

Temporal Database System

Jaymin Patel

MEng

Individual Project

18 June 2003

Department of Computing, Imperial College, University of London

Supervisor: Peter McBrien
Second Marker: Ian Phillips

Abstract

More and more organisations everyday are now storing achieves of data to help them make important business decisions. The Commercial Database Management Systems (DBMS) that are available in the I.T. market do not provide any significant and useful methods of storing and manipulate such data. Whilst there has been research into the subject of Temporal Databases, there is a lack of commercial tools.

The project aims to design and develop a Temporal Database Management System (TDBMS) that provides means to manipulate temporal data. The TDBMS will provide a Temporal Relational Algebra (TRA) structured query language that is based on and is an extension to Relational Algebra (RA) for extracting data from the Temporal Database. The TRA is to contain most the common operators that are associated with RA such as Cartesian Product, Natural Join, Union and many more. The TRA will introduce new temporal operators, Until and Since based on US logic, which are useful for querying Temporal Databases. US logic is the result of research work undertaken at Imperial College, which have been shown to be useful for Temporal Databases and is mathematically sound.

Acknowledgement

I would like to thank Dr Peter McBrien for the project proposal and supervising of my project. He has taken the time to show an interest and his guidance throughout the whole project has given me a greater understanding of Temporal Databases. He has helped me to develop an interesting system.

Contents

Chapter 1 Introduction	8
1.0 Overview.....	8
1.1 Database Systems and the Different Types	8
1.11 Relational Database	8
1.12 Object-Oriented Database.....	8
1.13 Spatial Database.....	8
1.14 Temporal Database	9
1.2 Temporal Database Introduction.....	9
1.3 Different Forms of Temporal Databases.....	9
1.4 Application domains of Temporal Data.....	9
1.5 Solutions in developing a Temporal Database.....	10
1.6 Research and Available TDBMS.....	10
1.7 Objectives of the Project.....	11
1.8 Report Structure	11
Chapter 2 Background.....	13
2.1 Temporal Dimension	13
2.11 Time Views.....	13
2.12 Continuous Time.....	13
2.13 Discrete Time.....	13
2.14 Chronon.....	13
2.15 Modelling Time	13
2.2 Temporal Database Forms and Data Model	14
2.21 Transaction time.....	14
2.22 Valid time.....	14
2.23 Data Model for Temporal Data.....	14
2.3 Temporal Structure	15
2.31 Other Temporal Structures.....	17
2.4 Temporal Normal Form	17
2.5 Temporal Query Language	18
2.51 Relational Algebra - RA	18
2.52 Temporal Relational Algebra – TRA.....	18
2.53 New Temporal Operators.....	19
2.54 US logic	19
2.6 Auxiliary Functions	19
2.61 Overlap.....	19
2.62 Min.....	20
2.63 Max	20
2.7 Querying Temporal Database	20
2.8 Implementing TRA in RA	21
2.801 Example Relations	22
2.802 Project π_t	23
2.803 Coalesce Function.....	23
2.804 Select σ_t	24
2.805 Product \times_t	25
2.806 Since Product S_x	26
2.807 Until Product U_x	26

2.808 Since and Until Product Explanation.....	26
2.809 Join \bowtie_t	27
2.810 Since Join $S \bowtie$	27
2.811 Until Join $U \bowtie$	28
2.812 Since and Until Join Explanation.....	28
2.813 Difference $-_t$	28
2.814 Union \cup_t	29
2.9 Derived Modal Operators	31
2.91 Past.....	31
2.92 Previous.....	32
2.93 Always in Past.....	33
2.94 Future	34
2.95 Next.....	35
2.96 Always in Future.....	36
 Chapter 3 Specification.....	 37
3.1 Introduction.....	37
3.2 System Architecture.....	37
3.3 Middleware Architecture	39
 Chapter 4 Design	 40
4.1 Analyser	40
4.2 Parser - TRA Tree Generator.....	41
4.3 Temporal Query Processor - TQP.....	43
4.4 User Interface.....	44
4.5 Temporal Database	44
 Chapter 5 Implementation	 45
5.1 System Technology.....	45
5.2 Development Approach	46
5.3 Implementation Approach	46
5.4 Temporal Database Development.....	47
5.5 TRA Grammar Development.....	47
5.6 Analyser	48
5.7 Parser.....	49
5.71 Validating Tokens against TRA grammar	49
5.72 Tree Generation	50
5.73 Nested Queries	52
5.8 Temporal Query Processor	54
5.81 Data Structures.....	54
5.82 Interpreting the Tree	55
5.83 Temporal Relational Operators.....	56
5.831 Natural Product, Since Product and Until Product	56
5.832 Join, Since Join and Until Join.....	58
5.833 Project	61
5.834 Union.....	61
5.835 Difference	62
5.836 Modal Operators - Past, Previous, AlwaysPast, Future, Next and AlwaysFuture	64
5.837 At.....	68

5.838 Coalesce	68
5.839 Rename	68
5.84 Operator Optimisation	69
5.85 Auxiliary Functions	69
5.86 Connecting to the Temporal Database.....	69
5.9 Interface	70
5.91 Command/Input Interface	70
5.92 Result/Output Interface	70
Chapter 6 Testing	71
6.1 Test Data Used	71
6.2 Test Specification.....	71
6.3 Validation and Verification of Modules	72
6.31 Analyser	72
6.32 Parser.....	72
6.321 Validating Tokens.....	72
6.322 Generating TRA Trees.....	73
6.33 Temporal Query Processor - TQP.....	73
6.331 Interpreting TRA Trees.....	73
6.332 Functionality of the Operators	73
6.333 Operator Verification	74
6.34 Error Handling	74
6.35 Interface	74
6.4 Defect Handling	75
6.5 System Performance	75
6.51 Performance Analysis	75
6.6 Stress Evaluation.....	76
Chapter 7 Evaluation	77
7.1 Module Evaluation.....	77
7.11 Analyser	77
7.12 Parser.....	77
7.13 Temporal Query Processor	78
7.131 Interpreting TRA Trees and Tree Optimisation.....	78
7.132 Overall TQP.....	78
7.14 Interface	79
7.2 Improvements	80
7.3 Problems	80
Chapter 8 Conclusion	81
8.1 Overall.....	81
8.2 Achievements.....	81
8.3 Extensions.....	82
 APPENDIX A - Considering Implementation Technology	 83
APPENDIX B - System Development Outline	85
APPENDIX C – Main content of TRA Grammar.....	87
APPENDIX D – Tree Node UML Diagram.....	88

APPENDIX E – Unary Node Types UML Diagram.....	89
APPENDIX F – Table data structure UML Diagram.....	90
APPENDIX G – Modal Operators UML Diagram	91
APPENDIX H – Command Interface	92
APPENDIX I - Output Interface.....	94
APPENDIX J - Unary Operator Performance	95
APPENDIX K - Binary Operator Performance.....	96
 Bibliography	 97
Publications.....	97
Websites	97

Chapter 1 Introduction

1.0 Overview

Conventional database management systems (DBMS) are responsible for the storage and processing of huge amounts of information. The data stored by these database systems refers to information valid at present time, valid *now*. It concerns data that is believed to be true in reality at the present moment. *Past data* refers to information that was stored in the database at an earlier time, data that is believed to have existed in the past, valid at some time before now. *Future data* refers to information considered to be valid at a future time instance, data that will be true in the near future, valid at some time after now. The commercial DBMS of today used by organisations and individuals, such as Oracle, DB2, Sybase, Postgres etc, do not provide models to support and process (retrieving, modifying, inserting and removing) past and future data.

1.1 Database Systems and the Different Types

A Database Management System (DBMS) is a collection of interrelated data and a set of programs to access data stored. A DBMS that stores data must have a well-defined format to model and store information. There are four well known modelling techniques to represent data for storage, a relational database, an object-oriented database, a spatial database and a temporal database.

1.11 Relational Database

A *relational database* stores data in tables, known as relations. Each table consists of rows, known as tuples and columns, known as attributes or fields. A row contains data about a specific entity. Thus, a relational database stores a set of tables.

1.12 Object-Oriented Database

Object-oriented database stores data about entities in objects. The type of object specifies the properties (attributes) the object contains. Sets of objects of the same type are called collections. Thus, an object-oriented database contains a set of collections.

1.13 Spatial Database

Spatial database concerns the storing of data in relation to *space*. It offers spatial data types and stores information relating to geometric or geographical space, for example, the 2-D abstraction of the Earth's surface or the 3-D space representing the arrangement of chains of protein molecules. Thus, a spatial database stores a collection of space related data.

1.14 Temporal Database

Temporal database stores data relating to time instances. It offers temporal data types and stores information relating to past, present and future time, for example, the history of the stock market or the movement of employees within an organisation. Thus, a temporal database stores a collection of time related data.

1.2 Temporal Database Introduction

A temporal database is formed by compiling, storing temporal data. The difference between temporal data and non-temporal data is that a time period is appended to data expressing when it was valid or stored in the database. The data stored by conventional databases consider data to be valid at present time as in the time instance “now”. When data in such a database is modified, removed or inserted, the state of the database is overwritten to form a new state. The state prior to any changes to the database is no longer available. Thus, by associate time with data, it is possible to store the different database states.

In essence, temporal data is formed by time-stamping ordinary data (type of data we associate and store in conventional databases). In a relational data model, tuples are time-stamped and in an object-oriented data model, objects/attributes are time-stamped. Each ordinary data has two time values attached to it, a start time and an end time to establish the time interval of the data. In a relational data model, relations are extended to have two additional attributes, one for start time and another for end time.

1.3 Different Forms of Temporal Databases

Time can be interpreted as valid time (when data occurred or is true in reality) or transaction time (when data was entered into the database).

A *historical database* stores data with respect to valid time.

A *rollback database* stores data with respect to transaction time.

A *bitemporal database* stores data with respect to both valid and transaction time – they store the history of data with respect to valid time and transaction time.

1.4 Application domains of Temporal Data

Examples of application domains dealing with temporal data are:

- Financial Applications – e.g. history of stock markets; share prices
- Reservation Systems – e.g. when was a flight booked
- Medical Systems – e.g. patient records
- Computer Applications – e.g. history of file back ups
- Archive Management Systems – e.g. sporting events, publications and journals
- ...etc

It is always possible to identify application domains of temporal data since any data can be represented as temporal data. The question is: when is such transition (considering all different states of data) required and important? The answer often is how vital it is to have temporal data for an organisation and the benefits that it will bring. Often, when organisations are storing archives of data then a Temporal Database Management System (TDBMS) will be useful for manipulating temporal data.

1.5 Solutions in developing a Temporal Database

The following approaches underline how a temporal database may be created:

1. Use the type *date* provided by a non-temporal (any commercial) DBMS.
2. Extend a non-temporal data model to a temporal data model by *attaching time attributes* to each data.
3. Develop a new temporal database system from scratch that provides a primitive data type time and handles the different states/time instances of data being stored.

The first and second solution does not involve any changes to existing database technology and may be simple to form as we just build new methods for temporal support on top of the existing database system that will be used.

The third solution involves developing a whole new database system with temporal support. This will be difficult as the underlying principles used by commercial DBMS to optimise operations must be reformed and a lot of theoretical work needs to be carried out to show that the new system is fully complete, all new and modified operations perform as required. The amount of time and manpower required for this approach is similar to that needed by commercial vendors to develop DBMS that we all are familiar with today.

Thus, this project cannot consider the third solution when developing a temporal database, as this is out of reach in terms of time and manpower available.

The project adopts the second solution, by using a relational database system to model and store temporal relations and hence, produces a temporal database.

1.6 Research and Available TDBMS

US logic and derived Modal operators - [1, 2, 3] gives an insight into new temporal operators that are specifically designed to manipulate temporal databases. These operators have been shown to be mathematically sound which makes them useful to extract temporal data. The publications discusses procedures to take in implementing these operators on relational databases systems using the logic associated with relational algebra.

Temporal and Modal Logic – [17, 18] describes the mechanics of temporal and modal logic. They discuss the different tenses (Past and Future) and give the underlying semantics of the logic.

Temporal Database – [4, 7, 8, 19] gives an insight on how to build a complete temporal database system and how to form TSQL, which is an extension of SQL to handle temporal components denoted in records. The publications also describe how to optimise queries and gives material covering the subject in broad.

TimeChain Technology – [15] provide a TDBMS that eases the difficult task of managing historical databases. The technology does not consider new temporal specific operators that are provided by US logic or the derived modal operators.

1.7 Objectives of the Project

The overall aim of the project is to develop a competent TDBMS that concentrates on manipulating and extracting temporal data.

The following points summarise the requirements of the project:

1. Implement a temporal database that can store temporal relations. Use current available commercial relational databases system to model the temporal database.
2. Revise and implement the relational operators that are common and supported by most commercial database systems based on relational algebra.
3. Introduce and implement new temporal operators based on US logic and including the derived modal operators.

The project does not attempt to develop a full and complete database system but implement features that allow manipulating and extracting temporal data. It should allow writing queries with ease and removing the burden of structuring length queries to extract meaningful data.

1.8 Report Structure

The report covers the research, design, implementation and evaluation of the project.

The following give an overview and a description of each chapter in the report:

Chapter 2 Background – gives the theoretical knowledge of creating temporal databases, US logic and derived modal operators each with tutorial style examples.

Chapter 3 Specification – provides an overview of the system requirements.

Chapter 4 Design – gives the core features of the each component created to form the complete system.

Chapter 5 Implementation – discusses in detail how, what and why system features were built. It provides algorithms of the useful operators implemented.

Chapter 6 Testing – describes how the whole system and the individual system components were tested.

Chapter 7 Evaluation – analyses the system's functionality and features. Gives the strength and weakness of each component and how they can be improved.

Chapter 8 Conclusion – summaries the project, achievements of the project and any extension that can be undertaken in the future.

Chapter 2 Background

This chapter aims to describe and illustrate with tutorial style examples the underlying principles that are related to creating and manipulating temporal databases.

2.1 Temporal Dimension

A temporal database [3,4] is a database system that provides special support for handling data with a temporal component, namely time. In a temporal database, objects can have different values at different times. Therefore at a specific time, we have a specific snapshot of the database. The important issue that needs to be addressed for creating a temporal database is the representation or modelling of time.

2.11 Time Views

Views of time can be considered to be either continuous or discrete time:

2.12 Continuous Time

A continuous time model [3] is considered to be similar to representing time with real numbers, it is always possible to define a new time point between two existing time points. This results in an infinite set of time points.

2.13 Discrete Time

A discrete time model [3] is considered to be similar to representing time with natural numbers (subset of real numbers), a set of equally spaced and ordered time points. Therefore, there is a concept of some atomic unit of time, known as a chronon.

2.14 Chronon

A chronon is the shortest duration of time [4] – it is a non-decomposable unit of time, cannot be further divided or broken to generate new time points. A chronon is used to build all units of discrete time.

2.15 Modelling Time

The discrete time model is used to interpret time because of the simplicity and relative ease of implementation and has also been widely adopted by many that have conducted research and implemented temporal databases [1,2,4]. If a continuous time model were used to represent time in a temporal database, there would be many problems in providing arithmetic support since there is an infinite precision of time.

2.2 Temporal Database Forms and Data Model

Another important issue relating to the representation of time is the role played by the time (what the time signifies) that is being supported by the temporal database system. There are two common perceptions of time, transaction time and valid time.

2.21 Transaction time

A database object is stored in a database at some point in time. *The transaction time of an object is the time when the object is stored in the database [3,4], the time that it is present in the database.* For example, in a banking system, the transaction time of a withdrawal would be from the time the clerk entered the payment of withdrawal into the database to the time that it was made invalid in the database. Another example would be, in a company situation, an employee receives a pay rise but it comes into effect when the payroll clerk enters this salary rise into the database. Transaction time values cannot be after the current time.

2.22 Valid time

The valid time of a database object is the time when the object is effective or holds (is true) in reality [3,4]. The time when the event occurred, took place in reality. For example, in a banking system, the payments and withdrawals made by a customer have a valid time associated with the time the customer performs the transaction at the bank. Another example would be, in a football competition, when did the clubs win the competition i.e. the times when Arsenal won the F.A Cup competition.

Objects in the temporal database system will have a time component associated to it; this will hold either the valid time or the transaction time.

2.23 Data Model for Temporal Data

The following concepts are established to determine a complete data model for temporal data. The concepts provide knowledge of the temporal database employed, and is used to implement a fully functioning temporal database management system with special temporal operators:

1. Each object or tuple in a database will have some time value associated with it, a timestamp. The time model employed is the discrete time model.
2. The time model will be bounded [3], is finite, which means that there is a first and a last time point and all timestamp values associated with tuples in the database fall within this range. A bounded model is chosen as oppose to an unbounded model since an unbounded model of time allows time to span over infinity. This will create implementation problems similar to those defined by using a continuous time model for the representation time [1,3].
3. A linear model of time will be used [3] which means that there is only one version of the data available for any time. A linear model is chosen as oppose to a

branching model of time that allows alternative versions of data to hold for any given time. Again the reason to avoid this branching model is that complications will arise when implementing a query language supported by the temporal database management system.

Thus, a discrete linear bounded data model approach is used for temporal data.

2.3 Temporal Structure

The temporal structure is the method of adding a temporal dimension to the relational model employed by most conventional database systems [1], the process of applying the discrete linear bounded data model. In the temporal structure we have a model with an ordered set of chronons where time, $t \in \{0, 1, 2, \dots, n\}$ that represents the different time points. Ordering and restricting the chronons t used for time points to be finite achieves a bounded linear model for the structure [3]. Each time point is used to index different snapshots of the database such that a historical database D_n (n is a natural number) is considered as a set of relational databases $\{D_t \mid 0 \leq t \leq n\}$ [1]. For shorthand, each instance of t is known as a tick.

Example 2.1: Temporal Structure

Consider the following *League* relation:

League (team, pos) - Each tuple in the relation consists of a football team name and the league position the team finished for the campaign at the specific tick.

The campaign for this example will be the football premiership title.

A tick represents a time point. For this example, let us consider 5 time points, each describe a new campaign. In modelled reality, each time point will signify the season of the campaign.

t	database	League
0	D ₀	(Arsenal, 2) (Blackburn, 19) (Chelsea, 3) (Man Utd, 1)
1	D ₁	(Arsenal, 2) (Chelsea, 5) (Man Utd, 1)
2	D ₂	(Arsenal, 2) (Chelsea, 6) (Ipswich Town, 5) (Man Utd, 1)
3	D ₃	(Arsenal, 1) (Blackburn, 10) (Chelsea, 6) (Ipswich Town, 18) (Man Utd, 3)
4	D ₄	(Arsenal, 2) (Blackburn, 6) (Chelsea, 4) (Man Utd, 1)

The table shows tuples that hold at specific ticks – results are taken from the actual football premiership competition for the years from season 1998-1999 to 2002-2003.

The range of the ticks 0 to 4 represents each season (in terms of years) 1998-1999 to 2002-2003.

The structure can be extended to hold a relation called “Time”, that stores a single attribute which represents or holds the mapping of the each tick to a more specific user defined time domain that corresponds to real world time [3]. For example, we could map the different natural numbers to mean a specific date (dd/mm/yyyy), the days of the week, particular years, century’s etc.

2.31 Other Temporal Structures

To alter the structure but still maintain the relations, we can use different notions [1] to store and maintain tuples, and the way they are linked to the tick values. For example the tuple (Arsenal, 2) from the relation League can be expressed as (Arsenal, 2) {0, 1, 2, 4} – listing the different ticks when the tuple is present in the historic database. Another form of expression can be as (Arsenal, 2) [0, 2] and (Arsenal, 2) [4, 4] – using an interval or time period to define the ticks for when the tuple holds in the historic database.

2.4 Temporal Normal Form

A particular and efficient method for creating temporal relational databases [1,2,3] is with the use of time intervals that are associated with the tuples. In short, it is proposed that tuples will have the following structure: $(a_1, \dots, a_n) [start, end]$ in a relation R where start and end are the temporal time attributes (referring to ticks) and define the time interval for when the tuple exists. This temporal structure is applied and stored in a relational database as $R(a_1, \dots, a_n, start, end)$, a set of attributes [3].

All intervals that are stored in the relational database are and must be maximal [2]. That is to say, for any tuple $R(a_1, \dots, a_n, start, end)$ there must be no other matching tuple $R(a_1, \dots, a_n, start', end')$ where the non-temporal attributes a_1, \dots, a_n are identical and the temporal attributes, the intervals overlap by confirming to $start \leq end' + 1$ and $end \geq start' - 1$ (that overlap or touch each other) [2]. Thus, adding the constraint that all such intervals are maximal gives the temporal normal form (TNF).

In order for an interval to be a valid interval in the bounded model, the fact $0 \leq start \leq end \leq n$ (n is the last tick t of the historic database D_n) must be obeyed.

The reasons for choosing the TNF (time interval associated with tuples):

1. Produces a compact database, making it efficient in storage, whereby overlapping intervals of two tuples are replaced by a single interval for a single tuple.
2. Without the normal form, the primary keys of relations would be seen to have duplicates at some time. For example, if we have a relation R with a single attribute that is also its primary key and have stored in the database $R = \{ (a) [2, 5], (a) [5, 10] \}$, then at tick 5, the primary key x is seen to be duplicated.
3. Implementation of the query language would be complicated. For example, if we want to find if $R(x)$ held for the interval [4,8], then in the TNF database we need only check once that $R(a)$ is stored in the database as holding for some interval that encloses [4, 8]. However, for a non-TNF database, we must merge all overlapping tuples to see if together they enclose [4, 8].

Example 2.2: TNF encoding of the League relation, Example 2.1

New relation formed **League (team, pos, st, et)** where st and et are time points and describe a time interval for which the data is valid.

League
(Arsenal, 2) [0, 2]
(Arsenal, 1) [3, 3]
(Arsenal, 2) [4, 4]
(Blackburn, 19) [0, 0]
(Blackburn, 10) [3, 3]
(Blackburn, 6) [4, 4]
(Chelsea, 3) [0, 0]
(Chelsea, 5) [1, 1]
(Chelsea, 6) [2, 3]
(Chelsea, 4) [4, 4]
(Ipswich Town, 5) [2, 2]
(Ipswich Town, 18) [3, 3]
(Man Utd, 1) [0, 2]
(Man Utd, 3) [3, 3]
(Man Utd, 1) [4, 4]

Notice how the new table represents data with time points. Each tuple has associated with it a time period, which signifies the time points for which the information is valid. E.g. the tuple *(Arsenal, 2) [0, 2]* means that Arsenal finished in position 2 between the time points 0 to 2 (seasons 1998-1999 to 2000-2001) inclusively.

The new table shows how TNF removes repeated data present at different time points by grouping them. Also reduces the number of tuples that need to be stored, forms a concise table.

2.5 Temporal Query Language

2.51 Relational Algebra - RA

Conventional relational databases have basic constructs of the SQL query language [5,6] that can be translated to relational algebra, which has strong semantics. Thus, the SQL language is well understood and consistent. Relational Algebra (RA) is the writing of logical expressions using set of relational operators [5] that perform operations over relations and returns results as relations.

2.52 Temporal Relational Algebra – TRA

TRA is the extension to RA. TRA is a set of operators that includes all the RA operators and with the addition of new temporal specific operators. The evaluation of a query in TRA [1] using the information of just a snapshot of the historic database (a particular tick) would mean that the classical relational operators¹ have their normal meaning, are consistent.

¹ Common relational operators are Union, Intersect, Difference and Product

2.53 New Temporal Operators

To project data through manipulation and linking of tuples from other databases $D_{t'}$ with the tuples from the present database D_t (where $t' \neq t$), we can introduce new temporal operators [2,3,4]. There is the introduction of two new temporal operators that can only be applied to historical temporal databases, which are known as *Until* and *Since* [2]. These operators and their logical interpretation, US logic (Until and Since logic) are first order complete for a discrete historic database, they are a simple extension of classical logic. Research into temporal logic and their properties has resulted in the creation of US logic that has been shown to be mathematically sound [1,2]. They have proven semantics, which makes them popular temporal operators to be used for querying temporal relational databases.

2.54 US logic

Operator	Semantics
A Until B	A must hold at all times until the time B holds. A holds for all future snapshots up to and including the snapshot when B holds. At time n , the result is false.
A Since B	A must have held at all times since B held. A holds for all past snapshots back to and including the snapshot when B held. At time 0, the result is false.

The Until and Since operators serve to query the ‘future’ and ‘past’ snapshots respectively, without the need of forming long and complicated queries involving the classical RA operators.

2.6 Auxiliary Functions

To manipulate and evaluate queries for a temporal relational database in TNF, we define auxiliary functions that evaluate the temporal components, the time interval.

2.61 Overlap

There are two types of overlap functions. Both overlap functions check if time intervals of tuples are overlapping but differ where on the fact that one of the overlap functions checks for touching time intervals while the other does not.

Touching Overlap Function: determines whether pair of intervals overlap or touch each other.

```
function overlap1 ([start, end], [start' end'])
    if start ≤ end'+1 and end ≥ start' -1
        then return true
    else
        return false
```

The above overlap function is useful for transforming temporal relations into TNF.

Non-touching Overlap Function: determines if a pair of intervals overlap (not interested if they touch).

```
function overlap2 ([start, end], [start' end'])
    if start ≤ end' and end ≥ start'
        then return true
    else
        return false
```

The above overlap function is used by temporal operators for considering compatibility of tuples for performing the temporal relational operation on them. A temporal operation on two tuples can only be conducted if they both hold at some common tick(s), present at the same snapshot(s) of the database –share time values.

2.62 Min

Function Min: determines the minimum value of a pair of time values

```
function min(a, b)
    if a ≤ b
        then a
    else
        return b
```

2.63 Max

Function Max: determines the maximum value of a pair of time values

```
function max(a, b)
    if a ≥ b
        then a
    else
        return b
```

2.7 Querying Temporal Database

In order to query a temporal database, the introduction of the temporal relational algebra (TRA) is considered. Defining new temporal operators such as Until and Since or using current, existing operators of the relational algebra (RA), need to be either defined or revised to manipulate temporal data (especially taking care of the time intervals) and extract correct information from a temporal database.

When querying a relation R in the temporal database to find tuples that holds for time q, we need only ask the relational database (to which the temporal structure has been

applied) the relational query $\sigma_{\text{start} \leq q \leq \text{end}} R(a_1, \dots, a_n, \text{start}, \text{end})$ where σ is the select operator.

The following section shows how TRA operators can be implemented in terms of RA operators to work on temporal databases.

2.8 Implementing TRA in RA

TRA operators are implemented to work on a temporal database that is in TNF [1,2]. Relations stored in a database will be in TNF, so we expect all the TRA operators not to violate but confide to TNF when returning any output relations. If this was not the case and the output of any temporal query or operator is feed as an input to another TRA operator, then unexpected and incorrect results will be calculated.

Publication [5] discusses the features and properties of RA. The publication [1] discusses the features and properties of TRA: describes how to extend RA to produce TRA. These publications are referenced when describing the temporal operators in the latter part of the background sections.

2.801 Example Relations

The following relations will be used and referred for all further examples (temporal operator) illustrated in this section.

League - the same relation described in the ‘TNF’ section. We have omitted the tuples regarding teams Chelsea and Man Utd to form a smaller relation for our examples to be illustrated later.

team	pos	st	et
Arsenal	2	4	4
Arsenal	1	3	3
Arsenal	2	0	2
Blackburn	6	4	4
Blackburn	10	3	3
Blackburn	19	0	0
Ipswich Town	18	3	3
Ipswich Town	5	2	2

(8 rows)

FACup (team, st, et) - Each tuple in the relation consists of a team name and two time points, st and et. The relation describes the winners of the *actual* F.A Cup competition.

team	st	et
Arsenal	3	4
Liverpool	2	2
Chelsea	1	1
Man Utd	0	0

(4 rows)

Time (time, year) - Each tuple in the relation consists of a time point and a year. The relation describes the representation used for each time point e.g. time point 2 means the season 2000-2001.

time	year
0	1998-1999
1	1999-2000
2	2000-2001
3	2001-2002
4	2002-2003

(5 rows)

2.802 Project π_t

$$\pi_{t A, B, C} R = \text{coalesce}(\pi_{A, B, C, \text{start}, \text{end}} R)$$

Query Example 1 - Project *team* from *League*

Result of Query

team	
Arsenal	[4, 4]
Arsenal	[3, 3]
Arsenal	[0, 2]
Blackburn	[4, 4]
Blackburn	[3, 3]
Blackburn	[0, 0]
Ipswich Town	[3, 3]
Ipswich Town	[2, 2]

The result performs a simple project operation where we retrieve the team names from the relation *League*.

Note. If the result of this query was used as an input to another query then this would violate the TNF definition - All time intervals that are stored in a relational database are and must be maximal.

Therefore, we need a function to transform output into TNF form. This function is known as Coalesce.

2.803 Coalesce Function

To ensure that the evaluation and results of a query on the temporal database does not violate the TNF [2], a special function known as Coalesce is applied to the results of a temporal query. The aim of the function is to arrange temporal data to abide by the TNF, remove any violations of the TNF.

Query Example 2 - Coalesce (Project *team* from *League*)

Result of Query

Team	
Arsenal	[0, 4]
Blackburn	[0, 0]
Blackburn	[3, 4]
Ipswich Town	[2, 3]

The result coalesces the output of the sub-query in the “brackets”. This query resolves the problem mentioned in Query1 – violation of TNF

Therefore, the objective of the coalesce function is to sort the results of a query on a relation based on its attributes and merge all tuples that have the same non-temporal attributes and overlapping intervals.

The following pseudo algorithm defines the coalesce function

```

function coalesce(R)

  R := R order by a1,...,an,start
  Rc = empty
  Size = count(R)
  i = 1

  repeat
    Ri = (a1,...,an) [start, end]
    while Ri+1 = (a'1,...,a'n) [start', end'] and a1=a'1,..., an=a'n and start'≤end+1
      end := max(end, end')
      i = i+1
    end_while
    insert (a1,...,an) [start, end] into Rc
    i := i+1
  while i ≤ size

  return Rc

```

2.804 Select σ_t

$$\sigma_{t A, \dots} R = \sigma_{t A, \dots} R$$

Query Example 3 - *League* Where $pos = 1$ and $team = 'Arsenal'$

Result of Query

team	pos
Arsenal	1 [3, 3]

The result retrieves the tuples where team name is Arsenal and position is 1. A query like this would be useful to determine when a club has won the league competition. The result for the above query shows that Arsenal won the competition for the season 2001-2002 represented by the time point 3.

Query4 - *League* Where $pos \leq 18$

Result of Query

team	pos
Blackburn	19 [0, 0]
Ipswich Town	18 [3, 3]

The result retrieves the tuples for when and what clubs have finished below the position 17. It is known that clubs, which finish below position 17, they are eliminated from the premier league competition and a query like this would be useful to determine when a club has been relegated. The result for the above query shows that Blackburn were relegated for the season 1998-1999 represented by the time point 0.

2.805 Product \times_t

$$R \times_t S = \pi_{R, S, \max(R.start, S.start), \min(R.end, S.end)} \sigma_{\text{overlap}(R,S)} (R \times S)$$

Query Example 5 - Product (*League*, *FACup*) - This result illustrates what club won the F.A at the time when data is valid in the League relation.

Result of Query

team	pos	team	
Arsenal	1	Arsenal	[3, 3]
Arsenal	2	Arsenal	[4, 4]
Arsenal	2	Chelsea	[1, 1]
Arsenal	2	Liverpool	[2, 2]
Arsenal	2	Man Utd	[0, 0]
Blackburn	6	Arsenal	[4, 4]
Blackburn	10	Arsenal	[3, 3]
Blackburn	19	Man Utd	[0, 0]
Ipswich Town	5	Liverpool	[2, 2]
Ipswich Town	18	Arsenal	[3, 3]

The result produces a temporal product of the two tables. Note that a temporal product is different from the non-temporal product that would bind each tuple from the League relation with every tuple from the FACup relation. A non-temporal product would therefore produce 32 tuples (8 * 4 – tuples from League and FACup respectively). For a temporal product, a tuple from the League table is appended to a tuple from the FACup table *if their respective time periods overlap*.

2.806 Since Product S_{\times}

$$R \ S_{\times} \ S = \text{coalesce} (\pi_{R,S, \max(R.\text{start}, S.\text{start})+1, R.\text{end}+1} \ \sigma_{\text{overlap}(R,S)} (R \times S))$$

Query Example 6 - Since Product (*League, FACup*)

Result of Query

team	pos	team	
Arsenal	1	Arsenal	[4, 4]
Arsenal	2	Chelsea	[2, 3]
Arsenal	2	Liverpool	[3, 3]
Arsenal	2	Man Utd	[1, 3]
Blackburn	10	Arsenal	[4, 4]
Blackburn	19	Man Utd	[1, 1]
Ipswich Town	5	Liverpool	[3, 3]
Ipswich Town	18	Arsenal	[4, 4]

2.807 Until Product U_{\times}

$$R \ U_{\times} \ S = \text{coalesce} (\pi_{R,S, R.\text{end}-1, \min(R.\text{end}, S.\text{end})-1} \ \sigma_{\text{overlap}(R,S)} (R \times S))$$

Query Example 7 - Until Product (*League, FACup*)

Result of Query

Team	pos	team	
Arsenal	1	Arsenal	[2, 2]
Arsenal	2	Arsenal	[3, 3]
Arsenal	2	Chelsea	[0, 0]
Arsenal	2	Liverpool	[0, 1]
Blackburn	6	Arsenal	[3, 3]
Blackburn	10	Arsenal	[2, 2]
Ipswich Town	5	Liverpool	[1, 1]
Ipswich Town	18	Arsenal	[2, 2]

2.808 Since and Until Product Explanation

The since and until product results produce a temporal since and until product of the two tables respectively. Note that these two new operators are a temporal related operators and can only be applied to temporal databases. Both since and until product are similar to the temporal product except that when integrating time periods of each tuple being used by the since and until product, different time intervals are produced to those found when performing a temporal product.

2.809 Join \bowtie_t

$$R \bowtie_t S = \pi_{R, S, \max(R.start, S.start), \min(R.end, S.end)} \\ \text{AND } R.common\text{-}attributes=S.common\text{-}attributes \sigma_{\text{overlap}(R,S)} (R \times S)$$

Query Example 8 - Join (*League*, *FACup*)

This result illustrates when did a club win the F.A cup and what position did they finish in the league. The above table show that Arsenal have won the F.A cup twice and finished first in the year 2002 (time point 3) and second in the year 2003 (time point 4)

Result of Query

team	pos
Arsenal	1 [3, 3]
Arsenal	2 [4, 4]

The result produces a natural temporal join of the two tables. Note that a temporal join is different from the non-temporal join that would bind only the tuples from the League relation with tuples from the FACup relation where joining on the common attributes (column names and types) shared by the two relations (in this query it is the attribute team) This is used to check if two tuples can be joined. A non-temporal join would therefore produce 3 tuples (as the team Arsenal exists in both relations – 3 tuples from League and 1 tuple from FACup respectively). For a temporal join, a tuple from the League table is joined to a tuple from the FACup table *if their respective time periods overlap* as well as obeying the existing conditions to perform a non-temporal join.

2.810 Since Join \bowtie_s

$$R \bowtie_s S = \text{coalesce} (\pi_{R,S,\max(R.start, S.start)+1, R.end+1} \\ \text{AND } R.common\text{-}attributes=S.common\text{-}attributes \sigma_{\text{overlap}(R,S)} (R \times S))$$

Query Example 9 - Since Join (*League*, *FACup*)

Result of Query

team	pos
Arsenal	1 [4, 4]

2.811 Until Join \bowtie

$$R \bowtie S = \text{coalesce} (\pi_{R,S, R.\text{end}-1, \min(R.\text{end}, S.\text{end})-1} \\ \text{AND } R.\text{common-attributes}=S.\text{common-attributes} \sigma_{\text{overlap}(R,S)} (R \times S))$$

Query Example 10 - Until Join (*League, FACup*)

Result of Query

team	pos
Arsenal	1 [2, 2]
Arsenal	2 [3, 3]

2.812 Since and Until Join Explanation

The since and until join results produce a temporal since and until join of the two tables respectively. Note that these two new operators are temporal related operators and can only be applied to temporal databases. Both since and until join are similar to the temporal join except that when integrating time periods of each tuple being used by the since and until join, different time intervals are produced to those found when performing a temporal join.

2.813 Difference \neg_t

$$R \neg_t S = \text{difference} (R, S)$$

There is no relational algebra to define how to perform a temporal difference of two relations.

Query Example 11 -

Difference (Coalesce Project *team* from *League* Where *team* = *Arsenal*,
Coalesce Project *team* from *League* Where *pos* = 1)

SubQuery A = Coalesce Project *team* from *League* Where *team* = *Arsenal*,

team
Arsenal [0, 4]

SubQuery B = Coalesce Project *team* from *League* Where *pos* = 1

team
Arsenal [3, 3]

Difference (SubQuery A , SubQuery B) - This result illustrates when did Arsenal not win the premier league competition.

Result of Query

team
Arsenal [4, 4]
Arsenal [0, 2]

The result performs the operation $a \text{ minus } b$ for tuples a from SubQuery A and tuples b from SubQuery B. Note that particular care must be taken of resulting tuples as new time intervals are created. Unlike a non-temporal difference, which just removes data from one relation that exists in the relation, which you are subtracting away, the temporal difference needs to subtract the data for the time periods that match in both relations. The data from SubQuery A exists for time interval 0 to 4 while the data in SubQuery B exists at only time period 3. Therefore, the result must remove the data existing at time period 3, which in effect forms two new time intervals with the data. This implies that in order to remove the time period 3 from the time interval 0 to 4, we create the time intervals 0 to 2 and 4 to 4, which ensures that the time point 3 is not in either time interval, as defined by and being the purpose of the difference operator.

2.814 Union \cup_t

$$R \cup_t S = \text{coalesce} (R \cup S)$$

Query Example 12 -

Union (Coalesce Project *team* from *League* Where $pos = 1$, *FACup*)

SubQuery A = Coalesce Project *team* from *League* Where $pos = 1$

team
Arsenal [3, 3]

Union (SubQuery A, *FACup*)

team
Arsenal [3, 4]
Chelsea [1, 1]
Liverpool [2, 2]
Man Utd [0, 0]
Arsenal [3, 3]

Coalesce (Union (SubQuery A, *FACup*)) - This result illustrates when did a club win either the premier league or the F.A cup competition.

Result of Query

team	
Arsenal	[3, 4]
Chelsea	[1, 1]
Liverpool	[2, 2]
Man Utd	[0, 0]

The result performs the temporal operation union of SubQuery A with the relation FACup. A temporal union behaves in the same manner as a non-temporal union. However, the resulting output must be coalesced else it fails to obey the TNF. For the above query, performing a union without coalescing the results, means that we have both the tuple Arsenal [3, 4] and the tuple Arsenal [3, 3] in the output to the query. The time interval for the former tuple ([3, 4]) covers the time interval of the latter tuple ([3, 3]) and this implies that repeated data is present and hence violating the TNF. Thus, a temporal union is equivalent to a non-temporal union with all results being coalesced.

2.9 Derived Modal Operators

Additional operators can be derived from the Until and Since operators (US Logic) [2] which are known as modal operators.

2.91 Past



$I \times R$ where I is the identity relation

Derived from: Since.

Semantics: R held at sometime in the past.

Description: Tuples of R from *some snapshot before* the current snapshot.

The Past operator is useful for querying a temporal database when we are interested in finding all tuples that existed in the past, where past is all the ticks or time points before the current time point.

Query Example 13 - The Past modal operator

Past (*League*)

Result of Query

team	pos
Arsenal	1 [4, 4]
Arsenal	2 [1, 4]
Blackburn	10 [4, 4]
Blackburn	19 [1, 4]
Ipswich Town	18 [4, 4]
Ipswich Town	5 [3, 4]

The result illustrates where have all the clubs finished in the *past* and when is the data true in the *past*.

Past (*FACup*)

Result of Query

team
Arsenal [4, 4]
Liverpool [3, 4]
Chelsea [2, 4]
Man Utd [1, 4]

The result illustrates which clubs won the F.A cup competition in the *past* and when in the *past*.

2.92 Previous

$R \text{ S}_\times I$ where I is the identity relation

Derived from: Since.

Semantics: R held at the previous moment.

Description: Tuples of R from *the snapshot before* the current snapshot.

The Previous operator is useful for querying a temporal database when we are interested in finding tuples that existed at the previous time point for all time points. Tuples previous to tick _{i} are tuples that exist at tick _{$i-1$} where i : all ticks (the range of ticks) of the temporal database.

Query Example 14 - The Previous modal operator

Previous (*League*)

Result of Query

team	pos
Arsenal	1 [4, 4]
Arsenal	2 [1, 3]
Blackburn	10 [4, 4]
Blackburn	19 [1, 1]
Ipswich Town	18 [4, 4]
Ipswich Town	5 [3, 3]

The result illustrates for all the clubs in the relation *League*, where did they finish in the *previous* time point for all time points.

The tuple Arsenal 2 [1, 3] implies that Arsenal finished in the second position for the time points *previous* to the interval [1,3] that are the time points 0, 1 and 2 (seasons 1998-1999, 1999-2000 and 2000-2001 respectively) or time interval [0, 2].

Previous (*FACup*)

Result of Query

team	
Arsenal	[4, 4]
Liverpool	[3, 3]
Chelsea	[2, 2]
Man Utd	[1, 1]

The result illustrates for all time points, which club won the F.A cup in the *previous* time point.

2.93 Always in Past

$$\pi_R (R \bowtie \sigma_{\text{time}=0} \text{ time}) \quad \text{where 0 is the first tick}$$

Derived from: Since.

Semantics: R held at all times in the past.

Description: Tuples of R present in *all snapshots before* the current snapshot

The Always in the Past operator is useful for querying a temporal database when we are interested in finding tuples that existed at all the time points from the current time point. Tuples that held at all tick_i where $i: 0 \leq i \leq n$ ticks of the temporal database (n is last tick). At the first time point the result is false.

Query Example 15 - The AlwaysPast modal operator

AlwaysPast (*League*)

Result of Query

team	pos
Arsenal	2 [1, 3]
Blackburn	19 [1, 1]

The result illustrates for all times points, which data in the relation League is *always true in the past*. This modal operator implies what data holds from the minimum time point to any time point

For our example, our minimum time point is 0, season 1998-1999 and thus the above query returns all data that holds from 0 to any other time point. i.e. Arsenal have *always in the past* finished second from the time point 3.

AlwaysPast (*FACup*)

Result of Query

team
Man Utd [1, 1]

The result illustrates for all times points, which data in the relation FACup is *always true in the past*. Only Man Utd has always won the F.A cup in the past from the time point 1 back to the start time point (minimum time point).

2.94 Future



$I \cup_{\times} R$ where I is the identity relation

Derived from: Until.

Semantics: R holds at sometime in the future.

Description: Tuples of R from *some snapshot after* the current snapshot.

The Future operator is useful for querying a temporal database when we are interested in finding all tuples that existed in the future, where future is all the ticks or time points after the first time point.

Query Example 16 - The Future modal operator

Future (*League*)

Result of Query

team	pos	
Arsenal	2	[0, 3]
Arsenal	1	[0, 2]
Arsenal	2	[0, 1]
Blackburn	6	[0, 3]
Blackburn	10	[0, 2]
Ipswich Town	18	[0, 2]
Ipswich Town	5	[0, 1]

The result illustrates where have all the clubs finished in the *future* and when is the data true in the *future*.

Future (*FACup*)

Result of Query

team	
Arsenal	[0, 3]
Liverpool	[0, 1]
Chelsea	[0, 0]

The result illustrates which clubs won the F.A cup competition in the *future* and when in the *future*.

2.95 Next ○

$R \cup_{\times} I$ where I is the identity relation

Derived from: Until.

Semantics: R holds at the next moment.

Description: Tuples of R from *the snapshot after* the current snapshot.

The Next operator is useful for querying a temporal database when we are interested in finding tuples that existed at the next time point for all time points. Tuples at the next tick to $tick_i$ are tuples that exist at $tick_{i+1}$ where i : all ticks (the range of ticks) of the temporal database.

Query Example 17 - The Next modal operator

Next (*League*)

Result of Query

team	pos	
Arsenal	2	[3, 3]
Arsenal	1	[2, 2]
Arsenal	2	[0, 1]
Blackburn	6	[3, 3]
Blackburn	10	[2, 2]
Ipswich Town	18	[2, 2]
Ipswich Town	5	[1, 1]

The result illustrates for all the clubs in the relation *League*, where did they finish in the *next* time point for all time points.

The tuple Arsenal 1 [2, 2] implies that Arsenal finished in the first position for the time points *next or after* the interval [2,2] which are the time points [3, 3] (season 2001-2002).

Next (*FACup*)

Result of Query

team	
Arsenal	[2, 3]
Liverpool	[1, 1]
Chelsea	[0, 0]

The result illustrates for all the time points, which club won the F.A cup in the *next* time point.

2.96 Always in Future □

$$\pi_R (R \cup_{\times} \sigma_{\text{time}=n} \text{time}) \quad \text{where } n \text{ is the last tick}$$

Derived from: Until.

Semantics: R holds at all times in the future.

Description: Tuples of R present in all *snapshots after* the current snapshot

The Always in the Future operator is useful for querying a temporal database when we are interested in finding tuples that existed at all the time points after the first time point. Tuples that held at all tick_i where $i: 0 \leq i \leq n$ of the temporal database (0 is first tick). At the last time point the result is false.

Query Example 18 - The AlwaysFuture modal operator

AlwaysFuture (*League*)

Result of Query

team	pos
Arsenal	2 [3, 3]
Blackburn	6 [3, 3]

The result illustrates for all times points, which data in the relation League is *always true in the future*. This modal operator implies what data holds from any time point to the maximum time point.

For our example, our maximum time point is 4, season 2002-2003 and thus the above query returns all data that holds from any time point to the maximum time point 4. i.e. Arsenal have *always in the future* finished second from the time point 3.

AlwaysFuture (*FACup*)

Result of Query

team	pos
Arsenal	[2, 3]

The result illustrates for all times points, which data in the relation FACup is *always true in the future*. Only Arsenal has always won the F.A cup in the future from the time point 2 until the end time point (maximum time point).

Chapter 3 Specification

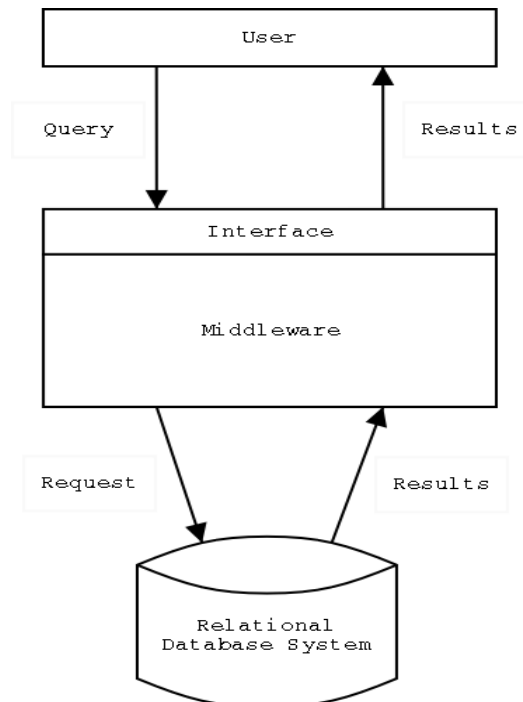
This chapter aims to give an overview of the system to be developed and the requirements of temporal management system (middleware).

3.1 Introduction

The overall aim of the project is to create a Temporal Database Management System (TDBMS). The objective of the TDBMS is to provide efficient storage of temporal data and support temporal querying over the information. The temporal data and relations stored by the database will be according to a linear discrete bounded data model and will reside on a conventional relational database. The temporal database managed by the TDBMS will be in historical form. The data stored in the temporal database will be with respect to valid time. The queries supported by the TDBMS will consist of Temporal Relation Algebra (TRA)² that includes the new temporal logic operators (US logic).

3.2 System Architecture

Diagram 3.1: An overview of the TDBMS



² TRA is the extension of Relation Algebra (RA) that is common to and supported by standard database systems. TRA includes new operators (being Until, Since and the modal operators based on the US logic) that are significant to manipulated temporal data. Conventional DBMS support queries in the form of SQL, which are translated by the DMBS into RA for evaluation.

The roles of the individual components:

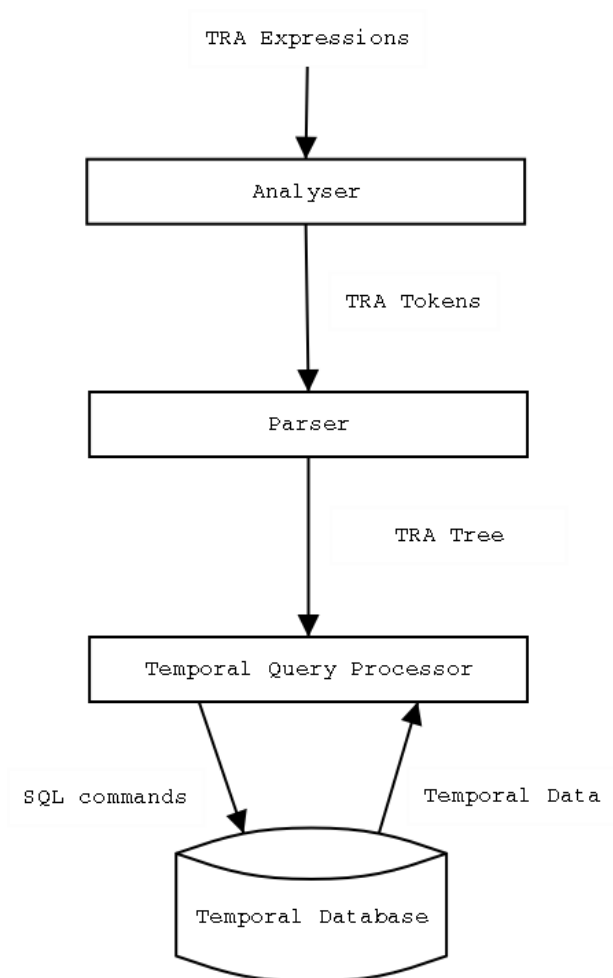
- User – interacts with the TDBMS.
- Interface – resides on the top of the Middleware. A friendly graphical user interface that allows the user to make requests to the TDBMS and displays the results. The interface should be simple to use and easy to learn. The user interface is the only way the user can communicate with the temporal database system.
- Middleware – the core engine that contains the functionality of the system, which processes the request from the user via the interface and retrieves information from the underlying conventional relational database.
- Relation Database System – stores temporal data and is configured to behave as a temporal database.

3.3 Middleware Architecture

The middleware provides the power to process user-entered requests and extract data for evaluation from the underlying temporal database.

The middleware must be created such that it does not depend on the type of physical database that being is used to store data and slight or no alterations would be required to the middleware if an alternative physical database is used.

Diagram 3.2: Modules of the Middleware



Chapter 4 Design

This chapter aims to give the core features of each component created to form the complete system.

4.1 Analyser

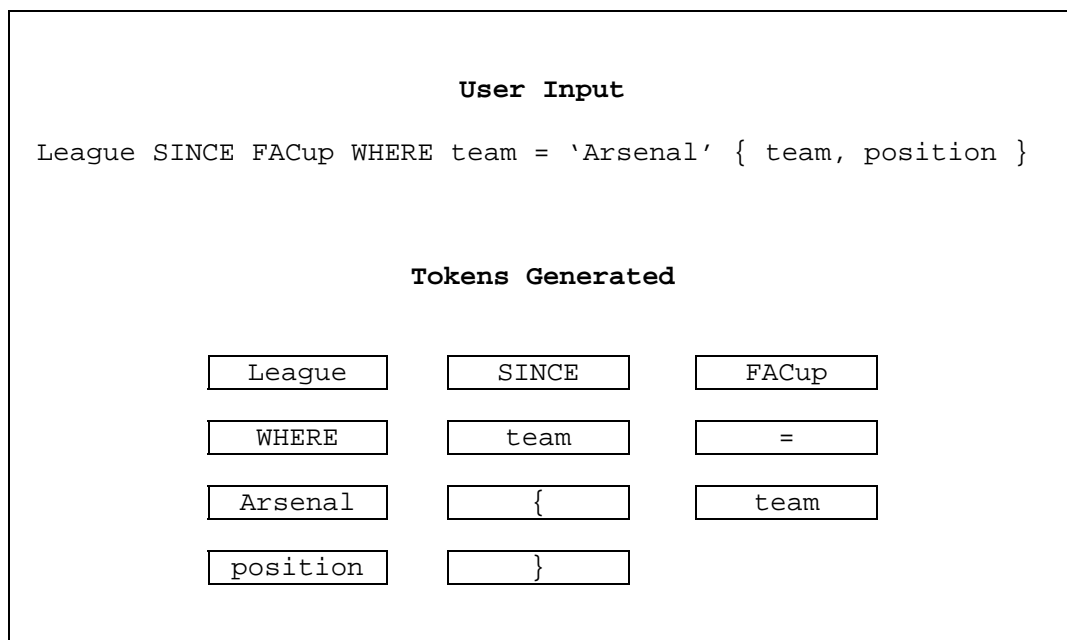
Input: TRA query as a single argument

Output: Lexical Tokens

Analyser designed with following properties:

1. Breaks up the TRA user input into individual tokens for verification and evaluation: checking against the grammar defined.
2. Tokens are defined for all operators or constants that are used in form TRA expressions, e.g. UNION, PRODUCT, JOIN, SINCE, UNTIL, PAST, FUTURE etc.
3. Tokens are defined for arguments or variables given in TRA expressions, e.g. relation names, attribute names, integers, real numbers, string values etc.
4. If a token cannot be found for a particular input then the system returns a user friendly error message.

Example 4.1: Generating tokens form a user input



4.2 Parser - TRA Tree Generator.

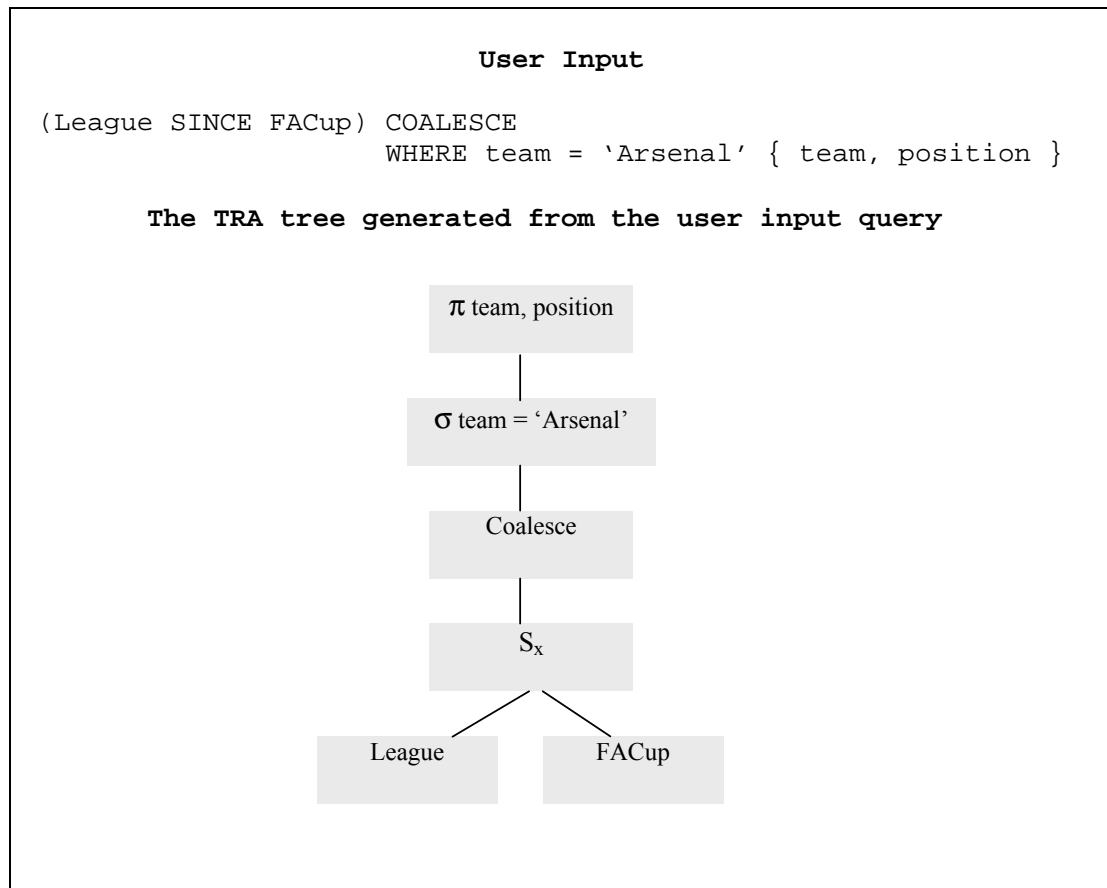
Input = Tokens

Output = TRA Tree

Parser designed with following properties:

1. Generates tree nodes from the tokens and links them to construct a TRA tree
2. Using the TRA language, the parser validates the tokens (the user-entered TRA query)
3. Produce the correct TRA tree from the given input to the parser, in principle, mapping the user-entered query to a tree structure.
4. The process of converting the user input into a TRA tree for evaluation is automatic and consistent.
5. The TRA tree generated is minimal and efficient meaning that the size and the length of the tree is small as possible. If it is possible that two or more operations can be combined for evaluation and be performed in one step, then the smaller solution is adopted. For example, if projecting a set of attributes from a relation, then it would be more efficient to project them all in one step than projecting each individual attribute in turn.

Example 4.2: Generating a TRA tree from user input.



4.3 Temporal Query Processor - TQP

Input: TRA Tree

Output: Results of executing the tree.

The main task of the TQP is to execute TRA trees by interpreting them. The TQP implements the functionality of the operators, which are used when executing the tree.

TQP designed with following properties:

1. Calls predefined and implemented methods to process each node.
2. After a node has been evaluated, its results are returned back to its adjacent, predecessor node.
3. Each parent node first evaluates its child nodes and use the results returned by its children when evaluating itself.
4. Having completed the interpretation of all nodes, the result of executing the TRA tree is outputted to the user via the user interface.
5. Executing and processing TRA trees is automatic.
6. Any nodes that cannot be evaluated due to the fact that the user has submitted invalid arguments to the query then the system notifies the user with a friendly error message.
7. Interacts with the underlying database system by performing SQL operations to retrieve temporal data.
8. Defines data structures to store the retrieved temporal data.
9. Executing and processing TRA trees is efficient: reduced and minimised communication to the temporal database.
10. Implements all of the following temporal operators listed:

Operator	Symbol	Operator	Symbol
Project	π	Since-Join	$S \bowtie$
Select	σ	Until-Join	$U \bowtie$
Union	\cup	Previous	\bullet
Difference	$-$	Past	\blacklozenge
Product	\times	Always-In-Past	\blacksquare
Since-Product	S_{\times}	Next	\circ
Until-Product	U_{\times}	Future	\diamond
Join ³	\bowtie	Always-In-Future	\square

11. Evaluation of TRA trees is consistent. Parsing the same TRA tree on different occasions does not generate different results.
12. The TQP is flexible, can handle a variety of temporal operations and different types of temporal queries such as nested queries.

³ The system must cater for both natural and non-natural join operations (also applies to since and until join operators). For the latter, the user must specify what attributes to compare and join data on.

4.4 User Interface

Input: TRA expressions

Output: Temporal Data – results of TRA queries

Interface designed with following properties:

1. Accepts TRA expressions to perform queries and other commands to configure the system to the user's requirements.
2. Provides a graphical user interface with buttons to make it easier for the user to learn and use.
3. Provides a user help option that describes how to use the system.
4. Returns the results of a query in a tabular format.

4.5 Temporal Database

Input: SQL statements

Output: Temporal Relations

The underlying physical database that stores the temporal data for manipulation is a relational database system. The physical database has been configured to store temporal relations and hence, create a temporal database. The temporal database is independent of the TQP and the technologies used to develop the TQP. Its sole purpose is to provide the TQP access to temporal relations it stores.

Chapter 5 Implementation

5.1 System Technology

After considering the arguments for and against the potential programming languages that can be used to build the system given in **APPENDIX A**, we have decided to use **Java** technology. The positive features of Java overshadow the features offered by the other languages and the author has some experience of programming in Java, which has also helped to arrive at this conclusion.

The reasons for selecting Java to develop our system:

- Object oriented features such as inheritance means that it will be easier and possible to reuse code. Certain operations that need to be offered by the system (such as product, since-product and until-product), it would be easier to define abstract classes making it simple for the programmer to design and maintain the system in the future.
- Java is robust and will allow our system to be more reliable. Java has error handling features to allow programmers to catch errors at earlier stages of development and the Java compiler is able to detect many problems that would first show up during run time in other languages. This would ensure that we produce a more complete system and are catching most errors.
- Java is portable which means that the system will run on most platforms without the need to recompile the code. This provides a greater depth to the usability of our system.
- Java is dynamic meaning that it is designed to adapt to an evolving environment. New methods and properties can be added freely in a class without affecting their clients. This will aid the programmer in producing a large system by allowing to code and test modules individually. This will help to reduce the time taken to integrate functionality to the system and provide the programmer with more valuable time to implement additional system features.
- Provides an extensive class library or APIs. This would be useful in defining data structures for our system, manipulating and parsing trees with ease. We can also use some predefined utilities when coding operations into the system.
- Java's JDBC drivers provide database interfaces to establish connection to the underlying database, which will prove vital for our system.
- The swing classes offered by Java will be beneficial in implementing a graphical user interface required for the system.

5.2 Development Approach

The development approach adopted to construct the system was an evolutionary and an incremental method. Initially, separate modules⁴ were first designed, implemented and then tested before integrating them together in order to achieve the complete desired system. When implementing a module, the design of the module was first performed offline before attempting to code it. Designs were modified as necessary and modules were re-implemented until a solid core and flexible structure was produce. The reason for this style of development was that at latter stages, it would be easier to add new functionality (accepting and processing additional query operations) without having to re-implement the system structure. If this style of development had not been adopted, there would have been a greater set back in implementation and the progress of the system.

5.3 Implementation Approach

A back to front end implementation approach was undertaken, building the modules in a reverse order; first setting up the temporal database, then beginning to implement the popular temporal relational operations (i.e. product, since, until, union, difference etc) and so on where at the end, finally developing the graphical user interface. This approach was adopted as it allowed the author to tackle the difficult tasks at the earlier stages of development, whereby providing more time in solving harder problems and leaving simple tasks that take less time to be implemented until the end. The main requirements of the project are to have more functionality as possible. Therefore, this approach of implementing temporal relational operators first, would be more ideal.

APPENDIX B - System Development Outline, describes how the system was implemented.

⁴ When referring to modules we mean the different components of the system: the temporal database, the analyser, the parser, the temporal query processor and the user interface.

5.4 Temporal Database Development

Postgres database system has been chosen on which to implement the temporal database. The main reason for selecting Postgres was the experience the author has with the database. The author has completed many projects in Postgres before creating this system and therefore, needs not to spend time learning the underlying database system that will be used for the project.

A temporal database was created on the Postgres relational database system. Temporal relations were implemented by attaching each tuple with a time interval, a start and end time value indicating when the data is valid. Information on how to form temporal data and relations is given in the background section.

5.5 TRA Grammar Development

The TRA grammar was designed by extending the RA [5] and implemented into the system. Thus, TRA queries are formed using this grammar defined. The TRA grammar was coded into the system such that the system's analyser and parser can use it to help break input into tokens, validate input and construct the TRA tree of the input.

APPENDIX C gives the main section of the TRA grammar that is implemented into the system.

The TRA grammar has been implemented with the following common characteristics:

- Unary operators are temporal relational operators that execute over 1 relation, e.g. Project, Past, Previous, Future, Next etc.
- Binary operators are temporal relational operators that execute over 2 relations, e.g. Product, Since, Until, Join, Union etc.
- Unary operators bind to the left sub expression of a query. [5, 6]
- Binary operators bind to the left and right sub expression respectively of a query. [5, 6]

5.6 Analyser

The analyser has been implemented to receive and process queries inputted by the user at the interface. The query is in a string form and the analyser has been implemented to break the string into tokens (substrings). Upon receiving the query, the analyser works by reading and splitting the input string from left to right into tokens.

The analyser only accepts the user input if it abides by and can be tokenised into constants and variables (arguments). It accepts the user input if it is able reach the end of the input string and is able to successfully split the whole input into tokens.

Example 5.1: Tokenising input

```
Input 1: league JOIN cup WHERE team = 'Arsenal'

Input 2: league JOIN cup WHERE team='Arsenal'5

Tokens Generated for input 1:
    <league> <JOIN> <cup> <WHERE> <team> <=> <Arsenal>

Tokens Generated for input 2: same as input 1

The analyser is successfully able to identify where to split
the string and can handle cases when input has spacing between
arguments in a query and when it does not.
```

⁵ Note. The substring <team = 'Arsenal'> in input 2 is not separated by spaces while in input 1 it is.

5.7 Parser

5.71 Validating Tokens against TRA grammar

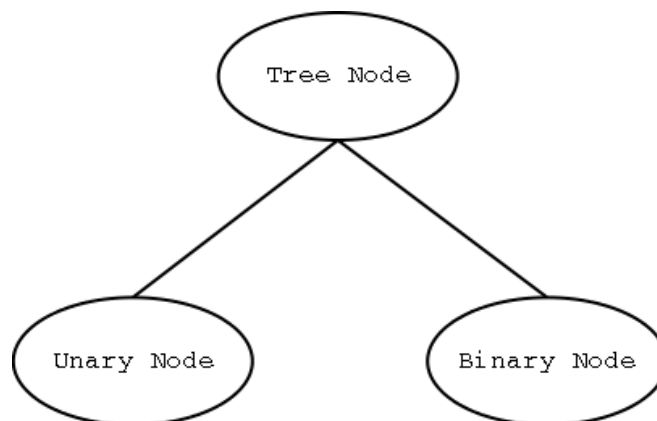
The parser validates each or a group of tokens (depending on the status of the current token having just been analysed) as they are generated, by matching them against the predefined grammar integrated into the system. While parsing tokens, if the parser is unable to recognise any part of the input, an error is returned back to the user via the interface highlighting the problem and the reason why the input has been rejected. For example, if the user has mistyped an operator name, then the analyser stops translating and validating any further input and returns an error message with this mistyped name.

5.72 Node Generation

The parser has been implemented to translate the tokens produced by the analyser into tree nodes, which are then used to create the TRA tree. The parser will produce nodes for each different operation that is supported by the system. Operations include Project, Product, Join, Since, Until etc, (temporal relational operators). Each operation has its own node type and stores any arguments that are required by operation. For example, performing a Project operation (part of a query) creates a 'Project' node and holds the list of attributes to project in the node.

Separate nodes have been implemented that define the different operations and hence the temporal functionality of the system. Operators that share the same structure i.e. the type of the arguments that the operator requires in order to carry out its functionality; we have implemented a single generic node for all such operators with common characteristics. By labelling the node, we can distinguish between the different operations the generic node can be used for. Using Java abstraction and inheritance properties, to achieve this desired effect.

Figure 5.2: The abstract types of Tree Nodes



Unary Nodes: For operators that carry out operations over a *single* sub expression or a relation. E.g. Project, Rename, Past, Future, etc require a single argument (a table of the query so far evaluated) on which to perform their operations. Unary nodes have a single pointer that is used to point to the node below it (known as child node); the rest of the tree.

Binary Nodes: For operators that require *two* sub expressions or relations. E.g. Product, Join, Since, Until, Union, Difference etc require two arguments (two tables representing the queries on the left and right to the operator) on which to perform their operations. Binary nodes have two pointers, which point to the left and right nodes below it (known as child nodes); the left and right sub tree respectively.

The Tree Node UML diagram in **APPENDIX D and E** shows the different nodes supported by the parser.

The parser has been implemented such that all temporal operators can be partitioned into either one of the two node types: unary and binary nodes. The top-level node (tree node) is an abstract node. The reason to have an abstract node implies that all the nodes are of one abstract class (subclasses extending the top-level abstract node) making it easier to link nodes when generating the TRA tree without having to know the concrete type or class of the node. Also this style of implementation eliminates the need to define different variants of a node (what other nodes they can be linked to) meaning creating more redundant and complex classes. For example, when creating a unary node and we are not using abstraction and inheritance, we must either develop two instances of the node where one allows the node to be linked to another unary node and the other allows the node to be linked to a binary node. The other way would be to implement a single complex unary node to allow the node to be linked to another unary or binary node which would prove difficult to design and time consuming to produce. Thus, without having an abstract node class, the system would have become difficult to manage and develop (too many unnecessary classes and code becoming messy).

In summary, nodes are implemented as classes and using inheritance properties to form a class relationships and a hierarchy structure. This modularises the code making it easier to maintain and then construct TRA trees.

5.72 Tree Generation

Once the nodes have been created, the system then connects them together to form a tree representation of the input known as TRA tree. Nodes are processed and linked to other nodes as they are generated. Each node generated is classified as a unary or binary node.

Algorithm 1 – Constructing TRA Trees

The following pseudo-algorithm illustrates how nodes are linked to construct TRA trees.

```
tokenType current = null;
int index = 0; // index to reference each token from the list

function BuildTree (tokens: Array of tokens): TreeNode
{
    boolean subexpr = false    // stop if reached the end of a sub
                                // expression
    RootNode root = new root node

    while(index < noOfTokens && !subexpr)
    {
        index++
        current = tokens[index]

        if current == binary operator
            bin = new binary node
            // assign the tree generated so far to the left of
            // the binary node
            bin.setleftTree = root.next
            // generate the expr on right of operator
            bin.setRight = BuildTree(tokens)
            // set the root pointing to this new node
            root.next = bin

        else if current == unary operator
            // get args need by the operator from the tokens
            ury = new unary node (tokens)
            // assign the tree generated so far to below the unary
            // node
            ury.setnext = root.next
            // set the root pointing to this new unary node
            root.next = ury

        // call a procedure to check for end of nested queries/sub
        // expressions
        else if checkBrackets(current)
            subexpr = true

        else
            halt and display appropriate errors

    } // end of while

    return root.next    // return the node at the root

} // end of the BuildTree function
```

The algorithm highlights the following facts:

1. A unary node represents a unary operator.
2. When a unary node is formed, its pointer is set to the tree that has been so far generated (the left sub query expression with respect to the operator).
3. A binary node represents a binary operator.
4. When a binary node is formed, its left pointer is set to the tree so far produced (the left sub query expression with respect to the operator). The right pointer is set to the remaining tree that will be produced from the remaining tokens (the right sub query expression with respect to the operator). The function to build a tree is called at the point when assigning the right pointer and passing along the remaining tokens to the call.
5. Nested queries are handled by checking for brackets. When a close bracket is encountered, the function stops build the tree and returns back to the calling point; calling the function recursively for handling nested expressions.
6. The tree is built bottom up.

Procedures and functions for building trees and detecting nested sub expressions were implemented by creating classes and methods in Java. The methods were integrated together and linked with the code for the analyser in order to tokenise input and build the tree concurrently to improve the overall performance of the system.

The tree generation functionality builds the tree as nodes are generated which improves and leads to an optimised parser as oppose to waiting until the all the tree nodes have been formed before constructing the tree. Also, it may prove difficult in connecting the tree nodes when they all have been created as we must keep an account of what nodes must be linked to which other nodes. The reason for representing the TRA query as a TRA tree is that a tree can be restructured after it has been built and before execution, to optimise the query and hence the performance of the system.

5.73 Nested Queries

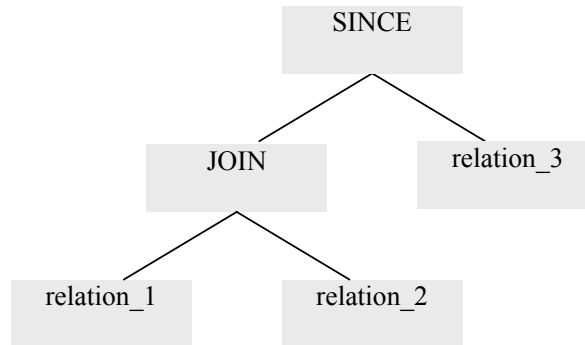
A nested query is a sub expression within a query that is embedded by parenthesis. The tree generator is design to handle nested queries where the start of a nested query is indicated by an open bracket, '(' and the end is indicated by a close bracket, ')' both generated by the analyser as tokens. The tree generation function keeps count of the number and type (open or close) of parenthesis it has encountered and using this knowledge (when it comes across a close bracket) it stops generating the tree and returns back to the point where the call was made. The tree generation function is used recursively to ensure that nested queries are correctly interpreted.

Parser Examples

Example 5.3: Parsing non-nested input queries.

Input: relation_1 JOIN relation_2 SINCE relation_3

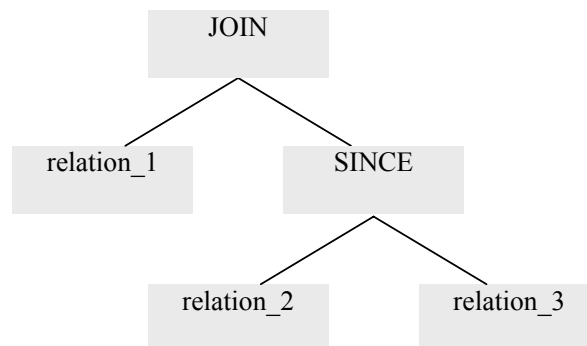
Tree Generated:



Example 5.4: Parsing nested input queries.

Input: relation_1 JOIN (relation_2 SINCE relation_3)

Tree Generated:

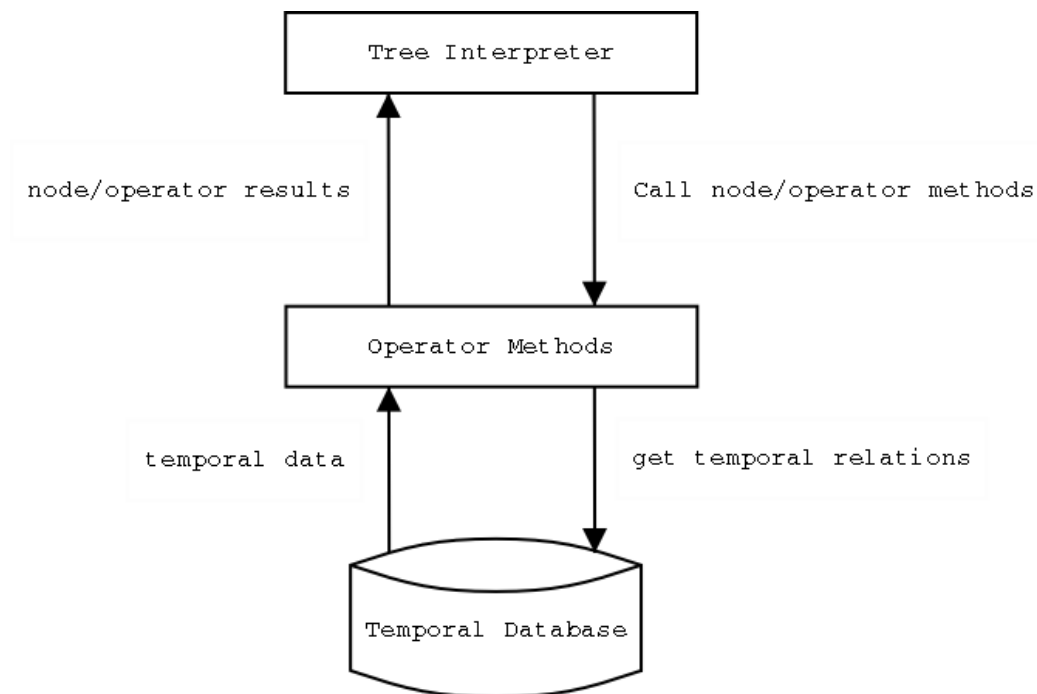


The above examples show how the parser behaves when query inputs contain parenthesis and when they do not. The non-nested query generates the tree abiding by the explanation of how unary and binary nodes are attached to the tree so far generated. The nested query, generates a TRA tree that interprets expressions in brackets as embedded or sub queries.

5.8 Temporal Query Processor

The TQP has been implemented to execute TRA trees that have been produced by the parser and display the results to the user using a graphical interface. The TQP walks over the tree and executes each node it visits and uses any arguments that the node has stored. TQP contains all the functions (coded as separate classes) that perform the relational operations (Product, Union, Project, Since, Until etc). Data structures have been implemented in the TQP to store the results of partial (as the tree is being processed) and the full evaluation of the TRA tree.

Diagram 5.5: TQP modules



5.81 Data Structures

A table data structure has been implemented to store the evaluation of a tree node. The results of a node are then passed as input to its predecessor node. The table data structure has been implemented to hold information typically associated with temporal relations, attribute details and tuple objects. A table It holds minimal data, meaning no repeated tuples.

Appendix F gives the UML diagram of the Table data structure.

Other small data structures have been created where necessary for implementing certain features of the system (e.g. a buffer to store and navigate TRA queries submitted to the system by the user – see section User Interface).

The data structure is implemented as classes and using the aggregation properties to form relationships between them. This allows code to be modularised and easy to maintain, update and use.

5.82 Interpreting the Tree

The tree is interpreted by walking over it in a bottom-up left-right, approach. The TQP receives a tree and evaluates the root by first executing its child nodes from left to right. A node is executed by calling the appropriate operator methods, which it represents. It makes use of the results of its children as input to the methods and passes in addition any arguments the node stores to the method called. Therefore, each node executes its child nodes before evaluating itself.

During interpretation of a tree where errors constitute of nodes posses incorrect user entered arguments, are handled using one of two approaches. Either the TQP highlights the problem and halts execution or on certain matters can make an assumption and continue the evaluation of the TRA tree. For example, the user has requested to perform a non-natural join of two relations and has supplied the attribute information on which to compare data. Consider the situation where the user has provided incorrect attribute information on which to compare data for joining. The system discards the incorrect, specified attributes to join on and performs a natural join of the two relations instead. The user is then notified of any such changes.

The interpretation of a tree is implemented as methods in TQP and coded in Java. It determines the instance of a node and calls the operator methods appropriately.

A bottom-up approach in parsing a TRA tree is adopted, as it is simple to create, use and understand for the requirements of the system. Adopting a top-down approach would be too complex to implement, maintain and may not be the correct way to interpret TRA tress. It will not provide any significant optimisation or enhance the performance of the TQP.

5.83 Temporal Relational Operators

The TQP implements the operators that the system can support and are implemented by coding specific algorithms in Java. Java Utilities, Java Language, Java SQL and other libraries/APIs were used to construct the different temporal operators. Class abstraction and inheritance features of Java are used to form relationships between operators (their classes) that are related.

The following sections describe the common operators that have been implemented in the system.

5.831 Natural Product, Since Product and Until Product

These operators take in as input two tables and perform the respective product operation. A single algorithm has been implemented as the three product operators have the same common structure for evaluating tables. Case statements are used to distinguish between the different types of products and manipulate the temporal objects of tuples with respect to the type of product being executed.

The algorithm implemented for executing product operators:

- Results are generated for overlapping tuples.
- Ensures that any row from either table, which has start and end time equal to the maximum or minimum time of the system does not generate a since or until product result respectively.
- The function implements the option of coalescing the results.

Algorithm 2 – Different Temporal Products

```
Function Product(A, B, flag):R {
A = set of rows each with 1...n data objects
B = set of rows each with 1...m data objects
flag = option to coalesce results
R = set of rows each with n*m data objects
maxTime = maximum time supported by the system
minTime = minimum time supported by the system
op = natural, since or until product

for(i=0, i<A.size, i++)
{
    for(j=0, j<B.size, j++)
    {
        if overlap(Ai, Bj)           // the time intervals of the
        {                             // two rows overlap
            switch(op)

            case PRODUCT:
                st = max(Ai.st, Bj.st)
                et = min(Ai.et, Bj.et)
                Rij = (ai1, ..., ain, bj1, ..., bjm, st, et)
                break;

            case SINCE:
                if not( (Ai.st==maxTime and Ai.et==maxTime)
                        or(Bi.st==maxTime and Bi.et==maxTime) )
                {
                    st = max(Ai.st, Bj.st)+1
                    et = Ai.et+1
                    if st>maxTime then st=maxTime
                    if et>maxTime then et=maxTime
                    Rij = (ai1, ..., ain, bj1, ..., bjm, st, et)
                    break;
                }

            case UNTIL:
                if not( (Ai.st==minTime and Ai.et==minTime)
                        or(Bi.st==minTime and Bi.et==minTime) )
                {
                    st = Ai.st-1
                    et = min(Ai.et, Bj.et)-1
                    if st<minTime then st=minTime
                    if et<minTime then et=minTime
                    Rij = (ai1, ..., ain, bj1, ..., bjm, st, et)
                    break;
                }

            default:
                Report error: unable to recognise 'op'
                break;
        } // end of if
    } // end of for B
} // end of for A

if (flag) then Coalesce(R)
return R
}
```

5.832 Join, Since Join and Until Join

These operators take in as input two tables and perform the respective join operation. The different join operators have been implemented to perform either natural join of each type of operator or perform a specific join on selected attributes that the user specifies. A single algorithm has been implemented as the three join operators have the same common structure for evaluating tables. The algorithm has been coded such that it accepts two lists of attributes, each list contains a set of column names from each table respectively. If both lists are empty or attributes are not compatible for a join, then a natural join (assumption made) of the specified join operator is performed. Otherwise, a join of the operator is performed on the selected attributes given in the lists.

The algorithm implemented for executing join operators:

- Results are generated for overlapping tuples.
- Ensures that any row from either table, which has start and end time equal to the maximum or minimum time of the system does not generate a since or until join result respectively.
- Before carry out any of the join operations, the tables are sorted by the attributes they are to be joined on. This helps to optimise and improve the performance of the operators as it makes it easier to search for tuples that can be joined without having to parse tables too many times.
- When comparing tuples for joining, if the tuple from first table is less than another, we move on to the next tuple in that table as no further tuples from the second table can be joined with this tuple.
- The function implements the option of coalescing the results.

Algorithm 3 – Different Temporal Joins

```
Function Join(A, B, attsA, attsB, flag):R
{
  A = set of rows each with 1...n data objects
  B = set of rows each with 1...m data objects
  attsA = set of attribute names of A
  attsB = set of attribute names of B
  flag = option to coalesce results
  R = set of resulting rows each with (n*m)-|atts|
      data objects
  maxTime = maximum time supported by the system
  minTime = minimum time supported by the system
  op = natural, since or until join

  if (attsA==empty or attsB==empty or NotCompatible(attsA,attsB))
  {
    // establish the common attributes shared by A and B
    common=GetCommonAttributes(A, B)
    attsA=common
    attsB=common
  }
  else
    display message that a natural 'op' join is performed

  quicksort(A, attsA)           // rows sorted according to attsA
  quicksort(B, attsB)           // rows sorted according to attsB

  k=0          // counter for results
  for(i=0, i<A.size, i++)
  {
    j=0
    stop=false

    while(not stop and j<B.size)
    {
      // row a greater than row b, continue searching for
      // joinable rows
      if Ai > Bj
        j++

      // the two rows have the same non-temporal data in
      // the common attributes shared by the rows time
      // and the intervals of the rows overlap
      else if AiattsA == BjattsB and overlap(Ai, Bj)
      {
        switch(op)

        case JOIN:
          st = max(Ai.st, Bj.st)
          et = min(Ai.et, Bj.et)
          // only include one copy of common attribute
          // data shared by the rows
          Rk = (ai1, ..., ain, bj1, ..., bjm, st, et)-BjattsB
```

```

        k++
        break;

    case SINCE JOIN:
        if not( (Ai.st==maxTime and Ai.et==maxTime)
            or(Bi.st==maxTime and Bi.et==maxTime) )
            st = max(Ai.st, Bj.st)+1
            et = Ai.et+1
            if st>maxTime then st=maxTime
            if et>maxTime then et=maxTime
            // only include one copy of common attribute
            // data shared by the rows
            Rk = ((ai1,...,ain,bj1...,bjm,st,et)-BjattsB
            k++
            break;

    case UNTIL JOIN:
        if not( (Ai.st==minTime and Ai.et==minTime)
            or(Bi.st==minTime and Bi.et==minTime) )
            st = Ai.st-1
            et = min(Ai.et, Bj.et)-1
            if st<minTime then st=minTime
            if et<minTime then et=minTime
            // only include one copy of common attribute
            // data shared by the rows
            Rk = (ai1,...,ain,bj1...,bjm,st,et)-BjattsB
            k++
            break;

    default:
        Report error: unable to recognise 'op'
        break;
}

// row a less than row b, no more joinable rows
else
    stop=true
} // end of while
} // end of for A

if (flag) then Coalesce(R)
return R
}

```

5.833 Project

The Project operator has been implemented to project the attributes of a table in the order of attributes given. The function takes a table and a set of attribute names as input. It only returns data in each tuple of the attribute names given. If an attribute name is not part of a table then the function stops executing and return this error.

The algorithm implemented for executing the project operator:

- Returns user-friendly errors highlighting any attribute that cannot be project.
- If a table has two or more attributes with the same name, the operator will project the first matching attribute name.
- All temporal objects (start and end time) for each tuple is maintained as it was prior to the project function call and returned in the results.
- The function implements the option of coalescing the results.

5.834 Union

The Union operator takes in as input two tables. The function has been coded so that both tables are sorted before carrying out union. The sorting of tables makes it easier to search for and remove duplicate tuples (both tables have the same copy of a tuple). This ensures that the performance of the union function is optimised and time taken to execute union has been improved. Once duplicate tuples have been removed, the two tables are appended together to produce the result.

The algorithm implemented for executing the union operator:

- The operator first checks if the two tables are of the same format and if not, it does not carry out the operation and returns a user-friendly error.
- Two tables have the same format if they have the same number of attributes and corresponding attributes of the tables have the same data type.
- Two tuples are identical (duplicates) if both have exactly the same data objects including the start and end time values of each tuple.
- Tuples of the second table are appended to the first table and hence the attribute information of the first table is used for the results returned.
- The function implements the option of coalescing the results.

5.835 Difference

The Difference operator takes in as input two tables. The function has been coded so that both tables are sorted before carrying out difference. The sorting of tables makes it easier to search for compatible tuples that can perform a difference. This ensures that the performance of the difference function is optimised. The difference operator only works if both tables have the same format. In effect, the operator function has been implemented so that it performs a temporal ‘A except B’ of the two tables A and B, given as input.

The algorithm implemented for executing the difference operator:

- Results are only produced if two tuples from the tables have the same data objects (except start and end time) and the time intervals of the tuples are overlapping.
- When subtracting one tuple from another, the resulting tuples contain the same data objects present in the tuples but new start and end time values. These new time values are the difference of the time intervals associated with the tuples being
- More than one instance (all data objects are the same except the start and end time values) of a tuple may be created if the difference of two time intervals produce partitioned time intervals.
- Two tables have the same format if they have the same number of attributes and corresponding attributes of the tables have the same data type.
- The function implements the option of coalescing the results.

Algorithm 4 – Temporal Difference

```
function Difference(A, B, flag):R
{
    A = set of rows each with 1...n data objects
    B = set of rows each with 1...n data objects
    R = set of resulting rows each with 1...n data objects
    maxTime = maximum time supported by the system
    minTime = minimum time supported by the system

    quicksort(A)
    quicksort(B)

    k=0          // counter for results
    for(i=0, i<A.size, i++)
    {
        j=0
        stop=false          // searching for subtract-able rows
        in B
        include=false       // whether to row Ai has been
        subtracted

        while(not stop and j<B.size)
        {
            // row a greater than row b, continue searching for
            // any subtract-able rows
```

```

        if  $A_i > B_j$ 
            j++

    // the two rows have the same non-temporal data
    else if  $A_i == B_j$ 
    {
        if( $A_i.st == B_j.st$  and  $A_i.et == B_j.et$ )
            // subtract the whole row by not including
it in R
            include=false
        else // create result of  $A_i$  subtract  $B_j$ 
        {
            if( $A_i.et > B_j.et$ )
            {
                st =  $B_j.et + 1$ 
                et =  $A_i.et$ 
                if  $st > maxTime$  then  $st = maxTime$ 
                 $R_k = (a_{i1}, \dots, a_{in}, st, et)$ 
                k++
            }
            include=false
        }
        if( $A_i.st < B_j.st$ )
        {
            st =  $A_i.st$ 
            et =  $B_j.et - 1$ 
            if  $et < minTime$  then  $et = minTime$ 
             $R_k = (a_{i1}, \dots, a_{in}, st, et)$ 
            k++
            include=false
        }
    } // end of else
    j++
} // end of else if

// row a less than row b, no more subtract-able rows
else
    stop=true

} // end of while loop

    if(include) // did not find any rows in B that can be
    {           // subtracted from row  $A_i$ 
         $R_k = A_i$ 
        k++
    }
} // end of for loop

if (flag) then Coalesce(R)
return R
}

```

5.836 Modal Operators - Past, Previous, AlwaysPast, Future, Next and AlwaysFuture

These operators take in as input a table and perform the respective modal operation. The different modal operators have similar characteristics, which are the fact that they all only update the time intervals of tuples. They first check if a tuple meets the requirements for evaluation else operating on them would violate the definition of the modal operator. The modal operators have been implemented as separate classes and linked (the relationship between classes) via an abstract class.

Appendix G gives the UML diagram of the Modal Operators.

The algorithm implemented for executing modal operators:

- Results generated maintain all the same data objects except the start and end time values the tuple had before modal evaluation.
- Past and Previous operators only evaluates and returns tuples that do not have start and end time equal to the maximum time of a time interval permitted by the our system.
- AlwaysPast operator only evaluates and returns tuples that do have start time equal to the minimum time of a time interval permitted by the our system.
- Future and Next operators only evaluates and returns tuples that do not have start and end time equal to the minimum time of a time interval permitted by the our system.
- AlwaysFuture operator only evaluates and returns tuples that do have start time equal to the maximum time of a time interval permitted by the our system.

Algorithm 5 – Temporal Past

```
function Past(A):R
{
  A = set of rows each with 1...n data objects
  R = set of resulting rows each with 1...n data objects
  maxTime = maximum time supported by the system

  k=0          // counter for results
  for(i=0, i<A.size, i++)
  {
    if not (Ai.st==maxTime and Ai.et==maxTime)
    {
      st = Ai.st+1
      et = maxTime
      if st>maxTime then st=maxTime
      Rk = (ai1,...,ain,st,et)
      k++
    }
  }
  return R
}
```


Algorithm 6 – Temporal Previous

```
function Previous(A):R
{
  A = set of rows each with 1...n data objects
  R = set of resulting rows each with 1...n data objects
  maxTime = maximum time supported by the system

  k=0          // counter for results
  for(i=0, i<A.size, i++)
  {
    if not (Ai.st==maxTime and Ai.et==maxTime)
    {
      st = Ai.st+1
      et = Ai.et+1
      if st>maxTime then st=maxTime
      if et>maxTime then et=maxTime
      Rk = (ai1, ..., ain, st, et)
      k++
    }
  }
  return R
}
```

Algorithm 7 – Temporal AlwaysPast

```
function AlwaysPast(A):R
{
  A = set of rows each with 1...n data objects
  R = set of resulting rows each with 1...n data objects
  maxTime = maximum time supported by the system
  minTime = minimum time supported by the system

  k=0          // counter for results
  for(i=0, i<A.size, i++)
  {
    if(Ai.st==minTime)
    {
      st = Ai.st+1
      et = Ai.et+1
      if st>maxTime then st=maxTime
      if et>maxTime then et=maxTime
      Rk = (ai1, ..., ain, st, et)
      k++
    }
  }
  return R
}
```

Algorithm 8 – Temporal Future

```
function Future(A):R
{
  A = set of rows each with 1...n data objects
  R = set of resulting rows each with 1...n data objects
  minTime = minimum time supported by the system

  k=0          // counter for results
  for(i=0, i<A.size, i++)
  {
    if not (Ai.st==minTime and Ai.et==minTime)
    {
      st = Ai.st-1
      et = minTime
      if st<minTime then st=minTime
      Rk = (ai1,...,ain,st,et)
      k++
    }
  }
  return R
}
```

Algorithm 9 – Temporal Next

```
function Next(A):R
{
  A = set of rows each with 1...n data objects
  R = set of resulting rows each with 1...n data objects
  minTime = minimum time supported by the system

  k=0          // counter for results
  for(i=0, i<A.size, i++)
  {
    if not (Ai.st==minTime and Ai.et==minTime)
    {
      st = Ai.st-1
      et = Ai.et-1
      if st<minTime then st=minTime
      if et<minTime then et=minTime
      Rk = (ai1,...,ain,st,et)
      k++
    }
  }
  return R
}
```

Algorithm 10 – Temporal AlwaysFuture

```
function AlwaysFuture(A):R
{
  A = set of rows each with 1...n data objects
  R = set of resulting rows each with 1...n data objects
  maxTime = maximum time supported by the system
  minTime = minimum time supported by the system

  k=0          // counter for results
  for(i=0, i<A.size, i++)
  {
    if(Ai.st==maxTime)
    {
      st = Ai.st-1
      et = Ai.et-1
      if st<minTime then st=minTime
      if et<minTime then et=minTime
      Rk = (ai1,...,ain,st,et)
      k++
    }
  }
  return R
}
```

5.837 At

The At operator takes in a single table and a time value as input. The operator has been designed to parse over the whole table and return tuples as results that have the time value occurring in the tuple's interval. In effect, the operator has been implemented to return tuples that are valid, present at the given time instance. The operator does not manipulate any of the data objects (including the start and end times) of tuples.

5.838 Coalesce

The Coalesce operator takes in a single table as input. The coalesce function has been implemented to group identical tuples with overlapping time intervals by combining them and creating one copy of the tuple. The time interval of the new tuple covers all the time intervals of the tuples merged. The algorithm, which implements this operator is described in the Background chapter under the heading 'Coalesce'.

5.839 Rename

The Rename operator takes in as input a table, and two lists of attribute names. The Rename operator has been implemented to format the names of attributes of a table. This function is useful when performing one of the temporal join operators. By renaming an attribute in one table permits the table to have attributes in common with other tables and hence carry out a join operation.

The algorithm implemented for executing the operator:

- Does not alter any tuple data of tables.
- One attribute list is the column names that exist in the given table.
- The second attribute list contains a set of user-defined names that replace the column names given by the first list.
- Returns user-friendly errors highlighting any attribute names that cannot be found in the given table for renaming or if the size of the two lists are not equal.
- If a table has two or more attributes with the same name, the operator will rename the first matching attribute name.

5.84 Operator Optimisation

Temporal operators that can potentially be optimised have been done so. To optimise most operators require sorting tables before manipulating them. This helps to reduce searching for tuples and the time taken to parse tuples in tables for evaluation. When considering a sorting algorithm, the following popular algorithms were analysed [16]:

Sorting Algorithm	Time Complexity	Space Complexity
Quicksort	$O(n \log n)$	$O(\log n)$
Insertion	$O(n^2)$	$O(1)$
Merge	$O(n \log n)$	$O(n + \log n)$
Bubble	$O(n^2)$	

The time and space complexity in the above table represents the average case performance of the sorting algorithm.

The quicksort algorithm was adopted for sorting tuples in tables by attributes specified. This is because quicksort offered better combination of time and space complexity than other the sorting algorithms. Although quicksort and mergesort have the same time complexity, the space complexity of quicksort is better than that offered by mergesort.

5.85 Auxiliary Functions

Auxiliary functions and methods are coded into the system that is required by the temporal relational operators to carry out their operation. These auxiliary methods are generic functions that are shared and used by any operator, if required. For example, the sorting of tables, determine if two tuples overlap, comparing the data objects of two tuples... etc. These auxiliary methods are coded in Java and making use of libraries available.

5.86 Connecting to the Temporal Database

A user entered TRA expression specifies the temporal relations on which to query. These relations must be retrieved from the underlying database system before querying and carrying out operations on them. The TQP is designed to use JDBC drivers to establish a connection with the underlying temporal database and makes use of Java sql library to interpret the temporal data. SQL statements are used to run a query on the temporal database to get temporal relations. The temporal data retrieved is stored in the table data structure designed and implemented in the TQP. The fetching of temporal data from the underlying DBMS is independent of the type⁶ of the relational DBMS used to store temporal relations. Note that the TQP is designed to only accept temporal data from relations, tuples must have a start and end time else the system notifies the user that such relation held by the DBMS is not a temporal relation.

⁶ By type we mean the commercial make of the DBMS e.g. Postgres, Oracle, DB2, Sybase etc.

5.9 Interface

The graphical user interfaces were implemented using Java Swing. The use of Swing libraries made it relatively easier to develop and code the interfaces.

The system has two types of interfaces:

5.91 Command/Input Interface

JFrames, JButtons, JTextAreas and other graphical components of Swing were used to develop the user interface. This interface has been designed to allow users to interact with the system and run TRA queries. The command interface has been implemented to display all error messages and pass user submitted input to the analyser as a single argument.

Details and screen dump of the command user interface is given in the **APPENDIX H**

A query buffer has been implemented to hold all TRA queries a user has executed on the system since the launch of the system, the current system session. The query buffer was coded using Java Utilities and integrated into the command interface. The purpose of the query buffer is to let the user navigate through the submitted TRA queries.

5.92 Result/Output Interface

JTables and JFrames were used to implement the graphical displaying of results. A JFrame is created to hold an instance of a JTable that contains the results of the user input. JTable made it easier to display results in a tabular format.

Different results are show on separate JFrames. The reason for using separate frames for results was that it made the system more flexible, as it made it possible for the user to compare results of different queries without the need to output the results to a file.

Details and screen dump of the result interface is given in the **APPENDIX I**

Chapter 6 Testing

This chapter aims to discover and discuss the capabilities of the system that has been implemented and explain how the system was tested. Testing is performed at two levels:

Low-level: testing the individual modules (code and algorithms) implemented.

High-level: testing the whole system from a user's perspective.

6.1 Test Data Used

The Temporal data used to construct test cases when testing all and individual parts of the system:

1. *League* and *FACup* relations used by the different examples given in the Background section. The relations were extended to hold more temporal data that covered the last 10 years of the football premiership and FA cup competition. The number of records in the *League* relation totalled to 204 for the 10 years. The *League* relation was extended to hold more information for each record such as the number of goals scored, conceded, games won, lost, drawn etc by a team. The data for these relations was obtained from the web site: <http://www.the-english-football-archive.com/>
2. The *account* and *movement* relations given in publication [3].
3. The *people* and *works for* relations given in publication [1,2].

Both sets of relations mentioned in points 2 and 3 were extended and altered as required.

6.2 Test Specification

During the technical testing of the system, we are trying to determine the following:

1. Validation – A complete system: has all functionality been fully implemented, are errors handled correctly and does the system boast the features and characteristics that are desired.
2. Verification – A correct system: does the system produce and calculate the correct results for all valid TRA queries.
3. Performance Evaluation: does the system execute and process TRA queries in reasonable time.
4. Stress and limitations of the system: the breaking point of the system, the processing weaknesses of the overall system.

6.3 Validation and Verification of Modules

In order to determine if the system has been correctly implemented and is complete requires the analysis of each individual module or component that has been built. As modules were created, individually, they were tested independently before integrating them to the rest of the system. This proved to be very effective and useful in delivering a complete system as defects were easily caught at earlier stages of the development and amended. Also, being able to test modules independently implies that the system is tested frequently and consistently during development, which leads to an improved system.

The system specification and design (described in their respective chapters) provides the material and a benchmark against which the system modules were tested and verified.

6.31 Analyser

Entering random valid and invalid TRA queries as input tested the analyser. Different types TRA queries were generated and entered into the analyser for testing. We ensured that for all valid queries entered, they covered all the operators the system is able to support. During progress and development of the system, new operators were added. The analyser was then updated and tested once again, by giving it as input, different structured TRA queries using the new operators.

6.32 Parser

The parser was tested with tokens that form valid and invalid queries. A mixture of random input representing tokens was used for testing. The test cases used to evaluate the parse ensured that they covered the whole set of temporal operators and a range of variables the system supports.

6.321 Validating Tokens

To detect that invalid queries were not processed, reordering tokens that form a valid query and placing different tokens next to other tokens that should not be there was used for testing. For example, when projecting a set of columns from a relation, the ‘{’ parenthesis denotes the start of projection and ‘}’ marks the end. In between these parentheses, only variable tokens can occur which are suppose to represent the columns to project and no constant tokens are permitted. Thereby, placing constant tokens such as those that represent operators or other types of parentheses, between the project tokens ‘{’ and ‘}’, we were able to test that errors were correctly caught and reported.

6.322 Generating TRA Trees

To ensure that the correct tree was being constructed, debugging style methods were applied to the part of the parser that builds the trees. Therefore, outputting the tree that the system generated on the command line screen and using parentheses and tab spacing made it easier to analyse the tree outputted for checking that correct trees were being created. In order to test if correct trees were formed, the query that was inputted was first sketched out on paper (the tree representation of the query) and then inputted into the system and checked against what the parser produces. A set of random examples were used for testing but ensuring that they cover nested queries that are placed in brackets.

6.33 Temporal Query Processor - TQP

The TQP was tested for the way it interprets TRA trees and the calculations made by the functionality of the operators that have been implemented.

6.331 Interpreting TRA Trees

Similar approach for testing the construction of TRA trees by the parser was used. A debug method was applied to print out the steps that need to be executed by the trees, the methods that need to be called and checked this against what was expected. Random covering majority of different types of trees that can be formed was used as input for testing.

6.332 Functionality of the Operators

Each operator was tested after it was implemented. They were each tested for the errors they are suppose to catch and the functionality that needs to be carried out.

The following summarises the test cases used:

1. Calculates the correct time values (start and end time of tuples) for the resulting tuples the operator generates.
2. Tuples that have times values set to the boundary of a time interval permitted by the system. For example, Since-Product, Since-Join, Past and Previous do not manipulate tuples that have both start and end time values as the maximum time value, which we allow are data to hold at.
3. Performing Join operations whereby joining tuples on specific attributes.
4. Performing natural Join operations whereby the Join operators must locate the common attributes shared by the relations.
5. Check the format of relations before carrying out the operation. For example, to perform a Difference of two relations, the relations must have the same number of attributes and each corresponding attribute must have the same data type.
6. Projection and renaming of attributes.

7. Tuples with different time intervals in a relation that consists of a mixture of overlapping, touching, non-overlapping and non-touching tuples. Useful especially to test the Coalesce function.

Note. Different operators need to be tested for different requirements so all the above test cases listed may not apply to every operator implemented.

6.333 Operator Verification

For validating the outputs of the operators, results were pre-calculated and checked against the results produced by the execution of the operators on the system. For relations that produced results with over 30 records, records at random were chosen and pre-calculated manually to ensure that the operation being conducted is correct.

6.34 Error Handling

Locating and reporting errors was tested for different type of errors that can persist. Test cases mostly consisted of invalid queries as inputs to the individual modules.

Testing for errors can be summarised in main two sections:

1. System errors: these are low-level errors such as unable to convert between data types. For example, all data objects extracted from the temporal database are held as an Object data type. When performing a Select operation, we need to convert the user entered and data object values into the correct data type for comparison i.e. into integer, floats, strings etc. Other low-level testing includes: unable to locate or update classes, methods, and variables.
2. User errors: these are high-level errors such as arguments given in a query input are incorrect. For example, wrong relation or attribute names. The purpose of these errors was to make sure that user-friendly error messages were being returned.

6.35 Interface

Having created the user interface and the display interface for results, each was tested to ensure that the buttons and any other components in the interface carry out their tasks. Test cases were set up as the interface was being developed. Their aim was to the passing of information between the interface and the rest of the system (all the modules) developed.

6.4 Defect Handling

All defects that were found during testing of the whole system and the individual modules were amended before the system was delivered.

6.5 System Performance

Performance testing aims to measure the average time taken to execute TRA queries and different TRA operators.

Three Linux and Windows platform based machines were used to test the performance of the system. Each individual test case was executed on all machines in order to compare the performance across the two platforms. The machines were chosen randomly from a selection of machines in the Computing department at Imperial College.

Relations used for the tests cases are discussed at the start of this chapter. The size of relations used for testing varied between 5 to 204 records (204 records for the full set of data held by the relation *League*).

APPENDIX J and K show the test results of Unary and Binary operations respectively.

6.51 Performance Analysis

The time taken to execute unary operators is about 1 second on either platform. The time taken to execute binary operators is about 1.75 seconds on the Linux machines and about 2 seconds on the Windows platform.

The average time to process queries is considered to be reasonable as we take into consideration the fact that there will be overheads of communicating to the temporal database via JDBC.

Also, by testing some of the operators (those that can be) in Postgres provided a benchmark to compare the performance times of the operators implemented by our system. For example, by running *A Product B* in Postgres gives the performance of the *Product* operator. Using this result, we can ensure and compare the *temporal Product* performance on our system. Note, this was only done for some operators and only to provide us with an idea of how reasonable the performance of operators need to be.

6.6 Stress Evaluation

Stress evaluation aims to test the breaking point of the system, how much data can it process before collapsing.

The system was tested for stress handling by executing a mixture of unary and binary operator defined queries and recursive queries. A recursive query is when an operator is recursively called such as, for example, *A Product B Product C* where A, B and C represent relations.

Stress evaluation showed that most operators were successful at processing and manipulating large amounts of records. The total number of records the system can handle are in the excess of 150,000 where each record has reasonable number of attributes. This result was obtained by executing *League Product League Product League* where the number of records in the *League* relation was 54⁷. The start and end time for each record was the same (set to 0 and 4 respectively – random five seasons of the football premiership competition) to ensure that a temporal product is produced for all tuples in the relation.

Processing records exceeding 200,000 resulted in the system crashing. This result was obtained by executing the above query mentioned where now the relation contained 60 records⁸. The reason for the system crashing was that the TQP (developed in Java) ran out of memory space to store and process results.

⁷ 157464 records produced by the query = $54 * 54 * 54$

⁸ 216000 records produced by the query = $60 * 60 * 60$

Chapter 7 Evaluation

This chapter gives an overview of the results and findings from the Testing Chapter. The chapter discusses the strengths and the weaknesses of the individual modules implemented. It mentions the problems encountered with the project and how they were overcome.

7.1 Module Evaluation

The following few sections give an evaluation of each module or component that has been developed.

7.11 Analyser

The task of the analyser is to break input into meaningful tokens for processing the user submitted query.

For all valid queries that were structured to include a variety of the operators and nested or sub queries, the analyser was successfully able to tokenise the input. The analyser evaluates input using one of the two following rules:

1. For all recognisable operators it tokenises them and represents them as constants.
2. For all other types of input, the analyser tokenises them as variables that represent or assumed to be arguments.

The main weakness of the analyser is that most operators that have been used in the query require a character space between them and the arguments, which are on either side of the operator. There are only a small number of cases where the analyser can handle where certain parts of the input has or does not have character spacing. For example, handles well for 'Where' statements that contain constraints where arguments are separated by character space or not.

Using Java made it easy to tokenise input where key query words were separated with character spaces. This made it possible to develop other parts of the system further in terms of operator functionality offered.

7.12 Parser

The main task of the parser is to validate the tokens against the grammar defined, create nodes using the tokens and construct a TRA tree.

The parse is successfully able validate tokens against the grammar that has been integrated into the system. It ensures by keeping an account of the last few tokens it has validated to help catch input errors at the early stage of query compilation rather than at run time and crashing the system. The parse then produces by converting

tokens or a collection of tokens into a node to represent a TRA operator with arguments supplied.

The parse has been effectively designed and implemented to build the tree as tokens are evaluated. This helps to improve the performance time of the system.

7.13 Temporal Query Processor

The main task of the TQP is to interpret TRA trees and evaluate them by calling the appropriate underlying functions of the operators.

7.131 Interpreting TRA Trees and Tree Optimisation

Through testing, we have found that TRA trees were being successfully interpreted. However, we would have liked to optimise TRA trees so that they are more efficient in executing the TRA query they represent. Optimising trees include having operators that filter or return smaller relations⁹ (reduce their size in terms of tuples) such as Select statements or Difference operators being placed lower down the tree which leads to performance improvement as other operators that use these results will have less work to do. We are discarding more tuples that will never contribute to the result, at earlier stages of the tree evaluation. This will mean that less data needs to be managed by the TQP and hence reduce overheads of managing and storing the data.

The reason the system has not been implemented to optimise TRA queries and its corresponding TRA tree is because it requires and involves a great deal of work in order to produce a complete and robust tree optimiser. It is not that simple to optimise TRA trees since it is not the case that all operators which return smaller results can be pushed down to the lower ends of a tree. They can only be placed lower down if they are applicable to the node (operator or relation) that will be below it. Also, methods that optimised trees will take some time to restructure them and hence, not improve system performance time as desired. As the project emphasises on implementing more functionality meant that there was not a lot of time to tackle this problem.

7.132 Overall TQP

The overall TQP is robust and makes the system independent to the temporal database. This is because the underlying relational database that behaves as a temporal database is only accessed once, at the start stage of evaluating a TRA tree. The TQP fetches all and the only relations that are needed by the TRA tree for evaluation. This has been possible since all operators have been implemented at the TQP level and not implementing any as procedures in the relational database system that stores the temporal data. Therefore, there is no need to move back and forth

⁹ By smaller relations we mean that the returned relation has size less than or equal to the relation that was used as input. At worst case the size of relation is the same as the one received as input to the operator.

between the relational database and the TQP. All is required by the TQP when retrieving relations from the temporal database, is that each relation has temporal components (start and end time) at any attribute position in the relation.

7.14 Interface

The interface aims to give control to the user to configure system properties and execute TRA queries.

The user interface provides the necessary buttons to execute queries. However there is not enough support offered by the interface to configure certain properties of the system. For example, if the user wishes to set new values for the minimum and maximum time values a time interval can take then they must do so manually by loading the code and updating it. The lack of these small features hinders the control of the system offered through the interface and makes the system less robust.

It would not be too difficult to add small features to the interface too give a user more control of the system. The reason for omitting or not having these useful features is because there was not enough time to add and test them. The interface was the last module implemented and at this stage, not much time was available as we approached the deadline that was set for completing the project.

Therefore, all is required for the interface to be complete is the fine-tuning and implementation of small features to give more control to the user.

7.2 Improvements

The weaknesses of each module described such as optimisation of TRA trees, handling of character spaces by the analyser, adding more user control features to the interface etc. can all be viewed as improvements that can be made to the system.

7.3 Problems

The following mention some of the problems encountered and overcome when developing the system.

1. As the project was research oriented, understanding concepts of temporal databases proved to be challenging. Especially US logic and the derived modal operators. The problem was overcome by holding frequent meetings with the project supervisor and asking questions relating to the subject area.
2. Engaged in other academic work concurrently such as sitting exams and coursework. By scheduling and spreading the work load made it easier to complete most of the project.
3. System compile and runtime errors. These types of problems were tackled by using intensive debugging methods. Printing out and testing data. Using the Java Sun, Postgres websites and Java Publications to help solve technical problems.
4. Implementing the TRA structure and handling the nested queries. This problem was tackled by doing some further background research on the features of Java and Object-Oriented languages, finding out more about abstraction and inheritance properties.

Chapter 8 Conclusion

This chapter aims to describe the project as a whole. It discusses the achievements and extensions that can be applied to the project in the future.

8.1 Overall

The system that has been developed for the project mainly focuses on and concerns the extraction and manipulation of temporal data. The ideas behind the resulting system have come from research-based work carried out at Imperial College on querying historical relational databases and implementing temporal related operators. The system developed has revised and implemented the RA operators common to most commercial relational database systems that now evaluate temporal data. The system has introduced new and specific operators that can only perform over temporal data and are based on US logic. The overall system is an independent software tool, more specific, a temporal relational database management system that allows itself to be attached to a temporal database, which has been modelled on a standard relational database system. This tool is appealing to organisations that store archives of data but find it difficult to extract required information.

8.2 Achievements

The achievements of the system has been based on two aspects:

1. The system is flexible and portable. Flexibility in terms of the tool being independent to the underlying relational database system used to hold temporal relations. The reason being is that all relations are retrieved from the underlying physical database and then processed. Portable in terms that the tool can perform across a variety of platforms. This is because Java technology was used to implement the system.
2. The variety of operations offered by the system. The tool covers most of the operators that are expected when using commercial database systems and the tool introduces new and useful operators making it easier to manipulate temporal data.

The project has been rewarding and the personal achievements include:

1. A more thorough understanding of temporal databases, US logic and the overall principles of databases.
2. Provided a good opportunity to use the software engineering skills developed at University and on work placement.
3. Has helped to further develop project management skills such as scheduling work and writing reports.
4. Gained experience and confidence in producing a large system. Being able to work and overcome technical difficulties on my own.
5. Provided a better understanding of Object-Oriented languages, Java especially.

8.3 Extensions

The following list the project extensions that can be undertaken:

1. Extend the TRA so that it covers all the RA operators and functionality commercial databases provide such as updating, deleting and inserting records.
2. Develop a TSQL language that incorporates and extends SQL (similar approach of extending RA to TRA).
3. Provide functionality to the system to optimise trees that represent queries.
4. Upgrade and test the system to executing over a range of different commercial relational database systems.
5. Develop a similar system that uses Object-Oriented databases to store temporal data.

APPENDIX A - Considering Implementation Technology

Before developing the TDBMS and implementing the core functionalities, we need to decide on the most appropriate programming language to use for implementing the system. There are a variety of programming languages that we can use to build the system, some will be useful while other will not be ideal for this type of project. We have identified the following potential candidates that can be used to program our system. Each language has a small description emphasizing the strengths and weaknesses of the programming language and any past experience the author has with the language.

Analysing potential technologies, which the author considers useful to use for developing the system:

C

Experience: A great deal. Developed industrial level applications and have completed a few large lab exercises at university.

Pros: The fact that C is a low-level language implies that we can have more control over memory and would in effect improve the performance of the system. Can help to reduce the time taken to execute queries and run operations.

Cons: The problem of using C is that writing the code can get messy and will prove difficult to maintain and test code as the system begins to grow. Also, the ODBC driver used to connect to the underlying database system is known to cause problems especially when migrating the system on to different platforms.

C++

Experience: None and very little knowledge of the language.

Pros: The positive points of C++ are that it does provide object oriented features such as inheritance and dynamic binding making it easier to test and implement individual modules. Being an extension to C, it has good system performance rates.

Cons: The weaknesses of C++ are that it is not a pure object oriented language (in fact an extension to C), has manual garbage collection and there is a current lack of development tools.

Java

Experience: Very little, completed a few small lab exercises at university. Have good knowledge of JSP and JDBC – developed web based tools.

Pros: An object-oriented language with dynamic binding, abstraction, extensive libraries and inheritance properties making it easier to reuse code. Also Java is

portable, which eliminates the effort of migrating the system onto different platforms and can be confident that it will run under the most (common) platforms.

Cons: The main disadvantage of Java is speed. Java needs to be compiled into byte code and then is executed by an interpreter. Therefore, the translation takes some amount of time and no matter how small a length of time this is, it is inherently slower than performing the same operation in machine code.

Perl

Experience: None and very little knowledge of the language.

Pros: The advantages of Perl is that it provides modularization and object oriented techniques.

Cons: Perl is widely adopted for programming when developing web-based applications. The language is pretty much overloaded, as the designers of Perl never hesitated to add new features thus making the syntax of Perl non-trivial. Therefore, considerable time may be required to master Perl.

Visual Basic (VB)

Experience: Some experience, developed a database application during pre-university studies.

Pros: The advantages of VB is that it helps the programmer to develop a large program without remember all variables and their relationship. Also, would be useful to implement our system since it has built in features (drag and drop) to develop graphical user interfaces with ease.

Cons: The main problems to VB are the limitation to the kind of application that can be coded, it is not a mature programming language and known to be slower at processing methods and applications.

APPENDIX B - System Development Outline

<p>Temporal Query Processor (TQP)</p> <ul style="list-style-type: none">• Data structures created to store the relations and results retrieved from the underlying temporal database. A top-level table data structure is created to hold rows and column information.• Implemented functions to connect to the underlying temporal database via the JDBC driver and fetch relations stored in the database.• Coded some of the temporal relational operators: PROJECT, PRODUCT, SINCE, UNTIL, UNION, COALESCE.• Implemented all additional and necessary auxiliary functions such as checking if two tuples overlap, a quick sort method to sort rows in a table etc required by temporal relational operators.• Implemented methods to format and display a table on the command line interface.
<p>Parser</p> <ul style="list-style-type: none">• Generated the different Tree Nodes that are required to form the TRA tree using the abstract and inheritance properties of Java. Tree nodes produced were of two main types, Unary nodes that have a single pointer and Binary nodes that have two pointers.• Implemented functionality to generate TRA trees; linking the nodes formed. <p>TQP was extended to walk over the TRA tree and display the nodes – useful to check that a correct tree is being formed.</p>
<p>Analyser</p> <ul style="list-style-type: none">• Functions implemented to analyse user entered TRA query and break the query into meaningful tokens. The tokens are then used to establish and form the Tree nodes by the parser.• Error handling coded to identify invalid queries and unrecognised symbols or tokens. Valid queries accepted where only for the relational operations that have so far been implemented by the TQP. <p>Note. The analyser and the parse modules were designed and implemented</p>

concurrently as their requirements are inter-linked.
<p>Having completed and built the solid structure of the above three modules, the modules were integrated and tested.</p> <p>Now having the framework of the 3 modules, additional functionality and operators were implemented (analyser and parser extended to handle new TRA queries). These being: JOIN, SINCEJOIN, UNTILJOIN, SELECT, RENAME¹⁰ and DIFFERENCE.</p>
<p>User Interface</p> <ul style="list-style-type: none"> • Implemented a graphical user command interface using Java Swing, which contains buttons to perform commands such as execute a query, display help etc and provide text areas for the user to enter queries and for the system to display messages. • Implemented graphical tables using Java Swing that returns the results for each query executed, in separate frames to the one created and used for the user command interface. <p>Having constructed the user interface, it was integrated to the rest of the system.</p>
<p>Having completed and integrated all the modules, the system was now more complete (feel, look and use) and meeting the project requirements set.</p> <p>Further and remaining temporal relational operators were added to the system (modules extended as required). The new temporal relational operators implemented are PAST, PREVIOUS, ALWAYS PAST, FUTURE, NEXT, ALWAYS FUTURE and AT.</p>
<p>Module Testing</p> <p>When implementing each module, the functionality was tested by running test cases and writing test files.</p> <p>When integrating the modules to form the desired system, testing was performed at a higher level, entering queries and running commands via the user interface.</p>

¹⁰ RENAME operator in a query provides the functionality of renaming attributes held in a table. A table stored in the TQP is a TRA query that has been executed or a sub query that has been executed.

APPENDIX C – Main content of TRA Grammar

<unary operation> ::=
 <rename> | <project> | <where> | <at time> | <modal operation>

<rename> ::=
 <relational expression> RENAME <attribute list> AS <attribute list>

<project> ::=
 <relational expression> { <attribute list> }

<where> ::=
 <relational expression> WHERE <attribute> <comparator> <value>

<at time> ::=
 <relational expression> AT <time>

<modal operation> ::=
 <relational expression> <modal operator>

<modal operator> ::=
 PAST | PREVIOUS | ALWAYS PAST |
 FUTURE | NEXT | ALWAYS FUTURE

<binary operation> ::=
 <relational expression> <binary operator> <relational expression>

<binary operator> ::=
 UNION | MINUS | TIMES | SINCE | UNTIL |
 JOIN | SINCE JOIN | UNTIL JOIN

<table> ::=
 <string>

<attribute list> ::=
 <attribute> | <attribute list> <attribute>

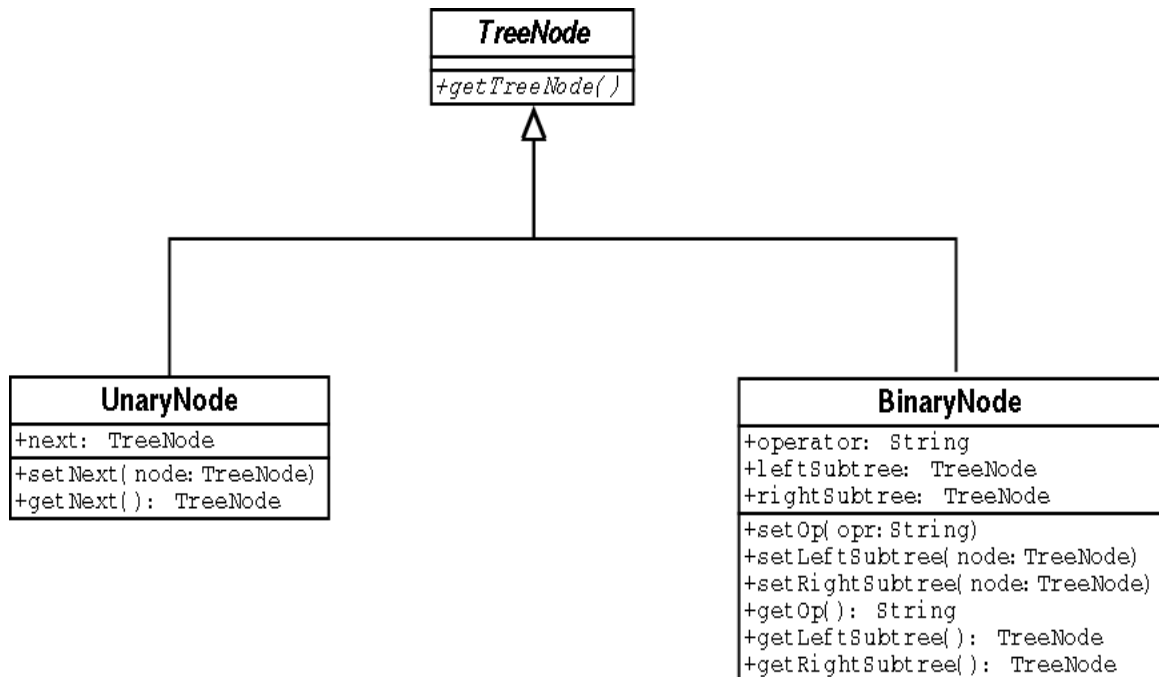
<attribute> ::=
 <string>

<comparator> ::=
 <= | < | = | > | >=

<data type> ::=
 <string> | INT | REAL | DATE | BOOLEAN | NULL

<string> ::=
 CHARS

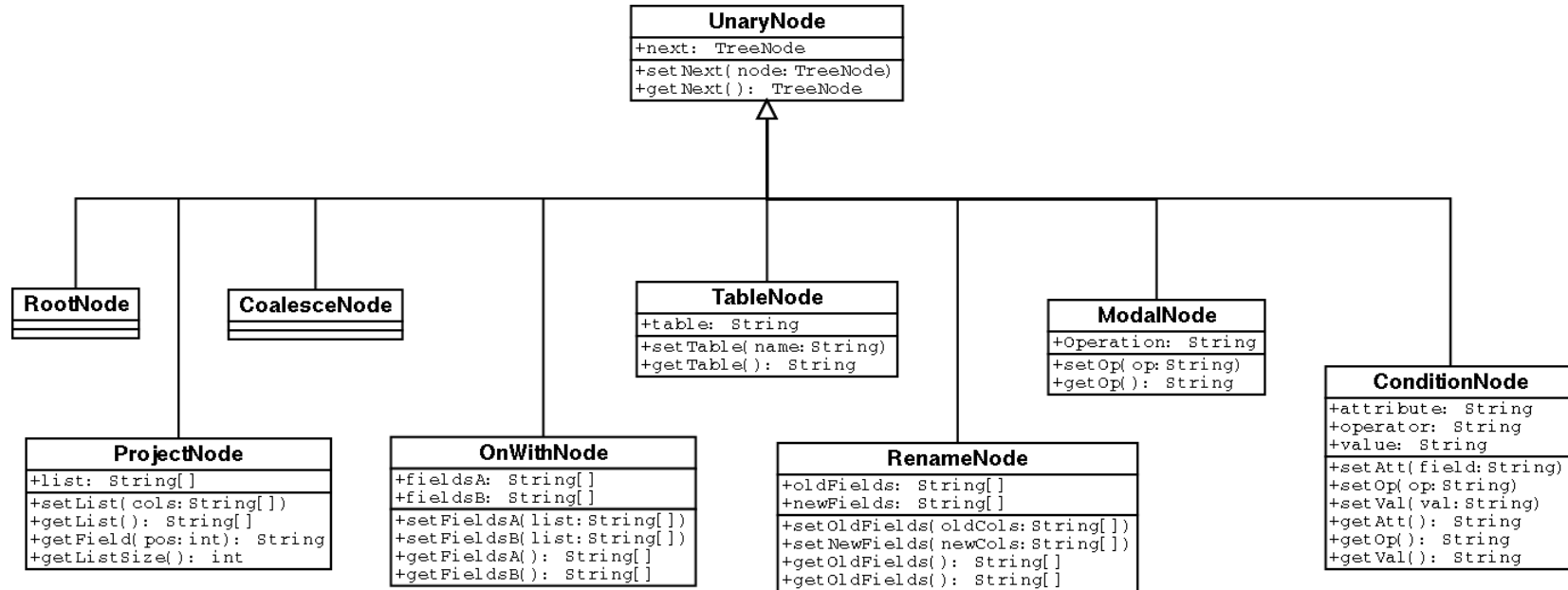
APPENDIX D – Tree Node UML Diagram



The binary node stores (labels the node) a string value, which represents the operation that we wish to conduct (given by the user in the input).

The binary node can only be labelled with a binary operator, which the system provides functionality for. The binary operators that the system supports are: UNION, MINUS, TIMES, SINCE, UNTIL, JOIN, SINCEJOIN, UNTILJOIN, WHERE, AND and OR.

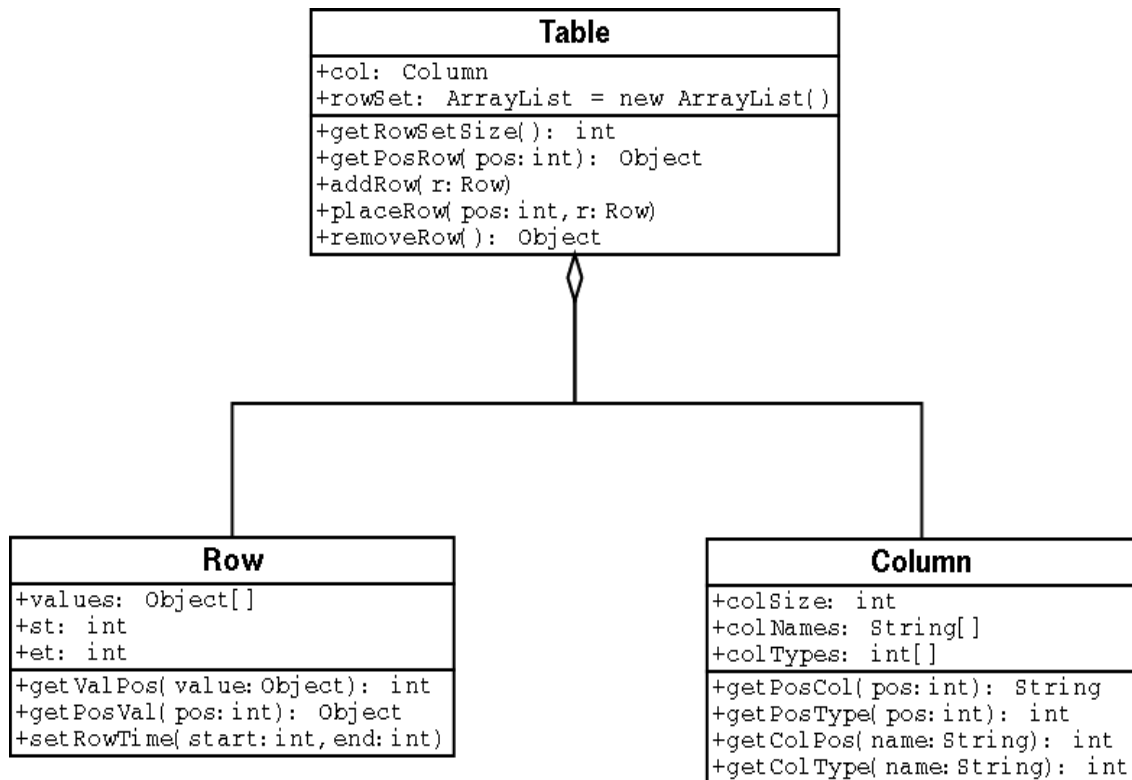
APPENDIX E – Unary Node Types UML Diagram



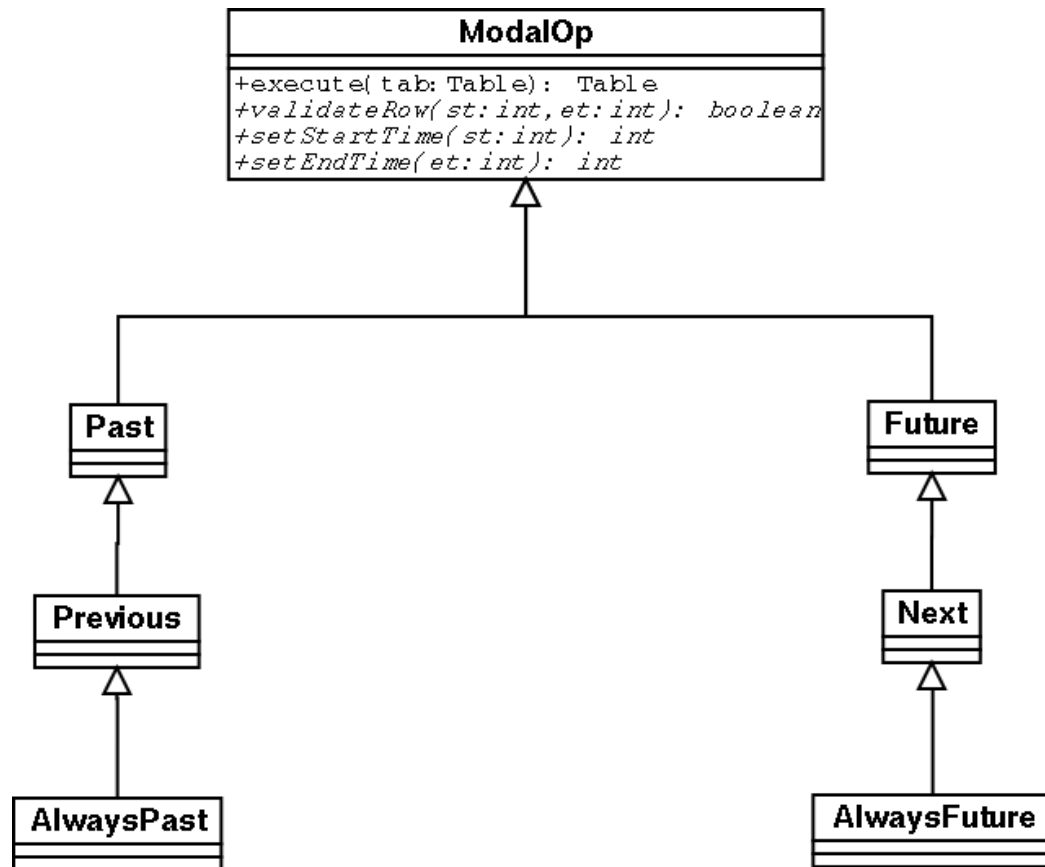
There are different types of unary nodes (Project, Rename, Modal¹¹ etc), which have been implemented by the system due to the fact that, unlike binary nodes, each different type of unary node requires and hold different sets of arguments (e.g. Project stores a single list of the attributes for projection while Rename stores two lists of attributes, one to specify which columns to rename and the other to provide the new column names to replace them with).

¹¹ Modal nodes represent the different temporal operators based on US logic (Until and Since) and are: Past, Previous, AlwaysPast, Future, Next and AlwaysFuture.

APPENDIX F – Table data structure UML Diagram



APPENDIX G – Modal Operators UML Diagram



APPENDIX H – Command Interface

A graphical user input interface has been implemented to let users interact with the system. The interface has been designed to allow users to submit TRA queries and execute them. All queries, which are invalid and rejected by the system, return a user-friendly error message displaying the reason for rejection. Queries that halt at run time or assumptions made by the system during execution due to the fact that the user has supplied invalid arguments, provide a debugging message to the user highlighting the problem and where the mistake has been made in the query. All messages returned by the system are displayed on the interface. All requests made by the user are passed to the analyser for tokenising and processing.

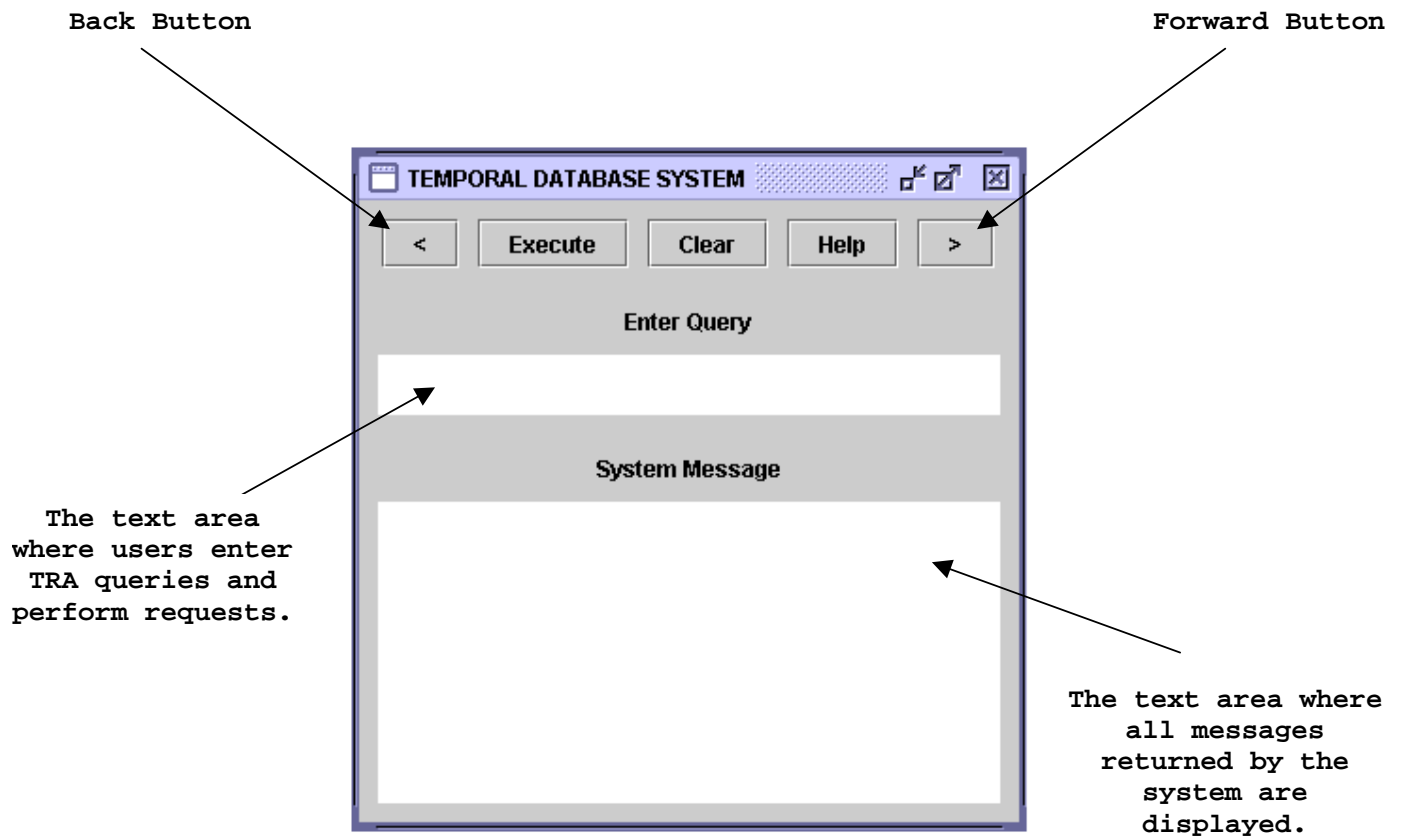
The following table describes the components in the user input interface:

Buttons	
Name	Description
Execute	Runs input that the user has entered in the 'Query' text area. Only evaluates non-empty input else the button has no effect.
Clear	Empties both text areas i.e. sets them as blank
Help	Displays the list of operators that the system can perform, how to use each operator and how to form valid queries.
Back '<' Forward '>'	Allows the user to navigate through the queries they have previously entered since the application was launched and displays them in the 'Query' text area.

Buttons are activated by clicking on them.

Text Areas	
Name	Description
Query	The area in which users enter requests (TRA queries). Accepts input from the keyboard. By pressing the <enter> key the same behaviour as the 'Execute' button is performed (runs a query).
Messages	Displays all messages returned by the system i.e. shows error messages.

The User Input Interface Diagram



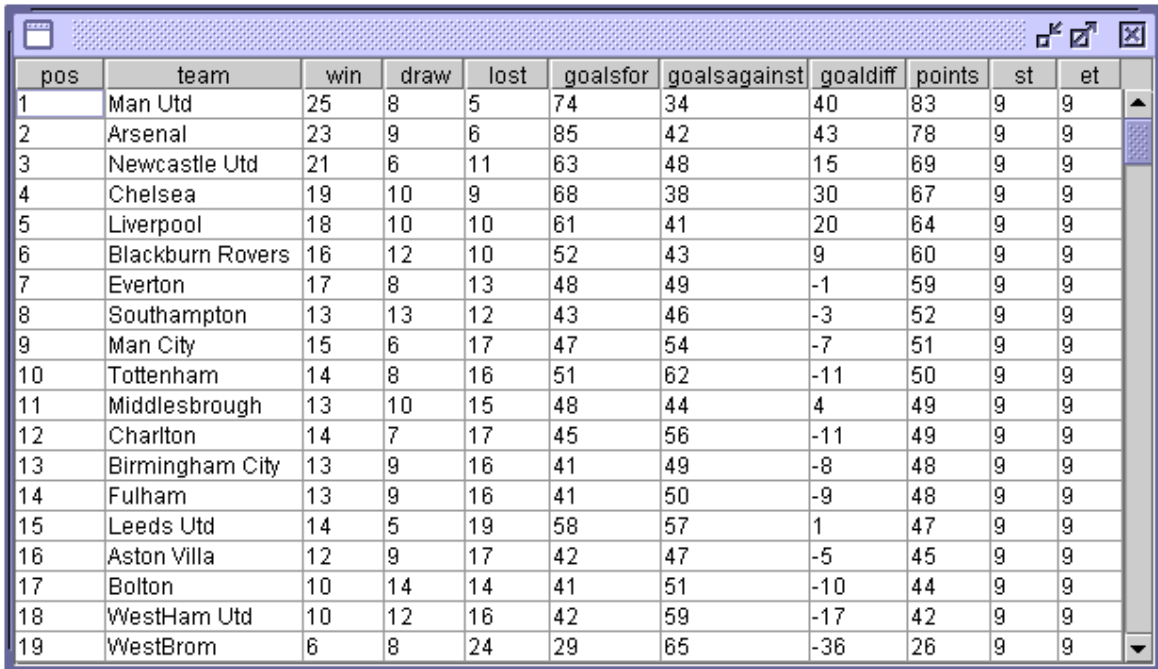
APPENDIX I - Output Interface

A graphical output interface has been implemented to return the results of a user request. The interface has been designed to display the results in a tabular format: column headings and temporal data of each tuple. Each tuple also displays its temporal objects (start and end time). When a TRA tree is evaluated by the TQP, the returning results of the root node are displayed using the output interface. All results are formatted as required before creating an instance of an output interface and then using it to display the temporal data. The output of results is returned in a separate window to that used for the user input interface.

Each time a new query is executed, a new output is created. The user must close the output interface of a query else it still remains viewable and present on the computer screen. This option of displaying results in a new output interface makes the system more flexible. A user can compare the output of two or more queries without simultaneously.

The output interface was implemented using Java Swing and JTable. A JFrame is created to hold an instance of a JTable that contains the results of a query.

The Output Input Interface Diagram



The screenshot shows a Java Swing window titled "Output" containing a JTable. The table displays football league data with 11 columns: pos, team, win, draw, lost, goalsfor, goalsagainst, goaldiff, points, st, and et. The data is sorted by position (pos) from 1 to 19. The table has a standard Java Swing look with a blue header and a scroll bar on the right.

pos	team	win	draw	lost	goalsfor	goalsagainst	goaldiff	points	st	et
1	Man Utd	25	8	5	74	34	40	83	9	9
2	Arsenal	23	9	6	85	42	43	78	9	9
3	Newcastle Utd	21	6	11	63	48	15	69	9	9
4	Chelsea	19	10	9	68	38	30	67	9	9
5	Liverpool	18	10	10	61	41	20	64	9	9
6	Blackburn Rovers	16	12	10	52	43	9	60	9	9
7	Everton	17	8	13	48	49	-1	59	9	9
8	Southampton	13	13	12	43	46	-3	52	9	9
9	Man City	15	6	17	47	54	-7	51	9	9
10	Tottenham	14	8	16	51	62	-11	50	9	9
11	Middlesbrough	13	10	15	48	44	4	49	9	9
12	Charlton	14	7	17	45	56	-11	49	9	9
13	Birmingham City	13	9	16	41	49	-8	48	9	9
14	Fulham	13	9	16	41	50	-9	48	9	9
15	Leeds Utd	14	5	19	58	57	1	47	9	9
16	Aston Villa	12	9	17	42	47	-5	45	9	9
17	Bolton	10	14	14	41	51	-10	44	9	9
18	WestHam Utd	10	12	16	42	59	-17	42	9	9
19	WestBrom	6	8	24	29	65	-36	26	9	9

APPENDIX J - Unary Operator Performance

Terms used in table showing the test results.

R: a relation

attributes: list of column names from relation.

time: time points (tick values).

constraints: single select statements or a combination of select statements formed using AND and OR.

The table shows the testing of queries constructed using the unary operators.

Unary Operator Queries	Average Execution time on Linux Systems (Seconds)	Average Execution time on Windows Systems (Seconds)
PROJECT <i>attributes</i> FROM R	0.56	0.51
R RENAME <i>attributes1</i> as <i>attributes2</i>	0.78	0.85
R WHERE <i>constraints</i>	1.45	1.89
R AT <i>time</i>	0.63	0.71
PAST R	0.78	0.86
PREVIOUS R	0.81	0.82
ALWAYSPAST R	0.83	0.91
FUTURE R	0.75	0.88
NEXT R	0.77	0.91
ALWAYSFUTURE R	0.83	0.93
Average time of Operators	0.82	0.92

The mean value for executing on Linux platform is 0.82 seconds.

The mean value for executing on Windows platform is 0.92 seconds.

APPENDIX K - Binary Operator Performance

Terms used in table showing the test results.

A and B: relations

The table shows the testing of queries constructed using the binary operators.

Binary Operator Queries	Average Execution time on Linux Systems (Seconds)	Average Execution time on Windows Systems (Seconds)
A PRODUCT B	1.35	1.56
A SINCE-PRODUCT B	1.59	1.78
A UNTIL-PRODUCT B	1.47	1.70
A JOIN B	2.03	1.98
A SINCE -JOIN B	2.22	2.24
A UNTIL-JOIN B	2.34	2.89
A UNION B	1.40	1.56
A MINUS B	1.67	1.89
Average time of Operators	1.76	1.95

The mean value for executing on Linux platform is 1.76 seconds.

The mean value for executing on Windows platform is 1.95 seconds.

Bibliography

Publications

- [1] Principles of Implementing Historical Databases in RDBMS, by P.J.McBrien.
- [2] Temporal Logic & Historical Databases, by P.J.McBrien and D.Gabbay.
- [3] Advanced Databases 2001 (Imperial Course Notes), by P.J.McBrien and J.McCann.
- [4] Temporal Databases: Theory Design and Implementation, by A.Tansel, J.Clifford, S.Jajodia, A.Segev and R.T.Snodgrass.
- [5] An Introduction to Database system, by C.J.Date.
- [6] A Guide to the SQL Standard, by C.J.Date.
- [7] Query Plans for Conventional and Temporal Queries Involving Duplicates and Ordering, by G.Slivinskas, C.S.Jensen and R.T.Snodgrass.
- [8] Adapting Query Optimisation and Evaluation in Temporal Middleware, by G.Slivinskas, C.S.Jensen and R.T.Snodgrass.
- [9] Core Java 2: Volume II Advanced Features, by Cay S. Horstmann and Gary Cornell.
- [10] Developing Java Software, by R.Winder and G.Roberts.
- [11] Database Programming with JDBC and Java (2nd ed.), by George Reese.

Websites

- [12] Java Technology - <http://java.sun.com>
- [13] JDBC - <http://java.sun.com/products/jdbc> and <http://java.sun.com/j2se/1.4/docs/guide/jdbc/getstart/GettingStartedTOC.fm.html>
- [14] PostgreSQL - <http://www.ie.postgresql.org>
- [15] TimeChain Technologies - <http://www.timechain.com>
- [16] Sorting Algorithms - <http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Sort/Quick>

- [17] Temporal Logic - <http://plato.stanford.edu/entries/logic-temporal>
- [18] Modal Logic - <http://plato.stanford.edu/entries/logic-modal> and
<http://www.doc.ic.ac.uk/~mrh/499/home.html>
- [19] TSQL - <http://www.cs.arizona.edu/people/rts>