, 1999; Wrembel and Bebel, 2007; Ahmed *et al.*, 2014). Kimball and Ross (2013) proposed for the first time three basic techniques for representing changing attributes in the dimension tables, together with five variations thereof. They vary in terms of complexity and the amount of historical information that can be captured. MultiDimER (Malinowski and Zimányi, 2008) is a conceptual model that provides rich support for modeling various temporal aspects in data warehouse dimensions, including the valid time of dimensional attributes, the lifespan of hierarchy levels, as well as the transaction time and the loading time into the data warehouse for both. As an example of handling data warehouse schema evolution, the COMET temporal model (Eder *et al.*, 2002) documents all changes by keeping multiple versions of the schema with timestamps. Koncilia (2003) extended the COMET model to a bi-temporal one supporting both valid time and transaction time.

Even if less prominent, there has also been some research on modeling and representing changes in fact data. The most widely used approach is that facts represent (point) events and are timestamped with a time point at some base granularity, e.g., days. However, some applications require storing data that differ from event facts. Hence more advanced concepts are needed.

Similarly to slowly changing dimensions, the work by Goller and Berger (2013) defines the principles of handling slowly changing measures to cover support for fact value changes. Such changes might be caused by the necessity to re-define the measurement function at certain time points (e.g., VAT value change or introduction of the euro) or to update already accumulated measure values. In particular, proactive versioning (slowly changing measures of Type 2) ensures a logical fragmentation of the fact table according to different measure definitions with a fixed valid time. Since the fact table follows an instant model, measure validity periods are implicitly derived from both the fact table and version dimension. Goller and Berger (2013) point out restrictions in rollup capabilities since measure values cannot be aggregated across different versions. Similarly, in the work of Malinowski and Zimányi (2008) the concept of valid time for facts to fix the same values in discontinuous time spans is represented by periods in an object-relational schema. However, an empirical evaluation of querying period-timestamped facts is not provided.

A more recent work by Goller and Berger (2015) proposes a new type of slowly changing measures, named Type 2.5 (or multiple active versions). Type 2.5 is a hybrid approach that combines the best features of Types 2 and 3, namely, simplicity and analytical power. In order to distinguish between different definitions of measures, version tuples are used instead of validity periods. The concept of maintaining multiple versions of the same measure (so-called several truths) ensures flexibility for OLAP querying. A major drawback of this approach is a noticeable increase in storage consumption compared to other types of slowly changing measures.

Garani *et al.* (2016) introduce the temporal starnest schema—a combination of the star and snowflake schema extended with temporal characteristics. To reduce the number of joins, the model does not explicitly store a time dimension. Instead, each fact has two time attributes representing respectively valid time and transaction time, expressed either as time points, time periods, or as temporal elements. Temporal nested queries are processed using an extension of SQL, named BTN-SQL, which is not available in current DBMSs. The main focus of this work is on selection and join queries with temporal predicates, such as, for instance, CONTAINS. Aggregation queries over time are not supported.

Koncilia *et al.* (2014) describe a model for storing and processing sequential data in an OLAP style using an SQL-like language. Though this approach supports both time points and time periods, the period data is limited to sequences of non-overlapping periods that are derived from point events, such as the time periods between two consecutive sensor measurements (e.g., all light and temperature sensor data are stored in one fact table as a sequence of events).

Bebel *et al.* (2015) focus on a proof of concept implementation of the Seq-SQL language. The study is targeted to analyze time point-based sequential data in an OLAP-like manner followed by an empirical evaluation of queries. Seq-SQL is particularly useful for querying ordered data, such as workflow management systems, indications of smart meters, etc. In the majority of cases, the processing time is linearly dependent on the data volume or query selectivity. In our study, the (input) data is period-based and its measures hold for a whole time period.

Blijute *et al.* (1998) present an analysis of queries executed on three different data warehouse schemas that represent the same temporal information. More specifically, a time series model, an instant fact model with one time attribute (called event model), and a period fact model with two time attributes (called state model) are presented. The advantages of the state model are a smaller number of records in the fact table, easiness of query formulation, and the absence of redundant information. This work is the most relevant one to ours. In our paper, we extend the problem of modeling and querying period-timestamped fact data by adding the period* model, which stores the periods as atomic elements in a new dimension table. The period dimension is extended with a date dimension table to explicitly associate the individual time points with each period, thereby combining the period and the instant fact model. Moreover, we analyze various types of aggregation queries, which is arguably the most important class of queries in data warehousing, whereas Bliujute *et al.* (1998) consider only selection queries.

3. Case study

This work is motivated by a case study in collaboration with "Landesverband der Tourismusorganisationen Südtirols"[1] (LTS), which is the umbrella association for tourist boards and tourism associations in South Tyrol. LTS maintains a database that stores various pieces of information about hotel bookings. The excerpt of the dataset we are using consists of $837,648$ bookings made by tourists in the period from January 1st, 2015, to December 30th, 2017. The data records *checkin* and *checkout* dates of the booked stay, the *destination* town, the *category* of the lodging, and optional information on the home *country* of the guest and the number of *adults* and *children* of the booking. A summary of the schema is shown in Table 2.

The aim of this case study is to develop a data warehouse solution that helps decision makers in the tourism sector to analyze the performance of individual hotels or entire geographic zones. This can be done by

[1] `https://www.lts.it/`.

Table 2. Schema of the dataset.

Field	Description	Mandatory
checkin	check in date	yes
checkout	check out date	yes
destination	town of the lodging	yes
category	category of the lodging	yes
country	home country of the guest	no
adults	number of adults	no
children	number of children	no

studying various key performance indicators, such as the number of guests, the number of overnight stays, or the number of bookings at different levels of detail.

A vital piece of information in these data is carried by the *time period* of the bookings formed by the checkin and the checkout date. Analyzing these periods reveals, among other things, insights into the average duration of the bookings, the number of bookings of at least one week, short bookings of one or two days, etc. These numbers are changing depending on the season and other parameters, e.g., for several years there has been a tendency towards more frequent but shorter stays.

These and other queries have varying complexity, and their performance and the ease of formulation are more or less favored by the underlying data model and representation. In the sequel, we will discuss different data models to store period-timestamped fact data, such as hotel bookings, and how to answer these types of queries.

4. Data model

In this section, we first present and discuss two different conceptual models and then three different logical models for representing period time-stamped facts, using our case study as running example.

4.1. Conceptual model. Depending on the focus of the analysis, the information about hotel bookings can be modeled in at least two different ways. First, each hotel booking can be modeled as a single *booking* fact/event, which is state-oriented data and has an associated time period. Second, each hotel booking is modeled as a set of *overnight stays*, one stay for each day in the booking period. This distinction is very similar to the concept of period-timestamped models (Lorentzos, 2009) and point-timestamped models (Toman, 2009) in temporal databases.

Figure 1 shows an excerpt of the dimensional fact model (Golfarelli and Rizzi, 2009a) for overnight stays. Each overnight event is time-stamped with a time instant, i.e., the date of the stay. A booking is modeled as one or more overnight stays. In order to associate the overnight stays with the correct booking, a degenerated dimension *bookingid* is used. Associating overnight stays to bookings is important for all queries that need to determine, for example, the first/last day of a booking or its duration. The date dimension corresponds to a traditional date dimension, representing individual days plus a number of dimensional attributes that are useful for the analysis. The remaining three dimensions are category, country of the guest, and destination. The country-continent hierarchy is an optional element, indicated by a dash in the edge. An overnight fact has two measures, namely, the number of adults and the number of children in the booking.

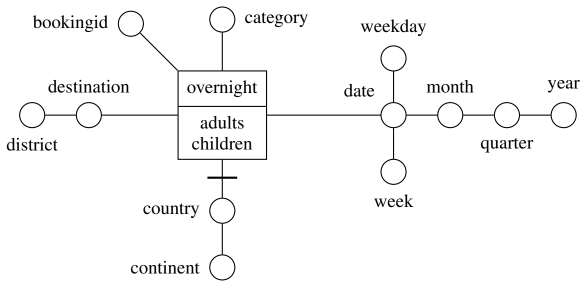

Fig. 1. Dimensional fact model for *overnight stays*.

Figure 2 shows an excerpt of the dimensional fact model for bookings. The booking fact is timestamped with a time period, represented by a checkin date and a checkout date. Date is a shared dimension between checkin and checkout, which is indicated by a doubled circle and the role names on the edges. Different from the model for overnight stays, there is no need for the degenerated dimension `bookingid` since *bookings* are the primary events. The remaining dimensions as well as the measures are identical to the model in Fig. 1.

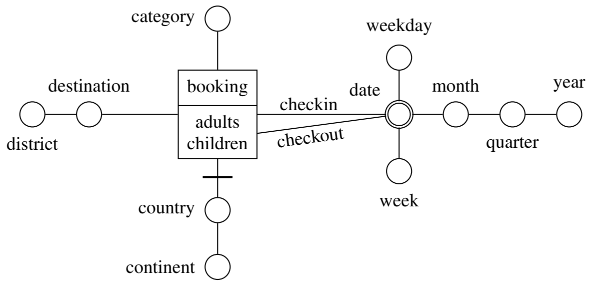

Fig. 2. Dimensional fact model for *bookings*.

4.2. Logical model. Starting from the conceptual models presented above, we now describe three different logical models. To keep the models simple, we show only the fact table and the temporal dimension.

The first model is termed the *instant model* and is shown in Fig. 3. It is derived from the conceptual model for overnight stays in Fig. 1. The fact table *i_overnight_fact* stores one row for each overnight stay of a booking, all of which have the same *bookingid*. In the dimension table *date_dimension* we use the surrogate key *dateid*, as suggested by Kimball and Ross (2013). Surrogate keys are integers that are sequentially assigned as needed. We highlight all surrogate keys with the suffix *id*. This model follows the traditional transaction model, where each row is timestamped with a time point at the granularity of the day. This kind of fact is also referred to as flow facts (Lenz and Shoshani, 1997) or event-oriented data (Bliujute *et al.*, 1998).

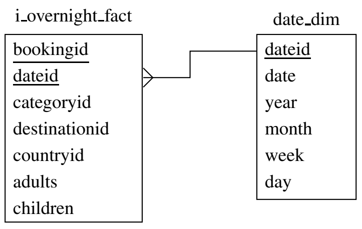

Fig. 3. Instant model.

The second model, termed the *period model*, is shown in Fig. 4 and derived from the conceptual model for bookings in Fig. 2. Each entry in the fact table represents a single booking event and is timestamped with a time period. The start and end points of the booking period are stored in the fields *checkinid* and *checkoutid*, respectively, which are both foreign keys to the date dimension. Thus, each row in the fact table has two foreign keys to the date dimension, but there are fewer entries in the fact table.

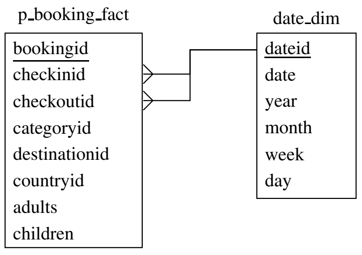

Fig. 4. Period model.

The third model, termed the *period* model*, is shown in Fig. 5 and tries to combine the advantages of the instant and the period ones. Similarly to the latter, each fact represents a single booking event that is timestamped with a time period. However, instead of storing the begin date and the end date of the booking period, a new dimension table *period_dim* is introduced, which stores all time periods of the bookings together with the checkin date, checkout date, and duration. Therefore, each fact contains only one foreign key to the period dimension, which further reduces the size of the fact table. Moreover, it is straightforward to obtain start and end dates of bookings as well as their duration. To efficiently support the analysis of overnight stays, each period in *period_dim* is explicitly associated with the days it is composed of. To represent this many-to-many relationship between the periods in *period_dim* and the days in the *date_dim* table, we use a bridge table (Kimball and Ross, 2013). Each period is related to multiple days, and each day can be part of many periods. The *checkinid* and *checkoutid* in *period_dim* are foreign keys to *date_dim*.

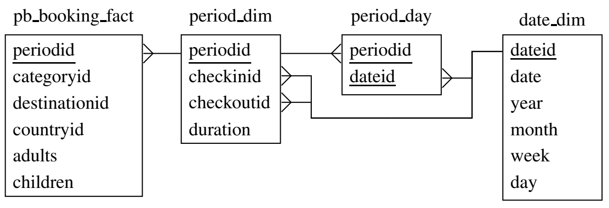

Fig. 5. Period* model.

5. Querying

In this section, we show how to query the three models introduced above. We use four different queries shown in Table 3, which are classified along two dimensions. First, we distinguish the query class depending on the grouping. The grouping may be either by each time instant of a period (i.e., time instant) or by period boundary. For instance, query "*Q1: Total number of guests for each day*" groups by time instants, since the guests of a booking count for each day of their stay. In contrast, query "*Q2: Total number of guests that checked in for each day*" groups by period boundary; more precisely, by the checkin day, i.e., the first day of the booking period. Second, we distinguish the query class depending on the aggregation, i.e., the attributes over which the aggregation function is applied. This can be either a nontemporal attribute or a temporal attribute. For instance, query Q1 aggregates over a nontemporal attribute, i.e., time is not used in the aggregation function. In contrast, query "*Q3: Average booking duration of guests for each day*" aggregates over a temporal attribute; more specifically, over the duration of the bookings. Similarly, query "*Q4: Average booking duration of guests that checked in for each day*" also aggregates over a temporal attribute, but in contrast to query Q3 it groups by a period boundary.

Query Q1. The SQL code for query Q1 on the instant model is shown in Fig. 6(a). This query is straightforward as the fact table stores the individual overnight stays. Hence, no transformation of time periods into time points is required.

To evaluate query Q1 on the period model, the SQL statement has to break the time period into time instants (cf. Fig. 6(b)). This transformation is achieved by first retrieving the dates of the period boundaries (checkin

Table 3. Query types.

Type	Grouping	Aggregation	Example
Type 1	Time instant	nontemporal	Q1: Total number of guests for each day
Type 2	Period boundary	nontemporal	Q2: Total number of guests that checked in for each day
Type 3	Time instant	temporal	Q3: Average booking duration of guests for each day
Type 4	Period boundary	temporal	Q4: Average booking duration of guests that checked in for each day

and checkout days) from the surrogate keys and then joining the period with all instants it contains (`d.date`). An alternative solution would be to use PostgreSQL's `generate_series` function[2] to generate the time instants.

The SQL code for Q1 in the period* model is shown in Fig. 6(c). The model contains both information about the period boundaries as well as the individual instants of each booking period. Answering query Q1 in this model is straightforward and requires only a join between the fact table and the dimension tables.

```
SELECT   d.date, SUM(adults+children)
FROM     i_overnight_fact f,
         date_dim d
WHERE    f.dateid = d.dateid
GROUP BY d.date
```

(a) instant model

```
SELECT   d.date, SUM(adults+children)
FROM     p_booking_fact f,
         date_dim din,
         date_dim dout,
         date_dim d
WHERE    f.checkinid = din.dateid AND
         f.checkoutid = dout.dateid AND
         d.date BETWEEN din.date AND dout.date
GROUP BY d.date
```

(b) period model

```
SELECT   d.date, SUM(adults+children)
FROM     pb_booking_fact f,
         period_dim p,
         period_day pd,
         date_dim d
WHERE    f.periodid = p.periodid AND
         p.periodid = pd.periodid AND
         pd.dateid = d.dateid
GROUP BY d.date
```

(c) period* model

Fig. 6. Query *"Q1: Total number of guests for each day"* on different fact models.

Query Q2. The SQL code for query Q2 for all models is shown in Fig. 7. The instant model first has to undergo a transformation (`transf`) to determine the checkin day of a booking, which is computed as the minimum over all days of the booking. Note that we also need to apply a "dummy" aggregation function on the number of adults and children; we use the (`MIN`) aggregation function, but others can be used as well since the numbers are the same for all instants. After this transformation, a simple aggregation evaluates the final result, summing up the number of guests for each checkin day.

For the period and period* model, query Q2 is rather native (cf. Figs. 7(b) and (c)). In both cases, the checkin day is first retrieved from the dimension table(s) and then used for the aggregation. Since facts are stored at the level of booking rather than overnight stays, no pre-aggregation is required.

```
WITH transf AS
(
SELECT   bookingid,
         MIN(d.date) checkin,
         MIN(adults) adults,
         MIN(children) children
FROM     i_overnight_fact f,
         date_dim d
WHERE    f.dateid = d.dateid
GROUP BY bookingid
)
SELECT   checkin, SUM(adults+children)
FROM     transf
GROUP BY checkin
```

(a) instant model

```
SELECT   din.date, SUM(adults+children)
FROM     p_booking_fact f,
         date_dim din,
         date_dim dout
WHERE    f.checkinid = din.dateid AND
         f.checkoutid = dout.dateid
GROUP BY din.date
```

(b) period model

```
SELECT   din.date, SUM(adults+children)
FROM     pb_booking_fact f,
         period_dim p,
         date_dim din
WHERE    f.periodid = p.periodid AND
         p.checkinid = din.dateid
GROUP BY din.date
```

(c) period* model

Fig. 7. Query *"Q2: Total number of guests that checked in for each day"* on different fact models.

Query Q3. The evaluation of query Q3 on the instant model needs, as in the previous query, determination of the start and the end date of each booking in order to be able to determine the duration of the booking (see Fig. 8(a)). This requires joining the overnight facts with the date dimension, grouping by `bookingid`, and computing the minimum and the maximum of the dates.

[2]https://www.postgresql.org/docs/10/static/functions-srf.html#FUNCTIONS-SRF-SERIES.

To evaluate query Q3 on the period model, the SQL statement has to break the period into time instants, although this is a period query (see Fig. 8(b)). The reason for this is that the result has to be grouped for each day within a booking.

Again as for Q1 and Q2, the evaluation of query Q3 on the period* model has a simple solution using standard data warehouse joins between fact and dimension tables. The model contains the complete information about duration of the bookings as well as the individual days of the booking. Hence, the result can easily be reported at the granularity of days without the need to generate the "missing" days. The corresponding SQL statement is shown in Fig. 8(c).

```
WITH transf AS
(
SELECT   bookingid,
         MIN(dd.date) checkin,
         MAX(dd.date) checkout
FROM     i_overnight_fact f,
         date_dim d
WHERE    f.dateid = d.dateid
GROUP BY bookingid
)
SELECT   d.date, AVG(checkout - checkin + 1)
FROM     transf,
         i_overnight_fact f,
         date_dim d
WHERE    transf.bookingid=f.bookingid AND
         f.dateid = d.dateid
GROUP BY d.date
```

(a) instant model

```
SELECT   d.date, AVG(dout.date - din.date + 1)
FROM     p_booking_fact f,
         date_dim din,
         date_dim dout,
         date_dim d
WHERE    f.checkinid = din.dateid AND
         f.checkoutid = dout.dateid AND
         d.date BETWEEN din.date AND dout.date-1
GROUP BY d.date
```

(b) period model

```
SELECT   d.date, AVG(duration)
FROM     pb_booking_fact f,
         period_dim p,
         period_day pd,
         date_dim d
WHERE    f.periodid = p.periodid AND
         p.periodid = pd.periodid AND
         pd.dateid = d.dateid
GROUP BY d.date
```

(c) period* model

Fig. 8. Query *"Q3: Average duration of bookings for each day"* on different fact models.

Query Q4. The SQL code for query Q4 on the different models is shown in Fig. 9. Also for this query the instant model has to undergo a transformation. The reason is that the period boundaries are required both for the grouping and for the aggregation function. For the other two models, query Q4 is rather simple, since only the period boundaries are required for the evaluation. Compared with the period model, similarly to query Q3, the period* model has the additional advantage of storing the duration with the period information. Hence no calculation is needed. Note that this cannot conveniently be supported by the period model, as it would be required to store the duration in the fact table, which substantially increases storage costs.

```
WITH transf AS
(
SELECT   bookingid,
         MIN(d.date) checkin,
         MAX(d.date) checkout
FROM     i_overnight_fact f,
         date_dim d
WHERE    f.dateid = d.dateid
GROUP BY bookingid
)
SELECT   checkin, AVG(checkout - checkin + 1)
FROM     transf
GROUP BY checkin
```

(a) instant model

```
SELECT   din.date, AVG(dout.date - din.date + 1)
FROM     p_booking_fact f,
         date_dim din,
         date_dim dout
WHERE    f.checkinid = din.dateid AND
         f.checkoutid = dout.dateid
GROUP BY din.date
```

(b) period model

```
SELECT   din.date, AVG(duration)
FROM     pb_booking_fact f,
         period_dim p,
         date_dim din
WHERE    f.periodid = p.periodid AND
         p.checkinid = din.dateid
GROUP BY din.date
```

(c) period* model

Fig. 9. Query *"Q4: Average duration of guests that checked in for each day"* on different fact models.

6. Rollup queries

A frequent operation in data warehouse applications are rollup queries in order to produce subtotals at different levels. For this, data need to be aggregated at various granularities along dimensional hierarchies. Since the focus of this paper is on temporal information, we show how to achieve aggregations at higher levels in the time dimension hierarchy.

For the aggregation, we have to distinguish between two semantics of the measures we want to aggregate on; this is similar to constant and malleable attributes in temporal databases (Böhlen *et al.*, 2006a; Dignös *et al.*, 2013). For measures that are additive along the time dimension, no changes (besides the different grouping) are required for the queries. An example of such an additive measure in our application scenario is the number of overnight stays. If we count the number of overnight stays per week and year instead of per day, this only affects the grouping of the aggregation. The reason for this is that each day (base granularity) of the data has to be accumulated to calculate the value at a higher granularity.

```
SELECT   week, year, SUM(adults+children)
FROM     (SELECT DISTINCT bookingid, adults, children,
                          week, year
          FROM i_overnight_fact f,
               date_dim d
          WHERE f.dateid = d.dateid) f
GROUP BY week, year
```

(a) instant model

```
SELECT   d.week, d.year, SUM(adults+children)
FROM     p_booking_fact f,
         date_dim din,
         date_dim dout,
         LATERAL( SELECT DISTINCT week, year
                  FROM date_dim d
                  WHERE d.date BETWEEN din.date
                                   AND dout.date) d
WHERE    f.checkinid = dout.dateid AND
         f.checkoutid = dout.dateid
GROUP BY d.week, d.year
```

(b) period model

```
SELECT   d.week, d.year, SUM(adults+children)
FROM     pb_booking_fact f,
         period_dim p,
         LATERAL( SELECT DISTINCT week, year
                  FROM period_day pd,
                       date_dim d
                  WHERE p.periodid = pd.periodid AND
                        pd.dateid = d.dateid) d
WHERE    f.periodid = p.periodid
GROUP BY d.week, d.year
```

(c) period* model

Fig. 10. Query *"Q1-yw: Total number of guests for each week and year"*; rollup of query Q1 for a non-additive measure for different fact models.

For measures that are non-additive along the time dimension hierarchy, duplicate elimination is required. An example of a non-additive measure in our application scenario is the number of guests in query Q1. For instance, if we count the number of guests per week and year instead of per day, the same guest that stays for several days in a week has to be counted only once and not for each of these days.

We now show how this aggregation of non-additive measures on higher levels of the time dimension hierarchy can be achieved by the three models, using query Q1 as an example. In particular, we show that for the period and period* model we can conveniently support it using lateral joins (Melton and Simon, 2002). These are are part of the SQL:1999 standard using the keyword `LATERAL` and allow to express correlations, for instance, in a nested subquery, within SQL's `FROM` clause. The lateral join is supported by many DBMSs such as Postgres as of version 9.3[3], Oracle as of version 12c, and IBM DB2 as of version 9.1. The Microsoft SQL Server also supports lateral joins, but uses a different keyword `OUTER APPLY` (Ben-Gan *et al.*, 2015). Figure 10(a) shows the query "*Q1-yw: Total number of guests for each week and year*" for the instant model. For this model, we need to de-duplicate the fact table *i_overnight_fact* based on the new granularity, since it possibly contains more occurrences of a single booking for the same week. After de-duplication, the aggregation can be applied similarly to query Q1.

For the period model, we can de-duplicate the join matches between the fact table and the date dimension. Figure 10(b) shows how this can be conveniently and systematically achieved using lateral joins. This ensures that the join will produce only one occurrence for each week and year.

Similarly, lateral joins can be used for the period* model as shown in Fig. 10(c). In this case, the source of possible duplicates resides in the bridge table, which are removed by the lateral subquery.

The approach adopted for query Q1 in Fig. 10 can be systematically applied to other granularities, such as months and years, and other queries with grouping on time instants, such as, e.g., query Q3. For groupings on period boundaries, such as queries of Type 2 (Q2) and type 4 (Q4), de-duplication is not required, since in all models queries are executed in the period representation (cf. Figs. 7 and 9), and thus each booking is only considered once.

7. Query execution

Before we move on to the empirical evaluation of the different models, we first review the execution plans of the most representative performance factors using our example queries on the different models. In particular, we focus on two main factors that affect the performance of the models. First, *time instant grouping* present in queries of Type 1 (Q1) and Type 3 (Q3): It requires the model and period* models to transform periods into instants before further processing, while it is naturally supported by the instant model. Second, *access to period boundaries* that is present in queries of types 2 (Q2), 3 (Q3), and 4 (Q4): It is very natural to the models that store periods, but requires a transformation for the instant model.

For the execution of the queries, we rely on index-based joins, which is common for data warehouse applications. To ensure this kind of joins for all query executions, we manually hint the optimizer to use index-based joins.[4] Almost all joins we perform are key-foreign key joins and by default PostgreSQL creates an index for each primary key. Hence, we do not need to create extra indices, with the single exception of an index on the `date` field of the date dimension. This index is used in the period model to join all instants within a period.

7.1. Time instant grouping.

To show the execution plans of the different models for time instant grouping, we use our example query Q1 as shown in Fig. 6.

[3]https://www.postgresql.org/docs/10/static/queries-table-expressions.html#QUERIES-LATERAL.

[4]https://www.postgresql.org/docs/10/static/runtime-config-query.html.

Instant model. The query plan for the instant model for query Q1 as shown in Fig. 6(a) is as follows:

```
                    QUERY PLAN
-------------------------------------------------------
HashAggregate
  Group Key: d.date
  -> Nested Loop
    -> Seq Scan on i_overnight_fact f
    -> Index Scan using dd_pk on date_dim d
          Index Cond: (dateid = f.dateid)
```

This query is very natural to this model, since instants are stored in the fact table. It requires only an index-join between the fact table and the date dimension, followed by an aggregation.

Period model. For the period model, query Q1 (cf. Fig. 6(b)) is executed as follows:

```
                    QUERY PLAN
-------------------------------------------------------
HashAggregate
  Group Key: d.date
  -> Nested Loop
    -> Nested Loop
      -> Nested Loop
        -> Seq Scan on p_booking_fact f
        -> Index Scan using dd_pk on date_dim din
             Index Cond: (dateid = f.checkinid)
      -> Index Scan using dd_pk on date_dim dout
           Index Cond: (dateid = f.checkoutid)
    ->Index Only Scan using date_idx on date_dim d
        Index Cond: ((date >= din.date) AND
                     (date <= dout.date))
```

First, the fact table is joined with the date dimension using an index-join to retrieve the checkin date from its surrogate key. A similar join is then performed to retrieve the checkout date. From these two dates, a third index-join retrieves all instants (days) between the checkin day and the checkout day. A final aggregation computes the result grouped by instants.

Period* model. For the period* model, the query plan for the same query Q1 (cf. Fig. 6(c)) is as follows:

```
                    QUERY PLAN
-------------------------------------------------------
HashAggregate
  Group Key: d.date
  -> Nested Loop
    -> Nested Loop
      -> Nested Loop
        -> Seq Scan on pb_booking_fact f
        -> Index Only Scan using pdimpk on period_dim p
             Index Cond: (periodid = f.periodid)
      -> Index Only Scan using pd_pk on period_day pd
           Index Cond: (periodid = p.periodid)
    -> Index Scan using dd_pk on date_dim d
         Index Cond: (dateid = pd.dateid)
```

Compared with the period model, this model requires an additional index-join, as it is required to first retrieve the period corresponding to a fact and then its corresponding instants. Different from the period model, this execution plan is based on equality joins only, while the period model performs a `BETWEEN-AND` index-join.

7.2. Access to period boundaries.

To show the execution plans of the different models when they access period boundaries, we use our example query Q2 as shown in Fig. 7.

Instant model. In order to be able to access period boundaries, as in this case for the grouping (cf. Fig. 7(a)), the instant model needs first to transform the set of instants into a period. The query plan is as follows:

```
                    QUERY PLAN
-------------------------------------------------------
HashAggregate
  Group Key: trans.checkin
  CTE trans
    -> HashAggregate
         Group Key: f.bookingid
      -> Nested Loop
        -> Seq Scan on i_overnight_fact f
        -> Index Scan using dd_pk on date_dim d
             Index Cond: (dateid = f.dateid)
  -> CTE Scan on trans
```

After joining the fact table with the date dimension, we have two aggregations. The first aggregates instants into periods and the second performs the final aggregation with grouping on the period boundary.

Period model. Access to period boundaries is a native operation to that of the period model. One index-join is required to retrieve from the date dimension the corresponding period boundary, which is then aggregated. This leads to the following query plan:

```
                    QUERY PLAN
-------------------------------------------------------
HashAggregate
  Group Key: din.date
    -> Nested Loop
      -> Seq Scan on p_booking_fact f
      -> Index Scan using dd_pk on date_dim din
           Index Cond: (dateid = f.checkinid)
```

Period* model. The execution plan of the period* model is again similar to that of the period model, the only exception being that one additional index-join is required to retrieve the data of a period boundary. The query plan is as follows:

```
                    QUERY PLAN
-------------------------------------------------------
HashAggregate
  Group Key: din.date
    -> Nested Loop
      -> Nested Loop
        -> Seq Scan on pb_booking_fact f
        -> Index Scan using pdimpk on period_dim p
             Index Cond: (periodid = f.periodid)
      -> Index Scan using dd_pk on date_dim din
           Index Cond: (dateid = p.checkinid)
```

7.3. Rollup.

To show the execution plans of the different models for rollup queries, we use our example query Q1-yw as shown in Fig. 10.

Instant model. For rollup queries on non-additive measures, such as the number of guests, duplicate elimination of instants has to be applied (cf. Section 6). The instant model stores instants at the fact level, thus duplicates need to be removed from the fact table. The following query plan shows the execution of the query from Fig. 10(a):

```
                    QUERY PLAN
-------------------------------------------------------
HashAggregate
  Group Key: d.year, d.week
    -> Unique
      -> Sort
         Sort Key: f.bookingid, f.adults, f.children,
                   d. week, d.year
        -> Nested Loop
          -> Seq Scan on i_overnight_fact f
          -> Index Scan using dd_pk on date_dim d
               Index Cond: (dateid = f.dateid)
```

First, the fact table is joined with the date dimension table to retrieve the week of a given instant of the fact, then duplicated instants of the same week and booking are removed, and finally the aggregation on weeks is performed.

Period model. Also for the period model we need a mechanism to remove duplicated instants once they arise from a join between fact table and the dimension table. The `LATERAL` feature however, allows applying duplicate elimination on a much smaller input. The query plan of the query from Fig. 10(b) is as follows:

```
                    QUERY PLAN
-------------------------------------------------------
HashAggregate
  Group Key: d.year, d. week
    -> Nested Loop
      -> Nested Loop
        -> Nested Loop
          -> Seq Scan on p_booking_fact f
          -> Index Scan using dd_pk on date_dim din
               Index Cond: (dateid = f.checkinid)
        -> Index Scan using dd_pk on date_dim dout
             Index Cond: (dateid = f.checkoutid)
      -> Unique
        -> Sort
           Sort Key: d.year, d. week
          -> Index Scan using date_idx on date_dim d
               Index Cond: ((date >= din.date) AND
                            (date <= dout.date))
```

After two index-joins between the fact table and the date dimension to retrieve for each booking period the checkin date and the checkout date, a lateral index-join is performed to retrieve the corresponding instants. Unlike for the instant model, duplicate elimination is not applied to the full join result, but multiple times; each time only on a much smaller set, namely, the instants of a single booking.

Period* model. Similar as for the period model, also the period* model uses the lateral feature in combination with duplicate elimination on a much smaller intermediate result compared to the instant model. For the period* model the query plan is as follows:

```
                    QUERY PLAN
-------------------------------------------------------
HashAggregate
  Group Key: d.year, d. week
  -> Nested Loop
    -> Nested Loop
      -> Seq Scan on pb_booking_fact f
      -> Index Only Scan using pdim_pk on period_dim p
           Index Cond: (periodid = f.periodid)
    -> HashAggregate
         Group Key: d.year, d. week
      -> Nested Loop
        -> Index Only Scan using pd_pk on period_day pd
             Index Cond: (periodid = p.periodid)
        -> Index Scan using dd_pk on date_dim d
             Index Cond: (dateid = pd.dateid)
```

8. Empirical evaluation and discussion

In this section, we compare the different models on storage costs, extraction-transform-load (ETL) performance, and query time. The focus is to provide an understanding as to which model is most suitable.

8.1. Setup and datasets.

We used a PostgreSQL server version 10.1 installed on a GNU/Linux machine with 47 GB RAM. All models were implemented in the same database. The common dimension tables, i.e., *date_dimension, time_dimension, country,* and *category* are shared by the three models. For the comparison of storage costs, the shared tables were not considered.

We use two datasets, a synthetic dataset, SYNTH and the real-world dataset TOURISM described in Section 3. SYNTH has the same schema as TOURISM. For the generation of SYNTH, we used the following parameters and distributions. The start time points of periods are uniformly distributed between 0 and $10,000$, and their duration follows a Zipfian distribution with parameter $\theta = 1.7$. The values of the remaining attributes are uniformly distributed. Attributes *adults*, *children* and *category* are uniformly distributed in the range from 1 to 5, and as a *destination* we assign a random value out of 118 different destinations.

We vary three parameters in the experiments. First, the cardinality of the facts stored in each of the three implemented models. The size is expressed in the number of bookings stored in the data warehouse. Second, we vary the ratio in percentage between the number of distinct periods and the total number of periods. A high percentage means that we have many different periods, i.e., 100% means all periods are different, and a low percentage means that many periods are the same. For this purpose we fixed the number of periods to 100k and normalized the start and end points of each period to multiples of integers in the range between 1 and 100, thus reducing the number of distinct periods. For larger normalization values the percentage of distinct periods decreases. Third, we vary the parameter θ of the Zipfian distribution of the period length of each fact skewed towards short periods. We vary θ between 1.4 (low skew) and 2.0 (high skew). For $\theta = 1.4$ the average period length is 126.4, and for $\theta = 2.0$ it is 4.25. Figure 11 shows the average period length and the resulting number of instant tuples for varying θ.

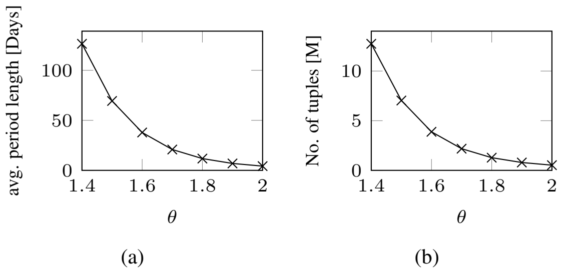

Fig. 11. Average period length (a) and the number of instant facts (overnights) (b) for varying θ.

8.2. Storage costs. We first analyze the storage costs of the three data models. We use the SYNTH dataset and vary the number of bookings from 10k to 100M. The results are shown in Fig. 12(a). The measured memory includes the data as well as the indices for primary key and key foreign-key relationships. As expected, the period

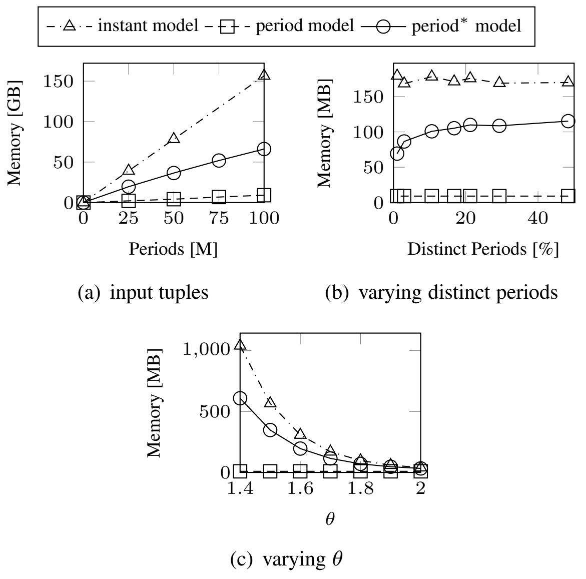

Fig. 12. Storage costs for the dataset SYNTH.

model is the best one in terms of storage costs, as it requires storing only one tuple for each booking in the input data (cf. Fig. 4). The instant model on the other hand requires storing for each period of the raw data d instant facts, where d is the average duration of periods (in base granularity). The average period duration of this dataset is about 20, therefore the instant model requires to storing $20 \times n$ instant tuples, where n is the number of periods. For the period* model, the storage costs depend on the number of distinct periods. In the extreme case when all periods are different, the period* model requires more space than the instant model due to the overhead of the additional bridge table. However, in practice this is rarely the case, as we will see in our real-world dataset.

Next we vary the number of distinct booking periods. The results are shown in Fig. 12(b). As expected, since the number of instant facts remains constant, the storage costs for the instant and period models do not depend on the number of distinct periods. For the period* model, on the other hand, the storage costs increase for a large number of distinct periods.

Figure 12(c) shows the impact of the booking durations on the three models. We vary the parameter θ of the Zipfian distribution. Low values of θ imply low skew and thus, on the average, longer periods, whereas high values of θ imply high skew and thus on average, shorter periods. We can see that the instant and period* model are affected by the duration of periods, while, as expected, the period model is not affected.

Table 4 shows the storage costs of the three models for our real-world dataset TOURISM. The instant model has the largest memory footprint as it requires to store a large number of instant facts. For each of the 5.77M tuples, 26 bytes are required, leading to a theoretical size of 143 MB. The effective space required by PostgreSQL instead is 332 MB, mainly due to the overhead per tuple, i.e., the `HeapTupleHeader` and *alignment padding*[5]. The period model has the lowest memory footprint, requiring only 15% of the instant model's storage. This is in line with the expectations. The average duration of bookings is 6.9 days. Thus the effective size of the fact table in the instant model is 6.9 times larger than the fact table in the period model. In the period* model, the fact table `pb_booking_fact` has similar size as the fact table in the period model. The overhead for the bridge tables is small. Also this is in line with the expectations. For this real-world dataset, many bookings cover the same time period. Only 10, 111 distinct periods for 835, 071 bookings exist. Thus only for 1.21% of the periods, instants have to be stored.

Table 5 shows the storage costs for the synthetic dataset SYNTH with 100M periods and parameters $\theta = 1.7$ and a period-normalization of 1, i.e., 2.5% of the periods are distinct. Also for this dataset the instant model has the largest memory footprint. It requires 18.9 times more storage than the period model which is in line with the average duration of periods that is 19.96. In contrast to the real world dataset, for this dataset the period* model requires about 42% of the storage of the instant model. The reason behind this is that the dataset has a large number of short periods that are identical while long periods are almost all distinct. This results in a much

[5]`https://www.postgresql.org/docs/10/static/catalog-pg-type.html`.

Table 4. Storage costs for the real-world dataset TOURISM.

Model	Relation	Number of tuples	B/tuple	Eff. table size	Eff. index size	Total
instant model	i_overnight_fact	$5,777,524$	24	332.0 MB	123.8 MB	**455.8 MB**
period model	p_booking_fact	$835,071$	30	54.4 MB	17.9 MB	**72.3 MB**
period* model	pb_booking_fact	$835,071$	30	41.6 MB	17.9 MB	
	pb_period	$10,111$	16	0.5 MB	0.23 MB	
	pb_periodday	$130,494$	8	4.6 MB	2.8 MB	
				46.7 MB	**20.93 MB**	**67.63 MB**

Table 5. Storage costs for the synthetic dataset SYNTH 100M periods.

Model	Relation	Number of tuples	B/tuple	Eff. table size	Eff. index size	Total
instant model	i_overnight_fact	$2,096,028,180$	24	102 GB	43.85 GB	**145.85 GB**
period model	p_booking_fact	$100,000,000$	30	6.4 GB	2.1 GB	**8.5 GB**
period* model	pb_booking_fact	$100,000,000$	30	4.86 GB	2.09 GB	
	pb_period	$2,544,529$	16	107.5 MB	54.5 MB	
	pb_periodday	$994,151,595$	8	33.6 GB	20.8 GB	
				38.54 GB	**22.94 MB**	**61.48 GB**

higher average duration of (distinct) periods compared with the average duration of bookings. Recall that the period* model only stores instants for distinct periods. The average duration of all bookings is 19.96 while the average duration of distinct periods is 390.7. The same phenomena can be observed as less pronounced in the TOURISM dataset, where the the average duration of the bookings is 6.9 while average duration of distinct periods is 12.9.

8.3. ETL performance. In the next experiment we compare the runtimes for the ETL process of the three models on the SYNTH dataset. The raw data are period-timestamped and the schema is shown in Table 2. For the comparison of the different models we do not consider data cleaning steps, as they would be the same for all data models. Rather we show the runtime of the ETL process that is specific to each approach, i.e., to import and transform the data in the correct data model. We also do not use any external tools, but only standard SQL statements. After the transformation step we add primary keys and key-foreign key constraints. The results for varying the different parameters are shown in Fig. 13. In general, we can see that the runtime in all cases is very similar to the storage costs (cf. Fig. 12) and thus affected by the additional data generated by the models. For all three models the runtimes increase linearly with the number of input tuples. As our raw data consists of periods, the ETL process for the period model is very simple and performs better than the other models. The instant model requires splitting each period into instant facts and thus has the highest runtime. To create instants from each period we use PostgreSQL's `generate_series` function.[6] The performance of the period* model lies between the period model and instant model. For the period* model the runtimes depend on the number of distinct periods. For datasets where almost all periods are different, the runtime of the ETL process is larger than for the instant model, due to the overhead of the additional tables for the model. On the other hand, if most of the periods are the same, the runtime for the period* model gets close to the period model. Each distinct period with the mapping to instants is stored only once in the bridge table and is reused for multiple facts. The overhead of the additional period dimension is relatively small.

For the real world dataset TOURISM, the number of distinct periods is very small (1.21%). The number of tuples required by the period* model and period model is thus very similar, and the runtime of the ETL process for these two models is almost the same: 2.5 seconds for the period model and 3.1 seconds for the period* model. The ETL process for the instant model, however, has a runtime of 18 seconds. The runtime difference is caused by the larger number of tuples that have to be inserted (cf. Section 8.2).

8.4. Query time. We now evaluate the performance for our example queries from Table 3 on the synthetic dataset SYNTH. First, we vary the size of the argument relation.

[6]`https://www.postgresql.org/docs/10/static/functions-srf.html#FUNCTIONS-SRF-SERIES`.

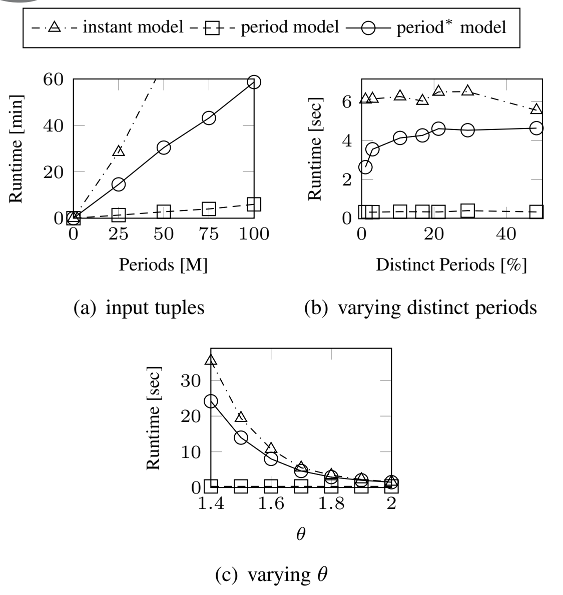

Fig. 13. Runtime of ETL for the SYNTH dataset.

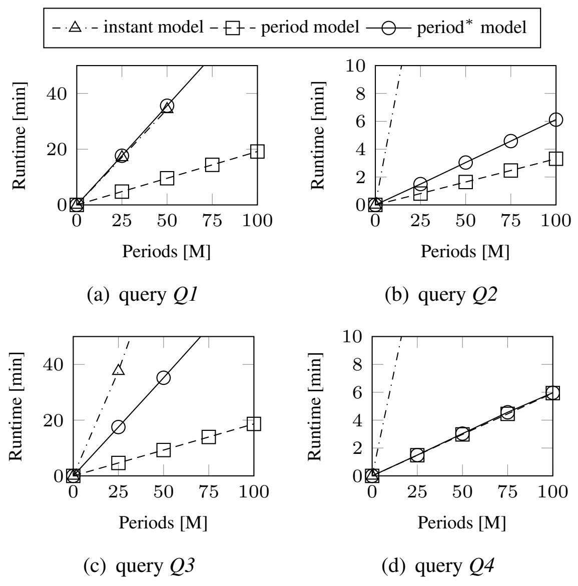

Fig. 14. Runtime for varying input size on the SYNTH dataset.

For all three models we generate the same number of booking facts. Next, we analyze the influence of the percentage of distinct periods in the argument relation, and in a third experiment we analyze the influence of varying duration of each period fact.

8.4.1. Size of the argument relation. In the first query time experiment, we evaluate the performance of the three models with our example queries for varying sizes of the argument relations. The result is shown in Fig. 14.

Figure 14(a) depicts the three models for query Q1 (type 1 queries). Despite the fact that this query groups by time instants and does not use period boundaries, the period model is the best one, while the instant and period* model have a higher runtime. This surprising result is due to the smaller number of index scans of the period model compared with the other approaches. For instance, for the 100M periods dataset (cf. Table 5), we have 100M booking facts and $2,096$M overnight facts. By consulting the query plans for this query in Section 7.1 we can see that the instant model performs one index-join. This index-join performs for each of the $2,096$M facts in the fact table one index scan on the date dimension, resulting in a total of $2,096$M index scans.

For the period model we have three index joins, but each with a much smaller number of index scans. The fact table for this model only contains 100M facts, which results in 100M index scans to retrieve the checkin date from the date dimension. The result of this join is 100M tuples that are joined again with the date dimension to retrieve the checkout date, yielding again 100M index scans. The result (100M tuples) is then joined with the date dimension to find all instants between the checkin date and the checkout date, using the index on the `date` field of the date dimension. In total, the period model performs only 300M index scans, which is much less compared with the instant model with $2,096$M index scans. For the period* model we have a similar behavior as for the instant model. Although the fact table contains only 100M facts, the index join with the `period_day` table produces an intermediate result of $2,096$M tuples. For each of these tuples in the intermediate result, one index scan with the date dimension is performed, which eventually results in an even higher number of index scans and runtime compared to the instant model.

The results for query Q2 (Type 2) are shown in Fig. 14(b). For the instant model that for this query is similar to that of query Q1. The dominating factor is the number of index scans for the index-join between the large fact table and the time dimension (cf. the query plan in Section 7.2) to retrieve, in this case, the checkin date. The period and period* model for this query have a comparable runtime, which is much smaller than for the instant model. Both models can use the period representation and avoid a large number of index scans. The runtime of the period* model is slightly higher than for the period one, since it requires one more index-join (cf. the query plan in Section 7.2).

For query Q3 (Type 3) the results are shown in Fig. 14(c). For the period and period* model, we have the same query plans and performance as for query Q1. The only difference for these models is the value that is aggregated, and in both cases it is available after the join. For the instant model the runtime is higher than for query Q1, since it has to perform twice an index-join for the large fact table, first to retrieve the period boundaries and then to retrieve the date from the surrogate key, yielding a large number of index scans. Also for this query the period model is the best one.

The results for query Q4 (type 4) are shown in Fig. 14(d). For this query the runtimes are similar to query Q2. For all models, we have the same joins as for query Q2.

8.4.2. Distinct periods. In this experiment, we analyze how the number of distinct periods affects query performance. The results are shown in Fig. 15. In contrast to storage and ETL, the number of distinct periods has no effect on the query time of the three models. In all cases the query time is constant.

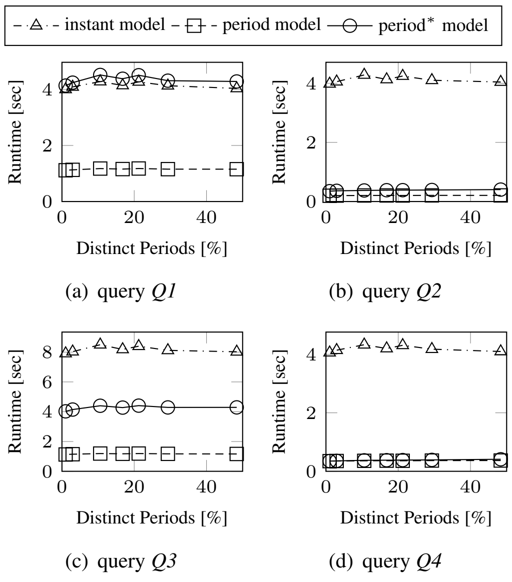

Fig. 15. Runtime for a varying number of distinct periods on SYNTH dataset.

8.4.3. Period length. Here we analyze the query time for different period lengths. The number of bookings is fixed to $100,000$. The result for the different queries and varying parameter θ for the Zipfian distribution of durations is shown in Fig. 16. Recall that for low values of θ the durations are, on the average, longer compared to high values of θ (cf. Fig. 11). The instant model is highly affected by the period length for all queries. For this model, the period length directly influences the size of the fact table. Therefore, longer periods cause a higher number of index scans during the joins for all queries. For the period and period* model the period length only affects query Q1 and Q3, where periods are joined with their instants. The period model is less affected, since in contrast to the period* model it does not need to join large intermediate results. For high θs and thus very short period lengths, the runtime of the three models converges. The reason for this is that the number of periods and instants, i.e., the number bookings and overnights, and thus the size of the fact tables become the same.

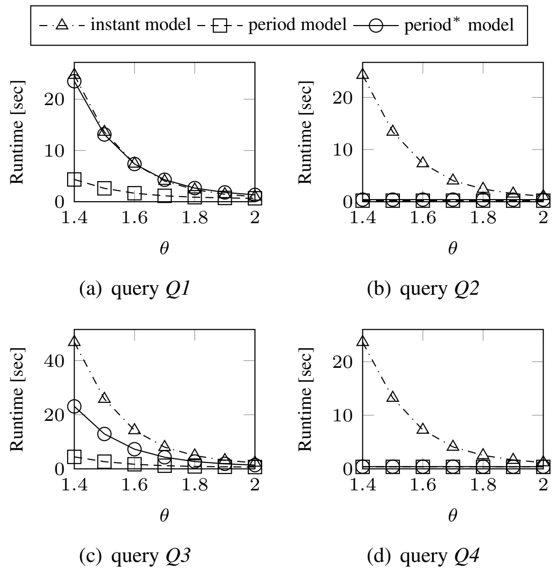

Fig. 16. Runtime for varying period lengths using parameter θ on the SYNTH dataset.

8.4.4. Rollup queries. Next, we show the performance of the three models for rollup queries. The results for a varying number of bookings are shown in Fig. 17. For all queries the period model performs best.

For the queries Q2-yw and Q4-yw, the runtimes are identical to the corresponding queries without rollup (cf. Fig. 14), since the grouping is on period boundaries, and thus duplicate elimination for the rollup is not required (cf. Section 6).

For the queries Q1-yw and Q3-yw, the runtimes of all approaches are higher than for the corresponding queries without rollup, since all approaches need to remove duplicates to perform the rollup (see Fig. 10 for Q1-yw). The gap between the instant model and the two other models widens. The reason for this is that the instant model needs to remove duplicates at the end on a much larger intermediate result (size of the overnight fact table) compared with the other models (cf. query plan in Section 7.3). The period and period* model use the lateral join feature and thus remove duplicates for each booking fact individually.

8.4.5. Real-world dataset. We evaluate the three models on our real-world dataset TOURISM. The results for our four queries (Q1, Q2, Q3, and Q4) and corresponding rollup queries (Q1-yw, Q2-yw, Q3-yw, and Q4-yw) are shown in Fig. 18.

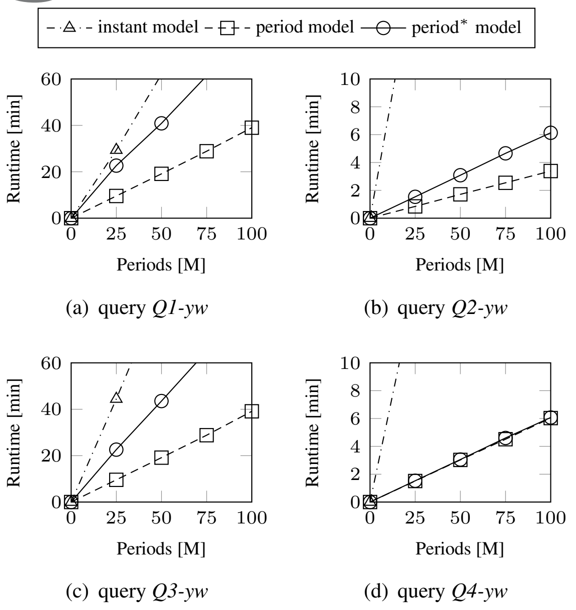

Fig. 17. Runtime of rollup queries for a varying input size on the SYNTH dataset.

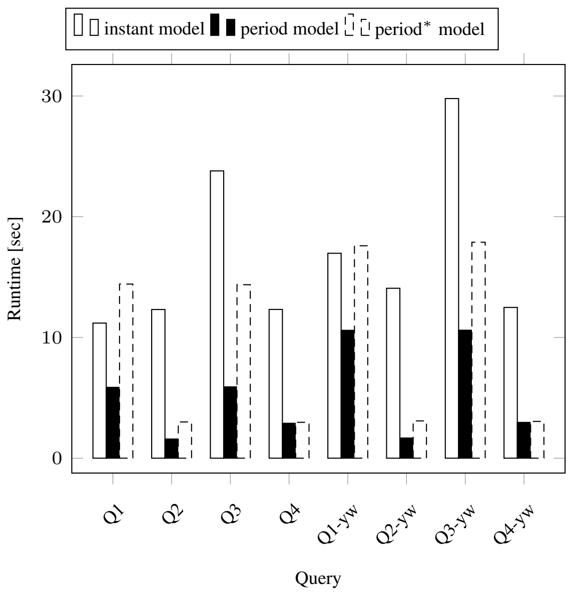

Fig. 18. Runtime for the real-world dataset TOURISM.

The results confirm our findings from the evaluation on the synthetic dataset. The period model is always the best one in terms of query time. The other models struggle with the high number of index scans for the queries with time instant grouping (Q1, Q3, Q1-yw, and Q3-yw). For the queries with grouping on period boundaries (Q2, Q4, Q2-yw, Q4-yw), the period* model is only slightly slower than the period one, since it has to perform one additional join for the bridge table. For the rollup queries with time instant grouping (Q1-yw and Q3-yw), the instant model deteriorates most compared to the others, since it applies duplicate elimination on a much higher intermediate result, while the other approaches use repeated duplicate elimination in combination with lateral joins.

8.5. Modeling duration as a measure. In the last experiment, we slightly modified the three models and stored the duration of the periods as an additional measure in the fact table. For our experiments we introduced one duration measure at the lowest granularity of days in each fact table, i.e., *i_overnight_fact*, *p_booking_fact*, and *pb_booking_fact*. We measured the impact of these models on the storage requirements as well as on the runtime of both the ETL phase and the sample queries Q3 and Q4 (and their rollup variants Q3-yw and Q4-yw) using the synthetic dataset with 25M periods. Queries Q1 and Q2 were excluded as they do not use the duration of periods.

In terms of storage costs, the instant model, which is already the most space-consuming one among the three models, shows the highest growth of almost 20%. For the period model, the storage increased by approximately 10%. The period* model has the lowest impact on the storage costs, with an increase of 1.37%.

In the ETL phase, the impact of storing the duration of periods in the fact tables is small, with an increase in the processing time of less than 5% for all three models.

The runtimes for queries Q3 and Q4 are summarized in Table 6, where a star indicates the queries over the model with the duration measure in the fact table. For

Table 6. Runtime in seconds for queries Q3 and Q4 with a duration measure in the fact table.

Query	Model		
	instant	period	period*
Q3	2,256	278	1,052
Q3*	1,042	271	1,044
Q3-yw	2,656	575	1,355
Q3-yw*	1,806	573	1,352
Q4	924	88	89
Q4*	1,079	50	89
Q4-yw	923	90	91
Q4-yw*	1,101	51	91

query Q3, which performs an aggregation on the duration in combination with instant grouping, the instant model takes advantage of the duration measure in the fact table. The nested query for retrieving the start and end of each period can be omitted, yielding a substantial reduction of the runtime: 60%, for Q3 and 38% for Q3-yw. Still, the instant model remains the slowest model and requires most storage. Queries Q4 and Q4-yw on the instant model become 25% slower with a duration measure in the fact table. This is due to two reasons: first, the fact table is bigger; second, without a duration measure an index only scan on the primary key index that contains *dateid* can be used (with the duration measure, a sequential scan is used.)

With the period model, query Q3 cannot take advantage of the additional attribute. Since the grouping is on time instants, each period has to be split into instants. Thus, queries Q3 and Q3-yw show more or less the same performance with and without a duration measure in the fact table. In contrast, queries Q4 and Q4-yw improved by 40% since the additional duration measure in the fact table saves one join.

For the period* model, we observed almost no performance improvement for our queries. The reason is that the join with the dimension table *period_dim* is still required. Since *period_dim* already contains the duration of the period, adding the duration measure to the fact table does not provide a further speed-up.

9. Conclusions

In this paper, we studied different ways of modeling and querying period-timestamped fact data in a data warehouse. We discussed three distinct logical models that model time periods, respectively, as a set of all time points in the time period (instant model), as a pair of start and end time points (period model), and as a combination of the two former models (period* model). We showed different classes of aggregation queries and their rollup variants, both of which are crucial for data analysis in data warehouses. We analyzed the three models based on their execution plans for the different queries and presented the results of a detailed empirical evaluation, with respect to storage costs, ETL performance, and query time using synthetic and real-world datasets from the tourism domain. Our analysis reveals the period model to be the clear winner in terms of modeling and querying complexity, ETL performance, and storage costs, as well as query performance, due to its lower number of index traversals compared to the other models.

Future work points in several directions. First, we will consider other time periods, such as seasons, i.e., time periods that either are associated to times of the year (e.g., summer, winter season) or holidays (e.g., Christmas, Easter holidays). Second, we want to consider additional facts mentioned by Höpken *et al.* (2013), which are valid in particular periods of time, e.g., feedback from the guests on bookings, the consumption of single products or services of any kind (e.g., food and drinks, tickets), provided capacities of products and services on a daily basis, or marketing activities represented by the time period and related to certain products or services. Third, it would be interesting to consider attribute data that varies over time and is valid in certain periods of time, such as the room price per day. How to integrate such changes of attribute data into the three models requires further investigations.

Acknowledgment

This work was supported by the Open Access Publishing Fund provided by the Free University of Bozen-Bolzano.

References

Ahmed, W., Zimányi, E. and Wrembel, R. (2014). A logical model for multiversion data warehouses, *Proceedings of the 16th International Conference on Data Warehousing and Knowledge Discovery, DaWaK 2014, Munich, Germany*, pp. 23–34.

Bebel, B., Cichowicz, T., Morzy, T., Rytwinski, F., Wrembel, R. and Koncilia, C. (2015). Sequential data analytics by means of Seq-SQL language, *Proceedings of the 26th International Conference on Database and Expert Systems Applications, DEXA 2015, Valencia, Spain*, Part I, pp. 416–431.

Ben-Gan, I., Machanic, A., Sarka, D. and Farlee, K. (2015). *T-SQL Querying*, Microsoft Press, Redmond, WA.

Blaschka, M., Sapia, C. and Höfling, G. (1999). On schema evolution in multidimensional databases, *Proceedings of the 1st International Conference on Data Warehousing and Knowledge Discovery, DaWaK 1999, Florence, Italy*, pp. 153–164.

Bliujute, R., Saltenis, S., Slivinskas, G. and Jensen, C.S. (1998). Systematic change management in dimensional data warehousing, *Proceedings of the 3rd International Baltic Workshop on DB and IS, Riga, Latvia*, pp. 27–41.

Böhlen, M.H., Dignös, A., Gamper, J. and Jensen, C.S. (2018). Temporal data management—an overview, *in* E. Zimányi (Ed.), *Business Intelligence and Big Data*, Springer International Publishing, Cham, pp. 51–83.

Böhlen, M.H., Gamper, J. and Jensen, C.S. (2006a). An algebraic framework for temporal attribute characteristics, *Annals of Mathematics and Artificial Intelligence* **46**(3): 349–374.

Böhlen, M.H., Gamper, J. and Jensen, C.S. (2006b). Multi-dimensional aggregation for temporal data, *Proceedings of the 10th International Conference on Extending Database Technology, EDBT 2006, Munich, Germany*, pp. 257–275.

Böhlen, M.H., Gamper, J., Jensen, C.S. and Snodgrass, R.T. (2009). SQL-based temporal query languages, *in* L. Liu and M. Tamer Özsu (Eds.), *Encyclopedia of Database Systems*, Springer, New York, NY, pp. 2762–2768.

Bouros, P. and Mamoulis, N. (2017). A forward scan based plane sweep algorithm for parallel interval joins, *Proceedings of the VLDB Endowment* **10**(11): 1346–1357.

Cafagna, F. and Böhlen, M.H. (2017). Disjoint interval partitioning, *The VLDB Journal* **26**(3): 447–466.

Dignös, A., Böhlen, M.H. and Gamper, J. (2012). Temporal alignment, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA*, pp. 433–444.

Dignös, A., Böhlen, M.H. and Gamper, J. (2013). Query time scaling of attribute values in interval timestamped databases, *Proceedings of the 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia*, pp. 1304–1307.

Dignös, A., Böhlen, M.H., Gamper, J. and Jensen, C.S. (2016). Extending the kernel of a relational DBMS with comprehensive support for sequenced temporal queries, *ACM Transactions on Database Systems* **41**(4): 26:1–26:46.

Eder, J., Koncilia, C. and Morzy, T. (2002). The COMET metamodel for temporal data warehouses, *Proceedings of the 14th International Conference on Advanced Information Systems Engineering, CAiSE 2002, Toronto, Canada*, pp. 83–99.

Faisal, S. and Sarwar, M. (2014). Handling slowly changing dimensions in data warehouses, *Journal of Systems and Software* **94**: 151–160.

Gao, D., Jensen, C.S., Snodgrass, R.T. and Soo, M.D. (2005). Join operations in temporal databases, *The VLDB Journal* **14**(1): 2–29.

Garani, G., Adam, G.K. and Ventzas, D. (2016). Temporal data warehouse logical modelling, *International Journal of Data Mining, Modelling and Management* **8**(2): 144–159.

Golfarelli, M. and Rizzi, S. (2009a). *Data Warehouse Design: Modern Principles and Methodologies*, McGraw-Hill, Inc., New York, NY.

Golfarelli, M. and Rizzi, S. (2009b). A survey on temporal data warehousing, *International Journal of Data Warehousing and Mining* **5**(1): 1–17.

Golfarelli, M. and Rizzi, S. (2011). Temporal data warehousing: Approaches and techniques, *in* D. Taniar and L. Chen (Eds.), *Integrations of Data Warehousing, Data Mining and Database Technologies—Innovative Approaches*, Information Science Reference, London, pp. 1–18.

Goller, M. and Berger, S. (2013). Slowly changing measures, *Proceedings of the 16th International Workshop on Data Warehousing and OLAP, DOLAP 2013, San Francisco, CA, USA*, pp. 47–54.

Goller, M. and Berger, S. (2015). Handling measurement function changes with slowly changing measures, *Information Systems* **53**: 107–123.

Höpken, W., Fuchs, M., Höll, G., Keil, D. and Lexhagen, M. (2013). Multi-dimensional data modelling for a tourism destination data warehouse, *Proceedings of the International Conference on Information and Communication Technologies in Tourism 2013, Insbrusck, Austria*, pp. 157–169.

Jensen, C.S., Pedersen, T.B. and Thomsen, C. (2010). *Multidimensional Databases and Data Warehousing*, Synthesis Lectures on Data Management, Morgan & Claypool Publishers, San Rafael, CA.

Jensen, C.S. and Snodgrass, R.T. (2009). Temporal database, *in* L. Liu and M. Tamer Özsu (Eds.), *Encyclopedia of Database Systems*, Springer, New York, NY, p. 2957.

Jensen, C.S., Soo, M.D. and Snodgrass, R.T. (1994). Unifying temporal data models via a conceptual model, *Information Systems* **19**(7): 513–547.

Kimball, R. and Ross, M. (2013). *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*, 3rd Edn., Wiley Publishing, Hoboken, NJ.

Kline, N. and Snodgrass, R.T. (1995). Computing temporal aggregates, *Proceedings of the 11th International Conference on Data Engineering, ICDE 1995, Taipei, Taiwan*, pp. 222–231.

Koncilia, C. (2003). A bi-temporal data warehouse model, *Proceedings of the 15th Conference on Advanced Information Systems Engineering, CAiSE 2003, Klagenfurt, Austria*, Vol. 74.

Koncilia, C., Morzy, T., Wrembel, R. and Eder, J. (2014). Interval OLAP: Analyzing interval data, *Proceedings of the 16th International Conference on Data Warehousing and Knowledge Discovery, DaWaK 2014, Munich, Germany*, pp. 233–244.

Lenz, H. and Shoshani, A. (1997). Summarizability in OLAP and statistical data bases, *Proceedings of the 9th International Conference on Scientific and Statistical Database Management, SSDBM 1997, Olympia, WA, USA*, pp. 132–143.

Lorentzos, N.A. (2009). Period-stamped temporal models, *in* L. Liu and M. Tamer Özsu (Eds.), *Encyclopedia of Database Systems*, Springer, New York, NY, pp. 2094–2098.

Malinowski, E. and Zimányi, E. (2008). A conceptual model for temporal data warehouses and its transformation to the ER and the object-relational models, *Data & Knowledge Engineering* **64**(1): 101–133.

Melton, J. and Simon, A.R. (2002). Advanced SQL query expressions, *in* J. Melton and A.R. Simon (Eds.), *SQL: 1999*, Morgan Kaufmann, Burlington, VA, pp. 265–353.

Moon, B., Vega Lopez, I.F. and Immanuel, V. (2003). Efficient algorithms for large-scale temporal aggregation, *IEEE Transactions on Knowledge and Data Engineering* **15**(3): 744–759.

Piatov, D. and Helmer, S. (2017). Sweeping-based temporal aggregation, *Proceedings of the 15th International Symposium on Advances in Spatial and Temporal Databases, SSTD 2017, Arlington, VA, USA*, pp. 125–144.

Piatov, D., Helmer, S. and Dignös, A. (2016). An interval join optimized for modern hardware, *Proceedings of the 32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland*, pp. 1098–1109.

Toman, D. (2009). Point-stamped temporal models, *in* L. Liu and M. Tamer Özsu (Eds.), *Encyclopedia of Database Systems*, Springer, New York, NY, pp. 2119–2123.

Wrembel, R. and Bebel, B. (2007). Metadata management in a multiversion data warehouse, *Journal on Data Semantics* **8**: 118–157.

Yang, J. and Widom, J. (2003). Incremental computation and maintenance of temporal aggregates, *The VLDB Journal* **12**(3): 262–283.

Zhang, D., Markowetz, A., Tsotras, V.J., Gunopulos, D. and Seeger, B. (2001). Efficient computation of temporal aggregates with range predicates, *Proceedings of the 20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 2001, Santa Barbara, CA, USA*, pp. 237–245.

Zhang, D., Tsotras, V.J. and Seeger, B. (2002). Efficient temporal join processing using indices, *Proceedings of the 18th International Conference on Data Engineering, ICDE 2002, San Jose, CA, USA*, pp. 103–113.

Giovanni Mahlknecht received his MSc degree in computer science from the Free University of Bozen-Bolzano in 2015. Since then he has been a PhD candidate there. His research interests include interval-timestamped databases, time series data, and temporal data analytics.

Anton Dignös received his MSc degree in computer science from the Free University of Bozen-Bolzano in 2010 and his PhD degree in computer science from the University of Zurich in 2014. Since then he has been an assistant professor at the Free University of Bozen-Bolzano in Italy. His current research interests include interval-timestamped databases, time series data, data summarization, and data visualization.

Natalija Kozmina received her PhD in 2015 in computer science from the University of Latvia in Riga. She has been a researcher at the Faculty of Computing of the University of Latvia since 2016. Previously she had worked as an assistant professor, teaching courses in the area of databases and software requirements analysis, and had held the position of a senior systems analyst of the IT Department, University of Latvia. Her research interests include data warehousing, big data, design of business intelligence solutions, and software requirement analysis.

Received: 18 April 2018
Revised: 13 August 2018
Re-revised: 6 October 2018
Accepted: 27 November 2018