Temporal Dimensional Modeling

LARS RÖNNBÄCK*
OLLE REGARDT

the Department of Computer Science, Stockholm University
lars.ronnback@anchormodeling.com

Abstract

Dimensional modeling has for the last decades been one of the prevalent techniques for modeling data warehouses. As initially defined, a few constructs for recording changes were provided, but due to complexity and performance concerns many implementations still provide only the latest information. With increasing requirements to record changes, additional constructs have been suggested to manage change, mainly in the form of slowly changing dimension types. This paper will show that the currently existing types may lead to undesired anomalies, either at read or at write time, making them unsuitable for application in performance critical or near real-time data warehouses. Instead, based on current research in temporal database modeling, we introduce temporal dimensions *that make facts and dimensions temporally independent, and therefore suffer from none of said anomalies. In our research, we also discovered the* twine*, a new concept that may significantly improve performance when loading dimensions. Code samples, along with query results showing the positive impact of implementing temporal dimensions compared to slowly changing dimensions are also presented.*

DIMENSIONAL MODELING · MICROBATCH · DATA WAREHOUSE · SLOWLY CHANGING DIMENSION
HIGH PERFORMANCE · TWINE · TEMPORAL DIMENSION · REAL-TIME ANALYTICS

1. INTRODUCTION

For the last 25 years, two techniques have been dominating when it comes to data warehouse implementations. (Inmon 1992) with his *normalized model* and (Kimball 1996) with his *dimensional modeling*. Initially, few constructs or guidelines for managing data that changes over time were available, but as such requirements became more common, temporal features were retrofitted onto these technologies (Bliujute et al. 1998; Body et al. 2002). This paper focuses on slowly changing dimension types (Kimball 2008; Ross 2013), which were introduced to manage change in dimensional modeling, and how well these fit with current requirements for high performance. Along with the traditional types of dimensions, a simple and new type of dimension is introduced, called a *temporal dimension*. It is based on current research in temporality (Hultgren 2012; Rönnbäck et al. 2010; Golec, Mahnič, and Kovač 2017), and suffers from none of the refresh anomalies associated with slowly changing dimensions (R. J. Santos and Bernardino 2008; Slivinskas et al. 1998). For most types of slowly changing dimensions the fact table and its dimension tables become temporally dependent, leading to update anomalies and performance degradation (Araque 2003). This is not the case when using temporal dimensions, in which the tables are temporally independent, such that existing relationships are preserved regardless of all changes. The authors, we, recommend that this new type of dimension should become the de-facto standard for managing change in dimensional modeling and particularly when considering real-time analytics, a growing requirement within data warehousing (Russom, Stodder, and Halper 2014; Azvine, Cui, and Nauck 2005). While much research has been done into strategies for loading data at near real-time (Jörg and Dessloch 2009; Vassiliadis and Simitsis 2009), little has been done with respect to finding the most suitable modeling techniques.

The temporal dimension and many of the

*Thanks to Up to Change (`www.uptochange.com`) sponsoring research on ANCHOR and TRANSITIONAL MODELING.

existing types support all three query categories that may be relevant when retrieving information that changes over time (Golfarelli and Rizzi 2009; Ahmed, Zimányi, and Wrembel 2015):

TIY Today is Yesterday, in which all facts reference the current dimensional value.

TOY Today or Yesterday, in which all facts reference the dimensional value that was in effect when the fact was recorded.

YIT Yesterday is Today, in which all facts reference the dimensional value that was in effect at certain point in time.

The paper has a running example and is structured as follows. This section served as an introduction to the topics discussed, Section 2 presents slowly changing dimensions, their models, and issues stemming from how they are implemented, and Section 3 introduces our new concept of a temporal dimension. Following that is Section 4, that presents results from reading from and writing to the different types of dimensions, Section 5 contrasting our work with related research, and Section 6 that ends the paper with conclusions and suggestions for further research.

2. SLOWLY CHANGING DIMENSIONS

A running example will be used to illustrate the differences found in the models of the slowly changing dimensions of type 1–7, the "junk" type, and the temporal dimension. The example model, containing a simple fact and dimension table, can be seen in Figure 1. Each column is contained in a box, which is lined and shaded when that column is part of the primary key in the table, and white with dotted lines if not. Slanted text is used for columns that have nothing to do with the keeping of a history of changes and upright text for the columns necessary to keep such a history. Relationships between tables are shown along with their cardinalities.

To keep the model uncluttered and to the point, it is the simplest possible that still conveys the construction of the different types. The dimension table has a single value, *dimProperty*, along with a primary key, dim_ID. The fact table references this key from a column with the same name, along with one measure, *factMeasure*, and the date, *factDate*, for which this measure was recorded[1]. The primary key in the fact table is a combined key, consisting of all dimension table references (in our case one) and the date that the measures were recorded.

The numbering of the types are taken from (Kimball 1996) for types 1 to 3, whereas 4 to 7 and "junk" come from (Ross 2013) and (Wikipedia 2019), favoring those definitions that actually capture a history of changes. There is, in fact, also a slowly changing dimension of type 0, but in that type changes are simply discarded, resulting in information that is not up to date. As this is less than desirable, type 0 will not be discussed further. It is well known that change is prevalent and integral to almost all information, so with no further ado, these are the types of solutions retrofitted onto dimensional modeling, trying to solve the problem of managing change.

2.1. TYPE 1

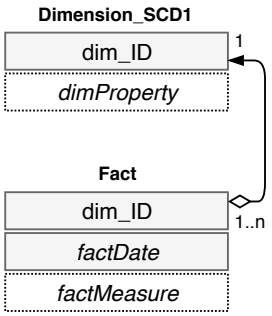


Figure 1: *Dimensional model, with a slowly changing dimension of type 1, also coinciding with type 0.*

In type 1, depicted in Figure 1, destructive updates of *dimProperty* are used to bring information up to date. The history is, of course, lost, and this type can only support TIY. This type is commonly used, despite its inability to keep previous versions, probably because the

[1]Normally dates are referenced from a fact table to a time dimension, but since that would only introduce additional and for all illustration and testing purposes equal complexity, it was simplified to a column directly in the fact table.

other types of slowly changing dimensions result in added complexity. However, as the requirement is to keep a history of the changes made, this type is discussed less, but will act as a point of reference when performing the experiments that measure performance of read and write operations.

2.2. TYPE II

In type 2, depicted in Figure 2, instead of updating *dimProperty* a new row is added with the new value. This row will have a different dim_ID, so in order to be able to correlate this change with the row containing the original value, some portion of the dimension must remain static over changes. This is represented by the dimStatic column. If all values can change, this may instead be introduced through a generated identifier. Furthermore, to know the order of changes and when they took effect, a dim_ValidFrom date is added.

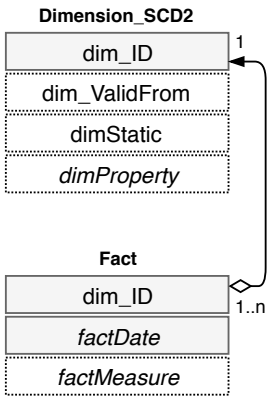


Figure 2: *Dimensional model, with a slowly changing dimension of type 2.*

Facts with *factDate* after dim_ValidFrom will reference the new value, and considering a model that regularly is receiving data, this will build a TOY connection between facts and dimension values. This can lead to confusion, particularly if aggregating measures over *dimProperty*, which now yields at least two different aggregates, given the circumstances. Bringing facts into a TIY or YIT view is possible through the combination of dimStatic and dim_ValidFrom, but the resulting query is complex and needs additional indexing in order to be performant.

If the use case of TIY is much more common, it is possible to update fact references when values change. This will instead take a heavy hit on performance at write time, since part of a primary key in the fact table is updated.

2.3. TYPE III

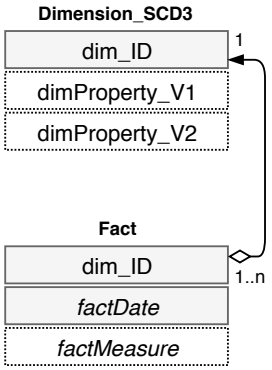


Figure 3: *Dimensional model, with a slowly changing dimension of type 3.*

In type 3, depicted in Figure 3, columns are added to manage change. The column *dimProperty* is split into dimProperty_V1 and dimProperty_V2, if only two versions are of interest. This limits the number of changes that can be kept track of, since regularly adding columns to the table will make it unwieldy. Any existing queries will also have to be continuously maintained in order to cope with the addition of new columns. Furthermore, if only a few of the rows in the dimension change, most rows will never have seen any changes and their columns then contain empty or dummy values for all but the first version, which is a waste of space.

2.4. TYPE IV

In type 4, depicted in Figure 4, a type 1 destructive update is used on the actual value in the dimension table producing a TIY connection. However, this is complemented by a second "history" table, to which the row with the value about to be updated first is copied. The primary key in the history table consists of the combination of the columns dim_ID and dim_ValidFrom. Historical rows thereby retain

their original dim_ID, making it easier to find them, but since the key now is composed it cannot be referenced from the fact table. The dashed reference line intends to show that a non-enforced relationship nevertheless exists.

The drawback is that a performance hit is taken at write time, since not only must a value be updated, but rows must also be copied between tables. As no referential integrity constraint exist between the fact and the history table, consistency must instead be guaranteed by other means. Queries for which YIT or TOY is of interest become less complex than in the case of type 2, but table elimination is no longer possible (Paulley 2000), due to the lacking referential integrity constraint. This may prove a problem in situations where many dimensions are slowly changing according to this type, since unneeded joins may be performed during execution of the query.

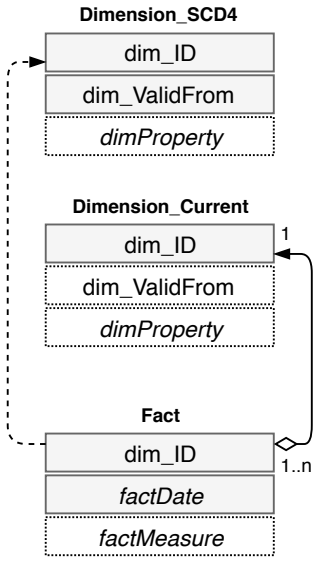


Figure 4: *Dimensional model, with a slowly changing dimension of type 4.*

2.5. TYPE V

Taking ideas from type 2, type 5 depicted in Figure 5, complements the dimension with the column dim_Current_ID, making it easy to find the current information. Where a single join from the fact to the dimension table yields a TOY connection, the TIY connection is only a second join away. Performance is hampered by the fact that searches need to be done over non-primary keys, which necessitates the need for secondary indexes. Furthermore, YIT still needs custom queries, and whenever a value changes updates are needed on existing rows, to repoint these towards the new latest row.

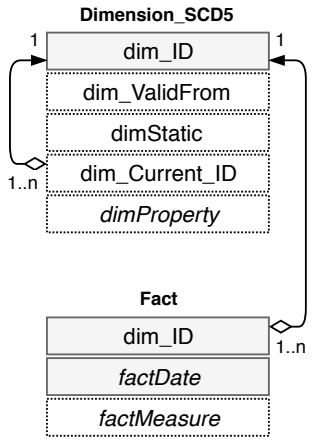


Figure 5: *Dimensional model, with a slowly changing dimension of type 5.*

2.6. TYPE VI

Trying to remedy some of the disadvantages of type 2, type 6 depicted in Figure 6 shares almost the same structure, but with a different primary key. The advantage of this type is that it is no longer needed for any part of the dimension to remain static, since the same dim_ID will remain across changes, thanks to dim_ValidFrom now being part of the primary key.

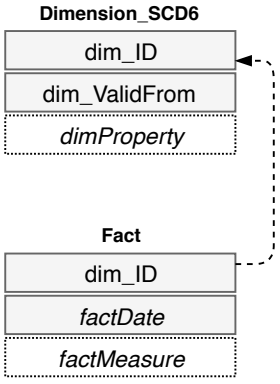


Figure 6: *Dimensional model, with a slowly changing dimension of type 6.*

The sacrifice being made is that referential integrity no longer can be maintained, as only half of that key is present in the fact table. While this gives a lot of flexibility, with no preference towards either of TIY, TOY, or YIT, problems arise if this model is the source for a cube, or any tool that requires referential integrity. A possible circumvention, although not mentioned in literature but probably existing in practice, would be to create regular views corresponding to TIY and TOY, which the fact table can reference using a logical key in the tool utilizing the model.

2.7. TYPE VII

Combining ideas from type 2 and type 5, type 7 depicted in Figure 7 also adds a column to keep track of the current information. Rather than to reference this from the dimension, as in type 5, the fact table is extended with a column. Compared to type 5 the current information is now a single join away instead of needing a double join.

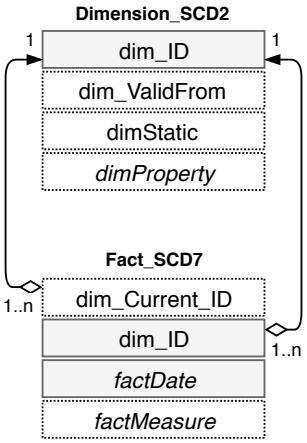


Figure 7: *Dimensional model, with a slowly changing dimension of type 7, which extends the fact table with a foreign key column referencing the current dimension information.*

Having the same issues as type 5 with additional indexing and requiring a static part, the reduced number of joins should, however, improve performance at read time. The tradeoff is that more rows need updating at write time, since now fact rows have to be updated instead of dimension rows, and they tend to be much more numerous.

2.8. JUNK AND MINI DIMENSION

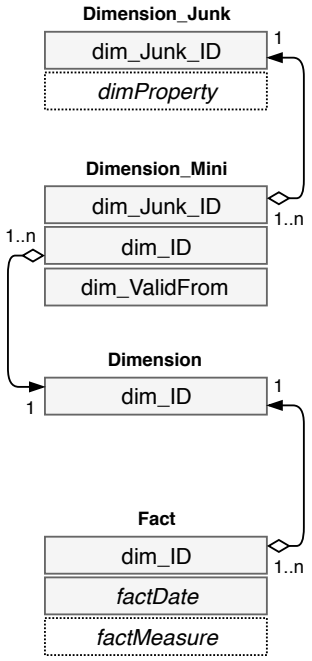


Figure 8: *Dimensional model, with a rapidly changing dimension, through mini and junk dimensions.*

A final approach depicted in Figure 8, not having been given a type, is still a construction that aims to manage change. The introduction of a "junk" and "mini" dimension is recommended when changes occur more rapidly than what is suitable for a slowly changing dimension. It is not clear exactly what rapidity necessitates this, but most likely when some part of the dimension changes considerably more often than other parts.

The parts that change rapidly, *dimProperty* in our case, are moved to a junk dimension, having its own primary key. Parts that are static or change slowly may remain in the original dimension, which is of a suitable slowly changing dimension type. All that remain in the original dimension in the example is dim_ID. A mini dimension is then introduced that references both the original dimension and the junk dimension, in which the primary key is extended with dim_ValidFrom.

This is actually the only model in which both referential integrity is maintained as well as being agnostic with respect to TIY, TOY, and

YIT. It suffers from now needing three joins in order to assemble the information, along with complex query logic. Furthermore, it relies upon the modeller making perfect assumptions about which columns will change rapidly, slowly or not at all. Of course, all columns could be moved to the junk dimension, but if they change at very different rates, a lot of information will be unnecessary duplicated.

3. TEMPORAL DIMENSION

Surprisingly, there is a model that is simpler in nature than many of the ones already presented that maintains referential integrity, is agnostic with respect to how the information should be temporally retrieved, and that has very respectable read and write performance. The key is the realization that temporal information benefits from having its immutable and mutable parts separated from each other and that minimal assumptions should be made concerning the future of the information being modeled.

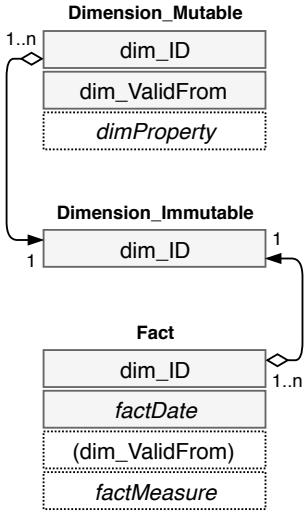


Figure 9: *Dimensional model, with the in this paper introduced temporal dimension, along with an optional "valid from" column in the fact table.*

The hereafter dubbed *temporal dimension* can be seen in Figure 9. It has two parts, a *mutable* table similar to a type 6 slowly changing dimension and an *immutable* table, similar to what would remain of the original dimension in the junk dimension scenario on the assumption that all columns may change. Should more properties be available in the dimension and those change at very different rates, each can reside in its own mutable table, still referencing the same immutable table.

While it may look strange to introduce an immutable table with a single column, in between the fact and dimension table, it ensures that referential integrity can be maintained by the database. Perhaps even stranger is the fact that the query optimizer can use these constraints to completely eliminate this table during the execution of retrieval queries, directly joining the fact and the mutable table. The immutable table will not be physically touched, and therefore behaves only like a logical modeling construct, not adversely affecting performance.

The optional dim_ValidFrom column in the fact table is used to retain which version is in effect with respect to the factDate. This will speed up TOY queries to some extent, at the cost of enlarging the fact table, which then slightly slows down TIY and YIT queries. Depending on the requirements this column may be omitted.

4. COMPARATIVE EXPERIMENT

Experiments have been made in which read and write performance have been tested for type 1–7 and junk compared with the presented temporal dimension. Read performance is further classified into the three query types necessary to retrieve information in TIY, TOY, and YIT. Write performance is measured separately for adding and updating rows in the dimension and fact tables.

4.1. CONFIGURATION

The tests were carried out on a server equipped with an Intel i7 4770 4-core CPU, 32 GB of RAM, an Intel 750 PCIe NVMe SSD as data disk, and four RAID-striped Samsung 840 Pro SATA SSD assembled as a temp disk. The server was running a 64-bit Microsoft Windows® operating system and the chosen database engine was Microsoft SQL Server® 2017 Developer Edition.

The initial dimension data consists of 2^{20}, 1 048 576 , unique values for dim_ID, for which one value changes 2^{20} times, two values change 2^{19} times, four values change 2^{18} times, ..., 2^{18} values change two times, and 2^{19} values change one time, thus representing both slowly and rapidly changing values. This resulted in 21 495 808 dimension rows in total. The times these changes were made, captured by dim_ValidFrom, will range from the end of 2017 back to January 2008, with the most rapidly changing every five minutes. The initial fact data consists of 10 rows added every minute over the same time range, but captured by *factDate*, with each row referencing a randomly selected dim_ID. This random distribution is squarely skewed towards the values in the dimension that change more often. This resulted in 52 427 836 fact rows in total.

These values were chosen in order to achieve query execution times in the range of seconds to minutes, which was considered long enough to not be disturbed by other factors and short enough to make the experiment feasible. They also represent the concept of microbatches, achieving near real-time updated data. For testing the write performance a larger batch had to be constructed to achieve consistent measures during the experiment. The method of loading is through regular SQL statements[2]. For writing to the dimension, all *dimProperty* values related to the 1 048 576 unique dim_IDs receive an update in the beginning of 2018, and equally many new values are added. One fact row per dim_ID is also added, totalling 2 097 152 new rows. In order to ensure that the tests are as equal as possible with respect to the data, the temporal dimension model was populated first, and the others populated from it.

Figure 10 begins with a code sample showing the creation of the tables for the temporal dimension. Both the fact and mutable dimension table is partitioned by respectively the year of *factDate* and the year of dim_ValidFrom, to liken them with actual implementations. Tables for type 1–7 and junk are created in a similar manner, following the models presented earlier. Partitioning can be and is used for all types where dim_ValidFrom is part of the primary key. All code for reproducing the tests is available online (Rönnbäck 2018) or by request to the corresponding author.

```
create table Dimension_Immutable (
    dim_ID int not null,
    primary key (dim_ID asc)
);

create table Dimension_Mutable (
    dim_ID int not null,
    dim_ValidFrom smalldatetime not null,
    dimProperty char(42) not null,
    foreign key (dim_ID) references
          Dimension_Immutable (dim_ID),
    primary key (dim_ID asc, dim_ValidFrom desc)
) on Yearly(dim_ValidFrom);

create table Fact (
    dim_ID int not null,
    factDate smalldatetime not null,
    factMeasure smallmoney not null,
    foreign key (dim_ID) references
          Dimension_Immutable (dim_ID),
    primary key (dim_ID asc, factDate asc)
) on Yearly(factDate);

create function pitDimension (
    @timepoint smalldatetime
)
returns table as return
select
    dm_in_effect.dim_ID,
    dm_in_effect.dim_ValidFrom,
    dm_in_effect.dimProperty
from (
    select
        *,
        row_number() over (partition by dim_ID order
              by dim_ValidFrom desc) as
              ReversedVersion
    from
        Dimension_Mutable
    where
        dim_ValidFrom <= @timepoint
) dm_in_effect
where
    dm_in_effect.ReversedVersion = 1;
```

Figure 10: *Code sample showing how to create the in this paper introduced temporal dimension, followed by the point in time parametrized view over the temporal dimension, through which the rows in effect at the given point in time can be fetched.*

[2]This is sometimes referred to as ELT, as in Extract, Load, then Transform. While Microsoft SQL Server comes with its own streaming data ETL tool, Integration Services, it only supports type 2 out of the box.

4.2. PREPARATION

To further liken the experimental setup with a real world scenario, parametrized views were created that encapsulate common operations. Such views are inlined by the database engine, "search and replaced" in the query, before it is passed to the optimizer, and should therefore have no performance impact compared to having typed the entire statement to begin with.

The parametrized view in the bottom half of Figure 10 is created using a windowed function that helps finding the information in effect. This is just one of many thinkable ways to find the information in effect at a given point in time and several methods were evaluated with respect to performance. For each type of slowly changing dimension, the best performing method of those described by (Hutmacher 2016) was picked for the experiment. For models in which the dim_Static column is present, this instead meant using a cross apply combined with an ordered top 1 operation.

While the point in time parametrized views provide an efficient way to find information in effect at a specific point in time, as for TIY and YIT, they proved to be a poor choice for TOY queries. Trying to cross apply such a view for every *factDate* yielded a query execution estimate in which billions of operations would take place. An attempt to run such a query was halted after four hours, during which numerous drawings saw the dawn of light on a whiteboard. Among these was one in which two timelines were drawn close together, from which the idea of a *twine* came.

The parametrized view in Figure 11 shows how two timelines are intertwined in order to create a lookup table between the fact and dimension keys for the information in effect. After twining the timelines, a conditional cumulative max-operation is used with a window to find the latest value from one of the timelines for each value in the other timeline over each *dim_ID*. Using a twine, the "never-ending" query now took seconds to run. As it is a simple trick, it is bound to have been discovered by practitioners, but to the best knowledge of the authors it is neither scientifically documented nor practically discussed in conjunction with slowly changing dimensions. As such, it should be of significant interest because of its tremendous performance boost. The authors have already had uses for twines in several situations outside of the dimensional modeling domain.

```
create function twineFact (
    @fromTimepoint smalldatetime,
    @toTimepoint smalldatetime
)
returns table as return
select
    in_effect.dim_ID,
    in_effect.factDate,
    in_effect.dim_ValidFrom
from (
    select
        twine.dim_ID,
        twine.Timepoint as factDate,
        twine.Timeline,
        max(case when Timeline = 'D' then Timepoint
            end) over (partition by dim_ID order by
            Timepoint) as dim_ValidFrom
    from (
        select
            dim_ID,
            factDate as Timepoint,
            'F' as Timeline
        from
            dbo.Fact
        where
            factDate between @fromTimepoint and
                @toTimepoint
        union all
        select
            dim_ID,
            dim_ValidFrom as Timepoint,
            'D' as Timeline
        from
            dbo.Dimension_Mutable
        where
            dim_ValidFrom <= @toTimepoint
    ) twine
) in_effect
where
    in_effect.Timeline = 'F';
```

Figure 11: *Code sample showing how to create the twine parametrized view, intertwining the timelines of the fact and dimension tables between the given time points by using a conditional cumulative max-operation, windowed to find the value in effect at each point of the timeline and for each dim_ID. It yields a significant performance gain for* TOY *type queries, compared to other known methods.*

4.3. APPROACH

With these views in place, the actual test was set up. For each type of query, reading TIY, YIT, TOY and writing to the dimension and fact tables, a loop per dimension type was set up. This loop was run four times when measuring read times, with the first iteration being a "dry run", after which all column statistics is updated. This is because the optimizer will automatically create statistics on the first run where it deems it suitable, slightly slowing down the query and thereby disturbing the test results. Furthermore, as this is done during execution, the statistics is only based on a sample in order to not affect the performance too adversely. The explicit update of the statistics is instead based on full table scans. When measuring write times, three runs without a "dry run" was used.

The loop for queries that read data is structured as follows. The first part clears all caches possible from within SQL Server, the second sets a starting time with nanosecond precision, the actual query is silently run with its results redirected to a temporary table, after which an ending time is set. Similarly, the loop for queries that write data is structured as follows. The first part clears all caches, a transaction is opened, a dimension starting time is set, the query writing data to the dimension table is run, an ending time is set, a fact starting time is set, the query writing data to the fact table is run, and ending time is set, after which the queries are rolled back. The rollback is necessary in order for each run to be equal and its time is not taken into account.

For both reading and writing, the difference in milliseconds between the starting and the ending time is stored for each iteration in a separate table. From these the median value is selected. A margin of error was also calculated and monitored. Should it have been large, indicating that something disturbed the testing, the tests would have been run again, but this never happened. Most of the tests were nevertheless run many times, between which refinements were made. Values for *dimProperty* and *factMeasure* were chosen in a way that verifications could be made to ensure that the results produced from the different queries were identical.

```
select
    in_effect.dimProperty,
    f.numberOfFacts,
    f.avgMeasure
into
    #result_tiy
from (
    select
        dim_Id,
        count(*) as numberOfFacts,
        avg(factMeasure) as avgMeasure
    from
        Fact
    where
        factDate between '20140101' and '20141231'
    group by
        dim_ID
) f
join
    pitDimension('20180101') in_effect
on
    in_effect.dim_ID = f.dim_ID;

select
    dm.dimProperty,
    count(*) as numberOfFacts,
    avg(f.factMeasure) as avgMeasure
into
    #result_toy
from
    twineFact('20140101', '20141231') in_effect
join
    Fact f
on
    f.dim_ID = in_effect.dim_ID
and
    f.factDate = in_effect.factDate
join
    Dimension_Mutable dm
on
    dm.dim_ID = in_effect.dim_ID
and
    dm.dim_ValidFrom = in_effect.dim_ValidFrom
group by
    dm.dimProperty;
```

Figure 12: *Code sample showing the performance test query for* TIY *and* YIT*, joining the point in time view for the temporal dimension (but with "20140101" passed to the view for* YIT*), followed by the test query for* TOY*, using a twine to find the version in effect.*

An example query, for TIY, is shown in Figure 12 using the point in time view from the temporal dimension. As seen, the query calculates the number of fact rows and the average of *factMeasure* per *dimProperty*. For the par-

ticular query, this resulted in 1 014 754 rows, which is slightly less than the unique number of *dim_ID*s, as the randomization did not exhaust all possible references between facts and dimension values. For some types, in which fact rows are already joined to the current version, TIY can be found through a direct join of the fact and dimension table. The same calculation is also used for YIT and TOY and for all types of dimensions, which for YIT is as simple as passing a different date to the point in time view, but for TOY may involve a twine, depending on the type.

4.4. RESULTS

To summarize the results, it clearly comes across that the different types of dimensions are geared towards specific use cases. These preferences fall into four categories; fast retrieval of the current version, the previous version, the historically correct version, or being temporally agnostic. In the first three, fact rows are temporally bound to a particular version in the dimension, whereas for the agnostic preference, fact rows and dimension rows are temporally independent. Temporal independence has faster write performance at the cost of slower read performance.

4.5. GENERAL PERFORMANCE

As can be seen in Table 1 there is no clear winner if taking both reading and writing into account. Favoring the current version are types 1, 3, 4, 5, and 7, favoring the previous is only type 3, and favoring the historically correct are types 2, 5, and 7. Favoring no particular version are temporal, optional, type 6, and junk. If only current values are of interest, type 1 is the best choice, and if only very few versions suffice, type 3 may be an option. If ad-hoc querying over versions is necessary, requirements have to be weighed against performance considerations. Whilst those types favoring a particular version perform particularly well when retrieving that version, they suffer at write time because of additional logic to include the required temporal bond.

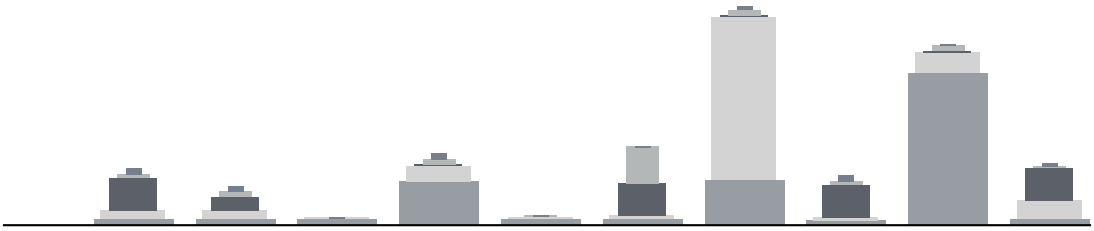
If, for example, in a read intensive environment where historically correct versions are necessary in most use-cases and only rarely the version in effect at an arbitrary point in time, then type 2 is the best choice, given that write performance can be sacrificed. Loading the type 2 dimension table is 50% slower than for temporal and six times slower than the update done by type 1. Loading the fact table is seven times slower than for temporal and type 1, due to a twine being necessary, but cannot be done as efficiently since finding the version in effect relies on dimStatic.

If both historically correct and current versions are common in use-cases, but still rarely the version in effect at an arbitrary point in time, then type 5 or type 7 could be a choice, but only after carefully considering the additional sacrifice in write performance. Writing to the dimension table of type 5 is seventeen times slower than temporal and writing to the fact table of type 7 is twenty-four times slower than temporal, due to the updates that both finds the version in effect and then sets it for each row. Maintaining a type 5 or type 7 dimension will over time become an insurmountable task.

If both current and the version in effect at an arbitrary point in time are common in use-cases, then our temporal and option, as well as type 6 and junk are good candidates. While junk has very similar read performance to temporal, the time is takes to load its dimension tables is doubled in comparison. For type 6, showing the best write performance of all types that retain a complete history of changes, the difference seen is explained by it having sacrificed referential integrity. However, as referential integrity must be guaranteed at some point, type 6 makes this "somebody else's problem", where it nevertheless is bound to hog some resources.

4.6. SUITABILITY FOR NEAR REAL-TIME

To determine the suitability of each type of dimension in a microbatched near real-time data warehouse, a hypothetical data warehouse system is envisioned. This hypothetical system can either perform reading or writing, with one blocking the other. Simplistic, but not entirely unlike a real system. Assuming such a system is approaching real-time, there are four possible scenarios; write often —



	Temporal	Optional	Type 1	Type 2	Type 3	Type 4	Type 5	Type 6	Type 7	Junk
TIY	10.4	10.2	0.8	9.1	0.7	0.7	3.6	9.9	2.8	6.6
YIT	6.4	6.7	—	9.1	0.7	60.2	9.1	6.3	9.1	4.1
TOY	53.5	23.7	—	3.2	—	53.7	3.0	53.2	2.9	53.8
Dim	14.2	14.0	3.3	23.9	4.2	7.0	271.2	7.2	31.8	30.1
Fact	10.8	10.9	10.8	74.2	10.9	10.9	74.9	7.2	254.2	10.8

Table 1: *Median execution times (in seconds) for queries reading (TIY, YIT, TOY) and writing (Dim, Fact) for the introduced temporal (and optional) dimension and each type of slowly changing dimension. The "skyline" presents the total execution time for all queries and type of query.*

read often, write often — read seldom, write seldom — read often, and write seldom — read seldom. Looking at the last of these, it is easy to be misled into thinking that such a scenario is never coinciding with the requirements of real-time decision support. This is *not* the case as the determining characteristic of real-time is latency (Bateni et al. 2011). Even in a write seldom — read seldom scenario, the rare read may take place arbitrarily close to the rare write and it may be of utmost importance that the data returned is as fresh as possible.

In order to reduce latency in the hypothetical system, the write needs to block as few reads as possible and likewise for the other way around. Type 5 and 7 are very imbalanced in this respect, also having the two poorest write performances, and cannot be recommended for near real-time requirements. The imbalance is also quite large for type 2, ranking in the bottom three for write performance together with types 5 and 7. Type 4 has poor read performance and types 1 and 3 may be disqualified for not maintaining a complete history of changes, leaving temporal, optional, type 6, and junk as the candidates.

We deliberately chose to ignore a potential issue when setting up the experiment. When rows are temporally bound, this depends on the assumption that data has been arriving synchronously over time, such that all existing fact rows always reference the correct version. In practice, this does not always happen, and retrospective modifications have to be made, adversely affecting the total maintenance time for types 1, 2, 3, 4, 5, and 7. As there is no way to estimate the impact, depending of how large the error is, this may range from a quick fix to lengthy outages, unacceptable in a near real-time environment. This again leaves temporal, optional, type 6, and junk as the candidates.

4.7. SIMPLICITY

Looking at the different models, junk is the most complex. Type 4 adds complexity in the form of a separate history table, type 2, 5, and 7 rely on some other column being static. Type 3 could lead to unwieldy tables due to the large number of columns. Type 6 is the second best with respect to model simplicity, following type 1, which naturally has the simplest model. The temporal dimension is very similar to type 6, adding an immutable table between the fact and dimension table, still keeping it on the simpler side. It also make no assumptions, allowing for every part of the dimension to change. Some lineage must exist if never before seen data is about to be loaded though, making it possible to determine if it represents a change or an addition.

Querying becomes slightly more complex for the temporal dimension, than in the temporally dependent types, but is helped by

"hiding" the logic behind the point in time and twine parametrized views. Every type needed a point in time view for YIT, except type 3, but under the presumption that keeping a single old version is enough. The types 4, 6, junk, and temporal all need the twine for TOY. The "valid from" column in optional adds complexity to the benefit of not having to twine. This again, though, assumes that once that time point is set it is correct and will never have to be changed. Given that reading times for TOY were only halved, this may not be advantageous enough to warrant its introduction.

5. RELATED RESEARCH

A lot of research has been done concerning performance of non-temporal dimensional models, many of which use Star Schema Benchmark, a modification of TPC-H (Rabl et al. 2013; O'Neil et al. 2009). More recent research tend to use the newer TPC-DS benchmark model (Nambiar and Poess 2006), but only one of its dimensions has temporal characteristics in the form of type 5. Given the state of established benchmarks, having forgotten about temporal requirements, the decision was made to produce our own test data.

A brief comparison of a subset of the types of slowly changing dimensions has been done by (V. Santos and Belo 2011), where they using only logical reasoning come to the conclusion that type 4 is superior to all other types. Given our findings, that seems like a poor conclusion. No previous work contrasting all types of dimensions could be found.

(Johnston 2014) discusses the need for new thinking when it comes to slowly changing dimensions and suggests a bitemporal dimension, albeit not readily implementable in current database technologies. His dimension is type 2-like as it relies on a "durable identifier" apart from the unique surrogate primary key, and therefore cannot be said to be completely temporally independent.

Big Data platforms, like Hadoop, rely on immutability of existing data. (Akinapelli, Shetye, and Sangeeta 2017) points out that slowly changing dimensions are particularly hard to manage in this respect. Thanks to the temporal decoupling between facts and dimensions when using our temporal dimension, databases can be insert only, which may open up for easier implementations on such platforms.

6. CONCLUSIONS AND FURTHER RESEARCH

Most, if not all, analytical tools have support for slowly changing dimensions, but to varying degrees with respect to different types. To name a few are: Tableau, MicroStrategy, Alteryx, SAP, Cognos, SAS, Analysis Services, and Power BI. Many database vendors also have guides for how to implement them, like Microsoft, Oracle, Vertica, Hive, SAP HANA, PostgreSQL, and IBM DB2 to name a few. ETL tools have special adapters for loading data into slowly changing dimensions, like Informatica, Microsoft Integration Services, Oracle Data Integrator, Pentaho, IBM DataStage, Apache Kafka, and others. This, if nothing else, should indicate that slowly changing dimensions are a hot topic, but as it stands, with little scientific grounding. We aimed to put them on better footing by challenging traditional types with a type of our own. There may still be situations where the traditional types may be of preference, but the introduced temporal dimension fares so well that we suggest it be the default choice when a history of changes is required. This suggestions is further accentuated when a data warehouse approaches near real-time updates. To summarize its advantages:

- It is temporally independent from the fact table, such that it may be refreshed asynchronously and favors no particular version in time.
- It has a simple model that extends upon already familiar types and it does not rely upon any part of the dimension being static.
- It is free from assumptions about the future and it may even hold future versions that will come into effect at some later time.
- It is modular and parts that change with different rates can be broken apart while keeping the same immutable connection to the fact.

- It maintains referential integrity in the database and when changes occur only inserts are used to capture these, never updates.
- Its performance is close to or better than all other types with similar capabilities, when both reading and writing is weighed in.

Along the way, we also discovered the twine; a new strategy for finding the version in effect for each fact by the times these facts were recorded. The twine is essentially a union of timelines that is sorted with an additional column containing a conditionally repeated value. It performs much better than any approach trying to produce a temporal join through some inequality condition. Twining may help many existing implementations gain better performance, as it can be used in any scenario where a temporal join is required.

There are many venues for further research. It would be interesting to utilize temporal tables, added in the ANSI:SQL 2011 standard (Kulkarni and Michels 2012), to see if the results would be the same. We used SQL to drive the loading of the dimensions, so an investigation into how streaming data ETL tools compare is warranted. Running the same data set and queries in other database engines could give insights into possible internal optimizations that may have affected our experiment. Extending the idea of our temporal dimension to become bitemporal should be straightforward as it is already temporally decoupled from the fact table. Finding the threshold for when changes are rapid enough compared to slower changing parts that breaking the mutable dimension apart would yield even better performance is also of interest. Skewing the data differently in our experiment may also produce different results. Combining the idea of temporal dimensions with slowly changing measures (Goller and Berger 2013) is also highly relevant.

References

Ahmed, Waqas, Esteban Zimányi, and Robert Wrembel (2015). "Temporal DataWarehouses: Logical Models and Querying." In: *EDA*, pp. 33–48.

Akinapelli, Sandeep, Ravi Shetye, and T Sangeeta (2017). "Herding the elephants: Workload-level optimization strategies for Hadoop." In: *EDBT*, pp. 699–710.

Araque, Francisco (2003). "Real-time Data Warehousing with Temporal Requirements." In: *CAiSE Workshops*, pp. 293–304.

Azvine, Behnam, Zheng Cui, and D Di Nauck (2005). "Towards real-time business intelligence". In: *BT Technology Journal* 23.3, pp. 214–225.

Bateni, MohammadHossein et al. (2011). "Scheduling to minimize staleness and stretch in real-time data warehouses". In: *Theory of Computing Systems* 49.4, pp. 757–780.

Bliujute, Rasa et al. (1998). *Systematic change management in dimensional data warehousing*. Tech. rep. Citeseer.

Body, Mathurin et al. (2002). "A multidimensional and multiversion structure for OLAP applications". In: *Proceedings of the 5th ACM international workshop on Data Warehousing and OLAP*. ACM, pp. 1–6.

Golec, Darko, Viljan Mahnič, and Tatjana Kovač (2017). "Relational model of temporal data based on 6th normal form". In: *Tehnički vjesnik* 24.5, pp. 1479–1489.

Golfarelli, Matteo and Stefano Rizzi (2009). "A survey on temporal data warehousing". In: *International Journal of Data Warehousing and Mining (IJDWM)* 5.1, pp. 1–17.

Goller, Mathias and Stefan Berger (2013). "Slowly changing measures". In: *Proceedings of the sixteenth international workshop on Data warehousing and OLAP*. ACM, pp. 47–54.

Hultgren, Hans (2012). *Modeling the agile data warehouse with data vault*. New Hamilton.

Hutmacher, Daniel (2016). *Last row per group*. URL: `https://sqlsunday.com/2016/04/11/last-row-per-group/` (visited on 07/11/2018).

Inmon, William H (1992). "Building the data warehouse". In:

Johnston, Tom (2014). *Bitemporal data: theory and practice*. Newnes.

Jörg, Thomas and Stefan Dessloch (2009). "Near real-time data warehousing using state-of-the-art ETL tools". In: *International*

Workshop on Business Intelligence for the Real-Time Enterprise. Springer, pp. 100–117.

Kimball, Ralph (1996). *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses*. New York, NY, USA: John Wiley & Sons, Inc. ISBN: 0-471-15337-0.

— (2008). "Slowly changing dimensions". In: *Information Management* 18.9, p. 29.

Kulkarni, Krishna and Jan-Eike Michels (2012). "Temporal features in SQL: 2011". In: *ACM Sigmod Record* 41.3, pp. 34–43.

Nambiar, Raghunath Othayoth and Meikel Poess (2006). "The making of TPC-DS". In: *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment, pp. 1049–1058.

O'Neil, Patrick et al. (2009). "The star schema benchmark and augmented fact table indexing". In: *Technology Conference on Performance Evaluation and Benchmarking*. Springer, pp. 237–252.

Paulley, Glenn Norman (2000). "Exploiting functional dependence in query optimization". In:

Rabl, Tilmann et al. (2013). "Variations of the star schema benchmark to test the effects of data skew on query performance". In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. ACM, pp. 361–372.

Rönnbäck, Lars (2018). *Temporal Dimensional Modeling Performance Suite*. URL: `https://github.com/Roenbaeck/tempodim/` (visited on 02/01/2019).

Rönnbäck, Lars et al. (2010). "Anchor modeling—Agile information modeling in evolving data environments". In: *Data & Knowledge Engineering* 69.12, pp. 1229–1253.

Ross, Margy (2013). "Design Tip# 152 Slowly Changing Dimension Types 0, 4, 5, 6 and 7". In: *Kimball Group*.

Russom, Philip, David Stodder, and Fern Halper (2014). "Real-time data, BI, and analytics". In: *Accelerating Business to Leverage Customer Relations, Competitiveness, and Insights. TDWI best practices report, fourth quarter*, pp. 5–25.

Santos, Ricardo Jorge and Jorge Bernardino (2008). "Real-time data warehouse loading methodology". In: *Proceedings of the 2008 international symposium on Database engineering & applications*. ACM, pp. 49–58.

Santos, Vasco and Orlando Belo (2011). "No need to type slowly changing dimensions". In: *IADIS International Conference Information Systems*, pp. 11–13.

Slivinskas, G et al. (1998). "Systematic Change Management in Dimensional Data Warehousing". In:

Vassiliadis, Panos and Alkis Simitsis (2009). "Near real time ETL". In: *New Trends in Data Warehousing and Data Analysis*. Springer, pp. 1–31.

Wikipedia (2019). *Slowly Changing Dimension*. URL: `https://en.wikipedia.org/wiki/Slowly_changing_dimension` (visited on 02/01/2019).