Systematic Change Management in Dimensional Data Warehousing

R. Bliujute S. Saltenis G. Slivinskas C. S. Jensen

Department of Computer Science, Aalborg University
Fredrik Bajers Vej 7E, DK–9220 Aalborg Øst, DENMARK
{rasa, simas, giedrius, csj}@cs.auc.dk

Abstract

With the widespread and increasing use of data warehousing in industry, the design of effective data warehouses and their maintenance has become a focus of attention. Independently of this, the area of temporal databases has been an active area of research for well beyond a decade. This article identifies shortcomings of so-called star schemas, which are widely used in industrial warehousing, in their ability to handle change and subsequently studies the application of temporal techniques for solving these shortcomings.

Star schemas represent a new approach to database design and have gained widespread popularity in data warehousing, but while they have many attractive properties, star schemas do not contend well with so-called slowly changing dimensions and with state-oriented data. We study the use of so-called temporal star schemas that may provide a solution to the identified problems while not fundamentally changing the database design approach. More specifically, we study the relative database size and query performance when using regular star schemas and their temporal counterparts for state-oriented data. We also offer some insight into the relative ease of understanding and querying databases with regular and temporal star schemas.

1 Introduction

Data warehousing [3] is one of the fastest growing segments of the database management market, which, in turn, is one of the biggest markets for software. The area of data warehousing has grown out of a need for decision support that was not met by existing database management technologies. Without data warehousing, the data of an enterprise resides in different databases that belong to systems aimed at supporting the day-to-day business of the enterprise. These databases are often termed operational data stores, and the systems are termed on-line transaction processing systems (OLTP systems) or operational systems. Paraphrasing Kimball [7], OLTP systems are designed to meet the needs of someone who *turns* the wheels of an enterprise. As a result, these systems do not meet the needs of someone who *watches* the wheels and tries to find the ways of making them turn even better. The objective of data warehousing is to meet this need by enabling enterprises to exploit the data that is entered into their operational data stores for decision support, allowing them to base business decisions on careful analyses of accumulated data capturing the past performance of the enterprise.

To enable such analyses, a data warehouse system typically employs its own hardware and software and is separate from the operational systems. Data is extracted from the different operational data stores, is integrated, and is loaded into the data warehouse database, where it is then accumulated to make the database contain data up to 5–10 years old. The process of loading operational data into the warehouse is typically performed at regular intervals, e.g., every night. This architecture implies that the life cycle of the data warehouse consists of two alternating phases: the relatively short loading phase, where new data is inserted, and the querying phase, where data is not updated, only queried.

Data in data warehouses must be organized so that it is possible to query and analyze it on-line in the ways required by the analysts. It should be easy to formulate and execute queries. Most often these queries will not have a large answer set, but their execution can involve scanning huge amounts of data. Naturally, the requirement is to make these queries run as fast as possible.

To fulfill these goals, the organization of data in a data warehouse must be considered very carefully. Dimensional data modeling [7] is one of the main techniques used for this purpose, and star schemas serve as the means of representing dimensional data in relational databases. While being widely used and possessing desirable features, star schemas do not contend well with so-called *slowly changing dimensions* [7] and with state-oriented data, which occur when we want to record information that remains valid for a duration of time, e.g., when we want to record account balances in a bank.

This paper studies the use of techniques from temporal databases [12] for solving these problems. It describes temporal star schemas and provides a case-based, empirical comparison of temporal star schemas with regular star schemas, considering database size and query performance, as well as the ease of formulating queries. The temporal star schemas considered here capture valid time [6, 11], the time when the facts stored in the database are true in the modeled reality, as an opposite to transaction time, which concerns when the facts are current in the database.

The novel notion of temporal star schemas was introduced in a white paper by Leep Technology, Inc. [9]. While the performance of various representation schemas for temporal data has been the object of study (e.g., [1]), we are not aware of any works that have studied the size, query performance, or user-friendliness of temporal star schemas.

The paper is outlined as follows. Section 2 introduces star schemas, the distinction between state-oriented and event-oriented data, and identifies problems that motivate this study. Section 3 then describes the proposed solution, namely temporal star schemas, covering both state-oriented and event-oriented data. Having presented the solution, Section 4 proceeds to describe the settings for the experimental comparison of temporal and regular star schemas. Sections 5 and 6 compare database size and query performance, respectively. Finally, Section 7 concludes the paper.

2 Star Schema Problems

As a precursor to presenting the identified problems with regular star schemas, these are introduced, and we consider the different data representations that are used for event-oriented and state-oriented data.

2.1 Star Schemas

A star schema is a collection of tables where one central table, the fact table, contains foreign keys to all other tables, the dimension tables, which do not contain any foreign keys. Fact table rows typically contain, in addition to their foreign key references, one or more measured values, typically numerical, that describe some business process. A row in a dimension table contains data, typically textual, that describes aspects of rows in the fact table. The presence of a time dimension that describes when the measured values are in effect is a defining property of a data warehouse [3].

Let us take as an example a company that has a chain of stores that sell various products. Each row in the fact table models a sale of a product and records the quantity sold. Dimension tables Product, Store, and Time describe the products being sold, the stores where products are sold, and the times when products are sold. The star schema is shown in Figure 1. A fact table row may thus store the information that "on April 28, store X sold 200 items of product Y."

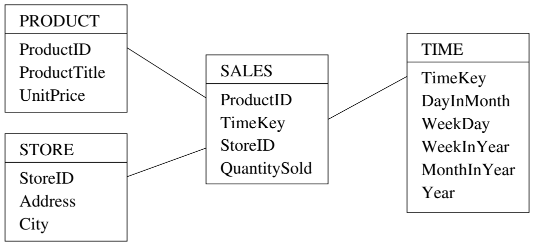

Figure 1: Sample Star Schema

The *granularity* of the measured values in the fact table rows is product by store by day. A single row then records all sales of a single product from a single store that occurred on a single day. Other possible granularities for the time dimension include minute, hour, week, and month. The choice of granularity affects the level of detail for the recorded information and the size of the database.

There is a difference between the attributes in fact versus dimension tables. The non-key attribute values in the fact table usually are numerical and vary continuously, e.g., a store *each day* sells *various* quantities of its products. In contrast the dimension table attribute values, such as store addresses and product titles, are textual, discrete, more or less stable, and serve as constraints in the business analyst's queries. However, sometimes the value of some attributes from the dimension tables change, and methods are needed to ensure that data in the database remains correct and consistent. Such dimensions, where some attribute values may change, are called slowly changing dimensions and are discussed in detail in Section 2.2 below.

We distinguish between two orientations of data: event-oriented data and state-oriented data. Examples of *event-oriented* data could be various types of buys, sales, inventory transfers, and financial transactions. The store example above thus concerns event-oriented data. Examples of *state-oriented* data include, e.g., prices, account balances, and inventory levels. The event and state models are complementary in the sense that having events, we can construct states that span between events; and likewise, having states we can construct events, representing the changes from one state to another.

The data in the star schema is *represented* either as events or as time-series, depending on the data orientation type. Event-oriented data is represented as events, i.e., each row in the fact table corresponds to an event in real life. State-oriented data can be represented as events, where a fact table row represents the event that caused the state of the object to change; or state-oriented data can be represented as time-series, where each row in the fact table represents a state at some fixed point in time, independently from the state changes of that object (this corresponds to sampling). We will revisit these issues in Section 2.3.

2.2 Problem 1: Ad-hoc Handling of Slowly Changing Dimensions

In the previous section, we assumed implicitly that all the dimension tables are independent of one another. However, this is not true: it can easily be seen that some, or most, dimensions are dependent on the omnipresent time dimension. For example, the prices of products in the product dimension may change over time. We term such time-dependent dimensions *slowly changing* [7].

Their presence leads to potential consistency and correctness problems, which should be handled. In the example, if the price of a product changes, the change must somehow be reflected in the Product dimension, to represent the correct price for all sales recorded in the fact table of the product. All fact table rows inserted before the change should still refer to the old price, while the new fact table rows (inserted after the change) should refer to the new price.

As proposed by Kimball [7], we can use three methods to handle slowly changing dimensions. Each of these methods provides a different degree of fidelity. Here, we describe only the most used method. In the description, we use a state-oriented star schema with a fact table, termed Balance, recording account balances and with dimension tables Time, Customer, and Account recording the times, customer informations, and account informations of the account balances, respectively. Assume that customer John Smith changed his address on January 15, 1997.

The most popular method for handling slowly changing dimensions proposes to create an additional dimension row at the time of the change with the new attribute value, getting an old description and a new description, see Figure 2.

Balance (before)

	GK	Bal
1/01	1	100
1/10	1	200

Customer (before)

GK	CId	Name	Address
1	1	John	Ulavej 1

Balance (after)

	GK	Bal
1/01	1	100
1/10	1	200
1/20	2	300

Customer (after)

GK	CId	Name	Address
1	1	John	Ulavej 1
2	1	John	Ritavej 5

Figure 2: Method Two: Balance and Customer Tables Before and After Changes

We create a new row in the Customer dimension for John Smith where the Address field has the new value. The new row must also have a unique key value. We cannot use, e.g., the social security numbers of customers as the key values because a customer such as John Smith has the same social security number independently of his address. There is a need for creating an artificial, generalized key (GK), which then requires special handling and consumes additional space in the data warehouse.

Another shortcoming is that the newly updated values in dimensions cannot be used on older facts and vice versa, which is sometimes meaningful. For example, the names of all bank branches may change, but there may still exist a need to track history in terms of the new branch names.

The described method is most generally applicable, but it, as the other two methods, has disadvantages, thus creating a need for a better solution to the problem of handling slowly changing dimensions.

2.3 Problem 2: Ineffective Handling of Change for State-oriented Data

We have mentioned that state-oriented data can be represented as either time-series or as events. Here, we identify potential data warehouse size and query performance problems that may arise from the management of state changes of state-oriented data for both representations.

Under realistic assumptions, the time-series representation inadvertently leads to loss of some information and repetition of other information. To see this, consider the banking example where we want to record the balances of all accounts. If we sample balances every hour then we will not have the complete account information because there can occur many transactions for a single account per hour. On the other hand, if we sample every minute then we will probably not loose information, but redundancy will be enormous because each account (there could be millions) will have a corresponding entry for each minute independently of whether the balance changed or not. In general, it can be difficult to choose a proper time granularity for the time-series representation, because some balances may change very often while others may change quite rarely. The time-series representation may be suitable when state transitions occur regularly in the real world.

With the event representation, there is a row in the fact table for each balance change. If some loss of information is acceptable, it is possible to record facts not for each event (change of a balance), but for several events together. For example, it may be adequate to record one fact for all changes to a balance per hour, independently of how many times the balance changed per hour, if it changed at least once. In the event representation, we also avoid redundant information because balances that do not change do not generate rows for the data warehouse.

However, the event representation creates query performance problems. With state-oriented data, it is likely that there frequently is a need to know the entire period during which a state persisted (e.g., the time when a balance remained at a constant value). We can efficiently find the Time dimension row giving the period's beginning, but finding the end of the period is neither simple nor cheap to compute.

In the next section, we will present the temporal star schema, which represents a possible solution to the problems mentioned in this section.

3 Temporal Star Schemas

Based on the previous section, we conclude that the problem of handling slowly changing dimensions does not have a fully satisfactory and conceptually clean solution using regular star schemas. In contrast, the temporal star schema systematically solves this problem. Besides, we have reasons to believe that the temporal star schema lets us achieve good query performance and a relatively compact data warehouse.

The idea behind the temporal approach to databases is simple: most of the things that we observe change in time, and the database system should capture the changes and represent the historical information in a natural way. The temporal star schema differs from the traditional one in its treatment of time. While the traditional star schema treats time as any other dimension, the temporal star schema omits the time dimension *table* and instead timestamps each row in every table of the schema, treating the fact table and the dimension tables equally with respect to time.

Adding detail to this design, we observe that event-oriented data and state-oriented data are represented differently in the temporal star schema. For event-oriented data, the event representation is used, meaning that each row in the fact table represents some event and has one timestamp, capturing the event occurrence time. For state-oriented data, we employ the state representation, meaning here that each row in the table describes some state and has two timestamps—the beginning and ending times of the period when the state persisted (see Figure 3).

Transaction Date	Transaction amount
1997/04/01	-100
1997/04/06	500

Begin Date	End Date	Account balance
1997/03/20	1997/04/01	10000
1997/04/01	1997/04/06	9900

Figure 3: Event and State Representations of Data

The event representation has the advantage of storing only one timestamp instead of two. Naturally this type of representation is well suited for event-oriented data and event-oriented queries. On the other hand, queries asking questions about states of some object in some periods of time will result in the comparison of timestamps from different rows and thus will execute slower. The state representation, while occupying more space, appears to be better suited for state-oriented data and state-oriented queries.

Taking a closer look at the data warehouse tables one can observe that dimension tables predominantly store state-oriented data. This implies having two timestamps in dimensions independently of whether the fact table represents events or states. For instance, in the banking example to the left in Figure 3, each fact table row represents one bank transaction and has one timestamp, while each Branch dimension row would have two timestamps. If some branch first became operational on July 14, 1990 and originally dealt only with savings accounts, but was upgraded to provide loans also on April 6, 1996, then the Branch dimension table will have two rows for that branch with timestamps July 14, 1990–April 6, 1996 and April 6, 1996–*now*.

Next, some dimension tables may not need timestamps at all. This occurs when a dimension models objects that do not change at all or when it does not describe real-world objects. An example of the latter could be the inventory status dimension in an inventory tracking data warehouse. This dimension lists only the possible values of the inventory status (received, inspected, boxed, etc.)

When using a temporal star schema, the problems associated with the handling of slowly changing dimensions disappear. Generalized keys are not needed and state-oriented data can be represented as states, diminishing the potential query performance and database size problems that may occur having events or time-series representations in the regular star schema. It is worth noting that the temporal approach is more informative than any regular star schemas—unlike the non-temporal star schemas, it associates the *exact* dates of the changes with all the historical information.

The presence of period timestamps raises some new issues. First, the value *now* has to be represented in one way or another—we choose the maximum date (9999/12/31) for this purpose. Second, referential integrity constraints [8] have to be maintained. Third, the primary key of any table augmented with timestamps is the original key plus one or two timestamps. Joins of such tables become more complex: for the two rows to join, their original keys must match, and their timestamps must overlap.

4 Settings for Comparison of Star Schemas

The previous sections compared regular and temporal star schemas at the conceptual level. Next, data warehouse size and query performance are considered. We investigate state-oriented data, which can be represented in two ways using regular star schemas and in one way using temporal star schemas, leading to a comparison of three data warehouses: a regular time-series warehouse, a regular events warehouse, and a temporal states warehouse. Even if all three warehouses attempt to represent the same information, their sizes as well as the execution speeds of queries on them will generally differ. In order to present concrete numbers of size and speed measures and to give a more real feeling of the differences, we will base the experiments on a case study.

We choose a data warehouse for a bank. We assume that there are 50 branches, 200 different customer types, and 5 account types. The warehouse does not keep information on single accounts and customers because its intended use is for decision support. Thus we have Branch, CustomerType, Time (only in the regular warehouses), and AccountType dimensions. We choose “week” as the time granularity. Each fact table row shows a total sum of all account balances on one fixed day of the week (i.e., Friday) for a certain type of account opened at a certain branch and owned by a certain type of customer.

In order to be able to load appropriate data into all three warehouses, additional assumptions are necessary. We assume that 60% of all aggregated balances change per week, i.e., the fact tables in both the events warehouse and the states warehouse have 40% less entries than the fact table in the time-series warehouse, because the latter one does not exclude duplicate values. For instance, if the total balance of all customers older than 60 and living in Aalborg remains the same for three weeks, in the time-series warehouse fact table, this will be represented by three rows, while in the events and states warehouses, there will be only one fact table row. We term the number 0.6 (which corresponds to 60%) the *actual change rate* (ACR).

We suppose that there are 7500 relevant combinations of AccountType, CustomerType, and Branch (5 account types · 150 customer types · 10 branches). One customer type is combined with only 10 branches of 50, because most likely those who live in, e.g., Aalborg never open accounts in the branches that are located in, e.g., Copenhagen. We assume that the AccountType and CustomerType dimensions do not change over time. The only slowly changing dimension is Branch.

The schemas of the two regular data warehouses are the same. In the temporal warehouse, the AccountType and CustomerType tables have the same schema as in the regular warehouses, but the Branch and the fact tables are different, and the Time table does not exist. The Branch tables in the regular warehouses contain a generalized key attribute, while in the temporal warehouse, the Branch table contains two timestamp attributes. The fact tables of the regular warehouses contain 4 foreign key attributes and a `Balance` attribute. The fact table of the temporal warehouse includes only 3 foreign keys (there is no Time table) in addition to two timestamp attributes and `Balance`.

We create an index on the fact table of each warehouse. For each of the regular warehouse fact tables, we create a primary index on the foreign key attributes, and for the temporal warehouse fact table we create an index on the foreign key attributes and `BeginDate`. For the temporal warehouse, we also experiment with primary index that additionally includes the `EndDate` attribute. SQL source code for creation of tables and indexes together with the size estimation of the attribute values of the tables can be found in reference [2]. The experiments were carried out using the Oracle DBMS (version 7.2.2.3).

5 Data Warehouse Size

We proceed to compare the sizes of the three warehouses. The data warehouse size is given by the sizes of the fact table, the dimension tables, and the indexes. Because dimension tables are generally very small [7] relative to the fact tables and have very little impact on the total size, we consider only the fact tables and indexes. The size of a table is determined by the number and size of the rows.[1]

The general formula for the computation of the size of a warehouse X is given next.

$$size(X) \quad = \quad rows(X) \cdot (FRL(X) + IEL(X)) \tag{1}$$

In the formula, X can be a temporal states warehouse (S), a regular events warehouse (E), or a regular time-series warehouse (TS). The function $rows(X)$ returns the number of rows in the fact table of warehouse X. If we assume that there are N rows in the time-series warehouse fact table, i.e., $rows(TS) = N$, then $rows(S) = rows(E) = \mathrm{ACR} \cdot N$.

[1] For simplicity, we use numbers of rows and sizes of rows instead of computing the exact numbers of data blocks used in Oracle. Using data blocks leads to the same conclusions.

In Section 4, we stated that the actual change rate in the banking case is 0.6, that there are 7500 relevant combinations of AccountType, Branch, and CustomerType, and that we record data once per week. For one year, there will thus be $7500 \cdot 52 = 390000$ rows in the time-series warehouse fact table and $0.6 \cdot 390000 = 234000$ rows in the fact tables of events and states warehouses. Thus the regular schema with the events representation and the temporal schema are better suited for representing changes that occur *irregularly* in the real world. In such cases, it is difficult or impossible to choose an appropriate sampling rate (time granularity) for the regular schema with the time-series representation.

The remaining functions in Formula 1 compute the fact table row length and the index entry length for a warehouse. Formulas for these functions are given below, and quantities used in the formulas are defined in Table 1.

$$\mathit{FRL}(X) = \mathrm{rh} + \mathrm{NonFKL}(X) + \mathrm{FKL}(X) + \mathrm{n}(X) \cdot 1\mathrm{byte}^2 \quad (2)$$

$$\mathit{IEL}(X) = \mathrm{eh} + \mathrm{rid} + \mathrm{NonFKLIndex}(X) + \mathrm{FKL}(X) + \mathrm{nIndex}(X) \cdot 1\mathrm{byte} \quad (3)$$

Notation	**Definition**
rh	row header size (3 bytes in Oracle)
$\mathrm{NonFKL}(X)$	the sum of the sizes of the non-foreign key attributes in the fact table of warehouse X
$\mathrm{FKL}(X)$	the sum of the sizes of the foreign key attributes in the fact table of warehouse X
$\mathrm{n}(X)$	the number of fact table attributes in warehouse X
eh	index entry header size (2 bytes in Oracle)
rid	row identifier size (6 bytes in Oracle)
$\mathrm{NonFKLIndex}(X)$	the sum of the sizes of the indexed non-foreign key attributes from the fact table of warehouse X
$\mathrm{nIndex}(X)$	the number of attributes in the index defined on a fact table of warehouse X

Table 1: Definition of Notations

The fact table row length (Formula 2) is the same in the regular events and time-series warehouses, but differs for the temporal warehouse. The difference is caused by (1) the presence of two timestamp attributes in the fact table of the temporal warehouse (S), (2) the presence of a generalized key attribute—which references the slowly changing Branch dimension—in the fact tables of the regular warehouses (TS and E), and (3) the presence of the foreign key reference to the Time table in the fact tables of the regular warehouses (TS and E).

The sizes of the fact tables, indexes, and the total sizes of three data warehouses for the banking case are given in Table 2.

Structure	*size*(TS)	*size*(E)	*size*(S)
Fact table	$390000 \cdot 21\mathrm{b} = 8\mathrm{Mb}$	$234000 \cdot 21\mathrm{b} = 4.8\mathrm{Mb}$	$234000 \cdot 33\mathrm{b} = 7.54\mathrm{Mb}$
Index	$390000 \cdot 20\mathrm{b} = 7.6\mathrm{Mb}$	$234000 \cdot 20\mathrm{b} = 4.57\mathrm{Mb}$	$234000 \cdot 24\mathrm{b} = 5.48\mathrm{Mb}$
Total size	15.6Mb	9.37Mb	13.02Mb

Table 2: Data Warehouse Sizes for One Year of Data in the Banking Case Study

Two characteristics define index size: the number and the length of index entries. The number of index entries is the same as the number of fact table rows. The index

[2] For each attribute its length value has to be stored. In Oracle for all attributes with average length less than 250 bytes, 1 byte is enough to store the length value itself.

entry length is affected by (1) the space usage of the indexed date-type attributes for the temporal warehouse, (2) the presence of foreign-key references to slowly changing dimensions, and (3) the presence of a foreign-key reference to the Time dimension table for the regular warehouses.

Based on Formula 1 and the numbers provided in Table 2, we can determine the sizes for the three warehouses as a function of the actual change rate, see Figure 4. It can be seen that the states warehouse size is smaller than the time-series warehouse size when the actual change rate is smaller than 0.72 (if the state warehouse index includes only one date-type attribute—the `BeginDate`).

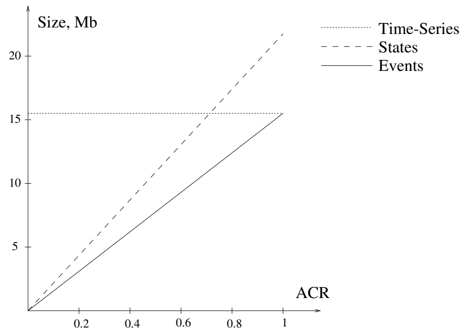

Figure 4: Data Warehouse Sizes as a Function of the Actual Change Rate

The inclusion of also the `EndDate` attribute in the index would increase the index size (consequently increasing the total size of the states warehouse about 14% and giving an actual change rate break-even point equal approximately to 0.62), but might also speed up queries. It is important to achieve an appropriate trade-off between space and speed; to do so, analyzing query performance is necessary.

6 Query Performance Experiments

In this section, we compare query performance when using each of the three data warehouses. We begin with general observations about the optimal query execution scenario. Then we give the types of queries used in the experiments, present the results, and draw conclusions.

6.1 General Observations

The special structure of a star schema, most notably the presence of one huge fact table and a number of relatively small dimension tables, greatly impacts the set of appropriate query execution strategies for data warehouse queries. Table size differences and distribution of values in tables have to be taken into account, and so cost-based optimization should be employed.

When dimensions are small, the optimal scenario for executing a query in a data warehouse begins with full scans of the dimension tables, identifying the dimension table rows that satisfy the constraints on dimension attributes given in the query. The next step is to perform the Cartesian product of the keys of the qualifying rows. These composite keys are then used for lookup in the fact table using its index, thus identifying the qualifying fact table rows.

The main idea is to access the fact table as late as possible, using its primary index as much as possible. The index usage may range from performing unique key lookup to not using the index at all. This depends on the ordering of the key attributes of dimensions in the composite primary key on which the fact table is indexed and on the set of dimensional constraints given in the concrete query. For example, it is not beneficial to use the index at all if there are no constraints on the dimension having its key first in the composite key of the fact table index.

To fairly compare query execution speeds for the three warehouses, the first step is to ensure that the queries are executed optimally. The execution times provided in Section 6.3 were thus measured only after extensive and careful tuning of each query.

6.2 Query Types and Queries

The goal of the query performance experiments was to find out what affects the query performance in the regular warehouses and in the temporal warehouse. To accomplish this, we define several broad query types and then investigate representative queries of these different types. Since the warehouses differ in how they handle (temporal) change, the types of queries are defined based on the types of temporal predicates they employ and what they retrieve. The chosen types of queries are given in Figure 5.

Query Type	*Type of Temporal Predicate*	*Retrieval of*
TYPE1	specified time point	non-temporal value
TYPE2	specified time period	non-temporal value
TYPE3	specified time duration	non-temporal value
TYPE4	specified nontemporal-attribute value	temporal value

Figure 5: Query types

We find it reasonable to assume that differences in how queries involve time (constraints on time, operations on time-attribute values) lead to differences in query performance in the three warehouses. We thus believe that the chosen types of queries allow us to cover a broad range of aspects of query performance and to identify potential differences among the three warehouses. We emphasize that, of course, not all queries of a given query type have the same performance.

Next, we give sample queries for each of the four query types (many more queries were studied; here, we give the most illustrative ones).

TYPE 1	
q.1.1:	*What is the balance—for customers living in City3, with zip code 9086, and whose age is between 25 and 40—of savings accounts in Branch9 on September 10, 1997?*
q.1.2:	*How much money was in all Branch10 savings accounts on September 10, 1997?*
q.1.3:	*Which customer type had the biggest amount in savings accounts in Branch26 on May 1, 1997?*
q.1.4:	*Compare the balances—for customers living in City3, with zip code 9086, and whose age is between 25 and 40—of savings accounts in Branch9 on September 10, 1997 and October 10, 1997.*

TYPE 2	
q.2.1:	*What were the balances—for customers living in City3, with zip code 9086, and whose age is between 25 and 40—of savings accounts in Branch9 from August 1, 1997 until November 1, 1997?*
q.2.2:	*Which customer types used two or more account types in Branch10 during 1997?*
q.2.3:	*What was the average April 1997 balance of savings accounts in Branch10 for customers younger than 18 that live in City5 and have zip code 5648?*

TYPE 3	
q.3.1:	*What customer types had the same balance in savings accounts in Branch2 for at least 12 weeks?*

TYPE 4	
q.4.1:	*When were the balances of savings accounts in Branch41 for customers younger than 18, living in City7, and with zip code 7700 bigger than 12000?*

6.3 Performance Results

We evaluated each of the queries given in the previous section a total of 10 times, measuring their actual execution times (see Figure 6). Execution times shown in the figure for queries on the temporal warehouse were measured using the index including both `BeginDate` and `EndDate` because this index yielded the best performance. The query execution times are normalized with respect to the longest execution time, which is given in seconds at the top of the appropriate column. Additional detail, including the actual tuned SQL queries and their execution plans, are provided elsewhere [2].

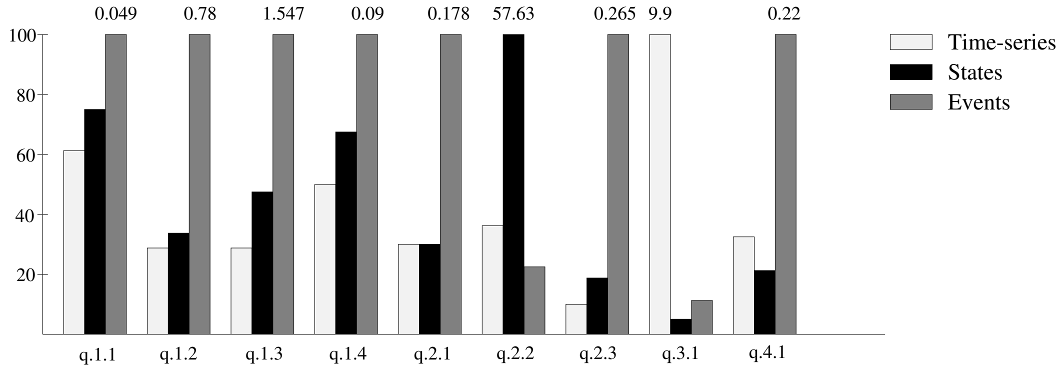

Figure 6: Query Execution Times in Percent of the Slowest Query

As we can see from Figure 6, queries on the temporal warehouse and the time-series warehouse are in most cases more efficient than queries on the events warehouse. The main reason is that in the events warehouse, it is difficult to find the time period when some state was valid. This is because one row in the fact table represents the beginning of the period; and in order to get an end of the period, which is located in another row, a subquery is needed. In addition to adversely affecting performance, subqueries reduce the readability and understandability of queries, especially if several subqueries are present and perhaps are nested.

In all queries (except Type 1) on the events warehouse, we also face the problems of finding the time period when the *current* balance state is valid. To accomplish this, either an additional subquery or a `UNION` with another `SELECT` statement is necessary to catch the "terminating" event. However, there are also factors that reduce query performance in the other two warehouses.

The time-series warehouse fact table contains duplicate values and is bigger than the fact tables in the other two warehouses. Therefore more fact table rows satisfy the given query constraints and more blocks are often read from disk. For example, this is the main reason for the poor performance of query q.3.1 in the time-series warehouse.

On the other hand, the fact table rows and index entries are bigger in the temporal warehouse; thus, more disk block reads are required for the same number of rows than in the regular warehouses.

Another difference between the presented warehouses is the absence of the Time dimension in the temporal warehouse, thus eliminating any joins with such a table. The price for not having a time table is to have two date attributes in the fact table, which can cause lower performance for some queries.

Many queries (e.g., of Type 1) on the time-series and events warehouses access only those fact table rows that affect the answer and do not access fact table rows to test time constraints. The same holds for the temporal warehouse with both dates in the index. But if only one of the dates is present in the index, these queries on the temporal warehouse are forced to access far more fact-table rows. This results in more disk accesses and slows down query execution.

We have experienced that date comparisons are expensive on their own in Oracle. This becomes clear when a lot of dates must be checked. This holds for query q.2.2 (the worst performance in the temporal warehouse) where the index cannot be used, meaning that all fact table rows are accessed; for the temporal warehouse, there is a need to check two timestamps of each row to determine whether they overlap with a given period.

Joins with slowly changing dimensions in the temporal warehouse are more complicated than those in the regular warehouses—additional checks of fact and dimension row timestamps are made to ensure that the fact row timestamps overlap with the dimension row timestamps. This is an issue particularly for queries of Type 2, 3, and 4. Temporal query language [13] would make it significantly easier to formulate temporal join queries.

The above-mentioned problems with temporal warehouses explain why query performance for the time series warehouse is usually slightly better than for the states warehouse.

The systematic handling of slowly changing dimensions in the temporal star schema leads to complex temporal joins, which are not attractive. But the handling of slowly changing dimensions in the regular star schema using generalized keys for the Branch dimension also complicates queries because values of these keys do not identify real Branch entities (there can exist several rows, with different generalized key values, for the same branch). If a query on a regular warehouse contains a subquery and there is a constraint on Branch in the main query, a join with Branch is needed in the subquery. This makes such queries difficult to write and hard to comprehend.

Another important point is that for all three warehouses, virtually every query requires manual tuning (using Oracle's hints) to achieve reasonable performance.

The general conclusion is that queries on the events warehouse always tend to become more complex and take more time to execute, while queries on the time-series and temporal warehouses stay simpler and run faster. The extended version of this paper [2] elaborates on the issues discussed in this section.

7 Conclusions and Future Research

Dimensional data modeling using star schemas is a typical approach when designing a relationally-based data warehouse. The regular star schema treats all dimensions, one of them being the Time dimension, equally and assumes them to be independent. A data warehouse may capture events or states; and events or states are represented as either time-series or events using the star schema. Star schemas have fundamental difficulties with the handling of slowly changing dimensions and change in state oriented-data: the time-series representation records potentially large amounts of redundant data, and the events representation has difficulties with query performance. Both representations fall short in capturing the actual times when dimensions change.

A possible solution to these problems, temporal star schemas systematically capture changes to fact and dimension tables alike. In temporal star schemas, time is not a separate, independent dimension table, but is a dimension of all tables and is represented via one or more time-valued attributes in all tables. Event-oriented data is represented as events (with one time attribute in the fact table), and state-oriented data as states (with two time attributes) in the temporal star schemas.

In the temporal star schema, real-world changes are handled in a natural and systematic way, allowing for the storage of the full and correct history of time-varying data. After investigating data warehouse size and query performance in three warehouses representing state-oriented data, we conclude that queries are easier to write and usually run faster in the temporal states warehouse than in the regular events warehouse; and the temporal states warehouse size is smaller than that of the regular time-series warehouse. The temporal states warehouse does not keep redundant information and is convenient for computing periods when some state was valid, which is usually needed in state-oriented queries.

Additional studies may shed further light on the properties of temporal star schemas. First, experiments can be carried out using real data and real queries. Also, the issue of efficient bulkloading may be studied. Updates to existing fact-table rows will be necessary when bulkloading the temporal warehouse, while, in the regular warehouses, it is only necessary to insert new rows. It would also be of interest to analyze temporal star schemas that include a time table. In this case, it would be enough to keep compact integer-type timestamps for each row, making comparisons more efficient. However, joins with the Time dimension table will be required.

Another research direction is to investigate advanced join techniques such as the STAR-join using the STARindex [10] or other techniques involving precomputed joins [14]. The performance gain resulting from the usage of precomputed joins would probably be more visible in the temporal warehouse than in the regular one, because temporal joins are more complex and expensive than regular joins.

Acknowledgements

This research was supported in part by the Danish Technical Research Council through grant 9700780 and by the CHOROCHRONOS project, funded by the European Commission DG XII Science, Research and Development, as a Networks Activity of the Training and Mobility of Researchers Programme, contract no. FMRX-CT96-0056.

References

[1] I. Ahn, R. T. Snodgrass. Partitioned storage for temporal databases. *Information Systems*, 13(4):369-391 (1988).

[2] R. Bliujute, S. Saltenis, G. Slivinskas, and C. S. Jensen. Systematic Change Management in Dimensional Data Warehousing (long version). URL: <`http://www.cs.auc.dk/~simas/tss.html`> (current as of 5 Feb 1998).

[3] W. K. Inmon. *Building The Data Warehouse*. John Wiley & Sons, 2nd ed. (1996).

[4] C. S. Jensen et al. A Consensus Test Suite of Temporal Database Queries. TR-R-93–2034. Dept. of Mathematics and Computer Science, Aalborg University (1993).

[5] C. S. Jensen et al., A Consensus Glossary of Temporal Database Concepts. *ACM SIGMOD Record*, 23(1):52-64 (1994).

[6] C. S. Jensen and R. Snodgrass. Semantics of Time-Varying Information. *Information Systems*, 21(4):311-352 (1996).

[7] R. Kimball. *The Data Warehouse Toolkit*. John Wiley & Sons (1996).

[8] T. Myrach, G. F. Knolmayer, R. Barnet. On Ensuring Keys and Referential Integrity in the Temporal Database Language TSQL2. *Proceedings of the Second International Baltic Workshop*, pp. 171–181 (1996).

[9] Leep Technologies, Inc. Supporting Temporal Data in a Warehouse. URL: <`http://www.iftime.com/wpaper2.htm`> (current as of 5 Feb 1998).

[10] Red Brick Systems, Inc. Star Schema Processing for Complex Queries. URL: <`http://www.redbrick.com/rbs-g/whitepapers/starschema.html`> (current as of 5 Feb 1998).

[11] R. T. Snodgrass, I. Ahn. A Taxonomy of Time in Databases. *Proceedings of ACM SIGMOD'85*, pp. 236–246 (1985).

[12] R. T. Snodgrass. Temporal Databases. *IEEE Computer*, 19(9):35-42 (1986).

[13] R. T. Snodgrass et al. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers (1995).

[14] P. Valduriez. Join Indices. *ACM TODS*, 12(2):218-246 (1997).