Aalborg University

Master Thesis

Temporal database support

Author:
Vujadin Vidović

Supervisor:
Assoc. Prof.
Kristian Torp

A thesis submitted in fulfilment of the requirements for the degree of Master of IT, Software Development

in the

Database and Programming Technologies
Department of computer science

June 2016

“The most important property of a program is whether it accomplishes the intention of its user."

C. A. R. Hoare

AALBORG UNIVERSITY

Abstract

Cassiopeia - House of Computer Science

Department of computer science

Master of IT, Software Development

Temporal database support

by Vujadin VIDOVIĆ

Support for temporal features has been added to the latest SQL Standard SQL:2011. Many DBMS vendors like IBM, Oracle, TeraData and now Microsoft provide some kind of temporal support. The goals of this thesis is to explore temporal concepts in general, examine temporal support as provided by the SQL Server 2016, and to find out if it is possible to work around any limitations that may be encountered. This SQL Server has been chosen because of practical reasons. To achieve these goals a relational model has been made, which is used as basis for the evaluation of the temporal support. Prototypes have been made for some of the missing temporal features that were discovered. The conclusion is that even if temporal support in the SQL Server is very limited, when considering the theoretical possibilities, it is still be very useful for some use cases.

Acknowledgements

First of all, I would like to thank Clienti A/S, company I currently work at for providing me with the opportunity to complete my master studies. They have been very flexible and understanding. Thanks also to my previous employer WebHouse ApS, that has made it possible for me to start on master studies.

I would also like to thank my advisor Assoc. Prof. Kristian Torp, whose lectures introduced me to the temporal concepts. Thanks for the patient guidance, encouragement and advice he has provided throughout my work on this thesis.

Finally, a very special thank to my son Stefan Vidović, who has been very kind to set time aside to discuss some of the mathematical concepts, and for proof-reading this thesis.

Thanks to all of you!

Contents

List of Figures

Code Listing

Chapter 1

Introduction

Database management systems (DBMS), that support time varying aspect of the data are said to be temporal. But, what does this entail, how well is it defined, how is it supported and what can be done to work around any possible deficiencies? Those are examples of some of the questions that needs to be addressed before any new technology is adopted.

Temporal data and DBMS support for this kind of data, is a research topic that has been heavily studied for at least last 30 years and continues to attract the attention of researchers even today. Many books and papers have been published on this subject. A publishing overview, outdated but comprehensive, is given in [1]. Most of the researchers have been proposing to extend the relational model. The earliest proposals, like HQuel, TQuel, DM/T, TempSQL and IXSQL are mentioned in [2]. Some controversies regarding those different models have surfaced [3]. Some proposals have even been heavily criticised. The best example of this is probably given by an article [4] written by Darwen and Date criticising the TSQL2 approach [5].

It seems that different camps have been formed in the research community that still exists today. Two of these are predominant in the research community. Both of these camps agree however on one thing, and that is that the SQL standard needs an extension. The biggest difference of opinion seem to be about the need for extending or modifying the relational model itself to achieve temporal support. Maybe that is one of the main reasons that major database vendors have been very slow to adopt temporal concepts in their products.

However, some vendors have been adding temporal support to their products. The most prominent representatives of these are Oracle, starting with flashback queries [6], DB2 [7] and TeraData [8]. They did it in spite of unavailable SQL standardisation. With the acceptance of the SQL:2011 standard, which includes temporal support [3], we can hope that more and more vendors will do the same. One of the vendors that has taken this path and joined other vendors is Microsoft with their SQL Server 2016 [9] product.

The most recently published overview, as of this writing, of the DBMS and their support for temporal data is given in [10]. Although not comprehensive it does give fairly good overall picture.

1.1 Motivation

In the 1980's researchers recognised the need to address temporal or time-varying aspects of the data [1]. Hardware was becoming cheaper, disk space larger and as a result more data could be saved and subsequently archived for longer periods. The tendency was clear. Archived data, once available, could then be analysed. New technologies such as data mining and data warehousing emerged in the 1990's and big data in 2000's.

There are many possible use cases for the temporal data. Some of them could be formulated as:

- Versioning of the data: When working on an article, the content management system is required to provide the possibility for the user to go back to previous versions of the article.
- Auditing: Medical journal systems are required to provide history on "who" did "what" and "when" for auditing purposes.
- Scheduling/planning: Scheduling software is required to disallow double bookings.
- Statistics: Company needs software to measure success of their online sales in regards to campaigns they are running.

More examples are listed in [7], with an interesting case that suggests that data had to be retained for longer periods because of the legal requirements. Moreover, it seems also that customers are expecting more and more from their temporal database vendors as the managed data sizes evolve and become larger. Those customers also have very different temporal needs, and therefore some interesting querying patterns have been observed and studied and possible solutions have been presented [11].

The mentioned use cases suggest that there may be many different scenarios and requirements. Difficulties possibly arise when working with temporal data but the need for temporal support is evident. Therefore if an organisation or company wants to adopt technologies supporting temporal data, an assessment is needed regarding temporal concepts in general and their concrete support in DBMS. Some vendors even suggest [7] that there are many work hours to be saved when using DBMS with temporal support versus coding an in-house custom solution in the application layer.

1.1.1 What is the problem?

In this section we are going to look into some examples, which illustrate the need for addressing temporal issues. Database tables are loosely described, but the focus is mainly kept on the temporal aspects, as to not worry too much about overall model, constraints, types, etc. Likewise, sample values shown do not necessarily represent values of the underlying types. They are mainly used for an easier interpretation of the sample values.

1.1.2 *Example 1*: Timestamps for created and updated events

We start of this example by looking at Figure 1.1). There are five attributes, and most of them should be self-explanatory. The primary key for this table is `PNO`. This is also indicated in the table header. The attribute `FEE` represents the membership fee for each player. For easier interpretation of the sample data by the reader, values for the column `FEE` are displayed with the currency symbol prepended. The column `CREATED` shows when a row has been recorded in the database and the column `UPDATED` when a row was last updated. The time part has been omitted from the output. Let us assume that values for those two attributes are set by the system, i.e. not by the end user.

PNO	NAME	CITY	FEE	CREATED	UPDATED
P1	Michael	Copenhagen	$100	2010-01-01	2012-01-01
P2	Simon	Berlin	$110	2011-04-01	2011-04-01
P3	Allan	Paris	$90	2012-01-01	2012-09-01

Figure 1.1: `PLAYER` with sample data

Looking into the sample data we can observe that it contains two date attributes. The date values for the attribute `CREATED` is created when the row is inserted and is never updated again, but the values for `UPDATED` can and will change over time. Let us now look at the same table after the `FEE` is updated for player `P1` by reducing it down to $80.

PNO	NAME	CITY	FEE	CREATED	UPDATED
P1	Michael	Copenhagen	**$80**	2010-01-01	**2016-01-01**
P2	Simon	Berlin	$110	2011-04-01	2011-04-01
P3	Allan	Paris	$90	2012-01-01	2012-09-01

Figure 1.2: `P1` row after update in `PLAYER` table

We notice two things in this example. First, the value for `UPDATED` has been updated to the current date value which is set to 1. January 2016. This becomes the valid or effective date for the new fee. Secondly, by updating `FEE` we have lost information about what fee `P1` was paying previously. Answering queries like "How many times the fee changed for the player `P1` during last year?", and "What was the total income from membership fees last year?" may be required, but neither of these queries can be answered correctly. Because data changes in this way some historical information will inevitably be lost, and because of that we will need to address this problem.

1.1.3 *Example 2*: Event time-stamping

For this example let us now look at the table `PLAYER_FEE_SINCE`, shown in Figure 1.3. This table has only three attributes. Those are `PNO`, `FEE` and `SINCE`. The attribute `SINCE` represents a year from which the fee has been valid and is updatable by the user. The primary key for this table consists of the attributes `PNO` and `SINCE`, since we can not have a player paying the fee more than once for the same year. We want again to change the fee for `P1` by reducing it to $80. Just like we did in the first example. Let us now look at what a possible scenario is.

PNO	FEE	SINCE
P1	$100	2012
P2	$110	2011
P3	$90	2012

(a) before

PNO	FEE	SINCE
P1	$100	2012
P1	**$80**	**2016**
P2	$110	2011
P3	$90	2012

(b) after

Figure 1.3: `PLAYER_FEE_SINCE` table before and after the fee is changed

This example illustrates a possible solution to the query problems we identified in 1.1.2. We have replaced the update statement with an insert, and redefined the primary key. This change seems to enable us to somehow construct the history around player's membership fees. We can say that player `P1`'s fee has been $100 from the year 2012 until the year 2016, and that fee is set down to $80 as of 2016, and will be current until this fact is changed. Let us now, for the moment, imagine that this player is no longer a member and has no fee, but we want to keep a record of him, i.e. we do not want to completely delete him from the table. How do we do this? It turns out that this is not possible by using the current model. Therefore there is a need to introduce two attributes to the model that will represent a *period*. A period with a designated start and end time.

1.1.4 *Example 3*: Need for two different time dimensions

In the examples we have presented so far, each had different semantics regarding the attributes `UPDATED` and `SINCE`. The first one, based on `UPDATED`, indicates transaction or log time of a change, and is not updatable by the end user. The second one, based on the `SINCE` attribute, is updatable and has been used to represent years for which some player fee has been valid. Sometimes it is necessary to support both of these semantically different time dimensions inside the same table.

PNO	CONTACT	FEE	SINCE	UPDATED
P1	JO	$100	2012	2011-12-01
P1	**MA**	**$80**	**2016**	**2015-12-01**
P1	**JO**	$80	2016	**2016-05-01**
P2	RI	$110	2011	2010-12-01

Figure 1.4: Changes across two different time dimensions.

The fee for player `P1` has been recorded on 1. December 2011 for the year 2012, and has remained the same up until year 2016, when it was reduced to $80. This was recorded on 1. December 2015. Furthermore, the contact person for this player has been changed from JO to MA. However, on may 1. the player has been reassigned to his previous contact person. Validity time for a player fee is implied by the attribute `SINCE`, contact information however is pertinent to specific valid time and is implied by the attribute `UPDATED`. To be able to record shown data we had to remove the `SINCE` attribute from the primary key. Once again, we will need a different model if we are to manage this kind of data.

1.1.5 *Example 4*: Need for periods

Let us now look into an example that uses two fully temporal and related tables. Table `PLAYER_LICENSE` stores licensed players. Licenses are obtained from an organisation and are valid for some period. Players can only join clubs if they are licensed. Second table `SQUAD` is used for storing player-club relations, which means that the player is under contract in that club for the specified period. The player must be licensed for the duration of the contract period. Tables are shown in figure 1.5. Attributes used for primary keys are underlined in the table headers. Foreign key references (player and club) are not shown but are assumed to exist.

PNO	FROM	TO
P1	2008	2012
P1	2014	2016
P2	2007	2015
P3	2010	2013

(a) `PLAYER_LICENSE`. Players with valid license periods.

PNO	CNO	FROM	TO
P1	C1	2009	2012
P1	C1	2014	2016
P2	C1	2007	2010
P3	C1	2011	2013

(b) `SQUAD`. Players with club contracts.

Figure 1.5: Licensed players with club contracts.

Let us now consider a few cases that would be relevant when working with tables like these two.

1. Should it be possible to insert `('P1', 2006, 2010)` into `PLAYER_LICENSE`?
2. Should it be possible to delete `('P1', 2008, 2012)` from `PLAYER_LICENSE`?
3. Should it be possible to insert `('P1', 'C2', 2010, 2011)` into `SQUAD`?

The short answer is no to all of these questions, and to prevent that from happening we will need to define and use temporal primary and foreign keys, and constraints. Generally, we want to prevent overlapping of period values in a specific table. However, for case (2) we want to ensure that periods in one table overlaps the period for relevant rows in another table. As we do not want a case where a player is under contract but not licensed.

1.2 Scope and problem statement

Necessity to address problems when dealing with time varying data was motivated by customer requirements in my current company. Furthermore, announced temporal support in the up-coming SQL Server 2016 [9], DBMS which our customers uses, has been the main reason to choose this platform for investigating temporal challenges and possible solutions. No other considerations or criteria have been used.

Let us now reiterate some of the points mentioned in the introduction so far. They are:

- Many temporal concepts are investigated and well described in the literature.
- Some temporal support is provided by leading database vendors.
- SQL:2011 standard includes section on temporal support.
- Customers are requesting support for temporal data. And tendency is rising.

Taking all this into account, it seems it would be worthwhile and generally useful, to try and implement full temporal support for an existing non-temporal database for which temporal requirements exist. Some of the questions that we want to address are: *"What are SQL Server limitations regarding temporal support if any?"*, *"Is it possible to work around those limitations?"* and in the end *"Is it worth the trouble?"*. Our relational starting point is described in Chapter 3.

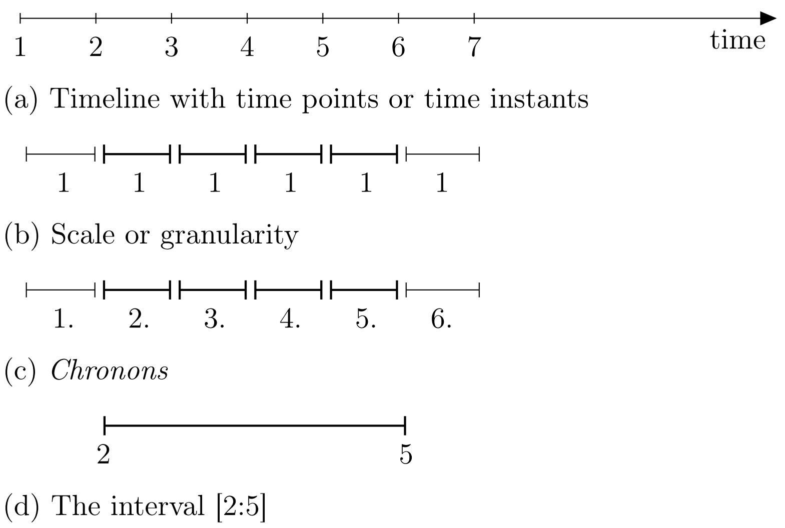
The hope is to be able to provide insight and give answers to those questions from an adopter's perspective. Temporal challenges have been dealt with in the application space for the most part so far. Now, the question is *"Was it preference or necessity?"*.

Chapter 2

Temporal concepts

2.1 Time point, scale

The concept of time in a temporal database world is somewhat different then our understanding of the time in the real world. In the real world we think of an instant of time or a time point as of something that does not have duration. For example if someone has a meeting at 10:00 in the morning we intuitively think of that point in time as one with no duration. In the temporal database world time is viewed as a finite set of discrete instants or time points in some time domain. Each time instant has a duration. The duration may be a year, a day, an hour, a microsecond, etc.



(a) Timeline with time points or time instants

(b) Scale or granularity

(c) *Chronons*

(d) The interval [2:5]

Figure 2.1: Illustration of different time concepts[1].

The time domain can be represented by integers, as in the Figure 2.1, or by the rational numbers with a defined *scale* [12]. For example, we could choose a numeric type `NUMERIC(8, 6)` to represent time points, let us say, minutes with a *scale* that supports representation of microseconds. Therefore, the *scale* of a real number, in this context, is also a *duration* of an instant and is also called a *granularity* or a *chronon* [13] in the temporal database literature.

2.2 Intervals

An interval in a temporal context is defined as the time between two events and represented as a set of contiguous *chronons* [13]. Another definition is that interval is an ordered set of discrete and contiguous time points, with a specified start and end. Data type representing an interval, on the other hand, also need to satisfy certain properties [12]. Those properties are: 1. Two unary operators `BEGIN` and `END` exist, that take an interval as a parameter and return the first and the last point respectively. 2. $\texttt{BEGIN}(i) \leq \texttt{END}(i)$. And 3. Defined binary operator $\in$ such that $p \in i$, if and only if $\texttt{BEGIN}(i) \leq p \leq \texttt{END}(i)$.

The duration of an interval can be defined as the number of time points between the start and endpoint. In Figure 2.1 we have marked an interval `[2:5]`, which includes the points `{2, 3, 4, 5}` and we can observe that the cardinality of this set is equal to the interval duration as represented on the timeline for the specified *scale*.

Intervals are unfortunately called periods in SQL:2011 [3] because the keyword interval has been taken. We will use the term *interval* when we are talking about the general concept outside of the specific SQL context, and *period* otherwise. Interval duration is also called *span* and *time distance* [13].

As we have already seen, an interval is denoted in notational terms by squared brackets surrounding the respective start and endpoint. If the start point of an interval is equal to the end point of an closed-closed interval, we say that interval duration is 1. In general all intervals with *unit* duration are called *unit intervals* [12].

[1]Slightly modified version of the relationship between time domain, *chronons* and interval as presented in [2].

There are four types of intervals. Their definitions regarding some point p with start in point s and end in point e are:

$$\begin{aligned}
[s:e] &= \{p:\ s \leq p \leq e\} \quad \text{and} \quad s \leq e \quad \text{(closed)}\\
[s:e) &= \{p:\ s \leq p < e\} \quad \text{and} \quad s < e \quad \text{(closed-opened)}\\
(s:e] &= \{p:\ s < p \leq e\} \quad \text{and} \quad s < e \quad \text{(open-closed)}\\
(s:e) &= \{p:\ s < p < e\} \quad \text{and} \quad s < e \quad \text{(open)}
\end{aligned} \tag{2.1}$$

2.3 Time concepts

2.3.1 User-defined time

In the temporal database literature [13] there are three main distinctions regarding different concepts of time. Those are transaction time, valid time and user-defined time. User-defined time or en event time is a concept that is known and already used in the database world. It could be used to represent a persons birthday, hiring date, or a date of a past or an upcoming event. As there is already support for this kind of time it will not be discussed further. Therefore, in what follows, we are going to focus on transaction and valid time only.

2.3.2 Transaction time

Transaction time is the time when a *fact* is stored in a database. It refers therefore only to the past. Tables supporting transaction time are also called history tables, as they preserve historical record of any data modification that might have been done. We can say that besides supporting access to the current state of the data transaction time tables also support access to the previous states or versions of the data. It is not possible to modify transaction time. No alternations to the history are possible. Transaction time is also called system time.

2.3.3 Valid time

Valid time represents a time when we believe that some *fact* is true. It is often used to represent the past but it can also be used to represent present and future time. As our beliefs can change so can valid time. Let us look at an example. We record in our database that some historical event took place in the period from the year 1956 to the year 1960. At some later point we learn that this was not true and that in fact that event took place from the year 1955 to 1960. Therefore it is necessary to correct it and make changes in the database. Another example could be that our vacation plans change and as a consequence we need to change the planned period to some other. Valid time is also called application or business time.

2.3.4 Bitemporal tables

We have seen in the previous two sections that transaction and valid time represent two semantically different time dimensions, orthogonal to each other. One represents the time of state changes inside the database and another represents the time when some *fact* is held to be true in the modelled world.

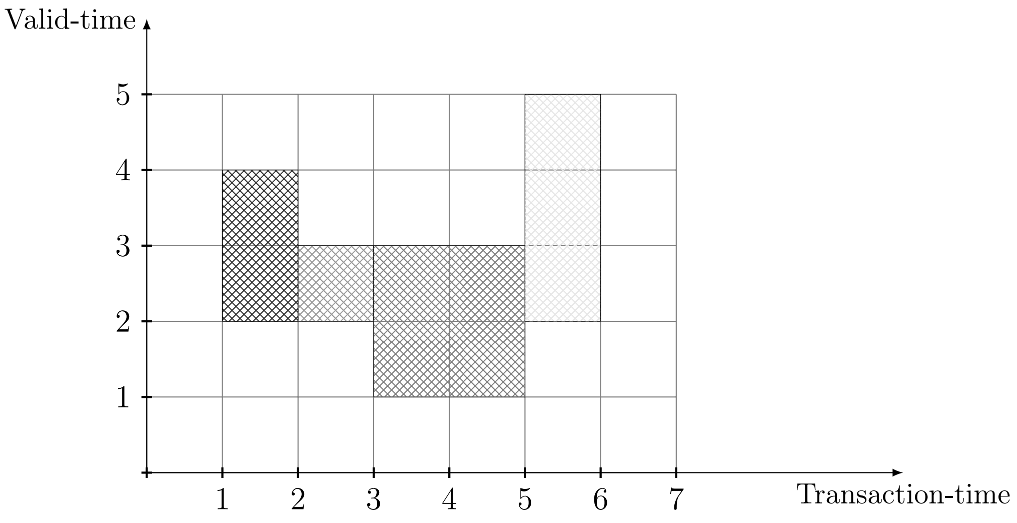


Figure 2.2: Validity of some *fact* according to transaction time

Tables that support both of these time dimensions are called *bitemporal*. Figure 2.2 represents valid time variation of a *fact* against the transactional time. Let us say that the current time is `7`, then we can observe that at transaction time `1`, `TT` for short, the system has recorded that valid time, `VT` for short, has the interval `[2:3]`. Furthermore, at `TT = [3:4]` we can observe that `VT = [1:2]`, and that at `TT = [6:6]` the valid time is not specified, indicating that valid time information for a *fact* has been deleted at time `6`. Different shades of gray in Figure 2.2 represent different values that are valid during valid time intervals.

Another way to represent bitemporal data is of course in tabular form as shown in Figure 2.3. The sample data shows four different fees for specified valid time and transaction time for when the fee was changed.

PNO	FEE	VT_FROM	VT_TO	SYS_FROM	SYS_TO
P1	$110	2	4	1	2
P1	$70	2	3	2	3
P1	$90	1	3	3	5
P1	$20	2	5	5	6

Figure 2.3: `PLAYER_FEE` with valid and transaction times.

Please note that there is no primary key constraints defined. We will discuss them later in this chapter. Each row shown in Figure 2.3 represents or states a *fact* about player fee for valid time. Therefor there must not be any overlapping periods. Overlapping and some other temporal predicates are also discussed later on.

2.4 Concepts of *now* and *until changed*

Now is a temporal variable managed by a DBMS and indicates that a fact is valid until present time. That means that, for example, for the interval [3:*now*] variable *now* is evaluated and becomes known when user asks for it. If current time is 10 then the interval is evaluated to [3:10]. That implies that variable *now* is bound to the current system time, and therefore changes as time progresses. In some temporal database literature [12] it is also called "*the moving point now*".

The temporal variable *UC*, which means *until changed* is a marker used for transactional time and indicates which row represents the current state of a *fact*. When a current state is changed then a *end* or *to* point is set to the current time and a new row is marked with *UC*. If we look at Figure 2.3 we can see that there are no current rows as all of them have been "closed". This "closing" time is indicated by the `SYS_TO` column.

Taking the same example from Figure 2.3 we can now add an extra row `(P1, $10, 5, now, 6, UC)`. This row now states that the fee for player `P1` is $10 from time 5 to present time and that it is recorded at time 6 and that this fact is current until changed. This change is shown in Figure 2.4.

The *UC* marker is usually represented by a maximal value of the underlaying data type. For the data type representing years it may be 9999, for dates it may be the value of 9999-12-31 and so forth. For examples used here it is taken to be 99.

However, the concepts *UC* and *now* have some semantical distortions [14]. For example, taking previously mentioned sample, if the current time is 10 and we are to ask "Which players will be charged $10 fee at time 11?", we will not get the correct answer. That

PNO	FEE	VT_FROM	VT_TO	SYS_FROM	SYS_TO
P1	$110	2	4	1	2
P1	$70	2	3	2	3
P1	$90	1	3	3	5
P1	$20	2	5	5	6
P1	$10	5	now	6	UC

Figure 2.4: `PLAYER_FEE` with a valid and transaction times.

is because the variable *now* will be evaluated to 10. If we are, at the same time, to ask question for how long player $P1$ has been charged a $10 fee we will again get the wrong answer. The duration of the period [6 : 99] does not represent actual duration in regard to the question we are asking.

2.5 Interval operators

In the temporal database world there are variety of interval operators that can be used to form temporal predicates. Some of the operators that will be discussed here are taken from Date et al. [12]. They have been largely based on Allen's operators [15], but there are a few name changes and additions. Some of them are shown in Figure 2.5. A more complete overview is given in Appendix A.

The intervals i and j are assumed to be *closed*. In fact, all intervals presented in this thesis are *closed* unless otherwise noted. Also those that can be inferred from the sample data. With $interval.s$ we denote start point of the interval, and with $interval.e$ the end.

The operators *equals*, *overlaps*, *meets* and *merges* are commutative. The operator *meets* is combination of Allan's operators *meets* and *met by*. Likewise, operator *overlaps* is a combination of *overlaps* and *overlapped by*. The operator *merges* is or'ed combination of *overlaps*, *overlapped by*, *meets* and *met by*.

The predicate for testing if two intervals *meet* (or are adjacent) includes the variable *scale*, as can be observed in Figure 2.5. In our examples so far the *scale* (or granularity) has been 1. It is worth noting that if intervals i and j have been defined as half-open, then the predicate could be formulated without a *scale*. It would simply be: $(i.e = j.s) \vee (i.s = j.e)$.

Intervals i and j	Operator	Condition
$\vdash i \dashv$ $\vdash j \dashv$	i `EQUALS` j	$i.s = j.s \wedge i.e = j.e$
$\vdash\!\!\!\longrightarrow i \longrightarrow\!\!\!\dashv$ $\quad\vdash j \dashv$	i `INCLUDES` j	$i.s \leq j.s \wedge i.e \geq j.e$
$\vdash i \dashv$ $\qquad\qquad\vdash j \dashv$	i `BEFORE` j	$i.e < j.s$
$\vdash i \dashv$ $\quad\vdash j \dashv$	i `OVERLAPS` j	$i.s \leq j.e \wedge i.e \geq j.s$
$\vdash i \dashv\vdash j \dashv$	i `MEETS` j	$(i.e + scale = j.s) \vee$ $(j.e + scale = i.s)$
$\vdash i \dashv\vdash j \dashv$ $\vdash i \dashv$ $\quad\vdash j \dashv$	i `MERGES` j	i `OVERLAPS` $j \vee i$ `MEETS` j
$\vdash i \dashv$ $\vdash\!\!\!\longrightarrow j \longrightarrow\!\!\!\dashv$	i `BEGINS` j	$i.s = j.s \wedge i.e \in j$

Figure 2.5: Interval operators

In Figure 2.6 we show some other scales/granularities as well. They are defined over sample data types. It should be noticed that, for type `DATE`, scale or granularity is shown as one day and not as an integer 1. That is because of the underlying data type. For type `SECOND` it is assumed to be an integer. On the other hand, for `MILISECOND` it is type `NUMERIC(3,3)` and for `MICROSECOND` it is type `NUMERIC(6,6)`.

`Type`	`Scale/Granularity`
`DATE`	`1 day`
`SECOND`	`1`
`MILISECOND`	`0,001`
`MICROSECOND`	`0,000001`

Figure 2.6: Scales/granularities for different types

2.6 Coalescing

The coalescing operation is somewhat similar to the duplicate elimination. When performing a duplicate elimination we consider if two rows are equivalent. Two rows are equvalent if all their attribute values are. In temporal tables two rows are *value-equivalent* if values of their non-timpestamp attributes, also called explicit attributes, are equivalent.

Coalescing is an unary operation. It takes temporal relation as input, and produces *normalized* or *packed* relation regarding interval values of it's timestamp attribute. The resulting relation is union compatible to the input relation [16].

PNO	FEE	VT_FROM	VT_TO
P1	$110	2	4
P1	$110	5	7

(a) Value-equivalent and adjacent rows

PNO	FEE	VT_FROM	VT_TO
P1	$110	2	7

(b) After coalescing

Figure 2.7: Coalescing two value-equivalent rows that have adjacent valid-times.

Temporal relation is coalesced when there are no *value-equivalent* tuples such that their timestamp intervals are adjacent (meet) or overlap. Performing coalescing operations on a relation which is already coalesced has no effect [17]. Figure 2.7 shows an example of a coalescing operation. If the fee for player `P1` is `$110` for time points $I1 = \{2, 3, 4\}$ and the same for time points $I2 = \{5, 6, 7\}$, then the set of all time points for which the fee is valid is $J = I1 \cup I2$. The resulting interval $[2 : 7]$ is then obtained by taking the smallest interval point for the start and largest for the end, i.e. interval $[MIN(J) : MAX(J)]$. The result can be observed in table b), from Figure 2.7. This operation, as already suggested, can only be applied to the rows that are *value-equivalent* and their intervals *meet* or *overlap*. If we were to make a union of two disjoint interval sets then the result will contain gaps and gaps are not allowed in intervals by definition.

Let us now consider what would happen if the second row from our running example has the following values: `(P1, $100, 3, 7)`. The first and second row now overlap. And coalescing on the timestamp attribute only, will give the same interval, namely interval $[2 : 7]$. However, because the rows are not *value-equivalent* we are not allowed to perform coalescing. Now, the question arises, should such a row be allowed into the table to begin with. We will look into this and some related issues in Section 2.7.

2.7 Key constraints

Date et al. in their book [12] identify three potential problems that need to be addressed when working with temporal data. Problems are identified as problem of: *contradiction*, *redundancy* and *circumlocution*. Let us now consider each of these by looking into examples representing those problems.

PNO	FEE	VT_FROM	VT_TO
P1	$110	2	4
P1	$100	3	6

Figure 2.8: Contradiction problem.

Example in Figure 2.8 shows two facts about player `P1`. The first one states that the fee is `$110` for time points 2, 3 and 4. Second one states that the fee is set to `$100` for time points 3, 4, 5 and 6. Those two statements are clearly in contradiction regarding points 3 and 4. The first one states that the fee for those points is `$110` and second one that the fee is `$100`. If we define a database table as a set of true propositions or facts, then those two statements will violate this definition.

The second example regarding *redundancy*, is shown in Figure 2.9. This example illustrates that the same fact is stated twice, namely that the fee for player `P1` is `$110` for time point 4. This problem could be solved by *coalescing* those two rows into one, stating that the fee is `$110` for points `[2:5]`.

PNO	FEE	VT_FROM	VT_TO
P1	$110	2	4
P1	$110	4	6

Figure 2.9: Redundancy in temporal data.

Last example, Figure 2.10 under (a), shows the problem of *circumlocution*. That is, using two statements or rows, to state something that could be stated using a single statement. This is a somewhat similar problem to the previous one, but with adjacent intervals with no time points in common. Two rows in this example could also be *coalesced*, but not always. It depends on the predicate of the table. If we consider a similar, valid time, table with presidents and their terms in office then we can not perform coalescing or packing, because in doing so we will lose information on their exact periods in office. This is illustrated in the same figure under (b).

The necessity for using some kind of temporal keys should be obvious from those examples. Furthermore, temporal foreign keys would also be necessary in some cases. Figure

PNO	FEE	VT_FROM	VT_TO
P1	$110	2	4
P1	$110	5	6

(a) can be coalesced

PRESIDENT	VT_FROM	VT_TO
Reagan	1981	1984
Reagan	1985	1988

(b) can not be coalesced

Figure 2.10: Example of two possible circumlocution cases.

2.11 shows two tables, first representing players and second player-club relationship. Let us now suppose that before a player can join a club for some period, the player needs to be "valid" in the player table for that very same period. To enforce this kind of requirement DBMS has to support foreign temporal keys. As an example, if player `P1` has joined a club `C1` during time points `[2:4]` it should not be possible to delete or modify the valid time period for this player in such a way that would violate the requirement.

PNO	VT_FROM	VT_TO
P1	2	4
P2	3	6

(a) player

PNO	CNO	VT_FROM	VT_TO
P1	C1	2	4
P2	C1	4	5

(b) squad

Figure 2.11: Dependant valid times in two different tables.

2.8 Querying in general

In conventional database systems querying is performed on the current *state* of the database. Temporal querying is the ability to query a specific *state* of the database regarding one or more time dimensions. If we take a look at the example shown in Figure 2.4, then we can perform queries on both valid and transaction time. A sample query regarding the valid time could be formulated as: "Get a player's fee for valid time period as of time point 3". Or, another one, "Get average fee for players during the valid time period [2:4]". Queries regarding transaction time could be formulated in a similar way, but they will always refer to present or past time. Combining both valid and transaction time periods we could query across two different time dimensions. Effectively asking about our modelled "beliefs" regarding valid times that are held and recorded at some specific transaction time. Taking our current example we could formulate this kind of query as: "What have we believed that the fee was for player `P1`, at transaction time 3 for the period [2:4]?".

2.9 Join and aggregate functions

Temporal join is a type of join between two tables, both containing compatible timestamp attributes.

Joins, in general, can be represented by the following statement [2]

$$R.A \quad \theta \quad Q.B \tag{2.2}$$

They are called theta joins where R and Q are relations and A and B are attributes with compatible data types. The theta operator θ is based on the following predicates/-operators: $=, \neq, <, >, \leq$ *or* $\geq$. The temporal join involves attributes of temporal data types, such as time intervals. Temporal joins can therefore be defined as joins that are based on interval operators. Operators like *before*, *overlap* and *meet*, which have been discussed in 2.5.

To illustrate temporal join we will reuse the example from Figure 2.4. We want to select all player pairs that were or will be paying members at some overlapping point in time. Listing in 2.1 assumes the existence of the type generator `INTERVAL` and an operator `OVERLAPS` in the DBMS.

```
SELECT P1.PNO, P2.PNO FROM
PLAYER_FEE P1, PLAYER_FEE P2
WHERE P1.PNO < P2.PNO AND
INTERVAL(P1.VT_FROM, P1.VT_TO)
 OVERLAPS INTERVAL(P2.VT_FROM, P2.VT_TO)
```

Listing 2.1: Temporal join

This temporal join can be represented in the more general form as

$$\Pi_{A_1, A_2, \ldots A_n}(\sigma_{C \wedge R.interval\ \mathtt{INTERVALOP}\ Q.interval}(R \times Q)) \tag{2.3}$$

Where `C` is a non temporal predicate and `INTERVALOP` is an interval operator. If we denote temporal predicate with `T` we can formulate temporal join even shorter.

$$R \bowtie_C^T Q \tag{2.4}$$

Aggregate functions, in general, could be divided into selective (`MIN`, `MAX`) and cumulative (`AVG`, `SUM`, `COUNT`) functions. In our example from Figure 2.4, regarding player fees, aggregate function could be used to compute `SUM` of the fees for specific time points. Or, to find the largest fee for each time point using the `MAX` function across all players.

Temporal aggregation can be based on six different *temporal ranges* [11]. Those temporal ranges are: *point in time*, *instantaneous*, three different *windowing* functions and *user defined* function. Computing the sum of the fees per time point would be an example of an aggregate function over*point in time* range. Kaufmann [11], in his thesis, suggests that the most common use case for temporal aggregation in the SAP system is of the *instantaneous* type. An example of this aggregation type is given by Kaufmann and is showed in an adopted version in Figure 2.12.

PNO	FEE	VT_FROM	VT_TO
P1	$110	2	4
P2	$90	3	6
P1	$120	5	7

(a) player fees

SUM	VT_FROM	VT_TO
$110	2	2
$200	3	4
$210	5	6
$120	7	7

(b) temporal sum

Figure 2.12: Instantaneous temporal aggregation.

Chapter 3

Tables and relationships

In this chapter we are going to introduce tables and relations between them forming a running example that will be used in subsequent chapters as a basis for discussing various temporal topics that we mentioned in Chapter 2.

Table data will be illustrated with some sample values. As we did in Chapter 2, in some cases values presented will not necessarily represent those of the underlaying data type. We wanted to make sample values more readable and keep the focus on temporal aspects.

A complete overview regarding the running example and its database diagram and relevant SQL create statements is given in Appendix B.

For all sample data the *UC* mark is set to be 9999-12-31. Current time, or *now*, is assumed to be 1. April 2016 unless otherwise noted.

3.1 Predicates

Table rows are true propositions/statements, or facts, about modelled data. Therefore, the table predicate has a generalised form for all propositions. When creating conventional database tables we do not often think about predicates in a formal way. However, regarding temporal data, they are necessary and very useful so that we can reason about data that can span across different time dimensions. This reasoning will also help us greatly in formulating necessary constraints. In the table descriptions that follow they are therefore used and formally stated.

3.2 Players

The table player represents the registration of players in an association. Table is of bitemporal type. Predicate for this table is: "Player `PNO` with the name `NAME` is registered and the membership fee is set to `FEE` during period `VT_FROM` to `VT_TO` as recorded during the period `SYS_FROM` and `SYS_TO`.". A player can be registered as a member without paying a membership fee. Paying members are provided with additional services. The figure shows some sample values for this table.

PNO	NAME	FEE	VT_FROM	VT_TO	SYS_FROM	SYS_TO
P1	Michael	$110	2000-01-01	9999-12-31	2016-01-01	9999-12-31
P2	Anders	$70	2002-04-15	9999-12-31	2016-01-01	9999-12-31
P3	Jeppe	$90	2001-06-01	9999-12-31	2016-01-01	9999-12-31
P4	Vladimir	$20	2004-10-01	2005-10-01	2016-01-01	9999-12-31
P5	John	$20	2018-01-01	9999-12-31	2016-01-01	9999-12-31

Figure 3.1: Sample data for table `PLAYER`.

3.3 Player licences

Player licences are recorded for their validity periods. To be able to join squads, i.e. sign a contract, a player must have a valid licence issued by some authoritative authority. There are different types of licences and they are also shown. Licences can be issued only to *registered* players. Predicate for this table is: "Registered player `PNO` has a licence of type `TYPE` during the period `VT_FROM` to `VT_TO`".

PNO	TYPE	VT_FROM	VT_TO
P1	Junior	2009-01-01	2013-12-31
P2	Senior	2008-04-15	9999-12-31
P3	Senior	2009-06-01	9999-12-31
P4	Goldie	2010-01-01	2012-01-01

Figure 3.2: Sample data for table `PALYER_LICENCE`.

3.4 Clubs

Table clubs stores data about registered clubs. The attribute `SINCE` indicates the time when the club has been registered. This is a special case of an *event* based table where we can deduce valid time periods. Clubs can not be logically unregistered as it can be done with players. The table represents a historical record of club registrations and is mainly used to illustrate the possibility to partially represent historical records using a single attribute.

CNO	NAME	SINCE
C1	Dial Square	1886-01-01
C1	Arsenal	1914-01-01
C2	Liverpool	1892-01-01
C3	Chelsea	1905-01-01

Figure 3.3: Sample data for table `CLUB`.

3.5 Club managers

This valid-time table is used for the club managers as the section title indicates. The club managers are registered with their contract periods that they signed with their respective clubs. The predicate for this table is: "Manager named `NAME` has signed a contract with club `CNO` for the period `VT_FROM` to `VT_TO`".

CNO	NAME	VT_FROM	VT_TO
C1	Wenger	1996-01-01	1999-12-31
C1	Wenger	2000-01-01	2011-12-31
C1	Wenger	2012-01-01	2018-01-01

Figure 3.4: Sample data for table `MANAGER`.

3.6 Squads

Only licensed players are able to join clubs, that is, sign contracts with those clubs. This is a valid time table. Valid time represents the duration of contracts. The duration of a contract in this table must be equal to or subinterval of the player's licence period. Predicate for this table is: "Licensed player `PNO` has signed contract with club `CNO` during the period `VT_FROM` to `VT_TO` for which player has a valid licence.".

PNO	CNO	VT_FROM	VT_TO
P1	C1	2010-06-01	2013-05-31
P2	C1	2008-10-01	2011-02-01
P4	C1	2010-01-01	2012-01-01

Figure 3.5: Sample data for table `SQUAD`.

Chapter 4

Temporal support

In this chapter we are going to look at and examine temporal support in Microsoft's SQL Server 2016 [18]. It has been released on June 1, 2016. We will refer to it as SQL Server from this point on. The SQL Server has been released when this thesis was being completed, and because of that there was not sufficient time to use it. However, the final version does not have any significant changes in regards to the temporal support when compared to the beta version.

4.1 System-time support

In the running example, which has been described in Chapter 3, the system time support is demonstrated for the table `PLAYER`. System time support is also called versioning in the SQL Server. This name perhaps also suggests the intended purpose of the system-time support.

4.1.1 Enabling history (versioning)

To support system time or versioning, Microsoft has provided extensions to both DDL and DML for the SQL Server [9]. System time attributes are defined with `DATETIME2` type (and with that type only). There are two of them, making a pair, that indicates start and end time of the period.The SQL extension `PERIOD FOR SYSTEM_TIME` enables user to define a period and specify which attributes are to be used to form it. Periods are defined as closed-open. When defining system time attributes, user also specifies via built-in extensions `ALWAYS AS ROW START` and `ALWAYS AS ROW END` that attribute values are to be generated as start and end value of the period. Furthermore, user also have the possibility to specify that those attributes need to be hidden. Hiding attributes will prevent them from automatically showing up in *select ** queries. However, if explicitly included in a select list, they will be shown to the user. They can be hidden by using the SQL extension `HIDDEN`. Parts of the relevant create statement is shown in Listing 4.1. Full listing for player create statement is shown in Appendix B.

```
-- From-to attributes
 SYS_FROM DATETIME2 GENERATED ALWAYS AS ROW START HIDDEN NOT NULL
 SYS_TO DATETIME2 GENERATED ALWAYS AS ROW END HIDDEN NOT NULL

-- Defining a period
  PERIOD FOR SYSTEM_TIME (SYS_FROM, SYS_TO)

-- Enabling versioning using specific name for history table
 SYSTEM_VERSIONING = ON (HISTORY_TABLE = dbo.PLAYER_HISTORY)
```

Listing 4.1: Parts of create statement enabling versioning.

System versioning is explicitly enabled by using `SYSTEM_VERSIONING = ON`, and name of the table that is created by the system in which the system will record any changes to the original table. The history table can also be created by the user but has to be *union compatible*. If the name of the history table is not specified, the system will create one following predefined naming pattern. By using two tables the system separates current and past states of the data.

After successfully enabling versioning, the user is free to modify schema of the versioned (current state) table. However, there are some minor limitations. The user can not change the type of an attribute to a different type that is not compatible. For example, it is allowed to change `VARCHAR` to `CHAR`, but changing `INT` to `NUMERIC` is not. Even if the table is empty. Any schema changes to the versioned table are automatically propagated to the history table as well. This shows that SQL Server supports schema evolution of the versioned tables, in such a way, that is completely transparent to the user.

The user has no system provided abilities to modify data in the history table directly. Every modification in the versioned table is reflected in the history table by creating previous versions. Modifications can be performed by: insert, update and delete statements. We will look into semantics of each in Section 4.1.2.

In the running-example diagram, found in Appendix B, the history table is shown as an support table without any foreign key relations to the table `PLAYER`. It should be noted that it is possible to disable and enable versioning as many times as the user wants. Each time the user enables versioning, the user can choose a different table for history as long as this table is *union compatible*. This provides users with an opportunity to partition history data across many tables, and could be very useful for very large data sets, that have been accumulated over long periods of time. However, only one history table can be active and used by the system at any given time.

The SQL Server also has built-in support for dealing with large data sets in an efficient way. Figure 4.1 shows temporal in-memory architecture that SQL Server uses to optimise access to current and recent history data. Memory optimised tables reside in memory and provide fast access. In addition, transactions performed on memory optimised tables are fully ACID compliant.

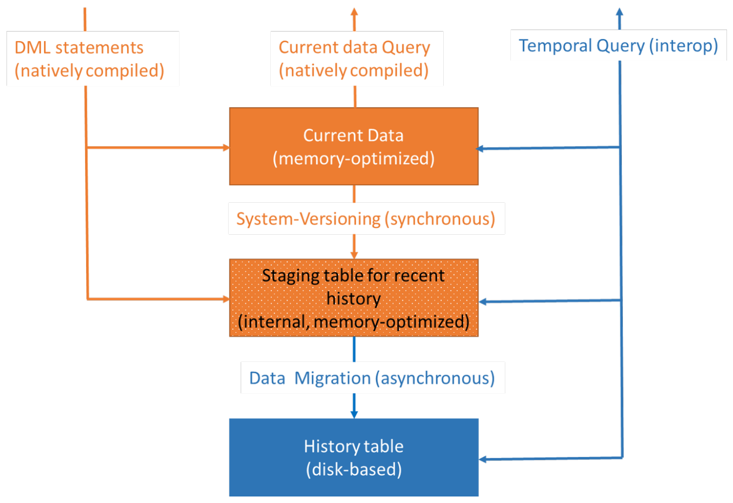

DML statements (natively compiled)
Current data Query (natively compiled)
Temporal Query (interop)
Current Data (memory-optimized)
System-Versioning (synchronous)
Staging table for recent history (internal, memory-optimized)
Data Migration (asynchronous)
History table (disk-based)

Figure 4.1: SQL Server temporal in-memory architecture. ***Source:*** *[19], used with permission from Microsoft.*

Further information on versioning can be found in reference [9], where some additional remarks are mentioned regarding creation of the versioned tables. Most importantly: 1. There must be primary key definitions for the versioned table 2. There is exactly one `PERIOD FOR SYSTEM_TIME` definition, and 3. `PERIOD` columns are always assumed to be non-nullable (even if not specified).

4.1.2 Modifying data

A general note before we go into different examples and considering the setup for the current running example. First, values for existing system time columns `SYS_FROM` and `SYS_TO` can not be modified in any way by the user[1]. Second, history table is completely

[1]During an insert user can provide default values.

read only from the user perspective[2]. Current time in the following examples is assumed to be 1. April 2016.

INSERTING: Semantics of inserting data into the player table is almost identical to the conventional insert. Data is inserted into target table and period columns are automatically updated. Nothing is recorded in the history table. Example of inserting data is shown in Figure 4.2.

```
INSERT INTO PLAYER VALUES('P1', 'Michael', '$110', '2000-01-01', '9999-12-31');
```

(a) SQL for inserting P1.

PNO	NAME	FEE	VT_FROM	VT_TO	SYS_FROM	SYS_TO
P1	Michael	$110	2000-01-01	9999-12-31	2016-04-01	9999-12-31

(b) Table player after inserting.

PNO	NAME	FEE	VT_FROM	VT_TO	SYS_FROM	SYS_TO

(c) Player history after inserting.

Figure 4.2: System-time insert.

If the user specifies system time period columns, in his insert statement, then values for those columns will be of built-in function `DEFAULT`.

UPDATING: To update the fee for player `P1`, the conventional update statement is used. What happens underneath is the following. Before a row is updated, it is inserted into history table with an adjusted period. Then, the row to be updated is removed and a new one is inserted. Periods for current and previous row will *meet*, after the operation is committed. The result of the operation is illustrated in Figure 4.3. Because we have assumed that current time is 1. April 2016, attributes `SYS_TO` and `SYS_FROM` in appropriate rows are set to this value.

In general, all updates are allowed in the current table. Even modification of primary key values, unless such modification will violate the foreign key or any other constraints.

DELETING: Deleting player `P1` will copy the relevant row to the history, close the period, and then remove original row from the current-state table. We have seen an example of period closing technique in Figure 4.3. It essentially means that end time of a period is set to the current system time. Figure 4.4 illustrates a delete operation.

[2]If versioning is turned off, user will be able to modify data in a regular way. After modification, user will be able to turn versioning on and reuse modified table.

```
UPDATE PLAYER SET FEE = '$100' WHERE PNO = 'P1';
```

(a) SQL for updating `P1`.

PNO	NAME	FEE	VT_FROM	VT_TO	SYS_FROM	SYS_TO
P1	Michael	**$100**	2000-01-01	9999-12-31	**2016-04-01**	**9999-12-31**

(b) Table player after updating `P1`.

PNO	NAME	FEE	VT_FROM	VT_TO	SYS_FROM	SYS_TO
P1	Michael	**$110**	2000-01-01	9999-12-31	**2016-01-01**	**2016-04-01**

(c) Player history after update.

Figure 4.3: System-time update.

```
DELETE PLAYER WHERE PNO = 'P1';
```

(a) SQL to delete `P1`.

PNO	NAME	FEE	VT_FROM	VT_TO	SYS_FROM	SYS_TO

(b) Table `PLAYER` after deleting `P1`.

PNO	NAME	FEE	VT_FROM	VT_TO	SYS_FROM	SYS_TO
P1	Michael	$110	2000-01-01	9999-12-31	2016-01-01	2016-04-01
P1	Michael	$100	2000-01-01	9999-12-31	2016-04-01	2016-04-02

(c) Table `PLAYER_HISTORY` after deletion.

Figure 4.4: System-time deletion.

It should be noted that the delete operation as illustrated, considering the sample data for our running example, will not succeed. The SQL Server will indeed prevent it, because of the foreign key constraint from `PLAYER_LICENCE` and `SQUAD` tables.

4.1.3 Querying

For querying the current and past states of versioned data, SQL Server provides five new temporal operators. Those operators are:

- AS OF *datetime*: Operator is used to return state as of specified timeslice.

- FROM *start_ datetime* TO *end_ datetime*: Operator is used to return rows where system time period overlaps with the user specified period. Periods are treated as closed-opened.
- BETWEEN *start_ datetime* AND *end_ datetime*. Similar to operator FROM TO, with the difference that periods are compared as closed-closed.
- CONTAINED IN (*start_ datetime*, *end_ datetime*): Operator that returns rows where the period is a subset of the user specified period.
- ALL: Operator that returns all states or versions, including the current state.

The first three operators listed correspond to operators defined in SQL:2011 [3]. The last two operators are SQL Server specific.

Figure 4.5 with the sample code shows the result of selecting players by using SQL `FOR SYSTEM_TIME` extension with operator `ALL`.

```
SELECT * FROM PLAYER FOR SYSTEM_TIME ALL;
```

(a) Select all players.

PNO	NAME	FEE	VT_FROM	VT_TO	SYS_FROM	SYS_TO
P1	Michael	$110	2000-01-01	9999-12-31	2016-01-01	2016-04-01
P1	Michael	$100	2000-01-01	9999-12-31	2016-04-01	2016-04-02

(b) Result of running select query.

Figure 4.5: System-time select by using operator ALL.

4.2 Valid-time support

The SQL Server 2016 does not provide built-in support for working with valid-time aspects of temporal data. In this section we will discuss challenges and possibilities when implementing valid-time support. Our implementation relied on the SQL Server extension and programmability infrastructure. We have also used the SQL Server integration possibility with CLR[3]. This integration possibility is described in [20].

Our implementation is based on theoretical concepts as discussed in Chapter 2, on SQL:2011 standard (temporal features) as discussed in [3], and on a concrete example of implementation as it has been presented in [7].

[3]Common Language Runtime, part of Microsoft's .NET framework

4.2.1 Temporal primary and foreign key constraints

As mentioned in Chapter 2, conventional primary and foreign key constraints are not enough to ensure that there are no duplicate or contradictory temporal statements in the table. Adding *from* and *to* attributes to primary key will not in it self prevent overlap. If we look at the table in Figure 4.6, we can see that player `P1` has two licences. First one issued while he was junior and the second when he became a senior. The problem is that the periods for these two different types of licences overlap. First row states that the player has been `Junior` in period `[2012:2013]`, and second that he was `Senior` at that same time. Two statements that clearly contradict each other. We need to prevent this from happening, and to do this we must add additional constraints to the table definition. Constraint that will ensure that licence periods for the same player do not overlap.

PNO	TYPE	VT_FROM	VT_TO
P1	Junior	2009-01-01	2013-12-31
P1	Senior	2012-01-01	2016-05-31

Figure 4.6: Overlapping player rows in `PLAYER_LICENCE`.

Listing in 4.2 adds a constraint that will do just that. The constraint relies on a user-defined function that takes player number, start and end date as parameters and returns zero or one depending on if relevant rows overlap. If an insert or update will violate overlap constraint function will return 1, and 0 otherwise. Listing for user-defined function `FnPLOverlaps` is included in Appendix C.

```
ALTER TABLE PLAYER_LICENCE
 ADD CONSTRAINT ChkPLOverlap
  CHECK (dbo.FnPLOverlaps(PNO, VT_FROM, VT_TO) = 0)
```

Listing 4.2: Adding constraint to prevent overlapping

Let us now consider the following temporal referential integrity problem. Players are not able to sign contracts with clubs if they do not have a valid licence for the signing period. Taking our running example, it concretely means that for a player to exists in table `SQUAD`, the very same player must also exist in the `PLAYER_LICENCE` table and his licence period must contain the period for which the player has signed a contract. Figure 4.7 illustrates this requirement for a single player `P1`.

The referential temporal integrity requirement must be enforced when a user modifies data in table `SQUAD` as well as when data is modified in table `PLAYER_LICENCE`. For example it should not be possible to change the licence period for a player when a player is under

PNO	TYPE	VT_FROM	VT_TO
P1	Junior	2009-01-01	2013-12-31

(a) P1's licence period

PNO	CNO	VT_FROM	VT_TO
P1	C1	2010-06-01	2013-05-31

(b) P1's contract period with C1

Figure 4.7: Contract period must be *contained in* licence period.

contract in such a way, that temporal integrity is violated. Specification of conventional foreign key from table SQUAD to table PLAYER_LICENCE referencing specific player without using *from* attribute is not possible. That is because the primary key for the table PLAYER_LICENCE is defined as the set of two attributes, namely PNO and VT_FROM.

Constraint check technique, and user-defined functions similar to the one used in the previous example can not be used to enforce temporal integrity in all cases. The reason is very simple. SQL Server does not enforce CHECK constraint when delete statements are executed. Because of that, the user could delete a contracted player from the PLAYER_LICENCE table without any checks being performed. So, to solve this problem triggers are used. For the PLAYER_LICENCE table we defined the triggers for insert, update and delete statements. For contracts in the SQUAD table defining a trigger for delete statements is not necessary.

```
IF NOT EXISTS (
 SELECT 1 FROM PLAYER_LICENCE PL INNER JOIN inserted I
 ON PL.PNO = I.PNO
 WHERE dbo.FnIsContainedIn(I.VT_FROM, I.VT_TO, PL.VT_FROM, PL.VT_TO) = 1
)
BEGIN
 RAISERROR ('Contract-licence period constraint violation.', 16, 1);
 ROLLBACK TRANSACTION;
END
```

Listing 4.3: Checking if contract period is contained in licenced period

The Listing in 4.3 shows the body of the trigger that is used to enforce the requirement that contract period is *contained in* licence period. Full trigger definition can be found in Appendix C, Listing C.3. The listing bellow shows system reporting an error if contract-licence constraint is violated by trying to insert an invalid contract period.

```
INSERT INTO SQUAD (PNO, CNO, VT_FROM, VT_TO)
 VALUES (1, 1, '2015-01-01', '2016-12-31')
-- Error message (shorten)
Contract-licence period constraint violation.
Msg 3609, Level 16, State 1, Line 125
The transaction ended in the trigger. The batch has been aborted.
```

Listing 4.4: Reporting error for an invalid contract period

The trigger for preventing licences to be deleted if their periods are referenced by contract rows is shown in C.4. This trigger relies on the fact that licence periods do not overlap, and that the contract period is *contained in* the licence period. Based on this we can then test if any contract is *contained by* the deleted licence period, and if so error is reported and transaction is rolled back.

4.2.2 Coalescing

The coalescing operation, also known as packing [12] and folding operation, is applied to periods that *overlap* or *meet*. By doing so we get longest possible periods, i.e. (*maximal intervals*), for *value-equivalent* rows. Figure 4.8 shows data based on the table `SQUAD`, with some additional rows. These new rows represent contracts that club `C2` obtained. Let us now suppose that we want to find clubs that had signed up players for the longest periods. From the Figure we can see that the longest period is obtained for club `C1`. This period is from 2008-10-01 to 2013-05-31, with a duration of about 5 years. To be able to answer these kind of queries we will need a generic operator that will, given a relation with periods, return a packed form of that relation.

PNO	CNO	VT_FROM	VT_TO
P1	C1	2010-06-01	2013-05-31
P2	C1	2008-10-01	2011-02-01
P4	C1	2010-01-01	2012-01-01
P2	C2	2011-03-01	2013-12-31
P3	C2	2014-01-01	2014-05-31

(a) unpacked contract periods

CNO	VT_FROM	VT_TO	DURATION
C1	2008-10-01	2013-05-31	~5 years
C2	2011-03-01	2014-05-31	~3 years

(b) packed contract periods

Figure 4.8: Packing of contract periods by club.

It is important to note that by packing the set of periods, as we have seen in the previous example, we in fact obtain *canonical form* of that set. By doing so we use the least number of rows to represent the same period information, as a full set of rows will do. Figure 4.8

demonstrates packing operation. Showing that the *canonical form* is in fact a unique representation of the set, in the most compact form, can also be shown with help of *unit intervals*. For example, with sets $A = \{[1:3],[2:4],[5:6]\}$, and $B = \{[1:6]\}$. Set of *unit intervals* for both sets is identical, namely: $A' = B' = \{[1:1],[2:2],[3:3],[4:4],[5:5],[6:6]\}$. Which implies that the *canonical form* of set A is in fact set B. And in general, two sets of intervals represent the same time points if their sets of *unit intervals* are equal [12].

An implementation of the coalescing operator `PACK`, as well as an implementation of the unpacking operator `UNPACK`, is shown and explained in Section 4.2.5. Unpacking a set of intervals is an opposite operation, where a new set of intervals is derived so that all intervals in that set are *unit intervals* 2.2.

4.2.3 Modifying valid-time records

In this section we are going to look into semantics of doing insert, update and delete operations on valid-time tables. For each of these operations we will present possible solutions. Discussion and solutions will be based on the table `PLAYER_LICENCE`. The sample data for this table is shown in Figure 3.3.

Inserting a new record into the temporal table is semantically equivalent to the conventional insert. That is, if all table constraints, including temporal ones, are satisfied the new record will be inserted into the table.

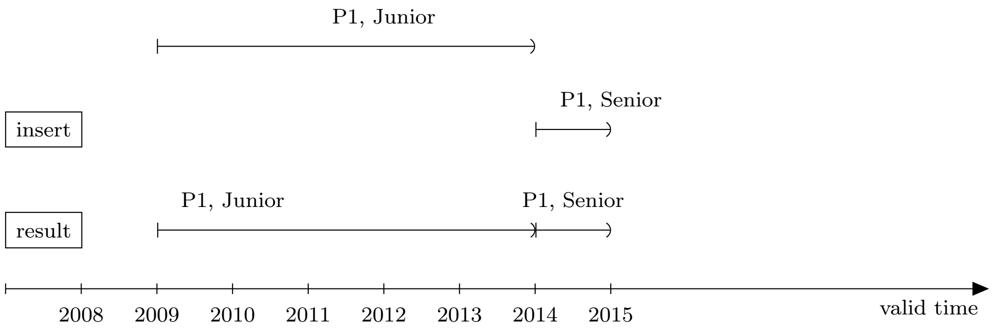


Figure 4.9: Inserting a new temporal record.

In Figure 4.9 a new row with following values is inserted `(P1, Senior, 2014-01-01, 2014-12-31)`. We can say, in general, that doing a temporal insert, if successful, will result in an increased cardinality of the table. Cardinality will be increased by one. That would semantically be equivalent to an ordinary insert statement.

Let us now consider the following update statement: "Player `P1` has been licensed as `Junior+` during the period from `2011-01-01` to `2014-05-31`", Listing 4.5. This update statement if executed as is, without any modifications by the system, will result in two rows being updated. The first row for the period when the player was licensed as a `Junior`, and second when the player was licensed as a `Senior`. This is clearly not what is intended, and will not be allowed because of the constraint that prevents overlap. For the user to be able to issue an update statement like this one, it needs to be modified on the fly by the system. In the SQL Server the semantics of a statement can be modified by using an `INSTEAD OF TRIGGER`. Using a trigger we would delete any existing rows for player `P1` for a given period and then perform an insert with provided values inside a single transaction. The problem with this approach however is that results of the update operation are not necessarily what the user intended. DB2 database system [7], DBMS that supports temporal data based on SQL:2011, will reject this statement and report an overlap error. In fact all conventional update and delete statements are also treated as such, even if temporal attributes are involved.

```
UPDATE PLAYER_LICENCE SET
 [TYPE] = 'Junior+',
 VT_FROM = '2011-01-01',
 VT_TO = '2014-05-31'
WHERE PNO = 1
```

Listing 4.5: Sample update

Listing 4.6 shows example of the same update statement that could be used to achieve the desired result in DB2. Statement is based on SQL:2011 extension formulated as `FOR PORTION OF <period> FROM <start_date> TO <end_date>` that can be used effectively for temporal update and delete operations.

```
-- DB2 example
UPDATE PLAYER_LICENCE
FOR PORTION OF BUSINESS_TIME FROM '2011-01-01' TO '2014-05-31'
SET TYPE = 'Junior+'
WHERE PNO = 1
```

Listing 4.6: Update for portion of time

Result of executing statement from Listing 4.6 with current data set is shown in Figure 4.10. Once again, this SQL statement is only used to demonstrate the effect of an update operation and is not supported by the SQL Server.

Updating the licence type for player `P1` has in our running example resulted in three rows. In general, the number of rows after an update *for a portion of time* can decrease, increase or remain unchanged. This will solely depend on the period that is being specified for the update.

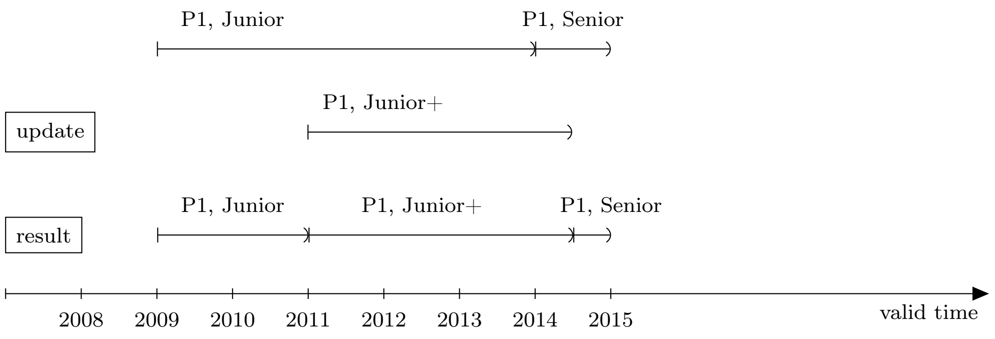


Figure 4.10: Updating valid-time record.

Semantics for delete operation are similar to update. They are differentiated from conventional delete statements by using *portion of time* extension. It should be noted that the result of the delete operation can decrease as well as increase the number of rows in the table. If we consider a row: `(P1, Junior, 2009-01-01, 2013-12-31)`, then deleting `P1` for period from `2010-01-01` to `2012-12-31` will produce two new rows. Namely `(P1, Junior, 2009-01-01, 2009-12-31)` and `(P1, Junior, 2013-01-01, 2013-12-31)`.

Listing 4.7 shows how the delete semantics of *for portion of time* can be achieved inside the SQL Server by using a parametrised stored procedure.

```
EXEC DeletePlayerLicence @PNO = 1, @FROM '2011-01-01', @TO '2014-05-31'
```

Listing 4.7: Deleting licence for specified period

4.2.4 Querying

In this section we are gonna look at three distinct queries concerning valid-time.

Query A: Get all pairs of players who were together in the same squad at some point.

SQL for this query is shown in Listing 4.8, and the result in Figure 4.11. We are joining the same table on club id, and then applying a temporal predicate to take only those records that overlap. Predicate `t1.PNO < t2.PNO` is used to eliminate redundant rows and rows where the same player matches.

```
SELECT t1.PNO AS PNO1, t2.PNO AS PNO2
FROM SQUAD t1 INNER JOIN SQUAD t2 ON t1.CNO = t2.CNO
WHERE
 dbo.FnIsOverlap(t1.VT_FROM, t1.VT_TO, t2.VT_FROM, t2.VT_TO) = 1
 AND t1.PNO < t2.PNO
```

Listing 4.8: SQL for Query A

PNO	CNO	VT_FROM	VT_TO
P1	C1	2010-06-01	2013-05-31
P2	C1	2008-10-01	2011-02-01
P4	C1	2010-01-01	2012-01-01

(a) source table

PNO1	PNO2	CNO
P1	P2	C1
P1	P4	C1
P2	P4	C1

(b) query result

Figure 4.11: Query A. Players that have played together.

If we wanted to find exact overlapping period for players then we could use the case statement shown in Listing 4.9.

```
(CASE
WHEN t1.VT_FROM > t2.VT_FROM THEN t1.VT_FROM ELSE t2.VT_FROM
END) AS OVERLAP_FROM,
(CASE
WHEN t1.VT_TO < t2.VT_TO THEN t1.VT_TO ELSE t2.VT_TO
END) AS OVERLAP_TO
```

Listing 4.9: Query A, overlapping period

The expression for finding overlapping periods can be expressed in more general terms as:

$$[MAX(i.start, j.start) : MIN(i.end, j.end)] \tag{4.1}$$

Let us now look at the next query, Query B. We want to get all players that are currently under contract, displayed by their contract length. That is, we want to find a player that has the longest contract.

Query B: Get all the active players (players under contract) ordered by the longest period of they club contract.

This query is shown in Listing 4.10, and the result is shown in the Figure 4.12.

```
SELECT *,
 DATEDIFF(day, VT_FROM, VT_TO) as NDAYS
FROM SQUAD
ORDER BY NDAYS DESC
```

Listing 4.10: SQL for Query B

If we wanted to find the longest contract length across all the clubs players played for, then we could aggregate over the result by summing contract length for each player.

PNO	CNO	VT_FROM	VT_TO
P1	C1	2010-06-01	2013-05-31
P2	C1	2008-10-01	2011-02-01
P4	C1	2010-01-01	2012-01-01

(a) source table

PNO	CNO	NDAYS
P1	C1	1095
P2	C1	972
P4	C1	760

(b) query result

Figure 4.12: Query B. Players displayed by their contract length.

We move now to the last query that involves *recursion*. Queries that involve recursion are not the most readable ones as we shall see. And they are also very hard to formulate and express in SQL.

Query C: Get all pairs of players who were at the same squad at the same year, and also display years when they were playing together and the squad they were playing for.

SQL for this query is shown in Listing 4.11, and the result is shown in the Figure 4.13.

```sql
WITH t1 AS (
 SELECT *, (YEAR(VT_TO) - YEAR(VT_FROM)) AS NYEARS FROM SQUAD
), t2 AS (
  SELECT PNO, CNO, YEAR(VT_FROM) AS VTS, NYEARS
  FROM t1
  UNION ALL
  SELECT  PNO, CNO, VTS, NYEARS - 1 AS NYEARS
  FROM t2
  WHERE NYEARS > 0
), t3 AS (
 SELECT PNO, CNO, (VTS + NYEARS) AS XYEAR FROM t2
)
SELECT x1.PNO AS PNO1, x2.PNO AS PNO2, x1.CNO, x1.XYEAR AS YEAR
FROM t3 as x1, t3 as x2
WHERE x1.CNO = x2.CNO AND x1.XYEAR =  x2.XYEAR AND x1.PNO < x2.PNO
ORDER BY x1.XYEAR, x1.PNO, x2.PNO
```

Listing 4.11: SQL for Query C

We start by computing period length in years of each contract and assigning the result to table `t1`. Then we iterate over the result reducing the year by one until `NYEARS` has reached zero. The result is assigned to table `t2`. Next, we calculate each year a player has been under contract by adding the number of years to the period start year and assign the result to table `t3`. At this point, table `t3` will contain the following entries for player `P1`: `{(P1, C1, 2010), (P1, C1, 2011), (P1, C1, 2012), (P1, C1, 2013)}`. The last step is to cross join table `t3` on club id where years are equal and produce the final result.

PNO	CNO	VT_FROM	VT_TO
P1	C1	2010-06-01	2013-05-31
P2	C1	2008-10-01	2011-02-01
P4	C1	2010-01-01	2012-01-01

(a) source table

PNO1	PNO2	CNO	YEAR
P1	P2	C1	2010
P1	P4	C1	2010
P2	P4	C1	2010
P1	P2	C1	2011
P1	P4	C1	2011
P2	P4	C1	2011
P1	P4	C1	2012

(b) query result

Figure 4.13: Query C. Players that have played together by year.

4.2.5 Interval data type

Interval (or period) as a data type does not exists in the SQL Server. In addition, SQL:2011 standard does not introduce and specify interval data type either. This shortcoming forces the user to conceptually think of two chosen attributes as start and end points of an interval, but unfortunately no support for any operations are provided. Furthermore, many very important properties of an interval are not exposed or obvious to the user. Properties such as kind of interval (open, closed, etc), duration of an interval (or scale), first and last element of underlaying data type, i.e. type that interval uses internally to represent interval points.

To overcome this problem I have created a user-defined interval and point data type in C# by utilising the CLR integration possibilities that the SQL Server provides.

Code in Listing 4.12 shows shortened create statements for the user-defined interval type and functions for creating intervals. The create assembly statement imports compiled .NET dll that provides necessary definitions and implementation. The interval type that has been implemented is based on integers and will be used to represent years in our case. Interval points are constrained to values from 1 to 9999.

```
CREATE ASSEMBLY [project] AUTHORIZATION [dbo]...
CREATE TYPE [dbo].[IntervalYear]...
CREATE FUNCTION [dbo].[IntervalYearCreate] (@s [nvarchar](MAX))
    RETURNS [dbo].[IntervalYear]...
CREATE FUNCTION [dbo].[IntervalYearCreateCC] (@b [int], @e [int])
    RETURNS [dbo].[IntervalYear]...
```

Listing 4.12: User-defined interval data type and utility functions.

There are many benefits that interval as a type provides. Some of them are: 1. User does not need to worry about which kind of interval it is when performing interval operations, because it is handled internally by the type methods. 2. Granularity or scale is handled automatically. 3. Support for min/max and previous/next interval points given an interval. 4. Automatic handling of *until changed* values. And 5. Given an interval point it is easy to test if the point is contained in the interval.

Listing 4.13 demonstrates some of the use cases. Statements and operators are shortly described within comments.

```
DECLARE @i1 IntervalYear, @i2 IntervalYear, @i3 IntervalYear

-- Create and assign an interval using type generator function
SET @i1 = IntervalYearCreateCC(2010, 2013)
-- Display user readable interval value
SELECT @i1.ToString()           -- [2010:2013]

-- Display min/max and begin/end point.
SELECT @i1.First(), @i1.Last()   -- 1, 9999
SELECT @i1.Begin(), @i1.End()    -- 2010, 2013

-- Check if specific year is contained in the interval.
SELECT @i1.Contains(2010)       -- 1

-- Create and assign some additional intervals using interval literals.
-- All four kind of intervals are supported.
SET @i1 = '[2010:2013]' SET @i2 = '[2008:uc)' SET @i3 = '(2010:2012)'

-- Display interval i2 and show duration. Current year is 2016.
SELECT @i2.ToString(), @i2.Duration()       -- [2008:9999], 8

-- Check if intervals overlap or meet. Those two operators are commutative.
SELECT @i1.Overlaps(@i2)       -- 1
SELECT @i1.Meets(@i2)          -- 0

-- Get intersection interval for two overlapping intervals.
-- If intervals are not overlapping, an error occurs.
SELECT @i1.Intersect(@i2).ToString()     -- [2010:2010]

-- Test if an interval is unit interval
SELECT @i1.Intersect(@i2).IsUnit()     -- 1
```

Listing 4.13: User-defined interval type demo.

Now, with the interval data type in place, we turn our attention to *pack* C.5 and *unpack* C.6 functions. These two functions are not implemented via CLR, but they both heavily depend on the new interval type. We will not go into implementation details, but rather discuss possible use cases for them. These two functions are, as mentioned, listed in full in Appendix C.

Let us suppose that we want to find the *maximal interval* for a set of intervals. We could, for example, be interested in finding a club with the longest contract interval across all active players. Listing 4.14 demonstrates starting point for a possible solution utilising the *pack* function.

```
DECLARE @TempTable dbo.IntervalBasedTable
INSERT INTO @TempTable VALUES ('[2010:2013]'), ('[2008:2011]'),
    ('[2010:2012]'), ('[2000:2001]')
SELECT DURING.ToString() AS DURING, DURING.Duration() AS DURATION
    FROM dbo.IntervalPack(@TempTable)
ORDER BY DURATION DESC
-- Result
   DURING     DURATION
[2008:2013]          6
[2000:2001]          2
```

Listing 4.14: Finding maximal interval.

Intervals are inserted into a temporary table, which is then packed and the result displayed. Rows are ordered by duration of the intervals.

When trying to solve the problem for *Query C* 4.13, rather complex SQL with recursion has been used. Let us now suppose that we wanted to find all the years for which the player `P1` had a licence but was without contract. SQL for this query with the result is shown in Listing 4.15. If we wanted to find the longest period (maximal interval) for which a player was licensed but without club contract we would just *pack* the result.

We start by creating two temporary tables, one for licence and one for contract intervals. We then *unpack* both and take the set difference by using the regular `EXCEPT` operator.

```
DECLARE @pl dbo.IntervalBasedTable
DECLARE @squad dbo.IntervalBasedTable
INSERT INTO @pl VALUES ('[2009:2013]'), ('[2014:2014]')
INSERT INTO @squad VALUES ('[2010:2013]');
WITH t (during) AS (
    SELECT during.ToString() FROM dbo.IntervalUnpack(@pl)
    EXCEPT
    SELECT during.ToString() FROM dbo.IntervalUnpack(@squad)
)
SELECT * FROM t
-- Result
[2009:2009]
[2014:2014]
```

Listing 4.15: Finding interval difference by interval points.

Chapter 5

Results and findings

5.1 Support and limitations

The SQL Server 2016 provides support only for system-time aspect of temporal data support. Enabling versioning on an existing table or creating a new one with history support is straight forward and easy to use. The possibility to automatically hide system-time attributes means that it will be easier to adopt this new technology because users are not forced to change existing code, and can adopt this new feature on an as needed basis. DML statements that have been used before enabling history will continue to work without any modifications after history is enabled. The provided features for system-time support seem adequate for the intended purpose, namely providing versioning support, but that may be too limiting for some user needs. If the user just wants to keep track of the data changes, or to save changes for some period of time it will be very easy to do so. Querying historical records for specific time-slices is supported via `FOR SYSTEM_TIME` extensions. Which means that the user can easily see previous states of the data and possibly rollback any changes by using conventional SQL statements.

If the user wishes to use more complex temporal queries involving period operators or wishes to aggregate over system-time data then such functionality needs to be built either using user-defined functions, procedures and types or in combination with the application code. Because the SQL Server does not provide any support for such requirements, the usefulness will be determined solely by the concrete user needs.

The SQL Server's support for memory-optimised temporal tables, which can provide fast and scalable access to temporal data is a very practical and useful feature. Because memory-optimised tables are also updatable, and are fully ACID compliant, they will be easily adoptable. Mainly because, seen from a user perspective, semantics of memory-optimised temporal tables regarding synchronisation of updates to the disk are handled automatically by the system. However, possibilities for conflict of two write operations do exist. If they conflict one will "win" and the other needs to be resubmitted by the user. Such conflicts can impact performance and scalability of the memory-optimised tables.

Because the SQL Server does not provide period data type there is no direct support for indexing periods as such. The SQL server does support various other index types that can be utilised and are optimised for fast memory-based index scans.

Support for valid-time and therefore for bitemporal data is not provided by the SQL Server, and the user is left alone to implement such support on its own. The extension possibilities that the SQL Server provides do help in this regard, but without built-in server support it is very hard, if not impossible, to build support as defined in the SQL:2011 standard and mentioned in the literature to be general enough, so that it can be reused across projects. Even if such support is built by the user it will be bound to the relational model under consideration and will not be reusable if the model changes.

Created prototype includes support for temporal primary and foreign keys. It has been demonstrated that triggers and stored procedures can be used to achieve additional temporal support regarding updates and deletes of temporal data. However, this support is pertinent to the relational model that has been used. The interval data type that has been implemented and demonstrated is general in nature, and could be reused across different models but most of the temporal operations using it will also need the knowledge of the table(s) where this data type is used. Because of that generalisation is lost.

With user-defined interval data type it has been demonstrated that it is fairly easy to provide support for:

1. Begin() and End() methods: which return start and the end of an interval respectively.
2. First() and Last() methods: which return first and last possible point for an interval respectively.
3. Duration()/Count(): which return duration of an interval.
4. Prev()/Next(): which return previous and next interval point respectively. Method Prev() will fail on an interval where start = First(), and Next() will fail on an interval where end = Last().

5. Interval operators such as: Overlaps(), Meets(), Contains().
6. Set-like operators like: Intersect(), Union() and Minus(). Which will either fail or return proper intervals.
7. Handling of different types of intervals
8. Handling of *until changed* values
9. Handling of interval constraints such as: $start \leq end$, and minimal and maximal allowed values.

Source code for the prototype is described in Appendix D and included as a zip package.

Chapter 6

Conclusion

6.1 Conclussion

System-time, or versioning support, in the SQL Server 2016 has been designed to solve specific temporal problems and in my view they have succeeded in doing that. Use cases like data auditing, where it is needed to keep track of changes, who made the changes and when. Or for example point in time analysis, where analyst may be interested in finding differences in trends over time are very well supported and will most probably satisfy user needs. Therefore, the SQL Server 2016 temporal support is definitely worth considering if an organisation has similar needs. In my own company we have already identified several specific scenarios where versioning will perfectly fit in solving our problems.

From my own implementation experience to provide temporal support for valid-time it seems that the problem is intrinsically difficult to solve because of the various ways that temporal data can be used and modified. However, it seems that interval data type is a fundamental building block in providing such support. It is therefore very unfortunate that SQL:2011 standard did not introduce interval data type and supporting constructs. An argument has been given in [3], that despite the recognised need for the definition of period data type (interval is a reserved keyword in SQL), considerations regarding compatibility with the existing software stack have prevailed. However, introduction and acceptance of new data types, such as DATE/TIME/TIMESTAMP and others, into standard is nothing new which makes the mentioned argument weaker. Working with pairs of attributes which designate the start and end mark of an interval is almost like using three attributes to represent year, month and a day rather then using a single attribute of type DATE. It is just not good enough at least from an data abstraction point of view. Furthermore, if an interval data type is not defined, is it then possible to create optimal interval-aware index structures that will be used to optimise access to temporal data?

6.2 Future directions

Interval data type presented in this thesis is based on integers and can be used to represent year-based intervals. It would be generally more useful if I had implemented an interval interface so that inheritance and other object-oriented techniques could be used to easily extend and create other interval types. Interval types based for example on date, date-time, or numeric types. Or perhaps to create more specialised cases of those types. In some applications it may be necessary to be able to represent for example money, pH or temperature intervals.

In my implementation of packing and unpacking operators we have solved a special case problem where the input is a set of intervals without any other attributes. For those operators to be generally useful, they must pack and unpack rows based on value-equivalence of the input rows. One possible solution to this problem is to introduce a new data type that will represent a set of unit intervals, which can then be used to store unpacked data per row.

Appendix A

Interval operators

Intervals i and j	Operator	Condition
$\vdash i \dashv$ $\vdash j \dashv$	i `EQUALS` j	$i.s = j.s \wedge i.e = j.e$
$\vdash\!\!\!\longrightarrow i \longleftarrow\!\!\!\dashv$ $\vdash j \dashv$	i `INCLUDES` j	$i.s \leq j.s \wedge i.e \geq j.e$
$\vdash i \dashv$ $\vdash\!\!\!\longrightarrow j \longleftarrow\!\!\!\dashv$	i `INCLUDED_IN` j	j `INCLUDES` i
$\vdash\!\!\!\longrightarrow i \longleftarrow\!\!\!\dashv$ $\vdash j \dashv$	i `PINCLUDES` j	*Properly includes* : $i.s < j.s \wedge i.e > j.e$
$\vdash i \dashv$ $\vdash\!\!\!\longrightarrow j \longleftarrow\!\!\!\dashv$	i `PINCLUDED_IN` j	*Properly included in* : j `PINCLUDES` i
$\vdash i \dashv$ $\vdash j \dashv$	i `BEFORE` j	$i.e < j.s$
$\vdash i \dashv$ $\vdash j \dashv$	i `AFTER` j	$i.s > j.e$
$\vdash i \dashv$ $\vdash j \dashv$	i `OVERLAPS` j	$i.s \leq j.e \wedge i.e \geq j.s$
$\vdash i \dashv \vdash j \dashv$	i `MEETS` j	$(i.e + scale = j.s) \vee$ $(j.e + scale = i.s)$
$\vdash i \dashv \vdash j \dashv$ $\vdash i \dashv$ $\vdash j \dashv$	i `MERGES` j	i `OVERLAPS` $j \vee i$ `MEETS` j
$\vdash i \dashv$ $\vdash\!\!\!\longrightarrow j \longleftarrow\!\!\!\dashv$	i `BEGINS` j	$i.s = j.s \wedge i.e \in j$
$\vdash i \dashv$ $\vdash\!\!\!\longrightarrow j \longleftarrow\!\!\!\dashv$	i `ENDS` j	$i.e = j.e \wedge i.s \in j$

Figure A.1: Interval operators as defined by Date et al. [12].

Appendix B

Running example

B.0.1 Database diagram

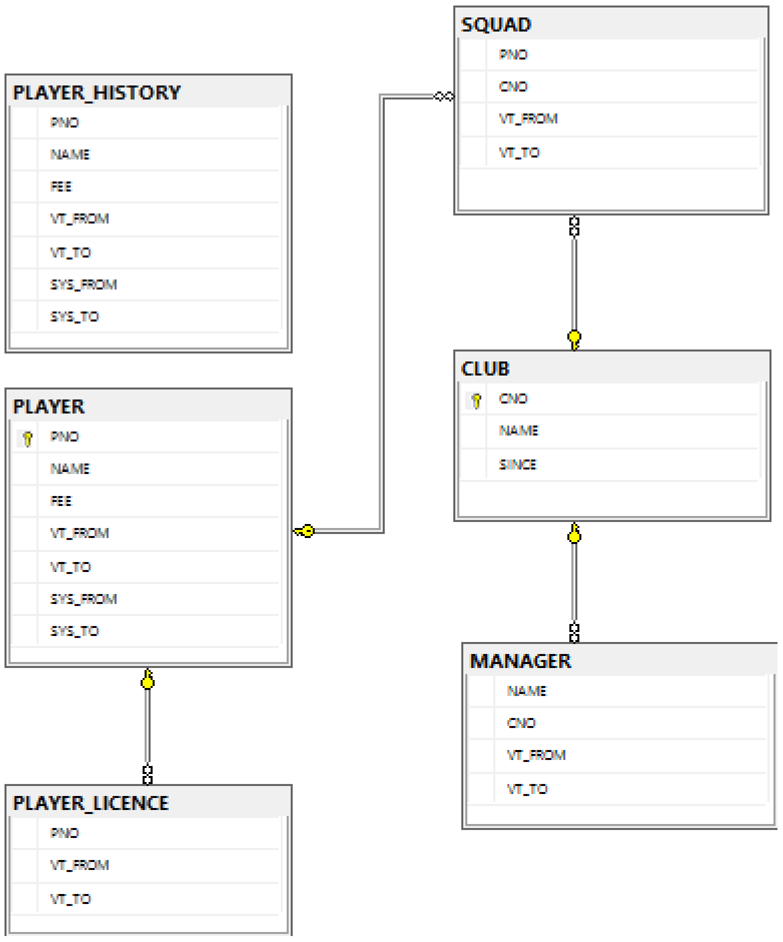


Figure B.1: Database diagram for running example.

B.0.2 Running example - DDL

```
CREATE TABLE dbo.PLAYER (
 PNO INT NOT NULL,
 NAME VARCHAR(255) NOT NULL,
 FEE MONEY NOT NULL,
 VT_FROM DATE NOT NULL,
 VT_TO DATE NOT NULL,
 SYS_FROM DATETIME2 GENERATED ALWAYS AS ROW START NOT NULL,
 SYS_TO DATETIME2 GENERATED ALWAYS AS ROW END NOT NULL,
 PRIMARY KEY (PNO),
 UNIQUE (PNO, VT_FROM),
 CONSTRAINT ChkPlayerFromLEQTo CHECK (VT_FROM <= VT_TO),
  PERIOD FOR SYSTEM_TIME (SYS_FROM, SYS_TO)
)
WITH
(
 SYSTEM_VERSIONING = ON (HISTORY_TABLE = dbo.PLAYER_HISTORY)
);

CREATE TABLE dbo.PLAYER_LICENCE (
 PNO INT NOT NULL,
 [TYPE] INT NOT NULL,
 VT_FROM DATE NOT NULL,
 VT_TO DATE NOT NULL,
 UNIQUE (PNO, VT_FROM),
 CONSTRAINT ChkPLFromLEQTo CHECK (VT_FROM <= VT_TO),
 CONSTRAINT FkPLRefPlayer FOREIGN KEY (PNO) REFERENCES PLAYER(PNO)
);

CREATE TABLE dbo.CLUB (
 CNO INT NOT NULL,
 NAME VARCHAR(255) NOT NULL,
 SINCE DATE NOT NULL,
 PRIMARY KEY(CNO),
 UNIQUE (CNO, SINCE)
);

CREATE TABLE dbo.MANAGER (
 NAME VARCHAR(255) NOT NULL,
 CNO INT NOT NULL,
 VT_FROM DATE NOT NULL,
 VT_TO DATE NOT NULL,
 PRIMARY KEY (NAME, VT_FROM),
 CONSTRAINT ChkManagerFromLEQTo CHECK (VT_FROM <= VT_TO),
 CONSTRAINT FkMangerRefClub FOREIGN KEY (CNO) REFERENCES CLUB(CNO)
);

CREATE TABLE dbo.SQUAD (
 PNO INT NOT NULL,
 CNO INT NOT NULL,
 VT_FROM DATE NOT NULL,
 VT_TO DATE NOT NULL,
 PRIMARY KEY (PNO, CNO, VT_FROM),
 CONSTRAINT ChkSquadFromLEQTo CHECK (VT_FROM <= VT_TO),
 CONSTRAINT FkSquadRefPlayer FOREIGN KEY (PNO) REFERENCES PLAYER(PNO),
 CONSTRAINT FkSquadRefClub FOREIGN KEY (CNO) REFERENCES CLUB(CNO)
);
```

Listing B.1: DDL for running example

Appendix C

Code examples

C.0.1 Temporal integrity

```
-- Test if [s1:e1] overlaps [s2:e2]
CREATE FUNCTION dbo.FnIsOverlap(
 @s1 DATE, @e1 DATE, @s2 DATE, @e2 DATE
)
RETURNS BIT AS
BEGIN
 RETURN (SELECT CASE WHEN (
   @s1 <= @e2 AND @e1 >= @s2
  ) THEN 1 ELSE 0 END
 )
END
-- Test if [s1:e1] contained in [s2:e2]
CREATE FUNCTION dbo.FnIsContainedIn(
 @s1 DATE, @e1 DATE, @s2 DATE, @e2 DATE
)
RETURNS BIT AS
BEGIN
 RETURN (SELECT CASE WHEN (
   @s1 >= @s2 AND @e1 <= @e2
  ) THEN 1 ELSE 0 END
 )
END
```

Listing C.1: User-defined helper functions used as interval operators

```
CREATE FUNCTION dbo.FnPLOverlaps(
 @pno INT, @start DATE, @end DATE
)
RETURNS BIT AS
BEGIN
 DECLARE @retval BIT
 SET @retval = 0
 IF EXISTS (
  SELECT * FROM PLAYER_LICENCE PL
  WHERE PL.PNO = @pno AND PL.VT_FROM != @start
  AND dbo.FnIsOverlap(PL.VT_FROM, PL.VT_TO, @start, @end) = 1
   ) BEGIN
    SET @retval = 1
   END
   RETURN @retval
END
```

Listing C.2: User-defined function for overalpping check

```
CREATE TRIGGER dbo.TrCheckPeriodContainedIn
ON dbo.SQUAD
FOR INSERT, UPDATE AS
BEGIN
 IF NOT EXISTS (
  SELECT 1 FROM PLAYER_LICENCE PL INNER JOIN inserted I
  ON PL.PNO = I.PNO
  WHERE dbo.FnIsContainedIn(I.VT_FROM, I.VT_TO, PL.VT_FROM, PL.VT_TO) = 1
 )
 BEGIN
  RAISERROR ('Contract-licence period constraint violation.', 16, 1);
  ROLLBACK TRANSACTION;
 END
END
```

Listing C.3: Trigger enforcing that contract periods are *contained in* licenced periods

```
CREATE TRIGGER dbo.TrCheckContractPeriodsContainedBy
ON dbo.PLAYER_LICENCE
FOR DELETE AS
BEGIN
 IF EXISTS (
  SELECT 1 FROM SQUAD S INNER JOIN deleted D
  ON S.PNO = D.PNO
  WHERE dbo.FnIsContainedIn(S.VT_FROM, S.VT_TO, D.VT_FROM, D.VT_TO) = 1
 )
 BEGIN
  RAISERROR ('Removing licence violates contract constraint.', 16, 1);
  ROLLBACK TRANSACTION;
 END
END
```

Listing C.4: Trigger enforcing that licences can not be deleted if their periods are referenced

```
CREATE FUNCTION IntervalPack
(
 @tbl dbo.IntervalBasedTable READONLY
)
RETURNS @xtbl TABLE (
    during IntervalYear NOT NULL
   )
AS
BEGIN
 DECLARE @during IntervalYear

 DECLARE MY_CURSOR CURSOR
   LOCAL STATIC READ_ONLY FORWARD_ONLY
 FOR
 SELECT during
 FROM @tbl
 ORDER BY during.[Begin]

 OPEN MY_CURSOR
 FETCH NEXT FROM MY_CURSOR INTO @during
 WHILE @@FETCH_STATUS = 0
 BEGIN
  --Do expanding
  IF NOT EXISTS (SELECT * FROM @xtbl WHERE during.Includes(@during) = 1)
  BEGIN
   IF EXISTS (SELECT * FROM @xtbl WHERE during.Merges(@during) = 1)
   BEGIN
    UPDATE @xtbl SET during = during.[Union](@during) WHERE during.Merges(@during)
    = 1
   END
   ELSE
   BEGIN
    INSERT INTO @xtbl VALUES(@during)
   END
  END
  FETCH NEXT FROM MY_CURSOR INTO @during
 END
 CLOSE MY_CURSOR
 DEALLOCATE MY_CURSOR

 RETURN
END
```

Listing C.5: Pack function

```
CREATE FUNCTION IntervalUnpack
(
 @tbl dbo.IntervalBasedTable READONLY
)
RETURNS @xtbl TABLE (
    during IntervalYear NOT NULL
   )
AS
BEGIN
 DECLARE @during IntervalYear

 DECLARE MY_CURSOR CURSOR
   LOCAL STATIC READ_ONLY FORWARD_ONLY
 FOR
 SELECT during
 FROM @tbl
 ORDER BY during.[Begin]

 OPEN MY_CURSOR
 FETCH NEXT FROM MY_CURSOR INTO @during
 WHILE @@FETCH_STATUS = 0
 BEGIN
  --Do expanding
  IF NOT EXISTS (SELECT * FROM @xtbl WHERE during.Equals(@during) = 1)
  BEGIN
   DECLARE @i int
   DECLARE @unitInterval IntervalYear
   SET @i = @during.[Begin]
   WHILE @i <= @during.[End]
   BEGIN
    SET @unitInterval = dbo.IntervalYearCreateCC(@i, @i)
    IF NOT EXISTS (SELECT * FROM @xtbl WHERE during.Equals(@unitInterval) = 1)
    BEGIN
     INSERT INTO @xtbl VALUES (@unitInterval)
    END
    SET @i = @i + 1
   END
  END
  FETCH NEXT FROM MY_CURSOR INTO @during
 END
 CLOSE MY_CURSOR
 DEALLOCATE MY_CURSOR

 RETURN
END
```

Listing C.6: Unpack function

Appendix D

Source code

Source code is attached in a zip file called source.zip. Visual studio project for interval implementation is in file IntervalYear.cs inside Database1 directory. To enable CLR in SQL Server 2016 please use CLRSupport.sql file. File Objects.sql contains definitions for various user-defined functions and triggers. Other file names should be self-explanatory regarding their intended usage.

```
Root
├── CLRSupport.sql
├── Database1
│   ├── IntervalKind.cs
│   ├── IntervalYear.cs
│   ├── IntervalYearCreate.cs
│   └── IntervalYearPointExtensions.cs
├── Intervals.sql
├── Objects.sql
├── Pack.sql
├── PackDemo.sql
├── QueryA.sql
├── QueryB.sql
├── QueryC.sql
├── TablesCreate.sql
├── TablesDrop.sql
├── Unpack.sql
└── UnpackDemo.sql
```

Bibliography

[1] Nick Kline. An update of the temporal database bibliography. 1993.

[2] Thomas F. Zurek. *Optimisation of Partitioned Temporal Joins*. PhD thesis, University of Edinburgh, 1997.

[3] Krishna Kulkarni and Jan-Eike Michels. Temporal features in sql:2011. *SIGMOD Record (Vol. 41, No. 3)*, 2012.

[4] Hugh Darwen and C. J. Date. An overview and analysis of proposals based on the tsql2 approach. 2005. Draft.

[5] Richard T. Snodgrass, editor. *The TSQL2 Temporal Query Language*, 1995.

[6] Flashback query in oracle 9i, 2002. URL http://www.oracle-developer.net/display.php?id=210.

[7] Cynthia M. Saracco, Matthias Nicola, and Lenisha Gandhi. A matter of time: Temporal data management in db2 10. *IBM developerWorks*, 2012. URL https://www.ibm.com/developerworks/data/library/techarticle/dm-1204db2temporaldata/.

[8] Richard T. Snodgrass. A case study of temporal data.

[9] Temporal tables - sql server 2016. URL https://msdn.microsoft.com/en-us/library/dn935015.aspx.

[10] Dušan Petković. Temporal data in relational database systems: A comparison. *Springer International Publishing, New Advances in Information Systems and Technologies*, 2016.

[11] Martin Kaufmann. *Storing and Processing Temporal Data in Main Memory Column Stores*. PhD thesis, ETH Zurich, 2014.

[12] C.J. Date, Hugh Darwen, and Nikos Lorentzos. *Time and Relational Theory*. Morgan Kaufmann, 2nd edition, 2014.

[13] C. S. Jensen, J.Clifford, S. K. Gadia, A. Segev, and R.T. Snodgrass. A glossary of temporal database concepts. *SIGMOD Record*, 21(3), 1992.

[14] Chengjie Mao, Hui Ma, Yong Tang, and Liangchao Yao. *Temporal Information Processing Technology and Its Application*, chapter Temporal Data Model and Temporal Database Systems, pages 69–89. Springer Berlin Heidelberg, 2010.

[15] Allen James F. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 1983.

[16] Christian S. Jensen et al. The consensus glossary of temporal database concepts - february 1998 version. 1998.

[17] Michael H. Böhlen, Richard T. Snodgrass, and Michael D. Soo. Coalescing in temporal databases. Technical Report R-96-2026, Aalborg Univeristy, 1997.

[18] Sql server 2016, 2016. URL `https://www.microsoft.com/en/server-cloud/products/sql-server/overview.aspx`.

[19] System-versioned temporal tables with memory-optimized tables. URL `https://msdn.microsoft.com/en-us/library/mt590207.aspx`.

[20] Introduction to sql server clr integration. URL `https://msdn.microsoft.com/en-us/library/ms254498(v=vs.110).aspx`.