Czech Technical University in Prague
Faulty of Electrical Engineering
Department of Computer Science and Engineering

Case Study: Implementation of Integrity Constraints in Actual Database Systems

Maedeh Sharif Khodaei

Master Study Program: Electrical Engineering and Information Technology M2612/2246. Specialization: Computer Science and Engineering

2007/2008

Supervisor: Ing. Michal Valenta, Ph.D.
Declaration

I, Maedeh Sharif Khodaei, confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g. ideas, equations, figures, text, tables, programmes) are properly acknowledged at the point of their use. A full list of the references employed has been included. I also agree that the department can use this thesis for non-commercial purposes in the future.

Signed:

Date: ................
Acknowledgements

First of all I would like to express my gratitude toward my supervisor, Ing. Michal Valenta PhD, who has helped and guided me through writing this thesis, I would also like to take this opportunity to thank my family for there continued support, financial and moral, throughout my year of study.

Furthermore, I would also like to thank my friends for their moral support throughout the duration of writing this thesis.
Abstract

The topic of this thesis is “Case study of: implementation of integrity constraints in actual Relational Database Systems”. This consists of designing a case application of a Library Management System and gives an overview of the integrity support in the previous SQL Standard SQL-92 and SQL: 99. It will also be shown as to what extent the different components of this standard can be found in major commercial relational database management systems; MySQL, Oracle, PostgreSQL, MS SQL and Firebird. Furthermore, this thesis will present general design guidelines showing how the integrity features provided by these systems should be utilized in order to implement an efficient integrity enforcing subsystem for a database. In this document only relational data model is discussed.

Abstrakt

Case Study: Implementation of Integrity Constraints in Actual Database System

Table of contents

TABLE OF CONTENTS

Declaration ........................................................................................................................ II
Acknowledgements ........................................................................................................... III
Abstract ........................................................................................................................... IV
Table of contents .............................................................................................................. V
List of Tables .................................................................................................................... X
List of Figures .................................................................................................................. X
Glossary of Terms and Abbreviations ............................................................................. XI
I. SUPPORTED THEORY ................................................................................................ 1
1. Introduction to Database Systems ................................................................................. 1
   1.1 Database Management System (DBMS) ................................................................. 2
   1.2 Relational Model ..................................................................................................... 3
      1.2.1 Relational database characteristics ................................................................... 4
   1.3 History of SQL ........................................................................................................ 5
      1.3.1 Advantages of SQL Databases ......................................................................... 7
      1.3.2 New Features of SQL-92 .................................................................................. 8
      1.3.3 New Features of SQL: 99 ................................................................................. 9
      1.3.4 New Features of SQL: 2003 ............................................................................. 11
2. Integrity support in SQL: 99 and commercial relational database management systems ................................................................................................................. 12
   2.1 Integrity Support in SQL: 99 ................................................................................. 12
      2.1.1 Referential Integrity - Referential Constraint Actions .................................... 14
      2.1.2 More about Referential Integrity Constraints ................................................. 16
   2.2 Stored Procedures, Trigger and User-Defined Functions ..................................... 16
      2.2.1 Stored Procedures ........................................................................................... 18
      2.2.2 User-Defined Functions ................................................................................. 18
      2.2.3 Triggers .......................................................................................................... 19
   2.3 Privileges, users, and security ............................................................................... 24
   2.4 data definition fundamentals ................................................................................. 26
      2.4.1 Schema and Catalogs ...................................................................................... 27
2.5 Some SQL: 99 DATA TYPES.............................................................................. 27
II. APPLICATION – A CASE STUDY .......................................................................... 29
3. LIBRARY MANAGEMENT SYSTEM.................................................................... 29
  3.1 Introduction ........................................................................................................... 29
  3.2 Application ............................................................................................................ 29
  3.3 CONCEPTUAL DESIGN: .................................................................................... 30
    3.3.1 Table relationships ......................................................................................... 31
    3.3.2 E-R Model ...................................................................................................... 32
  3.4 Entity description .................................................................................................. 33
4. SQL: 99 constraint support matrix .............................................................................. 36
  4.1 Check constraint and Assertion Limitations: ........................................................ 39
  4.2 Triggers: ................................................................................................................ 40
  4.3 Trigger on Views: .................................................................................................. 43
  4.4 Event schedulers: ................................................................................................... 43
  4.5 Automatic key generation: .................................................................................... 43
III. Comparison of Integrity Support in different Commercial Database Management Systems: ........................................................................................................................... 45
  5. Common Language for All Relational Databases ....................................................... 46
    5.1 Introduction to Oracle SQL ................................................................................... 46
    5.2 History of Oracle and Standard compliance ....................................................... 46
    5.3 Oracle Extensions to Standard SQL ...................................................................... 47
      5.3.1 Additional Parts of Standard Commands ....................................................... 47
    5.4 Oracle constraint support matrix ........................................................................... 47
    5.5 How Oracle Enforces Data Integrity ..................................................................... 48
      5.5.1 Nullability and Default Values ....................................................................... 48
      5.5.2 Unique/primary key constraint ....................................................................... 50
      5.5.3 CHECK Constraints ....................................................................................... 51
    5.6 Foreign Key Constraints and Restrictions ............................................................. 52
    5.7 Assertion: ............................................................................................................... 54
    5.8 Comparison of Trigger Features ............................................................................ 55
      5.8.1 Restrictions on Trigger Conditions ................................................................. 56
      5.8.2 Restrictions on BEFORE Triggers ................................................................. 57
      5.8.3 Restrictions on AFTER Triggers ................................................................. 58
    5.9 Trigger on view ..................................................................................................... 58
      5.9.1 Restrictions on INSTEAD OF Triggers ........................................................... 59
Case Study: Implementation of Integrity Constraints in Actual Database System

5.10 Automatic key generation .......................................................................................... 59
5.11 Oracle scheduler ...................................................................................................... 60
  5.11.1 The Database Scheduler .................................................................................. 60
6. Overview of the MySQL Database Management System ........................................... 62
  6.1 Introduction to MySQL ......................................................................................... 62
  6.2 The Main Features of MySQL .............................................................................. 62
  6.3 MySQL Standards Compliance ............................................................................. 63
  6.4 MySQL Differences from Standard SQL .............................................................. 63
  6.5 MySQL constraint support matrix ......................................................................... 64
    6.5.1 SQL Modes: ................................................................................................... 65
    6.5.2 Nullability and Default Values ...................................................................... 66
    6.5.3 PRIMARY KEY / UNIQUE Index Constraints ............................................. 66
    6.5.4 Constraints on Invalid Data ........................................................................ 67
    6.5.5 CHECK Constraints ..................................................................................... 69
    6.5.6 Foreign Key Constraints and Restrictions ................................................... 72
  6.6 MySQL Triggers .................................................................................................... 75
    6.6.1 Trigger Limitations ....................................................................................... 76
    6.6.2 Trigger Error Handling ................................................................................. 76
  6.7 Event Scheduler Overview ..................................................................................... 78
7. Overview of the Microsoft SQL Server Database Management System .................... 79
  7.1 Introduction to MS SQL ....................................................................................... 79
    7.1.1 Key Capabilities ......................................................................................... 79
  7.2 MS SQL Constraint Support ............................................................................... 80
    7.2.1 Primary Key Constraints .............................................................................. 81
    7.2.2 Unique Constraints ...................................................................................... 81
    7.2.3 Null Constraint ........................................................................................... 82
    7.2.4 Default Constraints ...................................................................................... 82
    7.2.5 Check Constraints ....................................................................................... 83
  7.3 Automatic key generation ..................................................................................... 84
    7.3.1 Identity Columns ......................................................................................... 84
  7.4 Foreign key and Referential Action ...................................................................... 85
    7.4.1 The Delete Rule ............................................................................................ 85
    7.4.2 The Update Rule .......................................................................................... 86
    7.4.3 SET NULL and SET DEFAULT ................................................................. 86
7.5 Triggers ............................................................................................................... 87
  7.5.1 FOR| AFTER Triggers ................................................................................... 88
  7.5.2 Inserted and deleted tables ........................................................................ 89
  7.5.3 INSTEAD OF Triggers ................................................................................. 91
  7.5.3 MS SQL Multiple Triggers ......................................................................... 91
  7.5.4 Trigger Limitations ..................................................................................... 92
  7.5.5 Downsides of Triggers .............................................................................. 92

8. Overview of Firebird Relational Database System .............................................. 93
  8.1 Introduction to Firebird 2.0 .......................................................................... 93
  8.2 SQL standard compliance .......................................................................... 93
  8.3 Important Features ....................................................................................... 93
  8.4 Firebird SQL ................................................................................................. 94
  8.5 Firebird integrity support ............................................................................ 94
    8.5.1 Domain support ...................................................................................... 95
    8.5.2 Null support .......................................................................................... 95
    8.5.3 Primary keys ......................................................................................... 95
    8.5.4 Unique Keys ......................................................................................... 96
    8.5.5 Foreign key and referential action ......................................................... 96
    8.5.6 Check constraint .................................................................................. 97
  8.6 triggers ........................................................................................................... 98
    8.6.1 General trigger Syntax .......................................................................... 98
    8.6.2 Context Variables ................................................................................ 98

9. Overview of the PostgreSQL Database Management System ................................ 99
  9.1 Introduction to PostgreSQL ....................................................................... 99
  9.2 Standard compliance .................................................................................. 99
  9.3 Data types ................................................................................................... 100
  9.4 Inheritance .................................................................................................. 100
  9.5 Constraint support matrix .......................................................................... 101
    9.5.1 Null/Default Values .......................................................................... 102
    9.5.2 Check Constraints .............................................................................. 104
    9.5.3 Unique/ primary Constraints .............................................................. 104
    9.5.4 Foreign Keys and Referential Action Support: .................................. 105
  9.6 Triggers ....................................................................................................... 107
    9.6.1 Compatibility with Standard SQL ....................................................... 107
Case Study: Implementation of Integrity Constraints in Actual Database System

9.6.2 BEFORE/ AFTER Trigger behaviour .......................................................... 109
9.7 Automatic Key-Generation ................................................................. 110
9.9 Event scheduler ............................................................... 110
10. Conclusion ......................................................................................... 111
References .............................................................................................. 113
List of Tables

<table>
<thead>
<tr>
<th>Table Name/ Number</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2.1 SQL: 99 trigger action</td>
<td>21</td>
</tr>
<tr>
<td>Table 2.2 SQL: 99 referencing clause</td>
<td>22</td>
</tr>
<tr>
<td>Table 2.3 SQL: 99 referencing clause</td>
<td>22</td>
</tr>
<tr>
<td>Table 2.4 Kinds of Protection</td>
<td>25</td>
</tr>
<tr>
<td>Table 4.1 Comparison of declarative integrity support</td>
<td>36</td>
</tr>
<tr>
<td>Table 4.2 Not null constraint</td>
<td>38</td>
</tr>
<tr>
<td>Table 4.3 Unique/ primary key constraint</td>
<td>38</td>
</tr>
<tr>
<td>Table 4.4 Triggers</td>
<td>40</td>
</tr>
<tr>
<td>Table 5.1 Declarative integrity constraint</td>
<td>48</td>
</tr>
<tr>
<td>Table 5.2 Not null constraint</td>
<td>50</td>
</tr>
<tr>
<td>Table 5.3 Unique/ primary key constraint</td>
<td>51</td>
</tr>
<tr>
<td>Table 5.4 Triggers</td>
<td>55</td>
</tr>
<tr>
<td>Table 6.1 Declarative integrity support</td>
<td>64</td>
</tr>
<tr>
<td>Table 6.2 Not null constraint</td>
<td>66</td>
</tr>
<tr>
<td>Table 6.3 Unique/ Primary key constraints</td>
<td>67</td>
</tr>
<tr>
<td>Table 6.4 Triggers</td>
<td>75</td>
</tr>
<tr>
<td>Table 7.1 Declarative integrity support</td>
<td>80</td>
</tr>
<tr>
<td>Table 7.2 Triggers</td>
<td>87</td>
</tr>
<tr>
<td>Table 9.1 Declarative integrity support</td>
<td>102</td>
</tr>
<tr>
<td>Table 9.2 Not null constraints</td>
<td>104</td>
</tr>
<tr>
<td>Table 9.3 Triggers</td>
<td>107</td>
</tr>
</tbody>
</table>

List of Figures

<table>
<thead>
<tr>
<th>Figure Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1 Trigger execution &amp; constraint checking</td>
<td>24</td>
</tr>
<tr>
<td>Figure 2 Abstraction in the SQL database environment</td>
<td>27</td>
</tr>
<tr>
<td>Figure 3.1 E-R Model</td>
<td>116</td>
</tr>
</tbody>
</table>
Glossary of Terms and Abbreviations

DBMS – Database Management System
ACID – Atomicity, Consistency, Isolation, Durability
RDBMS – Relational Database Management System
IC – Integrity Constraint
SQL – Structured Query Language
ANSI – American National Standards Institute
ISO – International Standards Organization
XML – Extensible Mark-up Language
OLAP – Online Analytical Processing
PL/SQL – Procedural Language/Structured Query Language
DB – Database System
MS SQL – Microsoft SQL
OS – Operating System
DBA – Database Administration
SPL – Stored Procedure Language
DDL – Data Declaration Language
ERM – Entity-Relationship Model
ERD – Entity-Relationship Diagram
DML – Data Manipulation Language
ODBC – Open Database Connectivity
CPU – Central Processing Unit
BLOB – Binary large object
PL/pgSQL – Procedural Language/PostgreSQL Structured Query Language
API – Application Program Interface

XI
I. SUPPORTED THEORY

1. Introduction to Database Systems

Today, more than at any previous time, the success of an organization depends on its ability to acquire accurate and timely data about its operations, manage this data effectively and to use it to analyze and guide its activities. The sheer magnitude of data and information that an organization will have and use can be completely overwhelming and the value of this data, as an organizational asset, is widely recognized. Yet without the ability to manage this vast amount of data and to quickly find the information that is relevant to a given question, as the amount of information increase, it tends to become a distraction and a liability rather than an asset. This paradox drives the needs for increasingly powerful and flexible data management systems. To get the most out of their large and complex datasets, users must have tools that simplify the tasks of managing the data and extracting useful information in a timely fashion. Otherwise, data can become a liability with the cost of acquiring it and managing it far exceeding the value that is derived from it.

A database is a collection of data pertaining to some well-defined purpose such that a computer program can quickly select desired pieces of data. A Relational Database is typically made up of many linked tables of rows and columns. In order to store data in a database, it is required that certain rules must be satisfied, namely [1]:

- Having a known format defined by the metadata describing the data that the database contains.
- Data must be stored, retrieved and modified only by a special type of computer program - the Database Management System.
- The data should be under transaction control. That is a formal set of rules and guidelines must ensure that integrity is maintained during and after operations on data.
1.1 Database Management System (DBMS)

A database management system or DBMS is software designed to assist in maintaining and utilizing a large collection of data. The need for such systems, as well as their use, is growing rapidly. A database management system should include [2]:

1. A modeling language to define the schema of each database hosted in the DBMS, according to the DBMS data model:
   - The four most common types of organizations are the hierarchical, network, relational and object-models. A given database management system may provide one or more of the four models. The optimal structure depends on the natural organization of the applications requirements.

2. Data structures optimized to deal with very large amounts of data stored on a permanent data storage device.

3. A database query language to allow users to interactively interrogate the database, analyze its data and update it according to the user privileges on data.
   - It also controls the security of the database.
   - Data security prevents unauthorized users from viewing or updating the database. Using passwords, users are allowed access to the entire database or subsets of it.
   - If the DBMS provides a way to interactively enter and update the database, as well as interrogate it, this capability allows for the managing of personal databases. However, it may not leave an audit trail of actions or provide the kind of controls necessary in a multi-user organization. These controls are only available when a set of application programs are customized for each day entry and updating function.

4. A transaction mechanism that ideally would guarantee the ACID properties (Atomicity, Consistency, Isolation, Durability), in order to ensure data integrity, despite concurrent user accesses, and faults.
   - It also maintains the integrity of the data in the database.
   - The DBMS can maintain the integrity of the database by not allowing more than one user to update the same record at the same time. The DBMS can help
prevent duplicate records via unique index constraints. One of the main advantages of DBMS is that it protects data from many different types of ill-formed operations, by applying transaction semantics to the operations.

1.2 Relational Model

Some important data models that are used in a database management system are hierarchical model, network model and the object models. Although there are many databases that use these models, the dominant model today is the Relational model. Edgar Frank Codd was a British computer scientist who made seminal contributions to the theory of relational databases model in 1970. At that time most database systems were based on one of two older data models (the hierarchical model and the network model); the relational model revolutionized the database fields and largely supplanted these earlier models. Prototype relational database management systems were developed through pioneering research projects at IBM and UC-Berkeley by the mid-70s, and several vendors were offering relational database products shortly thereafter. Today the relational model is by far the dominant data model and is the foundation for the leading DBMS products.

The relational model is very simple and elegant; a database is a collection of one or more relations, where each relation is represented by tables. This simple tabular representation enables even novice users to understand the content of a database and it permits the use of simple, high-level language to query the data.

The major advantages of the relational model over the older data models are its simple data representation and the ease with which even complex queries can be expressed. The relational model describes the data with its natural structure, without adding any additional structure for machine representation or implementation purposes. It provides a mathematical basis for the treatment of derivability, redundancy and consistency of relations. It also provides independence of the data from physical representation of the data, of the relationship between the data and of the implementation considerations related to efficiency and like concerns [3].

The relational model consists of two components:

- Relation schema(table definition)
The main construct for representing data in the relational model is a relation. A relation is a set of n-tuples, consisting of a relation schema and a relation instance. The relation instance is a table and the relation schema describes the column heads for the table. A database is only as good as the information stored in it and a DBMS must therefore help prevent the entry of incorrect information. An integrity constraint is a condition that is specified on a database schema and restricts the data that can be stored in an instance of the database. If a database instance satisfies all the integrity constraints specified on the database schema, it is a Legal instance. A DBMS enforces integrity constraints, in that it permits only legal instances to be stored in the database. Integrity constraints are specified and enforced at different times [4]:

- When an end user defines a database schema, he/she specifies the ICs that must hold on any instance of this database.
- When a database application is run, the DBMS checks for violations and disallows changes to the data that violates the specified ICs.

Many kinds of integrity constraints can be specified in the relational model:

1. Domain constraints
2. Key constraints
3. Referential integrity constraints
4. NOT NULL constraints
5. General assertions

### 1.2.1 Relational database characteristics

Relational databases offer a number of characteristics [5]:

- The relational model allows for the linking of different data tables using “keys” (primary and foreign keys), using the structured query language (SQL).
- Efficient storage of data. In a database design that adheres to the rules of the relational model, each data item is stored only once, that is in one location. Consequently, each data item can be changed, inserted or removed in only one location in the database.
• Using a relational database we can specify what kind of data a database column is allowed to contain. We can set data fields, numeric fields, text fields, etc. this gives us control over data integrity.
• Data integrity. By setting field properties, linking tables and applying data integrity rules we can increase the reliability of the data.
• Most relational database systems offer a right structure with which rights can be assigned to different users. Some of the operations that can be allowed, or disallowed, to a user are SELECT, INSERT, DELETE, ALTER, CREATE, etc. these rights correspond to the operations that can be performed using the SQL.
• In order to actually perform operations on the database, like storing new data, selecting present data and altering present data, SQL queries are used.

The relational model is a standard. By adhering to the rules of the relational model we ensure that our data can be transferred to other relational database systems relatively easy.

1.3 History of SQL

After Dr Codd published his seminal paper on the Relational model, research began at the IBM San Jose research laboratory on a language to implement that model; the language was called SEQUEL for Structured English Query Language. After a couple of years, a revised version called SEQUEL/2 was defined; the name of this version was later changed to SQL [6].

SQL is a data sublanguage for access to the relational database that is managed by the relational database management system (DBMS). Data sublanguage is one that is used in association with another language for specialized purpose of accessing data. The sorts of programming languages with which most people are familiar include FORTRAN, C, etc. These languages were designed to allow Application programmers to express their computational requirements efficiently.

None of these languages were designed specifically for manipulating data stored under the control of the database management system. However, a data language by itself is insufficient for writing real applications. In almost all cases, an application has a mixture
of the requirements: perform some calculations, manipulate some information and manage some data. In this common sort of application it is usually helpful if the application writer can build the calculation or manipulate portions of the application by using a language well-suited to the purpose, reverting to a specialized data language only for those parts of the application that require it. In this case the data language is often viewed as a data sublanguage with respect to the primary programming language. There are many relational data sublanguages to choose from, but only one has been formally standardized for access to the relational database: SQL. Other relational languages have been defined and implemented. Some of these have been commercial successes, but none of them has had as wide a following as SQL.

SQL was the query language of the pioneering system-R relational DBMS. Over the years, SQL has become the most widely used language for creating and manipulating, and querying relational DBMSs. Since many vendors offer SQL products, there is a need for standard that defines ‘official SQL’. The existence of a standard allows users to measure a given vendor’s version of SQL for completeness, it also allows users to distinguish SQL features that are specific to one product from those that are standards; an application that relies on non-standard features is less portable. The first SQL standard was developed in 1986 by the American National Standards Institute (ANSI), and was called SQL-86. There was a minor revision in 1989 called SQL-89, and a major revision in 1992 called SQL-92.

SQL-86 and SQL-89 were rightfully criticized for being incomplete. In fact, the language specified by those standards lacked many basic necessities for a commercial database system. Neither standard allowed an application to alter the definition of a database once it was defined. Users could not create new tables, add new user privileges or delete columns from tables. Other features that were widely implemented but in varying ways and thus were omitted from SQL-86 and SQL-89 include: dynamic SQL, system tables, a variety of set operations and extended error-handling facilities. The International Standards Organization (ISO) collaborated with ANSI to develop SQL-92. Most commercial DBMSs currently support SQL-92. SQL-92 has resolved many of the weak points found in the earlier versions of the SQL standards [7].
SQL-92 significantly increases the size of the original 1986 standard to include a schema manipulation language for modifying or altering schemas, schema information tables to make schema definitions accessible to users, new facilities for dynamic creating of SQL Statements and new data types and domains. Other new SQL-92 features include outer join, cascade update and delete referential actions, set algebra on tables, transaction consistency levels, scrolled cursors, deferred constraint checking and greatly expanded exception reporting. SQL-92 also removes a number of restrictions in order to make the language more flexible and orthogonal.

The work does not end there, in 1999; the work was completed on the latest version of the SQL standard, known as SQL:99, a major extension of SQL-92. The major features that are introduced in SQL-99 are regular expression matching, recursive queries, triggers, non-scalar types and some object-oriented features. Recently, in 2003, ANSI/ISO released the SQL-2003 standard also called SQL-200n. The important features of SQL-2003 are: more collection data types, cleaner object/relational specification and references to new parts such as XML. The big missing SQL-2003 feature is the SQL-99 standard BIT data type. Like most languages, SQL has some weak points and some very strong points.

1.3.1 Advantages of SQL Databases

- **High speed**
  SQL queries are designed to retrieve large amount of records from a database quickly and efficiently. With the help of simple SQL queries, you can retrieve even highly complicated combinations of data from the database. SQL databases are also much more adept at handling very large volumes of data and processing it quickly as compared to non-SQL databases.

- **Security**
  With the SQL database storage; there is the added security of all the data being in one place, which can be managed more easily.

- **Well-defined standards exist**
SQL database use the long-established SQL standard, which has been adopted by ANSI, the latest being SQL-2003. On the contrary non-SQL databases do not have clear standards to adhere.

- **Compatibility**
  Since well-defined and established standards exist and if the databases adhere to those, then portability from one SQL database to another is a trivial matter.

- **No coding required**
  Using standard SQL, it should be easier to move applications between different database systems without the need to rewrite a substantial amount of code [8].

### 1.3.2 New Features of SQL-92

The addition of all of these new features after SQL-89 makes the resulting standard much larger. SQL-92 runs about 580 pages in length, while SQL-89 is only 115 pages. A careful consideration of the actual content of the languages, adjusted to reflect the SQL-92 features that were already implemented by virtually every product and the rather more careful and complete specification of SQL-89 features in the SQL-92 document, suggests that the actual language is between 1.5 and 2 times as large as a realistic SQL-89. Pressure to keep the size of SQL-92 from being even larger than it was caused the deferral of several features such as triggers. These new features are [9]:

- Increased generality and orthogonality in the use of scalar-valued and table-valued query expressions.
- Support for capabilities (especially DROP, ADD, and ALTER statements) to define and redefine tables.
- Additional referential integrity facilities (including referential options), subqueries in CHECK constraint, separate assertion and user-controlled deferral of constraints.
- Support for dynamic execution of SQL statements.
- Improved diagnostic capabilities, in particular, new status parameters (SQLSTATE), a diagnostic area and supporting statements.
- Support for data type conversions.
• Additional data types (DATE, TIME, TIMESTAMP, BIT string, variable-length character and bit string and NATIONAL CHARACTER strings).
• Support for certain facilities required for remote database access (RDA), notably connection management statements and qualified schema names.
• Capabilities for domain definitions in the schema.
• Support for additional scalar operations, such as string operation for concatenation and substring, date and time operations and a form of conditional expressions.
• Additional set and join operators (for example union join, natural join, set difference, and set insertion).
• Support for scrolled cursors.
• A requirement for a flagging capability to aid in portability of application programs.
• Support for character sets beyond the one required to express the SQL language itself, support for explicit collation.
• Support for transaction consistency levels.
• Support for temporary tables.
• Additional security features.
• Additional programming language support.
• A better definition for direct invocation of the SQL language.

1.3.3 New Features of SQL: 99

SQL-92 resolved many of the weak points found in earlier versions of the SQL standard, but it was far from perfect; the subsequent work that was incorporated into SQL: 99 goes further in bringing the SQL standard in line with real-world database capabilities and user needs. Unlike SQL-92 and earlier versions of the SQL standard, SQL: 99 are not available in hardcopy form (printed on paper) from ISO or from ANSI. Instead is available from ISO and ANSI only in electronic or machine readable form. Syntax of SQL: 99 are significantly larger than SQL-92 syntax. Now let’s look at the new features offered by SQL: 99 [10]: 
• Object orientation.
• Triggers (Enhances integrity mechanism).
• Referential action NO ACTION.
• Recursive queries.
• New data types: BOOLEAN, ARRAY, ROW, CHARACTER LARGE OBJECT
  and BINRAY LARGE OBJECT, as well as the use of LOCATOR types to
  reference the LARGE OBJECT types.
• Distinct types.
• Use of Unicode as the character model, accompanied by several new named
  character sets: UCS2, UTF8, UTF16 and ISO10646.
• Use of functional dependencies to permit updating for more views.
• The addition of the keyword SENSITIVE for cursors that are neither
  SENSITIVE nor INSENSITIVE.
• Holdable cursors.
• Roles (Enhanced security mechanisms).
• Save points (Enhances user-controlled integrity).
• START TRANSACTION statement.
• The AND CHAIN option on COMMIT and ROLLBACK.
• SET SESSION CHARACTERISTICS statement.
• Bracketed comments (/*….*/).
• Additional build-in functions (CARDINALITY, ABS, MOD, OVERLAY,
  LOCALTIME, LOCALTIMESTAMP).
• Additional grouping options (ROLLUP, CUBE, GROUPING SETS).
• SIMILAR and DISTINCT predicates.
• Additional views in the information schema.
• Short-name views in the information schema.
• Representation of SQLs built-in functions in the information schema.
• A new conformance model for the SQL standard.
1.3.4 New Features of SQ: 2003

As the latest version of SQL standards, SQL 2003 is making major improvements in a number of key areas. Firstly, there are additional object relational features, which were first introduced in SQL 1999. Secondly, SQL 2003 standard revolutionizes SQL with comprehensive OLAP features. Thirdly, SQL 2003 delivers a brand new Part 14 for XML-Related Specifications (SQL/XML) to integrate popular XML standards into SQL. Finally, there are numerous improvements throughout the SQL 2003 standard to refine existing features. The SQL: 2003 standard is not freely available; it may be purchased from ISO or ANSI. Since the SQL: 2003 standard is relatively new; few DBMS implementations are able to claim full compliance with it yet. However, most vendors have partial compliance with the core SQL: 2003 features and sometimes have legacy, non-standard syntax equivalents that they had introduced prior to the SQL: 2003 standard. The SQL: 2003 standard makes minor modifications to all parts of SQL: 1999 (also known as SQL3) and officially introduces a few new features such as [11]:

- XML-related features.
- Window functions.
- The sequence generator, which allows standardized sequences.
- Two new column types: auto-generated values and identity-columns.
- The new MERGE statement.
- Extensions to the CREATE TABLE statement, to allow "CREATE TABLE AS" and "CREATE TABLE LIKE".
- Removal of the poorly-implemented "BIT" and "BIT VARYING" data types.
2. Integrity support in SQL: 99 and commercial relational database management systems

Integrity constraint specifications are typically translated into a constraint enforcing mechanism provided by the database management system used to implement the database. A database integrity constraint can be exploited for query optimization purposes and thus can lead to better performance for query evaluation. Integrity constraints have been studied from various aspects since the introduction of the relational data model [12]. Firstly, a profound overview of integrity support in the standard SQL: 99 shall be presented, then a comparison of major commercial database systems.

There is no such thing as a “perfect match” between the SQL standard and the features and syntax of a commercial RDBMS product. All of these products have, over the years, closed the gap between their respective syntactical constructs and that contained in the SQL standard. It is important to note that there is not, nor will ever be, a perfect match between the syntax of any of these products and that contained in the SQL standard. Therefore, it is very important that the syntax and behaviour of both the SQL standard and these individual products are understood.

The major similarities and differences among these systems with respect to the definition, semantic and actual enforcement of various classes of constraints at database run-time will be discussed. Therefore, some major guidelines and rules for designing and implementing integrity constraints using diverse type of constraint enforcing mechanisms, up to reactive constraint enforcement using triggers will be provided.

2.1 Integrity Support in SQL: 99

The databases are only as good as the integrity of the data contained within their tables and records. That is the reliability and consistency of the data is critical to our application, decision support and query tools and other aspects of information system environment that utilize the database. Outright errors in the data and incorrect reports can lead the database users to lose confidence in the overall database environment.
The integrity of the data can be ensured by applying constraints, or rules, to the structure of the database and its contents. There are two primary methods by which these constraints can be specified, applied and enforced [13]:

- Through the application programs and software systems that use the database.
- Through the database management system itself.

There are a couple of major advantages to database-enforced constraints as compared to those managed through procedural application code: Since tables in a relational database are used by many different applications, each and every constraint must be specified – and modified, whenever necessary – in each application program when the application–enforced model is used. In contrast, database-enforced constraints can usually be designated in a single location and still apply to all necessary applications. Application-enforced constraints are usually more complex to code and manage than database-enforced constraints are.

Every integrity constraint is associated with a descriptor consisting of the name of the constraint, initial checking mode and a flag indicating whether or not the checking of the constraint can be deferred until the end of the transaction [14].

1. Integrity constraints are schema objects such as tables or views. Within a database schema, an integrity constraint is uniquely identified by its name. If a name is not specified explicitly, then the system will implicitly provide an implementation-dependent name.

2. The checking mode determines the relative time when the integrity constraint has to be checked within a transaction. Two modes are distinguished: immediate and deferred.

   If the mode is immediate, then the integrity constraint is checked at the end of each database operation (that might violate the integrity constraint). Otherwise, the checking is deferred until the end of the transaction.

3. The initial checking mode defines the default mode that is valid at the beginning of each transaction. Only deferrable constraints can be set to deferred mode. The checking mode of a non-deferrable constraint is always immediate. The modes initially immediate and non-deferrable are implicit. If initially deferred is
specified, the \textit{non-deferrable} shall not be specified and thus, deferrable is implicit.

SQL: 99, like SQL-92 before it, allow us to declare that our constraints are deferrable. SQL: 99 provide the following keywords for declaratively specifying integrity constraints:

- \textbf{NOT NULL}: by default a column can hold null values; using constraint NOT NULL prevents a column from taking the null value.
- \textbf{DEFAULT}: sets the default value of a column.
- \textbf{UNIQUE}: defines that a column or set of columns must have unique values within a table.
- \textbf{PRIMARY KEY}: specifies the primary key of a table used to uniquely identify each row in a table.
- \textbf{FOREIGN KEY}: defines a foreign key whose values must match to the values of a unique/primary key, ensuring referential integrity of the data.
- \textbf{CHECK}: defines a general integrity constraint that must hold for each row of a table, ensuring that all values in a column satisfy certain conditions.
- \textbf{DOMAIN}: creates a (restricted) column domain.
- \textbf{ASSERTION}: defines a named, general integrity constraint that may refer to more than one table.

\textbf{2.1.1 Referential Integrity - Referential Constraint Actions}

Referential integrity is the state of a database in which all values of all foreign keys are valid. A foreign key is a key that is part of the definition of a referential constraint. A Referential Constraint is the rule that the values of the foreign key are valid only if:

- They appear as values of a parent key, or
- Some component of the foreign key is null.

The base table containing the parent key is called the \textit{parent table} of the Referential Constraint and the base table containing the foreign key is said to be a \textit{dependent} table. Referential Constraints can be defined in \texttt{CREATE TABLE} statements and \texttt{ALTER}
TABLE statements. Referential integrity reflects the purpose of constraints whose job it is to ensure that values stored in one place in our database reference existing data in some other place in the database. That is, references to data must be valid.

Through the use of foreign keys, which typically are used in conjunction with primary keys but may also be coordinated with non primary candidate keys if desired, several different types of referential integrity constraints may be specified at data definition time. These include cascading of data value modifications to related tables, automatically deleting certain logically linked rows, and setting various default or null values upon certain actions. When specifying a foreign key, we wish to prohibit the execution of certain SQL statements that might violate the referential integrity constraints; this feature is called referential action.

SQL: 99 allow defining different referential actions to react on updates and deletions performed on the referenced tables [15]:

- **SETTING DEFAULT**: the update/deletion of a referenced row requires that the foreign keys of all rows that reference this row are set to their default values.
- **SET NULL**: sets all the referencing values to NULL.
- **CASCADE**: update/deletion of a referenced row leads to an update/deletion of all rows that reference this row.
- **NO ACTION**: disallows the update/deletion of the row if it is referenced by a row in any table. This referential constraint is checked after all other referential constraints have been processed.
- **RESTRICT**: this referential constraint is checked before other referential constraints are checked.

All of these constraints shall be covered in the application system and looked at in more detail throughout this thesis.
2.1.2 More about Referential Integrity Constraints

By showing SQL’s expectation of statement behaviour, a better understanding of how referential integrities work shall be gained. Although various DBMS products might implement this in a different way, the end result will be the same. We shall think of every SQL statement as having three phases: a setup phase, an execution phase and a cleanup phase. During the setup phase the DBMS first identifies every row that will be directly affected by the statement and then identifies every matching row for every directly affected row (A matching row is a row in a referencing table whose foreign key values identify some directly affected row in the referenced table). During the execution phase the DBMS performs the action of the statement on the directly affected rows. No new rows are ever added to the list of matching rows as a result of any direct actions. During the cleanup phase, the referential actions are performed. In general, when a directly affected row causes some referential action, that action is performed, and any referential actions caused by that action are then performed. Although this might sound like a depth-first-search, referential integrity actions have been very carefully designed to allow them to be performed in any order at all to arrive at a correct, consistent state of the database [16].

2.2 Stored Procedures, Trigger and User-Defined Functions

By definition SQL is a nonprocedural programming language. Meaning that it lacks procedural constructs, specifically, the ability to perform operations in optionally named hierarchical logical blocks that can accept and return values, perform iterations, execute conditional statements and so on. The first RDBMS implementations did not have procedural language capabilities; all procedural database processing was done using embedded programming. All major procedural languages that were popular back then (C, COBOL, Pascal, etc.) had special extensions that allow the programmer to embed SQL statements directly into programming language code. The work of extensions is to translate SQL into appropriate language constructs that can be later compiled into binary code. However, while relational databases became increasingly sophisticated and more internal control was delegated to RDBMS, the idea arose to store procedural programming modules inside RDBMS in compiled form. The problem was that
ANSI/ISO standards did not specify any guidelines for these elements. As a result, each vendor implemented its own version of internal RDBMS procedural modules.

SQL: 99 added persistent stored routines and triggers to SQL standards when all major RDBMS vendors already had their own procedural languages. Oracle's procedural SQL extension is called PL/SQL; MS SQL Server uses Transact-SQL; and DB2 introduced its own version that does not seem to have an official name. Even though the basic syntax elements of these languages are similar, the advanced features differ significantly. The two main forms of RDBMS procedural routines are stored procedures and triggers that embody two different procedural programming approaches - linear and event-driven, correspondingly. A user-defined function can be envisioned as a special case of a stored procedure and a module is just a number of stored procedures and functions bundled together.

Stored procedures, user-defined functions and triggers can be used in many different ways and for many different reasons. For example, inserting or deleting table rows, updating certain columns, users logging in and out can trigger other events in a database. Some of their usages are:

- Stored procedures, functions and triggers can be used for database security purposes. A stored procedure (or function) is a separate database object with its own database privileges. That means we can design a stored procedure in such way that it would, for example, update only certain columns; or insert rows with NULL values for columns that a user who executes the procedure has no permission to update – the user would then only need the privilege to execute that particular stored procedure.

- Triggers are even more useful for security implementation. For example, they can be designed in such way that certain actions performed by users on certain objects are written to special database tables or OS files. These records can be reviewed later.

- Another important thing about stored routines is code reusability – once compiled, a stored procedure or user-defined function can be used over and over again by
multiple users (or applications), saving time on retyping large SQL statements and reducing the probability of human errors. Also, when a persistent module needs to be changed, the change won't affect the client programs that access it, as long as all the calling parameters remain the same [17].

2.2.1 Stored Procedures

Stored procedures are linear or sequential, their syntax varies from implementation to implementation but some common features can be emphasized. Stored procedures can accept parameters and allow local variable declaration; they are structured and allow the use of sub models; also they allow repeated and conditional statement execution [17]. A set of stored procedures can be implemented in such a way that every select, update and delete would go through the stored procedures. Business security rules that govern the way data is inserted, queried, updated or deleted can be implemented inside stored procedures. Stored procedures can even be used to create database objects. In some systems, stored procedures can be used to control transaction management; in others, stored procedures run inside a transaction such that transactions are effectively transparent to them. Stored procedures can also be invoked from a database trigger or a condition handler. For example, a stored procedure may be triggered by an insert on a specific table, or update of a specific field in a table, and the code inside the stored procedure would be executed. Writing stored procedures as condition handlers also allow DBAs to track errors in the system with greater detail by using stored procedures to catch the errors and record some audit information in the database or an external resource like a file.

2.2.2 User-Defined Functions

User-defined functions combine the advantages of stored procedures with the capabilities of SQL predefined functions. They can accept parameters, perform specific calculations based on data retrieved by one or more SELECT statement and return results directly to the calling SQL statement.
Each user-defined function carries certain properties or characteristics. The SQL standard defines the following properties [18]:

- **Language** - defines the programming language in which the user-defined function is implemented; examples are SQL, C or Java.
- **Parameter style** - defines the conventions that are used to pass the function parameters and results between the implementation of the function and the database system (only applicable if language is not SQL).
- **Specific name** - a name for the function that is unique within the database. Note that the function name does not have to be unique, considering overloaded functions.
- **Determinism** - specifies whether the function is deterministic or not. The determinism characteristic has an influence on the query optimizer when compiling a SQL statement. The rewrite of SQL statements may not change the number of function invocations for non-deterministic functions.
- **SQL-data access** - tells the database management system whether the function contains no SQL statements (NO SQL), contains SQL statements but does not access any tables or views (CONTAINS SQL), reads data from tables or views (READS SQL DATA) or actually modifies data in the database (MODIFIES SQL DATA).

User-defined functions should not be confused with stored procedures. Stored procedures allow the user to group a set of SQL commands. A procedure can accept parameters and execute its SQL statements depending on those parameters. A procedure is not an expression and thus, cannot be used like user-defined functions. Some database management systems allow the creation of user defined functions in languages other than SQL.

### 2.2.3 Triggers

This feature was implemented long ago by most SQL products but was not standardized until the publication of SQL: 99 [19]. The specification of triggers was complete before SQL-92 was published and all major SQL vendors were delivering versions of their
products that provided trigger support. But the trigger specification didn’t remain static in the lengthy interval between SQL-92’s publication and SQL: 99’s. In fact several improvements and enhancements were made, some to align with features that vendors had added and some to simplify the specifications. A trigger is a procedure that is atomically invoked by the database management system in response to specified database events. Triggers can be viewed as event-condition-action rules that allow users to implement application logic within the database management system. Triggers can be used to monitor the modifications of the database, to automatically propagate database modification, to support alerts or to enforce integrity constraints.

Triggers in SQL: 99 have the following components:

- A unique name identifying the trigger within the database.
- A triggering event which is INSERT, UPDATE or DELETE on a base table.
- The activation time which is BEFORE or AFTER, executing the triggering event.
- A trigger granularity which is FOR EACH ROW or FOR EACH STATEMENT.
- A trigger condition which can be any valid SQL condition involving complex queries.
- A triggered action which can be any valid sequence of SQL procedures statements.

The potential use of triggers is numerous, but we can narrow them down to these categories [20]:

- Logging and auditing: we can define triggers on certain tables - especially tables that have security implications, such as salary information - to record information about changes made to those tables. That information can be recorded in other tables and might include information as the authorization identifier under whose control the changes were made, the current time when the changes were made and so on.
Case Study: Implementation of Integrity Constraints in Actual Database System

- Consistency and cleanup: the application might benefit from allowing relatively simple sequence of SQL statements on certain tables to be supported by triggers whose responsibilities include making corresponding changes to other tables.
- Non-database operations: the triggers are not restricted to perform only ordinary SQL operations. The triggers can invoke procedures that send email messages, print documents or activate robotic equipment to retrieve inventory to be shipped.

We can define triggers that are invoked whenever we insert one or more rows into a specified table, update one or more rows in a specified table or delete one or more rows from a specified table. These triggers can take any kind of action that we find appropriate for our application. Triggers, even though they are “schema objects” themselves, are always associated with exactly one base table. SQL: 99 do not allow them to be associated with views, although some SQL products provide extensions to allow that capability. A table with which the trigger is defined is called the base table of the trigger and the SQL statement causing the trigger to be fired is called the triggering SQL statement. When the trigger is fired, it causes another SQL statement to be executed, called the triggered SQL statement.

*Table 2.1 shows a list of the possible trigger actions [21]:*

<table>
<thead>
<tr>
<th>STATEMENT</th>
<th>ROW</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BEFORE</strong></td>
<td>Trigger before statement: The trigger is executed only once before the execution of the operation that activates it.</td>
</tr>
<tr>
<td><strong>AFTER</strong></td>
<td>Trigger after statement: The trigger is executed only once after execution of the operation that activates it.</td>
</tr>
</tbody>
</table>

*Table 2.1: SQL: 99 Trigger Actions*

One aspect of SQL: 99 triggers is the optional syntax REFERENCING<old or new values alias>. This syntax gives the user the ability to reference values in the base table of the trigger, either in the trigger condition or in the triggered SQL statement. Values that existed in the base table before an update, delete or after an update and insert, can be
referred using the correlation names. OLD ROW or NEW ROW can be specified only with row-level triggers, only if FOR EACH ROW is specified.

The following Table 2.2 summarizes what value is retrieved when the column name is qualified by the old or by the new correlation name after various trigger events.

<table>
<thead>
<tr>
<th>Trigger Event</th>
<th>Old. column</th>
<th>New. column</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT</td>
<td>No value (error)</td>
<td>Inserted value</td>
</tr>
<tr>
<td>UPDATE (column updated)</td>
<td>Original value</td>
<td>Current value (U)</td>
</tr>
<tr>
<td>UPDATE (column not updated)</td>
<td>Original value</td>
<td>Original value (N)</td>
</tr>
<tr>
<td>DELETE</td>
<td>Original value</td>
<td>No value (error)</td>
</tr>
</tbody>
</table>

Table 2.2 SQL: 99 Referencing Clause

When a correlation name has no value, an error is issued only when an SQL or SPL statement referencing the undefined correlation is executed, rather than when the correlation name is declared. Refer to Table 2.3 in conjunction with Table 2.2.

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original value</td>
<td>Value before the trigger event</td>
</tr>
<tr>
<td>Current value</td>
<td>Value after the triggered action</td>
</tr>
<tr>
<td>(N)</td>
<td>Cannot be changed by triggered action</td>
</tr>
<tr>
<td>(U)</td>
<td>Can be updated by triggered actions; updated value might be different from the original value because of proceeding triggered actions.</td>
</tr>
</tbody>
</table>

Table 2.3 Reference Clause

Outside a FOR EACH ROW triggered-action list, we cannot qualify a column from the triggering table with either the old correlation name or the new correlation name; it always refers to the current value in the database.

2.2.3.1 Triggers and products

As mentioned above, most products have implemented triggers in spite of SQL-92’s failure to specify a standard for them. The result of such a thing was that various products implementation of triggers do not conform to one another. Different products implement triggers slightly differently. There are syntax variations, which cause more problems to application developers [22]. The difference in implementation of triggers in these products will be observed later in this thesis.
2.2.3.2 Constraint Checking and Trigger Execution

When a declarative constraint is not satisfied, an exception is raised. Such an exception usually leads to a rollback of the transaction that contains the violating database modification. In such a case of checking at commit time, the transaction is always terminated by an implicit rollback. It has already been mentioned that the checking of an integrity constraint can also be deferred to the commit time of the transaction. The SET CONSTRAINTS statement can be used to change the checking mode of deferrable integrity constraints with respect to the current transaction. When the keyword ALL is used instead of a list of constraint names, the mode of all deferrable integrity constraints is set to defer. The commit of a transaction implicitly executes the statement SET CONSTRAINTS ALL IMMEDIATE. In this way, all deferred integrity constraints are enforced at commit time. Since SQL-99 supports both declarative constraints and triggers, the semantics of their possible interaction has to be defined clearly. This includes the definition of the precise timing of constraint checking and trigger execution. Figure 1 sketches the processing order of declarative constraints and triggers.

The execution model ensures that the trigger actions are always executed on a consistent database state. BEFORE triggers are executed before the triggering operation's modifications are performed on the database. The actions of BEFORE triggers are not allowed to modify the database state. AFTER triggers are executed after the triggering operation has been completely executed on the database and all declarative constraints have been enforced. The order of the enforcement of all applicable declarative constraints is as follows: First the referential constraints with the RESTRICT semantics are evaluated. If these constraints are satisfied, then the referential constraints with the CASCADE, SET NULL, or SET DEFAULT semantics are enforced. Finally, if all previous integrity constraints are satisfied, not null constraints, unique/primary key constraints, check constraints, and referential constraints with the NO ACTION semantics are evaluated [23].
2.3 Privileges, users, and security

The very nature of databases, consolidating data into a logically cohesive group that many different applications and users can access, gives rise to security problems. There are many cases where certain users should not have one or more types of access privileges. SQL: 99 provide us with the capability to protect and control the access to various kinds of database objects such as [24]:

- Tables
- Columns
- Views
- Domains
- Character sets
- Collations
Case Study: Implementation of Integrity Constraints in Actual Database System

- Translations
- Triggers
- SQL-invoked routines
- User-defined types

Security properties have evolved greatly since SQL-86 and SQL-92. The general philosophy in SQL:99 are to hide information about schema objects from users who do not have any privileges to use those subjects. These kinds of protections are listed in Table 2.4 [24]:

<table>
<thead>
<tr>
<th>Kind of Protection</th>
<th>Privilege</th>
<th>Applies To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seeing</td>
<td>SELECT</td>
<td>Tables, Columns, SQL-invoked methods</td>
</tr>
<tr>
<td>Creating</td>
<td>INSERT</td>
<td>Tables, Columns</td>
</tr>
<tr>
<td>Modifying</td>
<td>UPDATE</td>
<td>Tables, Columns</td>
</tr>
<tr>
<td>Deleting</td>
<td>DELETE</td>
<td>Tables</td>
</tr>
<tr>
<td>Referencing</td>
<td>REFERENCES</td>
<td>Table, Columns</td>
</tr>
<tr>
<td>Using</td>
<td>USAGE</td>
<td>Domain, User-defined types, Character sets, Collations, Translations</td>
</tr>
<tr>
<td>Activating</td>
<td>TRIGGER</td>
<td>Tables</td>
</tr>
<tr>
<td>Executing</td>
<td>EXECUTE</td>
<td>SQL-invoked routines</td>
</tr>
<tr>
<td>Sub typing</td>
<td>UNDER</td>
<td>Structured types</td>
</tr>
</tbody>
</table>

Table 2.4 Kinds of Protection

The authIDS are the sort of surrogate for the users that SQL database system uses to identify users. Privileges are granted to users, represented by the authIDs. Privileges, once granted, could never be revoked in SQL-89; in SQL-92 that capability was added and SQL: 99 continued to support it. There is also the concept of the PUBLIC, which is a sort of pseudo-authID that identifies every authID that is now known to the DBMS. Privileges can be granted to PUBLIC just as they are granted to individual users. To
allow members to browse the list of books in the library, the seeing (SELECT) privileges should be granted to members on the appropriate tables. In the SQL model the creator of an object is always the owner of that object. The owner has every possible privilege on the object and the owner may grant some privileges on that objects to other users. If the owner wishes to have a certain level of security, privileges can be granted in a way that permits their grantees to grant the privilege to further users, either with or without the ability for them to pass the privilege on. The ability to pass a privilege onto others is called the *with grant option*. The rules in SQL: 99 with respect to these are:

- A user can perform DDL (Data Declaration Language) operations in a schema that they own.
- A user cannot perform any DDL operations in a schema that they do not own.
- A user can not override these rules.

SQL: 99 supports two forms of the GRANT statement: one form grants privileges to users or to roles while the other form grants roles to users or to other roles.

- Granting privileges
- Granting roles

Most SQL products do provide some sort of facility by such a user can override these restrictions, by giving others permission to perform DDL operations within schemas that the user owns; but the capability and syntax among these products varies.

### 2.4 data definition fundamentals

Tables are the basic unit of data management that will be encountered in SQL; therefore an understanding is needed of the structure in which tables are embedded. Tables are imbedded in a nested structure of increasing abstractions. *Figure 2* illustrates the relationship between the data objects. SQL: 99 databases consist of all data described by schemas under the control of a give DBMS at any one time [25].
2.4.1 Schema and Catalogs

Most applications require multiple tables, and most implementations support many applications, each with its own set of tables. In order to avoid problems arising from two applications choosing the same name for their tables, SQL provides the partitioned namespace solution. Tables exist in the concept of a higher-level object called schema. In SQL, a schema is simply the collection of all objects that share a namespace. Schemas have names and those names can be used to qualify the names of the objects contained within the schema. To address the problem of schema names conflicting SQL-92 addressed this problem by collecting schemas into a unit called a catalog. Many DBMS products do not implement catalog names, so expect to find a variance between DBMSs and the SQL: 99 Standard.

2.5 Some SQL: 99 DATA TYPES

Among the enhancements to SQL: 99 over its predecessor SQL-92 version are several new data types. Some data types that we will be used in the application sample supported by the SQL: 99 standards will be:
• Exact Numeric: The exact numeric data types are those that can represent a value exactly.
  o One category of these is the true integers, which are represented in SQL by INTEGER.
  o The other category is intended to permit fractional components, but still provide exact representations which are DECIMAL.
• Character string: depending on how we want to count, there are:
  o CHARACTER: This data type contains character strings. The specific character sets that can be stored in one of these are defined by your implementation.
  o CHARACTER VARYING: If we don’t wish to blank-pad our character strings, then we will want to use CHARACTER VARYING, which allows us to store exactly the number of characters that we have in our data.
• Date times:
  o DATE: This variation stores the year, month and day values of a data. The year value is exactly 4 digits and can represent the years 0001 through 9999. The month value is exactly 3 digits and is limited to values 01 through 12. The day value is also exactly 2 digits and is limited to values 01 through 31.
• Intervals:
  o An interval is broadly defined as the difference between two dates or times. We can specify the type of interval which may be, INTERVAL MONTH, INTERVAL DAY, INTERVAL YEAR, etc.
• Booleans:
  o Until SQL: 99 the SQL standard did not have a Boolean data type, which can have one of three different “Boolean literal” values:
    ▪ TRUE
    ▪ FALSE
    ▪ UNKNOWN
II. APPLICATION – A CASE STUDY

The aim of this thesis is to design an application system that demonstrates the integrity constraints from the case of standard SQL: 99 and other relational database systems and to gather information relevant for porting SQL from one product to another and/or possibilities and limits of 'cross product' SQL. Different products have, over the years, closed the gap between their respective syntactical constructs and that contained in the SQL standard. It is important to note that there is not - nor likely ever be - a perfect match between the syntax of any of these products and SQL standard syntax. General design guidelines shall be presented that show how the integrity features provided by one system are supported in another system. The following tables compare how different DBMS products handle various SQL (and related) features. If possible, the tables also state how the implementations should do things, according to the SQL standard. Therefore it is important to first understand the syntax and behavior of the SQL standard and then behavior of each product individually.

3. LIBRARY MANAGEMENT SYSTEM

3.1 Introduction

This chapter presents the design of an application to manage a university library system, able to perform many functions with certain levels of security. Firstly, the requirements for the library management system shall be defined.

3.2 Application

The university consists of many departments each providing a set of books to the library. Each department will appoint a manager who is in charge of the books offered by that respective department. The Library carries items such as textbooks and lecture notes, each specified by an ID. The system keeps a record of all the items with the specific information about each. The librarian is able to identify the type of item, the author,
subject, etc using this ID. There are also some restrictions on what types of items are allowed to be taken out of the library and what types are not. Members of the library are students and teachers. All members can register at the library and get a Library card. Only members with a valid card will be allowed to lend items from the library. Students renew their ID card every year where Teachers ID are valid as long as they are still employed at the university. The system allows the members to search for an item based on the author’s name, book name, etc. The system enforces some sort of security, members can not add/remove items to/from the library; this is handled by the librarian and the department manager. Items are loaned to members in various categories. The system has some restrictions and rules on the type of books borrowed and duration of which a book can be held by an individual. The system generates some alerts in certain situations such as when a member card is invalid or a member returns a book later than it was suppose to. The system keeps track of each member’s order history, which stores information about their borrowed/returned items, penalties, etc.

Before a member can place an order to a specific item, certain conditions must be checked to make sure no rules have been violated. Some of these conditions are: members having a valid library card and ordered item being available and allowed. If none of the conditions were violated, then the members are able to place their orders and borrow an item successfully from the library. Certain actions are then performed by the system such as updating the number of available books, updating information on the order history table and so on. The next step is identifying all the features that will be implemented into the library system.

3.3 CONCEPTUAL DESIGN:

The key issue in database design is to accurately model the appropriate aspects of a large enterprise as a relational database that can be efficiently accessed and updated by a large number of concurrently executing transactions in number of distinct applications, some of which might also include decision support queries. As in other engineering disciplines, the design process can be facilitated if it is performed according to some specific methodology and can be evaluated according to some objective criteria. In data modeling, an entity-relationship model (ERM) is a representation of structured data;
entity-relationship modeling is the process of generating these models. The end-product of the modeling process is an entity-relationship diagram (ERD) or ER diagram, a type of conceptual data model or semantic data model [26].

An entity represents a discrete object and is represented as a rectangle. A relationship captures how two or more entities are related to one another. Relationships are represented as diamonds, connected by lines to each of the entities in the relationship.

Entities and relationships can both have attributes. Every entity (unless it is a weak entity) must have a minimal set of uniquely identifying attributes, which is called the entity's primary key. Associative entity is used to solve the problem of two entities with a many-to-many relationship.

Unary Relationships - a unary relationship is a relationship between the rows of a single table. Entities may be either strong or weak.

- A normal entity as described above is alternatively called a strong entity, in that it can be uniquely defined by its attributes alone.
- A weak entity is an entity that isn't uniquely identified by its own attributes and therefore includes one or more of its relationships into its primary key.

Sometimes two entities are more specific subtypes of a more general type of entity. For example, students and teachers are both types of members in the library. To indicate this, the "ISA" hierarchy is used where the super class is member and subclasses are teacher and student. The first step in the E-R approach is to find the entities that will be used to model the enterprise. Book, member, card, orders, department, author, order_history and manager are entities used.

3.3.1 Table relationships

Relationships between tables in a database diagram show how the columns in one table are linked to columns in another table. In a relational database, relationships enable us to prevent redundant data. For example in the library system, there is a table called “books” that stores information about each book, such as books title, ISBN number, and department_id. There is also information that should be stored about each department, such as the department name, location, etc. This information is then stored in a separate table called “department”. A pointer is then put in the book table that references an entry in the “department” table. To ensure that the data in not out of sync; referential integrity
in enforced between the two tables. Referential integrity relationships help to ensure information in one table matches information in another. For example, each title in the “book” table must be associated with a specific department in the “departments” table. A book title can not be added to the database for a department that does not exist in the database.

3.3.1.1 Types of table relationships
A relationship works by matching data key columns - usually columns with the same name in both tables. In most cases the relationship matches the primary key from one table, with an entry in the foreign key of the other table. There are three types of relationships between tables, the type of relationship that is created depends on the how the related columns are defined [27].

- One-to-Many Relationships:
  A one-to-many relationship is the most common type of relationship. In this type of relationship, a row in table A may have many matching rows in table B, but a row in table B can have only one matching row in table A.

- Many-to-Many Relationships:
  In a many-to-many relationship, a row in table A can have many matching rows in table B and vice versa. We define such a relationship by defining a third table, whose primary key consists of the foreign keys from both table A and table B.

- One-to-One Relationship:
  In a one-to-one relationship, a row in table A can have no more than one matching in table B and vice versa. A one-to-one relationship is created if both the related columns are primary keys or have unique constraints. This type of relationship is not common because most information related in this way would be in one table.

3.3.2 E-R Model

3.3.2.1 Entity-Relationship Diagram
To build the database for this library system, firstly refer to the ER model in Figure 3.1 which is located in Appendix A.
3.4 Entity description

To understand the diagram better, the following section will describe each entity briefly:

**Member**

The information about the members, in the sample application domain, must be managed. The entity member has all the necessary attributes to identify members of the library system. Each member is identified by a unique member id (*mem_id*) and member number (*mem_num*). Members of the library are Students and teachers. SQL: 99 support the concept of sub-typing and sub-tables. In a table hierarchy, every table constraint implicitly holds for each sub table. That is, integrity constraints are inherited to the sub tables. A table shall contain at most one implicit or explicit primary key. Implicit primary key is the one being inherited from the super table.

Two subtypes of members should be represented:

- Student with attributes study_startdate, study_enddate, study_program, still_student and major.
- Teacher with attributes number of grants and specialized field.

**Student**

For each student, general information such as their *study_startdate*, *study_enddate*, *major*, etc has to be known. When a student starts the university, the attribute *Study_startdate* is inserted in the student table and they will not be allowed to borrow books from the library until the second month of their studies at the university. A student may use the library during their studies and once a student has ended their studies they no longer can borrow books from the library.

**Teacher**

The other type of member is teachers. For teachers records of their specialized fields and number of grants that they receive are kept.

**Card**

All members can register at the library and get a library card. Only members with a valid card will be allowed to lend items from the library. Teachers ID are valid as long as they are still employed at the university. Students ID are valid for a year since the date it was
created and the activation date differs for each member type. Each card is specified by a unique card_num. Information is stored about the cards create date, validity and status. The status is either activated or blocked. Each member has one card and each card belongs to one person. The attribute card_num is the primary key of the table and member_id is the foreign key. The two table’s “member” and “card” are connected together. Since the “card” table is dependent on the “member” table to exist, then the primary key of the “member” table shall be stored in the “card” table. Whenever a member’s details are updated or deleted the card belonging to that member shall be updated or deleted as appropriate.

**Order**

The “order” table is where each member is able to place their orders to a specific item in the library. For a member to be able to place an order certain conditions must be satisfied: The member must have a valid card; the book type must be in the list of allowed books that can be borrowed from the library, with the number of available copies greater than zero. When a member successfully borrows an item from the library, a corresponding record of this order will be copied to the “order_history” table. “Order” has relation with “card” and “book”. The primary key of this entity is order_id, and foreign keys are, Book_id and card_num. The order entity has attribute state, which has 3 set of values,’ order_processed, ’done’, ’Failed’, depending on the situation is updated.

**Book**

Book entity contains attributes such as book_title, ISBN_number, year of publication and so on; the book_id is the primary key and dep_id is the foreign key of the table. There is restriction on what type of book can be borrowed and taken outside of the library. There are two types of books in our library, ‘text book’ and ‘lecture notes’. Only text books are allowed to be borrowed by the members, lecture notes are not allowed. If the department id which provides the book is updated or deleted for some reason then the corresponding row in books table can only be updated by one referential constraint with a delete rule of SET NULL or SET DEFAULT.
**Department**
Each department is managed by a manager assigned by their respective department. Primary key is Dep_id. If the manager of the department for some reason resigns, then the manager id in the “department” table is updated as well.

**Manager**
All departments are managed by a manager who has the privilege to update, delete and insert books that their department offers. The manager is identified by the mgr_id, which is the primary key of the table.

**Author**
The relation between the tables “book” and “author” is n-ary relation. Meaning each book can be written by more than one author and each author can have more than one book. The primary key of this entity is the author_id. Each author is identified with a unique author_fullname.

**Book_auth**
Since Tables “book” and “author” have a many-to-many relationship between them, a book can have many authors and an author can have many books. This relationship is defined by the table “book_auth”. The primary key columns are copied from each of the other two tables to the relation table. Other columns can be added to this table, just as any other table.

**Order_history**
This table keeps a history about each member’s orders, the books they are holding, whether they have any penalties for returning a book late and etc. The primary key of this table is the history_id, and the foreign keys are, Card_num, book_id. As mentioned, whenever an order for a book has been successfully completed, the system automatically copies the record of that order to the “Order_history” table.
4. SQL: 99 constraint support matrix

In this chapter shall be presented a summary of the supported constraints in SQL: 99 with some examples. The portion of SQL: 99 Grammar related to integrity constraint is presented in Appendix B. Table 4.1, shows the declarative integrity support in SQL: 99:

![Table 4.1 Comparison of Declarative Integrity Support](image)

**Example 1:** In this sample application domain, information about members must be managed. Members are uniquely identified by a member id. Assuming every member has a first name and last name, members can have two types ‘student’ and ‘teacher’. The default member type is ‘student’. There are certain regulations about members, to be able to borrow books from the library, member’s age must be over 18 and they must have a valid address in Czech Republic and be living in ‘Prague1’ or ‘prague2’.
Domains are a sort of macro that can be defined to pull together a specific data type. The name of the domain can be used to define columns that inherit the data type. Later on we will see that not all DBMS products support this feature of the SQL Standard.

```
CREATE DOMAIN mem_types AS CHAR (12)
    DEFAULT 'student'
    CHECK (value IN ('student','teacher'));
CREATE TABLE member (  
    Mem_id INTEGER PRIMARY KEY,
    Mem_num VARCHAR (20) UNIQUE,
    Mem_firstname VARCHAR (15),
    Mem_lastname VARCHAR (30),
    Mem_address VARCHAR (30) NOT NULL,
    Mem_city VARCHAR (15) NOT NULL
        CONSTRAINT CHECK (mem_city IN ('PRAGUE')),
    Phone CHAR (16) DEFAULT 'UNKNOWN',
    Mem_age INTEGER NOT NULL
        CONSTRAINT CHECK (mem_age >= 18),
    Mem_type mem_types,
    Email VARCHAR (30) NOT NULL
);  
```

SQL-89 requires that to specify NOT NULL UNIQUE, PRIMARY KEY in the constraint definition; it was not permitted to say UNIQUE or PRIMARY KEY without saying the NOT NULL. Additionally, the ordering of the two constraints was restrictive; it could not be said UNIQUE NOT NULL, but rather had to place the NOT NULL first. SQL-92 relaxed that requirement and allowed UNIQUE, PRIMARY to be specified without accompanying specification of the NOT NULL. Primary and unique keys have the inherent property that no row can have a null value for the primary key, unique key column. The syntax and semantics for the UNIQUE, PRIMARY predicate continued into SQL: 99. SQL: 99 also define a UNIQUE, PRIMARY key constraint to be exactly the same as a CHECK constraint on the table. Each of SQL’s table constraints and column constraints may, optionally, have a name assigned to it. In reality, all constraints have a name, although not necessarily one assigned by the user. Constraint names were new to SQL-92. There are several different syntactical ways in which the constraints can be defined.
CREATE TABLE member (  
    . . .  
    CONSTRAINT check_city CHECK (mem_city IN ('PRAGUE')),  
    CONSTRAINT check_memage CHECK (mem_age >= 18),  
);  

The support of not null constraint has been summarized in Table 4.2:  

<table>
<thead>
<tr>
<th>NOT NULL on</th>
<th>SQL-99</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row type</td>
<td>✓</td>
</tr>
<tr>
<td>Reference type</td>
<td>✓</td>
</tr>
<tr>
<td>Collection type</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4.2 Not null constraints

Summarized in Table 4.3 is the applicability of unique/primary key constraints on columns of row, reference and collection types in SQL: 99.

<table>
<thead>
<tr>
<th>Unique/ Primary Key on</th>
<th>SQL-99</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row type</td>
<td>✓</td>
</tr>
<tr>
<td>Reference type</td>
<td>✓</td>
</tr>
<tr>
<td>Collection type</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4.3 Unique/ Primary key constraints

Example 2: As earlier mentioned the library has two types of members, students and teachers. It is to be made certain that the values in the mem_id column in the “student” table are different from those in the “teacher” table, meaning no student can have the same id as a teacher and vice versa.

A table constraint is normally used to make a restriction on the data that is stored in the table to which the constraint is attached to. It is possible to express restrictions involving multiple tables with regular CHECK constraint. This will be shown using the CHECK constraint:

CREATE TABLE student  
    mem_id INTEGER NOT NULL,  
    . . .  
    CONSTRAINT check_member CHECK ( mem_id <> ANY  
        (SELECT mem_id from teacher) )  
);
This sort of constraint allows the effective formation of a unique constraint across more than one table, but to express it as a table constraint is not that clear. First it has to be decided whether to put the constraint on the teacher table or the student table, so this is not recommended. Instead, a more natural way to express the same restriction is to state it as a standalone constraint - that is, an assertion [28].

```
CREATE ASSERTION student_never_teacher
    CHECK ((SELECT mem_id from student) <> ANY
        (SELECT mem_id from teacher)
    );
```

### 4.1 Check constraint and Assertion Limitations:

Check constraint is one of the most flexible, and therefore useful, constraints within SQL: 99. This allows the specification of a wide range of rules for the tables, such as range of values, list of values and others. There are a couple of limitations on check constraints that must be taken into account [29]:

- When the CHECK constraint is a column constraint, then it cannot reference any column other than the column with which it is associated.

- When the CHECK constraint is a table constraint, then it can reference only columns that are in the table with which it is associated.

SQL-92 standard permitted the use of sub queries within check constraints, thereby enabling check constraints to be used for multi-tuple and multi-table constraints; most products do not support this feature.

General constraints that are not specific to a single table can be specified by assertions, although again many products do not support this level of generality. Assertions, like other constraint, have some restrictions. For example, all of the values used must be literals or database values; host variables or parameters, date time functions, CURRENT_USER, SESSION_USER or SYSTEM_USER cannot be referenced [30].
4.2 Triggers:

The complete syntax of <trigger definition> is given in Appendix A. Currently triggers are available in all reference systems. However, since the concept of a trigger was not standardized until the recent SQL-99 standard, the implementations in the various systems differ with respect to syntax (and semantics) and capabilities. The grammar of trigger definitions in the various systems, are so different that no trigger specification that is formulated in one system can be used without any modifications in another system. Particularly the action parts differ in the various approaches because each system has its own trigger programming language. Figure 4.4 shows the capabilities of the supported Trigger mechanisms in SQL:99.

<table>
<thead>
<tr>
<th></th>
<th>SQL - 99</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Events per trigger</td>
<td>1</td>
</tr>
<tr>
<td>#Trigger per event</td>
<td>n</td>
</tr>
<tr>
<td>Trigger granularity</td>
<td>R/S</td>
</tr>
<tr>
<td>Activation time</td>
<td>B/A</td>
</tr>
<tr>
<td>Activation condition</td>
<td>√</td>
</tr>
</tbody>
</table>

**Tables 4.4 Trigger**

Number of events per trigger: Potential events that can lead to the firing of a trigger are INSERT, UPDATE, and DELETE operations. In SQL-99, a trigger is based on exactly one event (As shown later on, this is not the case in oracle).

- Number of triggers per events: SQL:99 systems do not have any restriction. (MySQL supports only one trigger per event)
- Trigger granularity: The granularity of a trigger can be FOR EACH ROW (R) or FOR EACH STATEMENT (S) (MySQL does not support statement triggers).
- Trigger activation time: A trigger is activated BEFORE (B) or AFTER (A) the specified event is handled.
- Trigger activation condition: The activation of a trigger can be associated with a condition.
- Trigger action can be a single SQL statement or a sequence of statements. In the second case, below is what must be used:

```
BEGIN ATOMIC
```
The use of the `BEGIN...END` in a triggered SQL statement requires the use of the keyword `ATOMIC` to act as a reminder that this particular `BEGIN ...END` block will always be handled as a unit.

These restrictions may be found surprising: a BEFORE trigger is defined in SQL:99, it is not allowed to be specified as either OLD or NEW TABLE (only references to row), nor may the triggers triggered SQL statement make any changes to the database. Those are allowed only for AFTER triggers.

Possible SQL: 99 Trigger Actions [22]:

- Trigger Before: data definition, data selection, procedure calls; it is however not possible to execute actions that modify the database state.
- Trigger After: everything that can be executed in the before trigger + data modification operations (INSERT, DELETE and UPDATE).

As will be shown later on this is not the case in some other products. The reason is perhaps a bit subtle. The transition tables implied by the OLD TABLE and NEW TABLE are too likely to be affected by referential constraints and referential actions that are activated by the changes being caused by the triggered SQL Statement; therefore, the values of the rows in that table are not stable or adequately predictable until after the triggering SQL statement has been executed.

Associating the “right” activation time with a trigger is highly system dependent. In almost all cases, AFTER triggers are sufficient. BEFORE triggers associated with a table are typically evaluated before declarative constraints are evaluated for the table. Note that here are several subtle differences among the implementation of the SQL-99 activation time feature in current systems. For example, in MYSQL the granularity FOR EACH STATEMENT is not supported, whereas in Oracle modifications of affected
rows are only allowed in BEFORE triggers and granularity FOR EACH STATEMENT triggers are supported.

Example 3: It must be ensured that any new order inserted on the orders table must come from a member with a valid card:

```
CREATE TRIGGER check_member_card
BEFORE INSERT on Order
REFERENCING NEW ROW AS Neworder
FOR EACH ROW
WHEN (Neworder.card_num) IN
(SELECT card_num from card
Where card_renew=’TRUE’
AND status=’Blocked’)
BEGIN ATOMIC
SIGNAL 1000 (‘Integrity violation: members’ card is not valid’);
END;
```

Example 4: Whenever a new card has been created the trigger fires, it sets the validity, status and activation date of the newly created card. As mentioned, the activation date of the card differs depending on the member type. For students the activation date is 60 days after their card has been created.

The following trigger is an example of a row-level INSERT trigger.

```
CREATE TRIGGER set_validity1
AFTER INSERT ON card
REFERENCING NEW ROW AS Newcard
For each ROW
WHEN (newcard.mem_id) IN
(Select mem_id from student
Where still_student=’TRUE’))
BEGIN ATOMIC
Update card
Set newcard.validity= Newcard.createdate + INTERVAL ‘360’ DAY
And newcard.activedate= Newcard.createdate+ INTERVAL’60’ DAY
END;
```
4.3 Trigger on Views:

A SQL: 99 views can be considered a virtual table, a table that does not physically exist, but rather is formed by a query expression against one or more tables. Views are part of the standard and they may be updated, as long as it 'makes sense'. SQL-92 was more restrictive, specifying that updatable views cannot be derived from more than one base table. SQL: 99 does not allow triggers to be associated with views, although some products provide extensions to allow that capability as we shall see later.

4.4 Event schedulers:

To execute a stored procedure at a specific time or in an interval - events are used. SQL: 99 do not support events.

4.5 Automatic key generation:

It is sometimes handy to have the DBMS handle generation of keys [31]. The DBMSs offer various means for this, which will be shown later, but the problem is each DBMS implementation seems to have its own way of declaring auto-incrementing fields. SQL: 99 Standard does not support this feature. In order to implement this feature a function which increments the value of the column should be used. In Standard SQL use:

```
CREATE TABLE orders (  
    Order_is INTEGER DEFAULT some_func() PRIMARY KEY  
    . . . );
```

Where “some_func()” is a function which finds 1 plus the currently largest value of column name. This feature was added later to the SQL: 2003 standard. The standard specified a column attribute of:

GENERATED ... AS IDENTITY

When creating a table, an IDENTITY clause may be declared for certain types of columns (INTEGER being one):

```
CREATE TABLE orders (  
    order_id INTEGER GENERATED ALWAYS AS IDENTITY  
    ... );  
```

Or

```
CREATE TABLE orders (  
```
order_id INTEGER GENERATED BY DEFAULT AS IDENTITY
);

The column with the IDENTITY attribute will be given values in increasing order; a base table may at most contain one column with the IDENTITY attribute. NOT NULL is implied for an IDENTITY column. Normally, a column declared with IDENTITY will also be declared PRIMARY KEY, but it's not implied. The examples differ in their 'ALWAYS' vs. 'BY DEFAULT' clauses:

- When ALWAYS is specified, the user cannot specify a value for the column which means that the DBMS can guarantee successful insertion of a unique value on each table insert.
- When BY DEFAULT is specified, the user may manually specify what value to put in the identity field of a row. The flip side is that the DBMS cannot guarantee that this will work.

The standard specifies several extended options which may be declared for a generated IDENTITY column. It will be shown later how this feature is supported in different systems.
III. Comparison of Integrity Support in different Commercial Database Management Systems:

Given the support for the specification and enforcement of integrity constraints in standard SQL: 99, the question now is how to utilize respective concepts and mechanisms effectively in different commercial relational database systems, in database design and implementation. In this section, some general guidelines and rules for utilizing the integrity maintaining mechanisms will be provided. However, a comprehensive overview of the role of integrity constraints in database design in general is not given, but rather a step-by-step approach helping to implement typical integrity constraints using the declarative and procedural mechanisms provided by the systems.

All major relational database management systems support SQL, so it is possible to transfer all skills gained with SQL from one database to another. In addition, all programs written in SQL are portable. They can often be moved from one database to another with little modification. Later on some basic issues regarding the acquisition and classification of different types of integrity constraints during database design in different products will be outlined and covered in detail.

Today, just about every useful database system supports SQL to some extent. In theory, SQL acts as a good unifier, since database applications written to use SQL as the interface to the database can be ported to other database systems with little cost in terms of time and effort. Commercial pressures however, dictate that database manufacturers distinguish their products one from another. This has led to SQL variations, not helped by the fact that the standard for SQL does not define commands for many of the database administration tasks that are an essential part of using a database in the real world. So there are differences between the SQL used by Oracle, MySQL, PostgreSQL, MSSQL and other database systems.
5. Common Language for All Relational Databases

All major relational database management systems support SQL, so all skills which have been gained with SQL can be transferred from one database to another. In addition, all programs written in SQL are portable. They can often be moved from one database to another with very little modification.

In this section, the semantic integrity features of SQL-99 [33] with those of the commercially available (object-) relational database management systems Oracle10g Server (Release 10.2) [34], PostgreSQL(Version 8.2.5) [35], Microsoft SQL Server 2005 [36], MySQL (Version5.1) [37] and firebird (Release 2.0) [38] are compared.

5.1 Introduction to Oracle SQL

Structured Query Language (SQL) is the set of statements with which all programs and users access data in an Oracle database. Application programs and Oracle tools often allow users access to the database without using SQL directly, but these applications in turn must use SQL when executing the user's request. This chapter provides background information on SQL used by Oracle DBMS.

5.2 History of Oracle and Standard compliance

SQL standards have been the most popular and enduring standards in computing. Although information processing has become increasingly more sophisticated and more complex over the past decades, SQL has continually evolved to meet the growing demands. In 1979, Relational Software Inc. (now Oracle) introduced the first commercially available implementation of SQL. Oracle's relational database was the world's first to support the Structured Query Language (SQL), now an industry standard. Oracle has grown from its humble beginnings as one of a number of databases available to the overwhelming market leader of today. In its early days Oracle Corporation was known as an aggressive sales and promotion organization. Over the years the Oracle database has grown in depth and quality.
Oracle strives to comply with industry-accepted standards and participates actively in SQL standards committees, as the latest version of SQL standards, SQL 2003, is making major improvements in a number of key areas. As the first commercial implementation of SQL; Oracle continues to lead the database industry in implementing SQL standards. In fact, many of the SQL 2003 new features had already been supported since Oracle Database 8/8i, Oracle Database 9i or Oracle Database 9i Release 2. The latest Oracle Database 10g supports additional SQL 2003 new features (advanced Multiset support) as well as several new SQL capabilities beyond the current SQL 2003 standard (e.g., additional statistical functions, regular expressions), making Oracle Database 10g the best implementation of SQL standards [39].

5.3 Oracle Extensions to Standard SQL

This section lists the additional features supported by Oracle that extend beyond standard SQL "Database Language SQL with Integrity Enhancement". This section provides information on these parts of the SQL language:

- commands
- functions
- operators
- pseudo columns
- data types
- names of schema objects

5.3.1 Additional Parts of Standard Commands

Oracle supports additional syntax and functionality for some standard commands that are not part of standard SQL [40].

5.4 Oracle constraint support matrix

When an SQL statement that modifies data in the table is issued, the Oracle Database ensures that the new data satisfies the integrity constraint without the need to do any checking within the program. Default values are also subject to integrity constraint checking. If the results of an INSERT or UPDATE statement violate an integrity
constraint, the statement will be rolled back. The following integrity constraints listed in Table 5.1 are supported by Oracle:

<table>
<thead>
<tr>
<th>FOREIGN KEY</th>
<th>ON DELETE</th>
<th>Oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NO ACTION</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RESTRICT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CASCADE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SET NULL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SET DEFAULT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NO ACTION</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RESTRICT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CASCADE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SET NULL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SET DEFAULT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHECK</th>
<th>column-level</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>row-level</td>
<td></td>
</tr>
<tr>
<td></td>
<td>table-level</td>
<td></td>
</tr>
<tr>
<td></td>
<td>database-level</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DOMAIN</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSERTION</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 5.1 Declarative Integrity Constraint**

### 5.5 How Oracle Enforces Data Integrity

Oracle enables the user to define and enforce each type of data integrity rule. Most of these rules are easily defined using integrity constraints or database triggers. Oracle uses No Action as its default action.

#### 5.5.1 Nullability and Default Values

- *Specification of not null constraints*: In oracle, a not null constraint is defined either by using the keyword NOT NULL directly in the column specification or by specifying a check clause of the form CHECK (column IS NOT NULL).
• **Default nullability:** Columns are nullable by default unless null values are explicitly disallowed.

• **Restrictions on the order of the specification of declarative constraints and default values:** In Oracle the specification of the default value must precede the specification of the other integrity constraints.

• **Nullability and object-relational type extensions:** Oracle provides object relational type extensions. Structured (row) types, the reference type and various collection types are supported by Oracle. Columns of these types can be defined as non-nullable in the same way as usual columns (of standard built-in data types). Not null constraints can also be specified individually for any component of a structured column [41].

5.5.1.1 Restrictions on NOT NULL Constraints

• NULL or NOT NULL cannot be specified in a view constraint.

• NULL or NOT NULL cannot be specified for an attribute of an object. Instead, use a CHECK constraint with the IS [NOT] NULL condition.

*Example 5:* The following check clause shows the definition of a not null constraint and a default constraint for the columns phone and mem_address in the member table.

```sql
CREATE TABLE member ( 

    mem_address    VARCHAR2 (30) NOT NULL,
    CONSTRAINT check_adress CHECK (mem_address IS NOT NULL),

    Phone          CHAR (16) DEFAULT 'unknown' NOT NULL,

);
```

*Table 5.2* lists the Oracle support of Not Null constraints:
Case Study: Implementation of Integrity Constraints in Actual Database System

<table>
<thead>
<tr>
<th>Not Null on</th>
<th>Oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row type</td>
<td>√</td>
</tr>
<tr>
<td>Reference type</td>
<td>√</td>
</tr>
<tr>
<td>Collection type</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.2 Not null constraint

5.5.2 Unique/primary key constraint

Compared to the other systems, Oracle additionally allows that there may exist two rows whose uniqueness columns completely consist of null values. However if a column is not explicitly defined as not null, then nulls can be inserted multiple times. Table 5.8 lists the oracle support for these 2 features [42]. The optional NULLs allowed feature is implemented: If the UNIQUE-constraint is imposed on a single column, then the column may contain any number of NULLs. However, if the UNIQUE-constraint is specified for multiple columns, then Oracle sees the constraint as violated if any two rows:

- Contain at least one NULL in a column affected by the constraint
- Identical, non-NULL values in the rest of the columns affected by the constraint

In Oracle, the definition of a primary key implicitly defines not null constraints on the corresponding uniqueness columns, meaning that primary key combines a unique and a not null constraint. This strategy is also proposed in SQL-99.

Integrity violation message in case of simultaneous violation of multiple uniqueness constraints: Oracle prints the name of the uniqueness constraint which has the oldest timestamp (with respect to its recent enabling time).

In Oracle, as in SQL-99, the creation of a uniqueness constraint is rejected if a uniqueness constraint that is defined on the same set of columns already exists.

Unique/Primary key constraints and object-relational type extensions: Oracle allows defining unique/primary key constraints on components of a structured column but not on the whole structure (column). Note that the components have to be based on standard built-in data types. Table 5.3 lists oracles primary, unique key support:
### Inheritance of unique/primary key constraints

Oracle follows the inheritance rules of SQL:99, that integrity constraints are inherited from supertables to subtables.

#### 5.5.3 CHECK Constraints

All reference systems support check constraints at the row level. A check constraint allows stating a minimum requirement for the value in a column. If more complicated requirements are desired, an insert trigger must be used.

##### 5.5.3.1 Multiple CHECK Constraints

A single column can have multiple CHECK constraints that reference the column in its definition. There is no limit to the number of CHECK constraints that you can define on a column.

*Example 6:* The following table allows only members, with age greater than 18, living in Prague1 or Prague2, and having type student or teacher.

```sql
CREATE TABLE member (
    mem_city       VARCHAR2 (15) NOT NULL
        CONSTRAINT check_city CHECK (mem_city IN ('Prague1', 'Prague2')),
    mem_age        INTEGER NOT NULL
        CONSTRAINT check_age CHECK (mem_age >= 18),
    mem_type      CHAR(12)  DEFAULT 'student' NOT NULL,
        CONSTRAINT check_memtype CHECK (mem_type in ('teacher', 'student'))
);
```
5.5.3.2 Restrictions on Check Constraints

- A check constraint for a view cannot be specified. However, the view can be defined using the WITH CHECK OPTION clause, which is equivalent to specifying a check constraint for the view.
- The condition of a check constraint can refer to any column in the table, but it cannot refer to columns of other tables.
- Conditions of check constraints must not contain the following constructs:
  - Subqueries and scalar subquery expressions.
  - Calls to the functions that are not deterministic (CURRENT_DATE, CURRENT_TIMESTAMP, DBTIMEZONE, LOCALTIMESTAMP, SESSIONTIMEZONE, SYSDATE, SYSTIMESTAMP, UID, USER and USERENV).
  - Calls to user-defined functions.
  - Dereferencing of REF columns (for example, using the DEREF function).
  - Nested table columns or attributes.
  - The pseudo columns CURRVAL, NEXTVAL, LEVEL or ROWNUM.
  - Date constants that are not fully specified.

5.6 Foreign Key Constraints and Restrictions

All reference systems implement the simple match rule for referential constraints. It is not possible to establish a foreign key on a global temporary table. If tried, Oracle issues an ORA-14455: attempt to create referential integrity constraint on temporary table. The referential actions supported by the FOREIGN KEY integrity constraints of Oracle are UPDATE and DELETE No Action, DELETE SET NULL and DELETE CASCADE. Other referential actions not supported by FOREIGN KEY integrity constraints of Oracle can be enforced using database triggers.

Example 7: The following tables “student” and “teacher” are subtypes of the “member” table. They integrity constraints are inherited from the supertable to these subtables. If the creation of the “member” table is observed in the previous SQL: 99 part, it can be
seen how domain for the mem_type was used. Since Oracle does not support domains; check constraint must be used instead.

Oracle does not support data type BIT and Boolean. A column cannot be defined in a table with type Boolean. Instead a table with a column of data type CHAR (1) must be created and ‘Y’ or ‘N’ then stored in that column to indicate TRUE or FALSE or a numeric column and store either 1 or 0 for TRUE or FALSE.

CREATE TABLE member (
    mem_id INTEGER NOT NULL primary key,
    mem_num VARCHAR2 (20) NOT NULL unique,
    mem_firstname VARCHAR2 (15),
    mem_lastname VARCHAR2 (30),
    mem_address VARCHAR2 (30) NOT NULL,
    mem_city VARCHAR2 (15) NOT NULL
    CONSTRAINT check_city CHECK (mem_city IN ('Prague1','Prague2')),
    phone CHAR(16) DEFAULT 'unknown',
    mem_age INTEGER NOT NULL constraint check_age
    CHECK(mem_age >= 18),
    mem_type CHAR(12) DEFAULT 'student' NOT NULL,
    CONSTRAINT check_memtype CHECK (mem_type in ('teacher','student')),
    Email VARCHAR2 (30) NOT NULL
);
FOREIGN KEY (mem_id)
REFERENCES member (mem_id)
ON DELETE CASCADE
);

Here the referential action ON UPDATE CASCADE shall be implemented by using triggers since it is not supported in oracle. After a member id is updated on the members table, the following id on the student/teacher must be updated as well.

CREATE OR REPLACE TRIGGER AFTR_updt_mmbbr_updy_tchr_stdnt
AFTER UPDATE OF mem_id ON member
REFERENCING OLD AS old NEW AS new
FOR EACH ROW
BEGIN
    UPDATE teacher
    SET mem_id=:new.mem_id
    WHERE mem_id=:old.mem_id;
    UPDATE student
    SET mem_id=:new.mem_id
    WHERE mem_id=:old.mem_id;
END;

5.7 Assertion:

As for general constraints, \texttt{ASSERTION} is not supported in Oracle.

\textit{Example 8}: One of the restrictions on the member table is that there is no student that can have the same member id as a teacher or vice versa. This rule was implemented by an Assertion in SQL: 99, but since the use of subqueries in the Check Constraint is not supported, assertion cannot be used and triggers must then be used to force this rule.

CREATE OR REPLACE TRIGGER student_id_before_insert1
BEFORE INSERT ON student
REFERENCING NEW AS new
FOR EACH ROW
DECLARE
    V_id NUMBER;
BEGIN
    SELECT count (*) INTO v_id
    FROM teacher
    WHERE teacher.mem_id = :new.mem_id;
IF (v_id > 0) THEN
raise_application_error(-20001, :new.mem_id || 'this id is being used by a teacher');
END IF;
END;

CREATE OR REPLACE TRIGGER teacher_id_before_insert1
BEFORE INSERT ON teacher
REFERENCING NEW AS new
FOR EACH ROW
DECLARE
v_id NUMBER;
BEGIN
SELECT count(*) INTO v_id
FROM student
WHERE teacher.mem_id = :new.mem_id;
IF (v_id > 0) THEN
raise_application_error(-20001, :new.mem_id || 'this id is being used by a student');
END IF;
END;

5.8 Comparison of Trigger Features

Triggers in Oracle differ in several ways from the SQL standard. Table 5.4 shows the capabilities of the supported Trigger mechanisms in Oracle [43]:

<table>
<thead>
<tr>
<th>Trigger Feature</th>
<th>Oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td># event trigger</td>
<td>n</td>
</tr>
<tr>
<td># trigger per event</td>
<td>n</td>
</tr>
<tr>
<td>Trigger granularity</td>
<td>R/S</td>
</tr>
<tr>
<td>Activation time</td>
<td>B/A</td>
</tr>
<tr>
<td>Activation condition</td>
<td>√</td>
</tr>
</tbody>
</table>

Table 5.4 Triggers

- Number of events per trigger: Potential events that can lead to the firing of a trigger are: INSERT, UPDATE and DELETE operations. Oracle, in contrast, allows defining a trigger on multiple events on the same table.
- Number of triggers per events: Oracle does not have any restrictions. Any given table can have multiple triggers of the same type for the same statement (this is not supported in MySQL).
• Trigger granularity: The granularity of a trigger can be FOR EACH ROW (R) or FOR EACH STATEMENT (S). Oracle supports both kinds of triggers. The default granularity is FOR EACH STATEMENT in Oracle.
• Trigger activation time: A trigger is activated BEFORE (B) or AFTER (A) the specified event is handled. Oracle supports both kinds of triggers.
• Trigger activation condition. The activation of a trigger can be associated with a condition. Oracle also supports the enabling and disabling of triggers.

Trigger specifications [44]:

• No OLD TABLE or NEW TABLE in REFERENCING clause.
• Referencing clause uses OLD instead of OLD ROW and NEW instead of NEW ROW.
• WHEN clause allowed only with FOR EACH ROW.
• No sub-queries in WHEN condition. <Action> is PL/SQL procedure with some restrictions.

5.8.1 Restrictions on Trigger Conditions

• If this clause is specified for a DML event trigger, then FOR EACH ROW must also be specified. Oracle Database evaluates this condition for each row affected by the triggering statement.
• Trigger conditions for INSTEAD OF trigger statements cannot be specified.
• Object columns or their attributes, array, nested table or LOB columns can be referenced. PL/SQL functions or methods in the trigger condition cannot be invoked [45].

The NEW and OLD keywords, when specified in the WHEN clause, are not considered bind variables, so are not preceded by a colon (:). However, NEW and OLD must be proceeded with a colon in all references other than the WHEN clause.

Example 9: After a book has been ordered the order date is set on the table, giving members 3 days from that date to pick up the ordered item. According to the pickup date
if it was on time or not the state of the order is updated. This is an example of multiple events per trigger.

CREATE OR REPLACE TRIGGER check_if_pick_up_ontime 
BEFORE INSERT OR UPDATE ON orders 
REFERENCING NEW AS New 
FOR EACH ROW 
BEGIN 
  IF :New.pickupdate < :new.ORDERDATE + INTERVAL '03' DAY THEN 
    :new.state:= 'Done'; 
  
  UPDATE book
  SET num_copiesavailable=num_copiesavailable - 1 
  
  ELSE 
  IF :New.pickupdate > :new.ORDERDATE + INTERVAL '03' DAY THEN 
    :new.state:='failed'; 
  
  END IF;
  END IF;
END;

5.8.2 Restrictions on BEFORE Triggers

- A BEFORE trigger cannot be specified on a view or an object view.
- NEW values can be updated.
- OLD values cannot be updated.

Example 10: A BEFORE INSERT trigger could be created as follows:

Whenever the member id on the members table is updated, update the corresponding id on the teacher/student table (Implementation of the referential action ON UPDATE CASCADE using triggers).

CREATE OR REPLACE TRIGGER bfr_updt_mmb_r_updt_tchr_stdnt 
BEFORE UPDATE OF mem_id ON member 
REFERENCING old as old new as new 
FOR EACH ROW 
BEGIN 
  UPDATE teacher 
  SET mem_id=:new.mem_id 
  WHERE mem_id=:old.mem_id;
  UPDATE student
SET mem_id=:new.mem_id
    WHERE mem_id=:old.mem_id;
End;

5.8.3 Restrictions on AFTER Triggers

- An AFTER trigger cannot be created on a view.
- NEW values cannot be updated.
- OLD values cannot be updated.

Example 11: An AFTER INSERT trigger could then be created as follows:
After an order has been successfully completed and the item taken from the library, a
record of such order must be created in the order_history table:

CREATE OR REPLACE TRIGGER after_updt_on_order
    AFTER UPDATE ON orders
    REFERENCING NEW as new
    FOR EACH row
    WHEN (new.state='Done')
    BEGIN
        INSERT INTO order_history (history_id, startdate, book_id, card_num)
        VALUES ( :new.order_id, sysdate, :new.book_id, :new.card_num );
    END;

5.9 Trigger on view

Some views cannot be directly modified through DML statements (INSERT, UPDATE
and DELETE). INSTEAD OF triggers allow this limitation to be worked around by
firing the trigger instead of executing the DML statement. INSTEAD OF triggers are valid
for DML events on views. They are not valid for DDL or database events. Oracle
Database fine-grained access control allows row-level security policies on views to be
defined. These policies enforce specified rules in response to DML operations. If an
INSTEAD OF trigger is also defined on the view, then the database will not enforce the
row-level security policies because the database fires the INSTEAD OF trigger instead of
executing the DML on the view.
The INSTEAD OF trigger is available for views only - they cannot be placed on a table. This is the only type of trigger to place on a view. The trigger will execute separately for each row in the view. These provide a transparent way of updating a normally non-updateable view.

5.9.1 Restrictions on INSTEAD OF Triggers

- INSTEAD OF triggers are valid only for views. An INSTEAD OF trigger cannot be specified on a table.

The: OLD and the: NEW value can be read, but the: OLD or the: NEW value cannot be written either.

5.10 Automatic key generation

It is sometimes handy to have the DBMS handle generation of keys [46]. The DBMS’s offer various means for this. Oracle does not support the SQL: 2003 standard’s IDENTITY attribute or the MySQLs Auto-Increment option. If an auto-incrementing column is required in Oracle, then a sequence should be created and used in a trigger associated to the table.

*Example 12:* For the table “orders”, it is required that the order_id column is to be of integer type, with an auto-incrementing value:

```sql
CREATE TABLE orders (  
    order_id INTEGER PRIMARY KEY,  
    ... (Other columns)  
);

CREATE sequence order_seq  
start with 1  
increment by 1  
nomaxvalue;

CREATE OR REPLACE TRIGGER order_inc_trigger  
BEFORE INSERT ON orders  
FOR EACH ROW  
BEGIN  
    SELECT order_seq.nextval into: new.order_id FROM dual;  
    :new.orderdate:=sysdate;  
    :new.state:='order_processed';  
END;
```
This will create an auto-incrementing column resembling the `GENERATED BY DEFAULT` variant from the SQL: 2003 Standard.

### 5.11 Oracle scheduler

Oracle Scheduler is a feature of Oracle database. It enables jobs running inside the database such as PL/SQL procedures or PL/SQL blocks as well as jobs running outside the database such as shell scripts to be scheduled. There are two interfaces for the Oracle Scheduler: a UI which is part of Enterprise Manager and an API (DBMS_SCHEDULER package). It is a free feature of the database [47].

#### 5.11.1 The Database Scheduler

The Scheduler enables database administrators and application developers to control when and where various tasks take place. These tasks can be time consuming and complicated, so using the Scheduler can help us to improve the management and planning of these tasks. In addition, by ensuring that many routine database tasks occur without manual intervention, operating costs can be lowered, implementing more reliable routines and minimizing human error. The Scheduler uses three main components:

- **A schedule** specifies when and how many times a job is executed. Similar to programs, schedules are database entities and can be saved in the database. The same schedule can be used by multiple jobs.

- **A program** is a collection of metadata about what will be run by the scheduler. This includes information such as the program name, the type of program, and information about arguments passed to the program.

- **A job** specifies what needs to be executed and when. For example, a PL/SQL procedure, an executable C program, a java application, a shell script or client-side PL/SQL. The program (what) and schedule (when) can be specified as part of the job definition or use an existing program or schedule instead.
In the application of the library system, three schedules are needed that run a procedure on a regular basis everyday and makes some changes to our tables.

**Example 13:** A schedule is required that runs this procedure every day and checks if there are any members who have reached their study end date and if so set the still_student to ‘N’.

Begin

```
  dbms_scheduled_job.create_job(
    job_name => 'a_job_name_which_you_choose',
    job_type => 'STORED_PROCEDURE',
    job_action => 'NoneStudent',
    start_date => SYSTIMESTAMP,
    repeat_interval => 'FREQUENCY=DAYLY; INTERVAL=1',
    enabled => TRUE);
```

End;

CREATE OR REPLACE PROCEDURE NoneStudent
IS
  s_date DATE;
  CURSOR c IS SELECT study_enddate FROM student WHERE study_enddate=sysdate;
BEGIN
  Loop
    IF NOT c%ISOPEN THEN
      OPEN c;
    END IF;
    FETCH c INTO s_date;
    EXIT when c%NOTFOUND;
    UPDATE student
    SET
    still_student= 'N';
  END loop;
END;
6. Overview of the MySQL Database Management System

6.1 Introduction to MySQL

MySQL[37], the most popular Open Source SQL database management system, is developed, distributed and supported by MySQL AB. MySQL AB is a commercial company founded by the MySQL developers. It is a second generation Open Source company that unites Open Source values and methodology with a successful business model.

- MySQL is a database management system.
- MySQL is a relational database management system.
- MySQL software is Open Source.
- The MySQL database server is very fast, reliable and easy to use.
- MySQL Server works in client/server or embedded systems.

6.2 The Main Features of MySQL

In this part some of the important characteristics of the MySQL Database Software shall be stated [48].

- Data types
- Statements and Functions
- Security
- Scalability and limits
- Partitioning
- Row based replications
- Event schedulers
6.3 MySQL Standards Compliance

MySQL Server has many extensions to the SQL standard and here it will be shown what they are and how they are used. Information will also be presented about functionality missing from the MySQL Server, and how some of these differences can be worked around. One of MySQL’s main goals with the product is to continue to work toward compliance with the SQL standard, without sacrificing speed or reliability. It is not afraid to add extensions to SQL or support for non-SQL features if this greatly increases the usability of MySQL Server for a large segment of the user base [48]. MySQL Server was originally designed to work with medium-sized databases (10-100 million rows, or about 100MB per table) on small computer systems. Today MySQL Server handles terabyte-sized databases, but the code can also be compiled in a reduced version suitable for hand-held and embedded devices. The compact design of the MySQL server makes development in both directions possible without any conflicts in the source tree. MySQL follows the standard Entry level SQL92. MySQL includes some extensions that cannot be found in other SQL databases. Be warned that if they are used, the code will not be portable to other SQL servers.

6.4 MySQL Differences from Standard SQL

It basically means that it is very hard to write code which supports both MySQL and SQL99 compliant database management systems. This is one of the major problems MySQL users are facing these days: their code can only run on MySQL back ends. It was attempted to make MySQL Server follow the ANSI SQL standard and the ODBC SQL standard, but MySQL Server performs operations differently in some cases. For example [48]:

- There are several differences between the MySQL and standard SQL privilege systems. For example, in MySQL, privileges for a table are not automatically revoked when you delete a table. You must explicitly issue a `REVOKE` statement to revoke privileges for a table.
- The `CAST()` function does not support cast to `REAL` or `BIGINT` and etc. ()
- The differences in integrity constraint support shall be covered later on.
6.5 MySQL constraint support matrix

In this section it will be shown how MySQL handles the integrity constraint differently for the standard and list integrity constraints supported by MySQL in Table 6.1.

```
<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>ON DELETE</th>
<th>ON UPDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT NULL</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DEFAULT</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>UNIQUE</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PRIMARY KEY</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FOREIGN KEY</td>
<td>NO ACTION</td>
<td>NO ACTION</td>
</tr>
<tr>
<td></td>
<td>RESTRICT</td>
<td>RESTRICT</td>
</tr>
<tr>
<td></td>
<td>CASCADE</td>
<td>CASCADE</td>
</tr>
<tr>
<td></td>
<td>SET NULL</td>
<td>SET NULL</td>
</tr>
<tr>
<td></td>
<td>SET DEFAULT</td>
<td>SET DEFAULT</td>
</tr>
<tr>
<td>CHECK</td>
<td>column-level</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>row-level</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>table-level</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>database-level</td>
<td>✓</td>
</tr>
<tr>
<td>DOMAIN</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ASSERTION</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
```

Table 6.1 Declarative Integrity Constraint

MySQL allows work to be done both with transactional tables that allow rollback and with non-transactional tables that do not. Because of this, constraint handling is a bit different in MySQL than in other DBMSs. This case must be handled when a lot of rows have been inserted or updated in a non-transactional table for which changes cannot be rolled back when an error occurs. The basic philosophy is that MySQL Server tries to produce an error for anything that it can detect while parsing a statement to be executed, and tries to recover from any errors that occur while executing the statement. The options MySQL has when an error occurs are to stop the statement in the middle or to recover as well as possible from the problem and continue. By default, the server
follows the latter course. This means, for example, that the server may coerce illegal values to the closest legal values. Several SQL mode options are available to provide greater control over handling of bad data values and whether to continue statement execution or abort when errors occur. Using these options, MySQL Server can be configured to act in a more traditional fashion that is like other DBMSs that reject improper input. The SQL mode can be set globally at server startup to affect all clients. Individual clients can set the SQL mode at runtime, which enables each client to select the behavior most appropriate for its requirements [50].

6.5.1 SQL Modes:

The MySQL server can operate in different SQL modes and can apply these modes differentially for different clients. This capability enables each application to tailor the server's operating mode to its own requirements. SQL modes control aspects of server operation such as what SQL syntax MySQL should support and what kind of data validation checks it should perform. This makes it easier to use MySQL in different environments and to use MySQL together with other database servers. The default SQL mode can be set by starting `mysqld` with the `--sql-mode="mode_value"` option. The mode can also be changed at runtime by setting the `sql_mode` system variable with a `SET [SESSION|GLOBAL] sql_mode='mode_value'` statement. MySQL Server supports transactions with the InnoDB transactional storage engine. InnoDB provides full ACID compliance [48].

The other non-transactional storage engines in MySQL Server (such as MyISAM) follow a different paradigm for data integrity called “atomic operations”. In transactional terms, MyISAM tables effectively always operate in AUTOCOMMIT=1 mode. Atomic operations often offer comparable integrity with higher performance. Because MySQL Server supports both paradigms, it can be decided whether the applications are best served by the speed of atomic operations or the use of transactional features. This choice can be made on a per-table basis. As noted, the trade-off for transactional versus non-transactional storage engines lies mostly in performance. Transactional tables have significantly higher memory and disk space requirements, and more CPU overhead. On
the other hand, transactional storage engines such as InnoDB also offer many significant
features. MySQL Server’s modular design allows the concurrent use of different storage
engines to suit different requirements and deliver optimum performance in all situations.

6.5.2 Nullability and Default Values

MySQL not null support is listed in Table 6.2:

<table>
<thead>
<tr>
<th>NOT NULL on</th>
<th>MYSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row type</td>
<td>√</td>
</tr>
<tr>
<td>Reference type</td>
<td>-</td>
</tr>
<tr>
<td>Collection type</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 6.2 Not null constraints

SQL NULL represents the absence of a value. In MySQL an explicit NULL may also
represent the next value of a pseudo-sequence and an implicit NULL may represent an
implicit default value (a zero or empty string) determined by MySQL. If no DEFAULT
value is specified for a column, MySQL automatically assigns one as follows. If the
column may take NULL as a value - the default value is NULL. If the column is
declared as NOT NULL - the default value depends on the column type [49]. Default
values can not be a function or an expression, it must be a constant. The default cannot
be set for a date column to be the value of a function such as now () of current_date.
Current_timestamp can be defined as default for a timestamp column.

6.5.3 PRIMARY KEY / UNIQUE Index Constraints

A PRIMARY KEY is a unique index where all key columns must be defined as Not
Null, if not explicitly declared as NOT NULL, MySQL declares them so implicitly. A
table can have only one PRIMARY KEY. If there is no PRIMARY KEY and an
application asks for the PRIMARY KEY in the tables, MySQL returns the first
UNIQUE index that has no NULL columns as the PRIMARY KEY. In the created table,
a PRIMARY KEY is placed first, followed by all UNIQUE indexes, and then the non-
unique indexes. This helps the MySQL optimizer to prioritize which index to use and
also more quickly to detect duplicated UNIQUE keys. A PRIMARY KEY can be a
multiple-column index. However, multiple-column index can not be created using the
PRIMARY KEY attribute in a column specification. Doing so only marks that single column as primary. In this case a separate PRIMARY KEY \((\text{index\_col\_name} \ldots)\) clause must be used. \textit{Table 6.3} lists the MySQL's unique/primary key support:

<table>
<thead>
<tr>
<th>Row type</th>
<th>MySQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference type</td>
<td>-</td>
</tr>
<tr>
<td>Collection type</td>
<td>√</td>
</tr>
</tbody>
</table>

\textbf{Table 6.3 Unique/primary key constraints}

If a PRIMARY KEY or UNIQUE index consists of only one column that has an integer type, then the column can also be referred to as row id in SELECT statements. An integer column can have the additional attribute \texttt{AUTO_INCREMENT}. When a value of \texttt{NULL} (recommended) or 0 is inserted into an indexed \texttt{AUTO_INCREMENT} column, the column is set to the next sequence value. Typically this is \texttt{value+1}, where \texttt{value} is the largest value for the column currently in the table. \texttt{AUTO_INCREMENT} sequences begin with 1 [46]. \textit{Example 1:}

```sql
CREATE TABLE orders (      
    Order_id INTEGER PRIMARY KEY AUTO_INCREMENT,
    ... 
);
```

Normally an error occurs when it is attempted to INSERT or UPDATE a row that causes a primary key, unique key or foreign key violation. If a transactional storage engine is being used, such as InnoDB, then MySQL will automatically roll back the statement. If a non-transactional storage engine is being used, then MySQL will stop processing the statement at the row which the error occurred and will leave any remaining rows unprocessed. To ignore such key violations, MySQL supports an \texttt{IGNORE} keyword for INSERT and UPDATE. In this case, MySQL ignores any key violations and continues processing with the next row.

\textbf{6.5.4 Constraints on Invalid Data}

By default, MySQL is forgiving of illegal or improper data values and forces them to legal values for data entry. However, the server SQL mode can be changed to select
more traditional treatment of bad values such that the server rejects them and aborts the statement in which they occur. This section describes the default (forgiving) behavior of MySQL, as well as the strict SQL mode and how it differs. If strict mode is not being used, then whenever an “incorrect” value has been inserted into a column, such as a NULL into a NOT NULL column or a too-large numeric value into a numeric column, MySQL sets the column to the “best possible value” instead of producing an error. The following rules describe in more detail how this works [48]:

- If an out of range value is trying to be stored into a numeric column, MySQL Server instead stores zero, the smallest possible value or the largest possible value, whichever is closest to the invalid value.
- For strings, MySQL stores either the empty string or as much of the string as can be stored in the column.
- If a string that doesn't start with a number is trying to be stored into a numeric column, MySQL Server stores 0.
- MySQL allows certain incorrect date values to be stored into DATE and DATETIME columns (such as '2000-02-31' or '2000-02-00').
- To try to store a NULL into a column that does not take NULL values, an error occurs for single-row INSERT statements. For multiple-row INSERT statements or for INSERT INTO ... SELECT statements, MySQL Server will store the implicit default value for the column data type. In general this is 0 for numeric types, the empty string ('') for string types and the “zero” value for date and time types.
- If an INSERT statement specifies no value for a column, MySQL inserts its default value if the column definition includes an explicit DEFAULT clause. If the definition has no such DEFAULT clause, MySQL inserts the implicit default value for the column data type.

The reason for using the preceding rules in non-strict mode is that these conditions cannot be checked until the statement has begun executing. If a problem is encountered after updating a few rows it cannot just be rolled back because the storage engine may not support rollback. The option of terminating the statement is not that good; in this case, the update would be “half done”, which is probably the worst possible scenario. In
this case it is better to “do the best you can” and then continue as if nothing happened. This is entirely up to the user to select an even stricter treatment of input values by using the `STRICT_TRANS_TABLES` or `STRICT_ALL_TABLES` SQL modes.

### 6.5.5 CHECK Constraints

In some other databases it is possible to use check constraint on a table to maintain data integrity and enforce rules on the type and range of data that is entered into them. For example, other types of constraints have been used with MySQL such as primary key or unique constraint which raised an error if a duplicate value had entered into a column. Check constraints are similar to that they are defined when creating the table, they differ however in the fact that the table creator can define how the constraint operates rather than being limited to the rules defined in the small set that already exists. Using a check constraint it can be checked simply when the insert takes place. SQL: 99 and Oracle both allow the use of check constraints and the table creation script has been seen for it in the previous cases [37].

Since MySQL does not support check constraints, a method of checking the values being inserted into the tables must be created. This can be done using an update statement but at that point it will not be know if the whole record should have been rejected or just the column which failed the check constraint check. What is required is a method of rejecting the record, or at least something telling the user that a column is failing the constraint check at the point the value is inserted or updated in the table.

The best way to do this is to use a trigger, however MySQL procedural language does not currently support the independent raising of an error, prior to 5.0.10 it was also not possible to call SQL to artificially raise an error, but even that is not an acceptable solution because the error wouldn’t have been specific enough to tell the user what went wrong. However with the release of 5.0.15 MySQL now offers a solution. Previously when an error was raised only the error number and standard message were returned to the user, from version 5.0.15 MySQL now also returns the value which you tried to insert. To do this, two things must be created:

- A table to produce the error against
- A procedure to insert into that table which will in turn raise an error.
The table definition is as follows:

```sql
CREATE TABLE `Error` (
    `ErrorGID`  INT (10) unsigned NOT NULL auto_increment,
    `Message`  VARCHAR (128) DEFAULT NULL,
    `Created`  timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP
        ON update CURRENT_TIMESTAMP,
    PRIMARY KEY (`ErrorGID`),
    UNIQUE KEY `MessageIndex` (`Message`)
) ENGINE=MEMORY
    DEFAULT CHARSET=latin1
    ROW_FORMAT=FIXED
```

The important things to note about this table are: the unique key in the Message column, this allows the error to be easily created and the fact that the table uses the MEMORY engine. This means the table is stored in memory and not in the file system, using that there is no need to clear the table down or worry about the table becoming full [51].

The next thing which needs to be created is a procedure to insert into this table to raise the unique key violation; this stage isn’t strictly necessary but makes the trigger code that will be produced next much more readable. It also means the same routine can be consistently called in any number of triggers (or even in Functions and Procedures).

```sql
DELIMITER $$
DROP PROCEDURE IF EXISTS `Fail`$$
CREATE PROCEDURE `Fail`(_Message VARCHAR(128))
BEGIN
    INSERT INTO Error (Message) VALUES (_Message);
    INSERT INTO Error (Message) VALUES (_Message);
END$$
DELIMITER;
```

The procedure accepts a parameter called _Message; this is then used as the message column value in the inserts. By calling the same insert statement twice the unique_key constraint is violated on the error table, because MySQL reports the column value in the error message this _Message parameter is added to the error message. This procedure can be called from the command line to see this in action.

```sql
mysql> call fail('age must be over 18');
ERROR 1062 (23000): Duplicate entry ‘age must be over 18’ for key 2
```

There are two things to note here; firstly the error message is returned which is great as it means the user can clearly see why the error was raised. The second thing to notice is
that the error number and description point to a duplicate entry error; this isn’t ideal but for now is the only way to raise the error. This table and procedure are now capable of raising an error safely without affecting any other part of the database. That is one of the key things here, the error could be raised without using a specific table or the procedure but doing so would not affect any other part of the database.

*Example 2*: Information about each member age is stored in the members table. A trigger shall be created which will enforce a constraint on the column member age so that the value cannot be less than 18. This now needs to be added to a trigger so that the error is raised when the constraint fails.

```
DELIMITER $$
Create trigger age_check before insert on member for each row
Begin
  If new.mem_age < 18 then
    call fail('age not in allowed range');
  End if;
End $$
DELIMITER;
```

The trigger simply checks the value of the new.mem_age column, if it’s not within the parameters allowed a call to the fail routine is called. This will then stop the record being inserted into the table. Let’s look at that in action:

```
mysql> insert into member (mem_id,mem_firstname,mem_lastname,mem_num,age)
  > Values (1002,'Penny','clarck','pent1',17);
ERROR 1062 (23000): Duplicate entry 'age not in allowed range' for key 2
```

### 6.5.5.1 ENUM and SET Constraints

**ENUM** and **SET** columns provide an efficient way to define columns that can contain only a given set of values. However, before MySQL 5.0.2, **ENUM** and **SET** columns did not provide true constraints on entry of invalid data [48]:

- **ENUM** columns always have a default value. If no default value is specified, then it is **NULL** for columns that can have **NULL**, otherwise it is the first enumeration value in the column definition.
Case Study: Implementation of Integrity Constraints in Actual Database System

- If an incorrect value is inserted into an `ENUM` column or a value is forced into an `ENUM` column with `IGNORE`, it is set to the reserved enumeration value of `0`, which is displayed as an empty string in string context.

- If an incorrect value is inserted into a `SET` column, the incorrect value is ignored.

With strict mode enabled, the definition of an `ENUM` or `SET` column does act as a constraint on values entered into the column. An error occurs for values that do not satisfy these conditions:

- An `ENUM` value must be one of those listed in the column definition or the internal numeric equivalent thereof.

- A `SET` value must be the empty string or a value consisting only of the values listed in the column definition separated by commas.

*Example 3:* Members of the library must have a valid address in prague1 or prague2, and member types are students or teachers.

Create table member (  
  mem_city  ENUM('PRAGUE1','Prague2') not null,
  mem_type   ENUM ('STUDENT','TEACHER') NOT NULL ,
  . . );

### 6.5.6 Foreign Key Constraints and Restrictions

In MySQL Server 5.1, the InnoDB storage engine supports checking of foreign key constraints, including CASCADE, ON DELETE and ON UPDATE. For storage engines other than InnoDB, MySQL Server parses the FOREIGN KEY syntax in CREATE TABLE statements but does not use or store it. Foreign key enforcement offers several benefits to database developers [37]:

- Assuming proper design of the relationships, foreign key constraints make it more difficult for a programmer to introduce an inconsistency into the database.

- Centralized checking of constraints by the database server makes it unnecessary to perform these checks on the application side. This eliminates the possibility that different applications may not all check the constraints in the same way.
• Using cascading updates and deletes can simplify the application code.
• Properly designed foreign key rules aid in documenting relationships between tables.

MySQL gives database developers the choice of which approach to use. If foreign keys are not needed and want to avoid the overhead associated with enforcing referential integrity, another storage engine can be chosen instead, such as MyISAM.

Foreign keys definitions are subject to the following conditions [37]:

• Both tables must be InnoDB tables and they must not be TEMPORARY tables.
• Corresponding columns in the foreign key and the referenced key must have similar internal data types inside InnoDB so that they can be compared without a type conversion. The size and sign of integer types must be the same. The length of string types need not be the same. For non-binary (character) string columns, the character set and collation must be the same.
• In the referencing table, there must be an index where the foreign key columns are listed as the first columns in the same order. Such an index is created on the referencing table automatically if it does not exist.
• In the referenced table, there must be an index where the referenced columns are listed as the first columns in the same order.
• Index prefixes on foreign key columns are not supported. One consequence of this is that BLOB and TEXT columns cannot be included in a foreign key, because indexes on those columns must always include a prefix length.
• If the CONSTRAINT symbol clause is given, the symbol value must be unique in the database. If the clause is not given, InnoDB creates the name automatically.

InnoDB rejects any INSERT or UPDATE operation that attempts to create a foreign key value in a child table if there is no a matching candidate key value in the parent table. The action InnoDB takes for any UPDATE or DELETE operation that attempts to update or delete a candidate key value in the parent table that has some matching rows in the child table is dependent on the referential action specified using ON UPDATE and
ON DELETE sub-clauses of the FOREIGN KEY clause. When the user attempts to delete or update a row from a parent table, and there are one or more matching rows in the child table, InnoDB supports five options regarding the action to be taken:

- **CASCADE**: Delete or update the row from the parent table and automatically delete or update the matching rows in the child table. Both ON DELETE CASCADE and ON UPDATE CASCADE are supported. Between two tables, several ON UPDATE CASCADE clauses should not be defined that act on the same column in the parent table or in the child table.
- **SET NULL**: Delete or update the row from the parent table and set the foreign key column or columns in the child table to NULL. This is valid only if the foreign key columns do not have the NOT NULL qualifier specified. Both ON DELETE SET NULL and ON UPDATE SET NULL clauses are supported. When a SET NULL action is specified.
- **NO ACTION**: In standard SQL, NO ACTION means no action in the sense that an attempt to delete or update a primary key value is not allowed to proceed if there is a related foreign key value in the referenced table. InnoDB rejects the delete or update operation for the parent table.
- **RESTRICT**: Rejects the delete or update operation for the parent table. NO ACTION and RESTRICT are the same as omitting the ON DELETE or ON UPDATE clause.
- **SET DEFAULT**: This action is recognized by the parser, but InnoDB rejects table definitions containing ON DELETE SET DEFAULT or ON UPDATE SET DEFAULT clauses.

*Example 4*: In the book table the members id information is kept, which is a foreign key on this table:

```sql
CREATE TABLE book (
    Book_id    INTEGER PRIMARY KEY,
    Dep_id    INTEGER,
    Year_of_publication  DATE,
    Price     decimal (5.2),
    ISBN_num   CHAR (20) NOT NULL,
    Book_title  VARCHAR (30) NOT NULL,
);```

74
Publisher VARCHAR (30),
Book_type ENUM ('text book', 'lecture note') NOT NULL,
Num_copiesavailable INTEGER NOT NULL,
CONSTRAINT FOREIGN KEY (dep_id) REFERENCES department (dep_id)
ON DELETE SET NULL ON UPDATE CASCADE
) ENGINE=InnoDB;

6.6 MySQL Triggers

MySQL triggers are activated by SQL statements only. The trigger becomes associated with the table named tbl_name, which must refer to a permanent table. A trigger cannot be associated with a TEMPORARY table or a view [37]. MySQL's trigger support is listed in Table 6.4:

<table>
<thead>
<tr>
<th></th>
<th>MySQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>#events per trigger</td>
<td>1</td>
</tr>
<tr>
<td>#triggers per event</td>
<td>1</td>
</tr>
<tr>
<td>Trigger granularity</td>
<td>R</td>
</tr>
<tr>
<td>Activation time</td>
<td>B/A</td>
</tr>
<tr>
<td>Activation condition</td>
<td>√</td>
</tr>
</tbody>
</table>

Table 6.4 Triggers

trigger_time is the trigger action time. It can be BEFORE or AFTER to indicate that the trigger activates before or after the statement that activated it.

trigger_event indicates the kind of statement that activates the trigger. The trigger_event can be one of the following: INSERT, UPDATE and DELETE.

trigger_stmt is the statement to execute when the trigger activates. For executing multiple statements, BEGIN ... END compound statement construct is used. This also enables the use of the same statements that are allowable within stored routines.

OLD and NEW are MySQL extensions to triggers. Columns in the subject table (the table associated with the trigger) can be referred to by using the aliases OLD and NEW. OLD.col_name refers to a column of an existing row before it is updated or deleted. NEW.col_name refers to the column of a new row to be inserted or an existing row after it is updated [37].
6.6.1 Trigger Limitations

There are other limitations on the types of triggers that can be created in MySQL:

- In particular, there cannot be two triggers for a table that has the same activation time and activation event. For example, two `BEFORE INSERT` triggers or two `AFTER UPDATE` triggers cannot be defined for a table (as it was seen earlier, this was not the case in Oracle triggers). This should rarely be a significant limitation, because it is possible to define a trigger that executes multiple statements by using the `BEGIN ... END` compound statement construct after `FOR EACH ROW`.

- All triggers are For Each Row; the trigger is achieved for each row that is inserted, updated or deleted. MySQL does not support triggers using For Each Statement.

6.6.2 Trigger Error Handling

MySQL handles errors during trigger execution as follows:

- If a `BEFORE` trigger fails, the operation on the corresponding row is not performed.
- An `AFTER` trigger is executed only if the `BEFORE` trigger (if any) and the row operation both execute successfully.
- An error during either a `BEFORE` or `AFTER` trigger results in failure of the entire statement that caused trigger invocation.
- For transactional tables, failure of a trigger (and thus the whole statement) should cause rollback of all changes performed by the statement. For non-transactional tables such rollback cannot be done, so although the statement fails any changes performed prior to the point of the error remain in effect.

*Example 5:* Before an order can be placed by a member the members card validity and the books state, whether it is available or not, must be checked:
In the previous case Oracle, our trigger was implemented in a way that if a member placing the order has a problematic card or a book being ordered is not allowed or available, then it would raise an application error indicating what the problem is. Since MySQL does not have a support for raising application errors two attributes, card_problem and book_problem, are added to the orders table to keep the members informed if any problem arises.

Delimiter $$
Create trigger bfr_insrt_order before insert on orders
For each row
Begin
    Declare v_notfound bool default false;
    Declare v_id integer;
    Declare b_id integer;
    Declare cur1 cursor for select card_num from card where status='blocked' or card_renew='yes';
    Declare cur2 cursor for select book_id from book where book_type='lecture note' or num_copiesavailable=0;
    Declare continue handler for not found set v_notfound:=true;
    Open cur1;
    Open cur2;
    Cursor_loop: loop
        Fetch cur1 into v_id;
        Fetch cur2 into b_id;
        If v_notfound then leave cursor_loop;
        End if;
        If new.card_num=v_id then
            Set new.card_problem='yes'
        End if;
        If new.book_id=b_id then
            Set new.book_problem='yes';
        End if;
        If new.card_num <> v_id and new.book_id<>b_id then
            Set new.orderdate=current_date();
            Set new.state='order_processed';
        End if;
    End loop cursor_loop;
    Close cur1;
    Close cur2;
End$$ DELIMITER;
6.7 Event Scheduler Overview

MySQL Events are tasks that run according to a schedule. Therefore, they are sometimes referred to as scheduled events. When an event is created, a named database object is created containing one or more SQL statements to be executed at one or more regular intervals, beginning and ending at a specific date and time. Scheduled tasks of this type are also sometimes known as “temporal triggers”, implying that these are objects that are triggered by the passage of time. Events should more specifically not be confused with “temporary triggers”. Whereas a trigger is a database object whose statements are executed in response to a specific type of event that occurs on a given table, a (scheduled) event is an object whose statements are executed in response to the passage of a specified time interval. While there is no provision in the SQL Standard for event scheduling, there are precedents in other database systems and some similarities may be noticed between these implementations and those found in the MySQL. The global variable event_scheduler determines whether the Event Scheduler is enabled and running on the server. Event scheduling can be stopped by setting the value of event_scheduler to OFF. When the server is running event_scheduler can be toggled between ON and OFF (using SET) [37].

Example 6: To be able to keep track of students who have reached their study_enddate, a schedule needs to be created that runs this procedure regularly every day. Now an event scheduler must be created that checks the students study_enddates and sets the still_student value:

Delimiter $$
Create event e_daily
On schedule
Every 1 minute
Comment’ student finished study?’
Do begin
Update student
Set still_student= ‘false’ where study_enddate= current_date;
End $$
7. Overview of the Microsoft SQL Server Database Management System

7.1 Introduction to MS SQL

Microsoft SQL Server is a relational database management system (RDBMS) produced by Microsoft. Its primary query language is Transact-SQL, an implementation of the ANSI/ISO standard Structured Query Language (SQL) used by both Microsoft and Sybase. SQL Server 2005 is the new edition of Microsoft’s SQL Server client-server relational database. It is a major release - the first in five years. SQL Server 2005 has many new features that help manage a relational database and, in many editions, adds important new business intelligence functionality. [36] Microsoft SQL Server 2005 is comprehensive integrated data management and analysis software that enables organizations to reliably manage mission-critical information and confidently run today’s increasingly complex business applications. SQL Server 2005 allows companies to gain greater insight from their business information and achieve faster results for a competitive advantage. MS SQL conforms to at least standard SQL-92.

7.1.1 Key Capabilities

- High Availability: Ensure business continuity with the highest levels of system availability through technologies that protect data against costly human errors and minimize disaster recovery downtime [36].
- Performance and Scalability: Deliver an infrastructure that can grow with the business and has a proven record in handling today's large amounts of data and most critical enterprise workloads.
- Security: Provide a secure environment to address privacy and compliance requirements with built-in features that protect your data against unauthorized access.
- Manageability: Manage infrastructure with automated diagnostics, tuning and configuration to reduce operational costs while reducing maintenance and easily managing very large amounts of data.
• Developer Productivity: Build and deploy critical business-ready applications more quickly by improving developer productivity and reducing project life cycle times.

• Business Intelligence: Gain deeper insight into the business with integrated, comprehensive analysis and reporting for enhanced decision making.

**7.2 MS SQL Constraint Support**

Constraints limit or control the types of data that the user can enter into the tables. There are seven main categories of constraints. They include primary key constraints, foreign key constraints, default constraints, not null constraints, check constraints, rules and unique constraints. MS SQL allows constraining the values allowed in a particular column so that inappropriate data is not permitted. MS SQL’s Declarative Integrity Support matrix is presented in *Table 7.1:*

```
<table>
<thead>
<tr>
<th>Constraint</th>
<th>NOT NULL</th>
<th>DEFAULT</th>
<th>UNIQUE</th>
<th>PRIMARY KEY</th>
<th>FOREIGN KEY</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ON DELETE</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NO ACTION</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RESTRICT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CASCADE</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SET NULL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SET DEFAULT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>ON UPDATE</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NO ACTION</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RESTRICT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CASCADE</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SET NULL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SET DEFAULT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>CHECK</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>column-level</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>row-level</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>table-level</td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>database-level</td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ASSERTION</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

*Table 7.1 Declarative Integrity Support*
The text that follows covers each of these constraint types in detail. SQL Server 2005 supports these types of constraint:

- Not null
- Unique
- Primary/foreign key
- Check constraint

### 7.2.1 Primary Key Constraints

When a table is created, typically a primary key is added to the table. A table may have only one primary key. A column intended as a primary key cannot have NULL values and each value in the column must be unique. If the table has data and these conditions are not met, the attempt to create a primary key on that column causes an error and no primary key is created. [52]

Every table in the database should have a primary key constraint. Furthermore, it is best if the primary key meets the following criteria: short, stable and simple.

- Short means that it should be composed of as few fields as possible and the smaller the field type is, the better. In fact, the optimal primary key is a single int field.
- Stable means that the data within the field never changes. A great candidate for a primary key is an identity column.
- Simple means that it is easy to remember and deal with.

### 7.2.2 Unique Constraints

A unique constraint requires that each entry in a particular column be unique. Each table can have 249 unique constraints. A unique constraint can be created by creating a unique index. MS SQL server follows the standard support of unique constraint with a twist: MS SQL offers the NULLs allowed feature but allows at most one instance of a NULL-value, if NULLs are allowed. [46]

Create table member (mem_id INT Not Null PRIMARY KEY, Mem_num VARCHAR (20) NOT NULL UNIQUE, ....);
MS SQL has no restrictions concerning the definition of multiple uniqueness constraints on overlapping sets of columns.

### 7.2.3 Null Constraint

Although not a constraint in the strictest definition, the decision to allow NULL values in a column or not is a type of rule enforcement for domain integrity. SQL NULL or NOT NULL can be used on a column definition to explicitly set the nullability of a column. In the following example table, the member FirstName column will accept NULL values while member LastName always requires a non NULL value. Primary key columns require a NOT NULL setting, and default to this setting if not specified [53].

```sql
CREATE TABLE member (  
    Mem_ID   INT PRIMARY KEY,  
    mem_FirstName  VARCHAR(50) NULL,  
    mem_LastName  VARCHAR(50) NOT NULL,  
    . . )
```

If a column is not explicitly set to allow or disallow NULL values, the database uses a number of rules to determine the nullability of the column, including current configuration settings on the server. It is recommended to always define a column explicitly as NULL or NOT NULL in the scripts to avoid problems when moving between different server environments.

Columns in a unique constraint can be declared to allow NULL values. However, the constraint checking considers NULL values as equal, so on a single column unique constraint; the database allows only one row to have a NULL value. In nearly all of the RDBMSs columns are nullable by default unless null values are explicitly disallowed. MS SQL has no restriction on the order of the specification of declarative constraints and default values.

### 7.2.4 Default Constraints

A default constraint is a value that SQL Server automatically places in a particular field in a table. A default value can be a constant, Null, or a function. All fields except identity and time stamp fields can contain default values. Each column can have one
default constraint. For example the default value can contain GetDate(), Null, 7, ‘hello’ and etc. An example of the default constraint is shown below:

Create table member (  
  mem_id       INT Not Null PRIMARY KEY,  
  Phone        CHAR (16) DEFAULT ‘UNKNOWN’,  
  . . .)

7.2.5 Check Constraints

Check constraints contain an expression the database will evaluate when a row is modified or inserted. If the expression evaluates to false, the database will not save the row. Building a check constraint is similar to building a WHERE clause. Similar operators such as (>, <, <=, >=, <>, =) can be used in additional to BETWEEN, IN, LIKE, and NULL. Also expressions can be build around AND, OR operators. Check constraints can be used to implement business rules and tighten down the allowed values and formats allowed for a particular column.

Create table member (  
  mem_id       INT Not Null PRIMARY KEY,  
  Mem_city     VARCHAR (15) NOT NULL CONSTRAINT check_city CHECK (city IN (‘prague1’,’prague2’)),  
  Mem_type     CHAR (12) NOT NULL CONSTRAINT check_type CHECK (mem_type IN (‘student’,’teacher’)),  
  . . .)

Check constraints limit the range of values that a user can enter into a column. Many check constraints can be created for a particular column. SQL Server evaluates the check constraints in the order in which you entered them. For example check constraints can contain check condition “@State in (’CA’, ’AZ’, ’UT’, ’CO’)” which limits the value entered to CA, AZ, UT and CO.

7.2.5.1 Restrictions on Check Constraints

Although check constraints are by far the easiest way to enforce domain integrity in a database, they do have some limitations, namely [53]:

- A check constraint cannot reference a different row in a table.
- A check constraint cannot reference a column in a different table.
7.2.6 Rules
Whereas Check Constraints apply only to the table for which they are entered, rules apply to multiple tables. Microsoft is phasing out support for rules. Therefore it is not allowed to create new rules. Instead of using rules, check constraints and triggers must be used.

7.3 Automatic key generation
It's sometimes handy to have the DBMS handle generation of keys. The DBMSs offer various means for this. Note, however, that some database authorities warn against - at least some variants of - auto-generated keys; this is a classic database discourse [46]. MSSQL offers IDENTITY as a column property, but with a different syntax (not as intuitive and with less option) than the standard's specification. An example of creating a table with an IDENTITY column:

CREATE TABLE orders (
    order_id INT IDENTITY PRIMARY KEY
    ...
)

With MSSQL's IDENTITY attribute, the user cannot manually insert/change the value, unless the user has first run SET IDENTITY_INSERT tablename ON MSSQL’s IDENTITY type is closest to the SQL: 2003 standards GENERATED ….ALWAYS AS IDENTITY variant.

7.3.1 Identity Columns
Identity columns provide an auto incrementing value for a table. An identity column should be used as the primary key field for any table that has no natural primary key that is short, stable and simple. Identity columns are often of the Int data type. The properties of the field are used to designate a column as an identity column. When a column is being designated as an identity column both the identity seed and the identity increment can be designated. The identity seed is the starting value for the field. The identity increment is the value by which each automatically assigned value is incremented. For
example, an identity field with an identity seed of 100 and an identity increment of 5 assigns the values 100, 105, 110 and so on.

This property is used with the CREATE TABLE and ALTER TABLE Transact-SQL statements. IDENTITY [(seed, increment)]

Seed: The value that is used for the very first row loaded into the table.
Increment: The incremental value that is added to the identity value of the previous row that was loaded. Both the seed and increment must be specified or neither. If neither is specified, the default is (1, 1). If an identity column exists for a table with frequent deletions - gaps can occur between identity values. If this is a concern, do not use the IDENTITY property.

CREATE TABLE orders (Card_num INT NOT NULL,
                      Order_id INT IDENTITY NOT NULL PRIMARY KEY , Book_id INT NOT NULL,
                      ...);

7.4 Foreign key and Referential Action

SQL Server enables defining rules that dictate what will happen when the user deletes or updates a record. These rules can be found under the INSERT and UPDATE Specification node of the Foreign Key Relationships window. [54]

7.4.1 The Delete Rule

Setting the Delete rule, determines what happens when the user deletes a record on the one side of a one-to-many relationship. For example, by setting the Delete rule to Cascade the rule can be established so that the user can delete a record on the one side of a one-to-many relationship, even if related records exist in the table on the many side of the relationship. The user can delete a member even if the member has existing orders, for example. Referential integrity is maintained between the tables because SQL Server automatically deletes all related records in the child table. Setting the Delete rule to Cascade is not always appropriate. It is an excellent feature, but it must be used prudently.

CREATE TABLE orders (...
                      Constraint orders_card_fk
                      FOREIGN KEY (card_num)
REFERENCES CARD (card_num) 
    ON DELETE CASCADE 
    ON UPDATE CASCADE,
Constraint orders_book_fk
FOREIGN KEY (book_id)
REFERENCES book(book_id)
    ON DELETE CASCADE 
    ON UPDATE CASCADE);

7.4.2 The Update Rule

With the Update rule set to Cascade, the user can change the primary key value of the 
record on the one side of the relationship. When the user makes an attempt to modify the 
field joining the two tables on the one side of the relationship, the change is cascaded 
down to the foreign key field on the many side of the relationship. This is useful if the 
primary key field is modifiable. Other options for the Delete and Update rules include 
No Action, Set Null and Set Default. No Action, the default value, does nothing and 
therefore does not allow the deletion of parent records that have children or the 
modification of the key field(s) of parent records that have children. Set Null sets the 
value of the foreign key field to Null. Finally, Set Default sets the value of the foreign 
key field to its default value.

7.4.3 SET NULL and SET DEFAULT

The referential actions SET NULL and SET DEFAULT compensate for modifications 
to the primary table by setting the related columns in the child rows in the secondary 
table to NULL or to their default value. These actions were not supported as declarative 
referential constraints in SQL Server 2000; they were implemented by using triggers.

CREATE TABLE order_history(
    
    CONSTRAINT fk_book1 
        ON DELETE SET NULL 
        ON UPDATE CASCADE, 
    CONSTRAINT fk_card1 
    FOREIGN KEY (card_num) references card(card_num) 
        ON DELETE SET NULL 
        ON UPDATE CASCADE );
Declarative cascading actions were not supported prior to SQL Server 2000 and they have been on the wish lists of many programmers and DBAs. At last, they are supported as of SQL Server 2000.

**7.5 Triggers**

*Triggers* are used to help maintain data integrity and enforce business rules (a trigger is a special kind of stored procedure that executes in response to an event inside SQL Server.) They complement the protection of data integrity that constraints, defaults and rules can provide. A trigger is associated with a particular table. When a specified event occurs, the trigger executes.

CREATE TRIGGER TriggerName  
On TableName  
FOR [INSERT], [UPDATE], [DELETE]  
AS  
--Trigger Code

The MS SQLs supported trigger mechanism is presented in *Table 7.2*:

<table>
<thead>
<tr>
<th></th>
<th>MS SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td># event per trigger</td>
<td>n</td>
</tr>
<tr>
<td># trigger per event</td>
<td>n</td>
</tr>
<tr>
<td>Trigger granularity</td>
<td>S</td>
</tr>
<tr>
<td>Activation time</td>
<td>A</td>
</tr>
<tr>
<td>Activation cond.</td>
<td>√</td>
</tr>
</tbody>
</table>

*Table 7.2 Triggers*

Triggers are broadly divided into two groups:

- **DDL triggers**: Data Definition Language triggers.
- **DML triggers**: Data Modification Language triggers.

Triggers are classified as follows:

- **INSTEAD OF triggers**: These execute instead of the statement to which they are related.
- **AFTER triggers**: These execute after the statement to which they are related.
A DML trigger is executed in response to an event associated with a Data Modification Language (DML) statement. A DML trigger is associated with one of the following statements:

- INSERT
- UPDATE
- DELETE

DML triggers can be used either to replace a DML statement or to execute after a DML statement. A trigger that replaces a DML statement is called an INSTEAD OF trigger. A trigger that executes after a DML statement is called an AFTER trigger.

**Example 1:** Once a new student is created this trigger automatically sets the students study end date according to the study program that they are enrolled in.

```sql
create trigger set_study_enddate
    on student
    for insert, update
    as
    declare @new char(10),
            @newdate datetime
    set @new= (select study_program from inserted)
    set @newdate= (select study_startdate from inserted)
    if( @new = 'bachelors')
        begin
            update student
            set study_enddate=(Select dateadd(year,3,@newdate))
        end
    else if (@new = 'masters')
        begin
            update student
            set study_enddate=(Select dateadd(year,5,@newdate))
        end;
```

### 7.5.1 FOR| AFTER Triggers

AFTER specifies that the trigger is fired only when all operations specified in the triggering SQL statement have executed successfully. All referential cascade actions and constraint checks also must succeed before this trigger fires. AFTER is the default when FOR is the only keyword specified, AFTER triggers cannot be defined on views [54].
Example 2: When a new card has been created for a student/teacher this trigger automatically sets the card's validity, active date and status.

```sql
ALTER trigger [dbo].[set_student_card_validity]
    on [dbo].[card]
    for insert, update
    as declare @new int,
               @new_id int
    set @new=(select mem_id from inserted)
    if ( select count(*) from student where student.mem_id=@new and student.still_student=1)>0
        begin
            update card
            set validity= (Select dateadd(year,1,createdate))
                          , activedate=(select dateadd(day,60,createdate)),
                          status='blocked'
            where card.mem_id=@new
        end
    else if (Select count(*) from teacher where teacher.mem_id=@new )>0
        begin
            update card
            set activedate=getdate(),
                status='activated'
            where card.mem_id=@new
        end
    else begin
        raiserror 13002 'student finished their studies'
        rollback
    end;
```

7.5.2 Inserted and deleted tables

DML trigger statements use two special tables: the deleted table and the inserted tables. SQL Server 2005 automatically creates and manages these tables. These temporary, memory-resident tables can be used to test the effects of certain data modifications and to set conditions for DML trigger actions. Data can not directly be modified in the tables or data definition language (DDL) operations can not be performed on the tables, such as CREATE INDEX. In DML triggers, the inserted and deleted tables are primarily used to perform the following:

- Extend referential integrity between tables.
• Insert or update data in base tables underlying a view.
• Test for errors and take action based on the error.
• Find the difference between the state of a table before and after a data modification and take actions based on that difference.

The deleted table stores copies of the affected rows during DELETE and UPDATE statements. During the execution of a DELETE or UPDATE statement, rows are deleted from the trigger table and transferred to the deleted table. The deleted table and the trigger table ordinarily have no rows in common. The inserted table stores copies of the affected rows during INSERT and UPDATE statements. During an insert or update transaction, new rows are added to both the inserted table and the trigger table. The rows in the inserted table are copies of the new rows in the trigger table.

Example 3: This trigger checks if a new student’s member id does not belong to any teacher.

create trigger stdnt_id_bfr_insrt
on student
for insert, update
as
declare @v_id int
set @v_id=(select mem_id from inserted)
if (select count(*) from teacher where teacher.mem_id=@v_id)>0
begin
    RAISERROR 13000 'id being used by teacher'
    rollback transaction
    return
end;

An update transaction is similar to a delete operation followed by an insert operation; the old rows are copied to the deleted table first and the new rows are then copied to the trigger and inserted table. When trigger conditions are set, the inserted and deleted tables are used appropriately for the action that fired the trigger. Although referencing the deleted table when testing an INSERT or the inserted table when testing a DELETE does not cause any errors, these trigger test tables do not contain any rows in these cases. Deleted and inserted are logical (conceptual) tables. They are structurally similar to the table on which the trigger is defined, that is the table on which the user action is
attempted and hold the old values or new values of the rows that may be changed by the user action [37].

7.5.3 INSTEAD OF Triggers

An INSTEAD OF trigger fires in the place of triggering action, it executes after the SQL Server creates the inserted and deleted tables, but before SQL Server takes any other actions. SQL Server executes INSTEAD OF triggers before constraints are applied. This enables performing pre-processing that supplements existing constraints. At most, one INSTEAD OF trigger per INSERT, UPDATE or DELETE statement can be defined on a table or view. INSTEAD OF triggers are not allowed on updatable views that use WITH CHECK OPTION. SQL Server raises an error when an INSTEAD OF trigger is added to an updatable view WITH CHECK OPTION specified. The user must remove that option by using ALTER VIEW before defining the INSTEAD OF trigger. \{ [DELETE ] [ , ] [ INSERT ] [ , ] [ UPDATE ] \} specifies the data modification statements that activate the DML trigger when it is tried against table or view. At least one option must be specified. Any combination of these options in any order is allowed in the trigger definition. For INSTEAD OF triggers, the DELETE option is not allowed on tables that have a referential relationship specifying a cascade action ON DELETE. Similarly, the UPDATE option is not allowed on tables that have a referential relationship specifying a cascade action ON UPDATE.

7.5.3 MS SQL Multiple Triggers

SQL Server allows multiple triggers to be created for each data modification event. For example, if CREATE TRIGGER FOR UPDATE is executed for a table that already has an UPDATE trigger, then an additional update trigger is created. In earlier versions, only one trigger for each data modification event (INSERT, UPDATE and DELETE) was allowed for each table. SQL Server is very good at handling sets of data. For example, a single UPDATE statement can be used to update many rows of data. There are times when loops through a series of rows perform processing for each row; in this case
Cursors are used. Cursors are the SLOWEST way to access data inside SQL Server. They should only be used when accessing one row at a time is truly needed [37].

7.5.4 Trigger Limitations

- CREATE TRIGGER must be the first statement in the batch and can only apply to one table. A trigger is created only in the current database; however, a trigger can reference objects outside the current database.
- If the trigger owner name is specified (to qualify the trigger) - qualify the table name in the same way.
- The same trigger action can be defined for more than one user action (for example, INSERT and UPDATE) in the same CREATE TRIGGER statement.
- INSTEAD OF DELETE/UPDATE triggers cannot be defined on a table that has a foreign key with a cascade on DELETE/UPDATE action defined.
- Any SET statement can be specified inside a trigger. The SET option chosen remains in effect during the execution of the trigger and then reverts to its former setting.

7.5.5 Downsides of Triggers

Many developers avoid triggers entirely. Probably the biggest disadvantage of triggers is that they get buried in the database and are difficult to debug and troubleshoot. This has been improved slightly in SQL Server 2005 Express because triggers appear as a node under the table that they are associated with [37]. Triggers also slow down database operations. Furthermore, they often lock data for relatively long periods of time, increasing the chance of concurrency problems. For these reasons, most developers utilize stored procedures functions, or even middle-tier components to replace the role of triggers in the applications that they build.
8. Overview of Firebird Relational Database System

8.1 Introduction to Firebird 2.0

Firebird is a relational database management system offering many ANSI SQL-2003 features. It runs on Linux, Windows and a variety of UNIX platforms. Started as a fork of Borland's open source release of InterBase, the Firebird codebase is maintained by the Firebird Project at SourceForge. In this document the most recent stable version Firebird 2.0 is used. This Firebird branch contains a large number of new features, including derived tables, support for Execute Block, increased table sizes, improved index code (no more 252-byte index length limit), expression indices, optimizer improvements, enhanced security features, support for on-line incremental backups as well as numerous other improvements and bug fixes [55].

8.2 SQL standard compliance

Firebird conforms to entry-level SQL-92 requirements. It has support for formal, cascading referential integrity constraints, updatable views and full, left and right outer joins. Client applications can link to the Firebird API, a messenger function library for client-server communication. Several extended SQL features are also implemented. Some of them (e.g. stored procedures and triggers, SQL roles and segmented blob support) anticipate SQL99 extensions [56].

8.3 Important Features

- Full support of Stored Procedures and Triggers [56].
- Full ACID compliant transactions.
- Referential Integrity.
- Multi Generational Architecture.
- Fully featured internal language for Stored Procedures and Triggers (PSQL).
- Support for External Functions (UDFs).
- Little or no need for specialized DBAs.
- Almost no configuration needed - just install and start using.
- Big community, therefore lots of places to get free and good support.
Native support for all major operating systems, including Windows, Linux, Solaris and MacOS.

- Incremental Backups.
- Full cursor implementation in PSQL.

### 8.4 Firebird SQL

Every database management system has its own idiosyncrasies in the ways it implements SQL. Firebird adheres to the SQL standard more rigorously than most other RDBMSes. Developers migrating from products that are less standards-compliant often wrongly suppose that Firebird is quirky, whereas many of its apparent quirks are not quirky at all [57].

### 8.5 Firebird integrity support

A constraint can be specified for each column (or columns) in a table, to guarantee the mechanism described above. Constraints can be domain or column-based and the specified conditions must be met when new data sets are inserted or an existing data sets is modified. They are used to verify data integrity. If a condition is not met an exception is then raised. Firebird internally generates a trigger for each check condition. Constraints can be defined as follows:

1. **Primary Key/Unique** - Specification of the unique option forces a unique entry in this column (these columns) for each data set (i.e. duplicate field entries are not allowed).
2. **Foreign Key** - The foreign key option determines that the column(s) is/are linked by a referential integrity relationship to the primary key of another table (i.e. the input data is only accepted if it already exists in the primary key column(s) in the referenced table).
3. **CHECK** - The check option enables each data set to be examined for validation of an expression specified in brackets. Check constraints in tables are identical to check constraints in domains. Only one constraint is permitted per column. If the column including a constraint is based on a domain also containing a constraint, both constraints are active. The specification of the keyword CONSTRAINT and the name is optional for all constraints.
If no name is specified Firebird then generates a name automatically. All constraint names are stored in a system table called DB$RELATION_CONSTRAINTS. It is only necessary to name constraints if they are to be deactivated at a later date using the ALTER TABLE DROP statement. Cascading referential integrity is also supported in firebird.

8.5.1 Domain support

Firebird supports domain as per standard SQL: 99.

8.5.2 Null support

Firebird 2 allows the use of NULL literals in every context where a normal value can also be entered. In a column or domain definition it can be specified that only non-NULL values may be entered by adding NOT NULL to the definition:

Create domain mem_type as varchar(30) not null

To know whether a variable, field or other expression is NULL, the following syntax can be used: `<expression> IS [NOT] NULL.” “… = NULL” must not be used to test for nullness. This syntax is illegal and gives the wrong result in Firebird 2 [58]. Setting a field or variable to NULL is done with the “=” operator, just like assigning values. The important thing to remember about the null values is: The comparison operator “=” should not and can not be used to test if something is NULL. The assignment operator “=” should and must be used to set something to NULL.

8.5.3 Primary keys

 NULLS are never allowed in primary keys. A column can only be (part of) a PK it has been defined as NOT NULL, either in the column definition or in a domain definition. Note that a “CHECK (XXX IS NOT NULL)” constraint will not do: a NOT NULL specifier is needed right after the data type.
8.5.4 Unique Keys

Unique keys and Unique Indices allow NULLs and what’s more: they even allow multiple NULLs. With a single-column key or index, it is possible to insert as many NULLs in that column, but each non-NULL value can only be inserted once. If the key or index is defined on multiple columns in Firebird:

- Multiple rows can be inserted where all the key columns are NULL;
- But as soon as one or more key columns are non-NULL, each combination of non-NULL values must be unique in the table.

8.5.5 Foreign key and referential action

Foreign keys impose no restrictions with respect to NULLs. Foreign key columns must always reference a column (or set of columns) that is a primary key or a unique key. A unique index on the referenced column(s) is not enough. Even if NULLs are absolutely forbidden in the target key (for instance if the target is a PK), the foreign key column may still contain NULLs, unless this is prevented by additional constraints.

Alter table student
add constraint UNQ_student UNIQUE (field_for_fk);

In Firebird, cascading referential integrity is also supported. When a foreign key relationship is specified, the user can define which action should be taken following changes to, or deletion of, its referenced primary key. ON UPDATE defines what happens when the primary key changes and ON DELETE specifies the action to be taken when the referenced primary key is deleted. In both cases the following options are available:

1. NO ACTION
2. NULL - The foreign key column should be set to its default value
3. CASCADE - The foreign key column is set to the new primary key value. The CASCADE option also deletes the foreign key row when the primary key is deleted.
8.5.6 Check constraint

A check is a database examination, which ensures data consistency in the tables among each other. It can be executed automatically and so ensures that data contents are kept consistent by testing them before they are stored in the database. The check constraint option enables each data set to be examined for validation of the expression in brackets following the check constraint. Check constraints in tables are identical to check constraints in domains. A check constraint can be specified for each column in a table, to guarantee the mechanism described above. It includes an expression that must be true, so that the data set following an insert or update can be written. The field contents must be included in the permissible values, which can be specified in a list. It is also possible to test the value for a minimum and maximum value. Furthermore, the value can be compared to values in other columns; in order to test dependencies. A check constraint can be created directly when creating a table. When creating a check constraint, the following criteria should be taken into consideration:

- A check constraint cannot reference a domain.
- A table column can only contain one check constraint.
- A check constraint defined by a domain, cannot be overridden by a local check constraint. However additional constraints can be specified.

Referential integrity declarations and primary key definitions are special check constraint compositions. Only one constraint is permitted per column. If the column is based on a domain containing a constraint, both check constraints are active. It is only necessary to name constraints, if they are to be deactivated at a later date using the ALTER TABLE DROP statement [59].

Create table Member (

Mem_city varchar(30) not null check ( mem_city in ‘PRAGUE1’ ),
...
)
8.6 triggers

Trigger types refer to the trigger status (ACTIVE or INACTIVE), the trigger position (BEFORE or AFTER) and the operation type (INSERT or UPDATE or DELETE).

- Triggers are self-contained routines (special Stored Procedures), which are connected to a table or view.
- A trigger is automatically executed ("fired") when a record is inserted, updated or deleted from a table/view.
- Triggers are never called directly but only through one of the commands INSERT, UPDATE or DELETE.
- Triggers are written in Stored Procedure and Trigger Language and can use Exceptions [60].

8.6.1 General trigger Syntax

- ACTIVE/INACTIVE: A trigger can be deactivated e.g. for development. Can also be used with ALTER TRIGGER.
- BEFORE/AFTER: Tells the system if the trigger must be fired before or after the Insert/Update/Delete operation
- POSITION: Trigger calling order. Triggers with low numbers are fired first. Triggers with the same position number are fired in an undefined order. The position defaults to 0.
- trigger_body: Same as body in a Stored Procedure.

8.6.2 Context Variables

- NEW.colname: New column value for INSERT or UPDATE operations.
- OLD.colname: Old column value for UPDATE or DELETE operations.

Example 1: Insert a Generator value for a newly inserted record:

```
CREATE TRIGGER Create_order_ID FOR orders
BEFORE INSERT
AS BEGIN
NEW.ID = GEN_ID (LogIdGenerator, 1); END;
```
9. Overview of the PostgreSQL Database Management System

9.1 Introduction to PostgreSQL

In this part one of the most successful open-source software products of recent times will be discussed, a relational database called PostgreSQL. PostgreSQL is an excellent implementation of a relational database, fully featured, open source, and free to use. PostgreSQL can be used from just about any major programming language including C, C++, Perl, Python, Java, Tcl and PHP. It very closely follows the industry standard for query languages, SQL92 and is currently implementing features to increase compliance with the latest version of this standard, SQL: 2003. Traditional relational database management systems (DBMSs) support a data model consisting of a collection of named relations, containing attributes of a specific type. In current commercial systems, possible types include floating point numbers, integers, character strings, money and dates. It is commonly recognized that this model is inadequate for future data processing applications. The relational model successfully replaced previous models in part because of its "Spartan simplicity". Postgres offers substantial additional power by incorporating the following four additional basic concepts in such a way that users can easily extend the system: Classes, Inheritance, Types and functions. These features put Postgres into the category of databases referred to as object-relational. Note that this is distinct from those referred to as object-oriented, which in general are not as well suited to supporting the traditional relational database languages. So, although Postgres has some object-oriented features, it is firmly in the relational database world. In fact, some commercial databases have recently incorporated features pioneered by Postgres. Throughout this study, the PostgreSQL version 8.2.5 will be used [38].

9.2 Standard compliance

Postgres understands a good subset of SQL92/99 plus some object-oriented features to these subsets and closely follows the industry standard for query languages, SQL92 and
is currently implementing features to increase compliance with the latest version of this standard, SQL: 2003.

### 9.3 Data types

PostgreSQL has a rich set of native data types available to users. Users may add new types to PostgreSQL using the `CREATE TYPE` command. Each data type has an external representation determined by its input and output functions. Many of the built-in types have obvious external formats. However, several types are either unique to PostgreSQL, such as geometric paths, or have several possibilities for formats, such as the date and time types [57].

### 9.4 Inheritance

PostgreSQL is a DBMS that supports many advanced features such as table inheritance. However, the table inheritance implementation poses several problems when it comes to using it in some real-world projects. The problems encountered when using PostgreSQL’s inheritance implementation are explained as well.

**Example 1:** The student table contains students who are also members. Naturally, the students table should inherit from cities.

```sql
CREATE TABLE member (  
    Mem_id INTEGER PRIMARY KEY,  
    Mem_firstname VARCHAR (15),  
    Mem_lastname VARCHAR (30),  
    phone CHAR(16) DEFAULT 'UNKNOWN',  
    . . .);

CREATE TABLE student (  
    Study_startdate DATE,  
    Study_program CHAR (10) NOT NULL CONSTRAINT chk_studyprogram CHECK study_startdate DATE DEFAULT CURRENT_DATE,  
    still_student BOOLEAN DEFAULT TRUE,  
    study_enddate DATE  
) INHERITS (member);
```
In this case, a row of student *inherits* all attributes (first name, last name, phone,...) from its parent, member. Students have extra attributes, study_startdate, study_enddate, study_program, major etc. In PostgreSQL a table can inherit from zero or more other tables and a query can reference either all rows of a table or all rows of a table plus all of its descendants. All check constraints and not-null constraints on a parent table are automatically inherited by its children. Other types of constraints (unique, primary key and foreign key constraints) are not inherited. A parent table cannot be dropped while any of its children remain. Neither can columns of child tables be dropped or altered if they are inherited from any parent tables. If a table and all of its descendants need to be dropped, one easy way is to drop the parent table with the `CASCADE` option. A serious limitation of the inheritance feature is that indexes (including unique constraints) and foreign key constraints only apply to single tables, not to their inheritance children. This is true on both the referencing and referenced sides of a foreign key constraint. Considerable care is needed in deciding whether inheritance is useful for our problem or not. Three issues are really critical as they prevent PostgreSQL's table inheritance to be used without having to create intermediary tables as workarounds: foreign key constraints and triggers not being propagated, which can lead to incoherence; primary keys and unique constraints not being propagated and finally, the impossibility to create foreign keys that reference both a table and its children. In order to work around these limitations, a set of PL/PgSQL functions are created that will automatically propagate foreign keys and generate functions and triggers that will enforce primary key and unique keys amongst all inherited tables as well as handle foreign keys to table hierarchies. In the PostgreSQL based application the use of inheritance in the tables shall be avoided.

### 9.5 Constraint support matrix

In this part the basics of using SQL to store and access data in PostgreSQL is presented. The declarative integrity support is presented in *Table 9.1*: 
9.5.1 Null/Default Values

A column can be assigned a default value. When a new row is created and no values are specified for some of the columns, the columns will be filled with their respective default values. A data manipulation command can also request explicitly that a column be set to its default value, without having to know what that value is. If no default value is declared explicitly, the default value is the null value. In a table definition default values are listed after the column data type. For example:

```
CREATE TABLE member (  
    Mem_id    INTEGER,  
    phone     CHAR(16) DEFAULT 'UNKNOWN',  
    ...);  
```

Table 9.1 Declarative Integrity Support
The default value may be an expression, which will be evaluated whenever the default value is inserted. A common example is that a timestamp column may have a default of `NOW()`, so that it gets set to the time of row insertion. Another common example is generating a serial number for each row. In PostgreSQL, this is typically done by:

```
CREATE TABLE orders (  
    Order_id integer DEFAULT nextval ('orders_order_id_seq'),
    ...);
```

A not-null constraint is always written as a column constraint. A not-null constraint is functionally equivalent to creating a check constraint `CHECK (column_name IS NOT NULL)`, but in PostgreSQL creating an explicit not-null constraint is more efficient. The drawback is that explicit names cannot be given to not-null constraints created that way. Of course, a column can have more than one constraint. Just by writing the constraints one after another:

```
CREATE TABLE member (  
    Mem_id integer NOT NULL,
    Mem_num VARCHAR(20) NOT NULL UNIQUE,
    mem_city VARCHAR(15) NOT NULL CONSTRAINT check_city CHECK (mem_city IN('PRAGUE1','PRAGUE2')),
    ...);
```

The order does not matter. It does not necessarily determine in which order the constraints are checked. The NOT NULL constraint has an inverse: the NULL constraint. This does not mean that the column must be null, which would surely be useless. Instead, this simply selects the default behavior that the column may be null. The NULL constraint is not defined in the SQL standard and should not be used in portable applications (it was only added to PostgreSQL to be compatible with some other database systems). For example, create the table with Null and then insert the NOT keyword where desired.

```
CREATE TABLE member (  
    Mem_id INTEGER NULL,
    Mem_num VARCHAR(20) NULL,
    Phone CHAR (16),
    ...);
```

PostgreSQL's Not Null support is listed in table 9.2.
9.5.2 Check Constraints

A check constraint is the most generic constraint type. It allows it to be specified that the value in a certain column must satisfy a Boolean (truth-value) expression. *Example 2.* For instance, to require positive salary for managers the following could be use:

```sql
CREATE TABLE manager (  
  Mgr_id   INTEGER,  
  Mgr_fullname  VARCHAR (30),  
  salary   INTEGER CHECK (salary > 0));
```

The constraint definition comes after the data type, just like default value definitions. Default values and constraints can be listed in any order. Column constraints can also be written as table constraints, while the reverse is not necessarily possible, since a column constraint is supposed to refer to only the column it is attached to.

9.5.3 Unique/primary Constraints

Unique constraints ensure that the data contained in a column or a group of columns is unique with respect to all the rows in the table.

```sql
CREATE TABLE member (  
  Mem_id   INTEGER,  
  Mem_num  VARCHAR (20) UNIQUE,  
  ...);
```

The above example is when written as a table constraint. In general, a unique constraint is violated when there are two or more rows in the table where the values of all of the columns included in the constraint are equal. However, null values are not considered equal in this comparison. That means even in the presence of a unique constraint it is
possible to store an unlimited number of rows that contain a null value in at least one of the constrained columns. This behavior conforms to the SQL standard but as seen before, other SQL databases may not follow this rule [46]. Care must be taken when developing applications that are intended to be portable. A primary key constraint is simply a combination of a unique constraint and a not-null constraint. Therefore, the following two table definitions accept the same data:

```sql
CREATE TABLE orders (    Order_id INTEGER UNIQUE NOT NULL,
    . . .);
CREATE TABLE orders (    Order_id INTEGER PRIMARY KEY,
    . . .);
```

A primary key indicates that a column or group of columns can be used as a unique identifier for rows in the table. This is a direct consequence of the definition of a primary key. Note that a unique constraint does not, by itself, provide a unique identifier because it does not exclude null values. A table can have at most one primary key (while it can have many unique and not-null constraints). Relational database theory dictates that every table must have a primary key. This rule is not enforced by PostgreSQL, although it is usually best to follow it.

### 9.5.4 Foreign Keys and Referential Action Support:

A foreign key can also constrain and reference a group of columns. As usual, it then needs to be written in table constraint form [38].

**Example 3:** The following is a contrived syntax example:

```sql
CREATE TABLE orders (    card_num INTEGER NOT NULL,
    order_id SERIAL PRIMARY KEY,
    book_id INTEGER NOT NULL,
    State VARCHAR (15) DEFAULT 'failed' NOT NULL
    CONSTRAINT check_state CHECK (STATE IN ('order_processed','done','failed')),
    orderdate DATE,
    Pickup DATE,
```
FOREIGN KEY (book_id) REFERENCES book
    ON UPDATE CASCADE
    ON DELETE CASCADE,
FOREIGN KEY (card_num) REFERENCES card
    ON UPDATE CASCADE
    ON DELETE RESTRICT
);

Of course, the number and type of the constrained columns need to match the number
and type of the referenced columns. Owners own name can be assigned for a foreign key
constraint in the usual way. A table can contain more than one foreign key constraint.
This is used to implement many-to-many relationships between tables. The referential
actions supported by PostgreSQL are explained below.

Restricting and cascading deletes are the two most common options. RESTRICT
prevents deletion of a referenced row. NO ACTION means that if any referencing rows
still exist when the constraint is checked, an error is raised; this is the default behavior if
nothing is specified. The essential difference between these two choices is that NO
ACTION allows the check to be deferred until later in the transaction, whereas
RESTRICT does not. CASCADE specifies that when a referenced row is deleted, row(s)
referencing it should be automatically deleted as well. There are two other options: SET
NULL and SET DEFAULT. These cause the referencing columns to be set to nulls or
default values, respectively, when the referenced row is deleted. Note that these do not
excuse the observation of any constraints. For example, if an action specifies SET
DEFAULT but the default value would not satisfy the foreign key, the operation will
fail. Analogous to ON DELETE there is also ON UPDATE, which is invoked when a
referenced column is changed (updated). The possible actions are the same. Finally, it
should be mentioned that a foreign key must reference columns that either are a primary
key or form a unique constraint. If the foreign key references a unique constraint, there
are some additional possibilities regarding how null values are matched [57].
9.6 Triggers

Table 9.3 lists the PostgreSQL trigger properties:

<table>
<thead>
<tr>
<th>Postgresql</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td># events per trigger</td>
<td>n</td>
</tr>
<tr>
<td># triggers per event</td>
<td>n</td>
</tr>
<tr>
<td>Activation time</td>
<td>B/A</td>
</tr>
<tr>
<td>Trigger granularity</td>
<td>R/S</td>
</tr>
<tr>
<td>Activation condition</td>
<td>√</td>
</tr>
</tbody>
</table>

Table 9.3 Triggers

Name: The name to give the new trigger, this must be distinct from the name of any.

Before/After: Determines whether the function is called before or after the event.

Event: One of INSERT, UPDATE or DELETE; this specifies the event that will fire the trigger. Multiple events can be specified using OR.

For each row/ statement: This specifies whether the trigger procedure should be fired once for every row affected by the trigger event, or just once per SQL statement. If neither is specified, FOR EACH STATEMENT is the default.

Function-name: A user-supplied function, that is declared as taking no arguments and returning type trigger, which is executed when the trigger fires.

Arguments: An optional comma-separated list of arguments to be provided to the function when the trigger is executed. The arguments are literal string constants. Simple names and numeric constants may be written here too, but they will all be converted to strings.

9.6.1 Compatibility with Standard SQL

The CREATE TRIGGER statement in PostgreSQL implements a subset of the SQL: 1999 standard. The following functionality is missing [38]:

- SQL: 1999 allows triggers to fire on updates to specific columns (e.g. AFTER UPDATE OF col1, col2).
• SQL: 1999 allows defining of aliases for the "old" and "new" rows or tables for use in the definition of the triggered action. Since PostgreSQL allows trigger procedures to be written in any number of user-defined languages, access to the data is handled in a language-specific way.

• PostgreSQL only allows the execution of a user-defined function for the triggered action. SQL: 1999 allows the execution of a number of other SQL commands, such as CREATE TABLE as triggered action. This limitation is not hard to work around, by creating a user-defined function that executes the desired commands.

• SQL: 1999 specifies that multiple triggers should be fired in time-of-creation order. PostgreSQL uses name order, which was judged more convenient to work with.

• The ability to specify multiple actions for a single trigger using OR are a PostgreSQL extension of the SQL standard.

The trigger function must be defined before the trigger itself can be created. The trigger function must be declared as a function taking no arguments and returning type trigger. Once a suitable trigger function has been created, the trigger is established with CREATE TRIGGER. The same trigger function can be used for multiple triggers.

Example 4: When a student is created based on their study program the student’s study_enddate is updated.

CREATE OR REPLACE FUNCTION set_study_enddate() RETURNS TRIGGER
AS$set_study_enddate$
BEGIN
    IF new.study_program='bachelors' THEN
        new.study_enddate=new.study_startdate+ interval '3 years';
    ELSEIF new.study_program='masters' THEN
        new.study_enddate=new.study_startdate+interval '5 years';
    END IF;
    RETURN NEW;
END; $set_study_enddate$ LANGUAGE plpgsql;

CREATE TRIGGER set_study_enddate
BEFORE INSERT OR UPDATE ON student FOR EACH ROW EXECUTE PROCEDURE set_study_enddate();
Trigger functions invoked by per-statement triggers should always return \texttt{NULL}. Trigger functions invoked by per-row triggers can return a table row to the calling executor, if they choose. A row-level trigger fired before an operation has the following choices:

- It can return \texttt{NULL} to skip the operation for the current row. This instructs the executor to not perform the row-level operation that invoked the trigger (the insertion or modification of a particular table row).

- For row-level \texttt{INSERT} and \texttt{UPDATE} triggers only, the returned row becomes the row that will be inserted or will replace the row being updated. This allows the trigger function to modify the row being inserted or updated.

A row-level before trigger that does not intend to cause either of these behaviors must be careful to return as its result the same row that was passed in (that is, the NEW row for \texttt{INSERT} and \texttt{UPDATE} triggers, the OLD row for \texttt{DELETE} triggers). The return value is ignored for row-level triggers fired after an operation and so they may as well return \texttt{NULL}.

\textbf{9.6.2 BEFORE/ AFTER Trigger behaviour}

Typically, row before triggers are used for checking or modifying the data that will be inserted or updated. Row after triggers is most sensibly used to propagate the updates to other tables or to make consistency checks against other tables. The reason for this division of labor is that an after trigger can be certain it is seeing the final value of the row, while a before trigger cannot; there might be other before triggers firing after it. If there is no specific reason to make a trigger before or after, then the before case is more efficient since the information about the operation does not have to be saved until the end of the statement. When a trigger is being defined, arguments can be specified for it. The purpose of including arguments in the trigger definition is to allow different triggers with similar requirements to call the same function. Each programming language that supports triggers has its own method for making the trigger input data available to the trigger function. This input data includes the type of trigger event (e.g., \texttt{INSERT} or \texttt{UPDATE}) as well as any arguments that were listed in \texttt{CREATE TRIGGER}. For a row-
level trigger, the input data also includes the NEW row for INSERT and UPDATE triggers and/or the OLD row for UPDATE and DELETE triggers. Statement-level triggers do not currently have any way to examine the individual row(s) modified by the statement [38].

### 9.7 Automatic Key-Generation

Generating a serial number for each row in PostgreSQL is typically done by something like [46]:

```
CREATE TABLE orders (  
    Order_id INTEGER DEFAULT nextval ('orders_order_id_seq'),  
    ...);  
```

Where the `nextval()` function supplies successive values from a sequence object. This arrangement is sufficiently common that there is special shorthand for it:

```
CREATE TABLE orders (  
    Order_id INTEGER SERIAL ...);  
```

### 9.9 Event scheduler

PostgreSQL, like any database software, requires that certain tasks be performed regularly to achieve optimum performance. For example, one of the maintenance tasks is creation of backup copies of the data on a regular schedule, running a function on regular intervals etc. PgAgent is a job-scheduling agent for PostgreSQL, capable of running multi-step batch/shell and SQL tasks on complex schedules. PgAgent runs 'jobs', each of which consists of steps and schedules [58].
10. Conclusion

In this document a detailed comparison of integrity constraint features in the SQL: 99 and major Relational Database Management Systems (DBMS) Oracle10g Server (Release 10.2), PostgreSQL (Version 8.2.5), Microsoft SQL Server 2005, MySQL (Version 5.1) and Firebird (Release 2.0) are discussed and implemented. The source code of each individual system can be found in the presented CD. General guidelines are provided for the implementation and designing of the integrity constraints using the functionality and mechanisms provided by these relational database management systems. As it has been pointed out earlier in this document, in today’s applications there is a strong need to ensure the correctness of the data managed by the database system, as this is vital to any application utilizing data for business, research and decision making purposes.

Current commercial database management systems basically support the entry level of SQL-92. Meaning that with regard to the specification of declarative constraints these systems support the specification of default values, not null constraints, primary and foreign keys, unique constraints and check constraints on the column level. The specification of referential actions as a simple type of reactive constraints enforcement is supported in some systems as well.

The biggest discrepancy between the current SQL: 99 and the relational database management systems presented in this document is on the specification and the usage of database triggers. The reason to this is that triggers have been provided by most systems prior to the introduction to SQL: 99. Most products have implemented triggers in spite of SQL-92’s failure to specify a standard for them. That is why as a result of such a thing almost all of these systems not only differ in the syntax of trigger specification but also they differ in functionality of the triggers in terms of relations. This raises another problem that different products implement triggers slightly differently with syntax variations, causing more problems to application developers. The specification of check constraints on the table database level and assertion is not supported in these relational database management systems.
In conclusion, although sufficient means for implementing constraint-enforcing mechanisms, either declaratively or in the form of database triggers are provided by all the relational database management systems, the role and importance of integrity constraints must not be underestimated in database designs. In today’s applications the need for correctness and usability of business critical data managed by the database systems is stronger than ever.

Let me conclude that the thesis specification was fully accomplished.
References

(1999); 408-409, Morgan Kaufmann publishers
[31] http://troels.arvin.dk/db/rdbms/
[34] Oracle corporation, Oracle® Database SQL Reference10g Release 2 (10.2)
[35] The PostgreSQL Global Development Group, PostgreSQL 8.2.5 Documentation Copyright © 1996-2006
[36] Microsoft corporation. SQL Server Development Centre Microsoft SQL Server 2005 Documentation
[38] PostgreSQL 8.2.5 Documentation http://www.postgresql.org/docs/8.2/interactive/index.html
[44] www.cs.purdue.edu/homes/akamra/cs541/active_DB.ppt
[50] Welling, L, Thomson, L; MySQL tutorial (2003); Sams Publishing
[56] http://www.ibexpert.info/documentation
[57] Matthew, N, Stones, R; Beginning databases with PostgreSQL From Novice to Professionals; Second Edition
Appendix A:

The Entity-relationship model presented in Figure 3.1:
Appendix B:

In the following the portion of SQL:99 grammars that are related to the integrity constraints are summarized:

```sql
<column-constraint-def> ::= [CONSTRAINT <constraint-name>]
                          <column-constraint>
                          [<characteristics>]

<column-constraint> ::= NOT NULL
                       | <uniqueness-spec>
                       | <references-spec>
                       | <check-constraint>

<uniqueness-spec> ::= UNIQUE | PRIMARY KEY

<references-spec> ::= REFERENCES <table-name> ( <attribute-list> )
                     [MATCH {SIMPLE | PARTIAL | FULL}]
                     [<referential-triggered-action>]

<referential-triggered-action> ::= ON UPDATE <referential-action>
                                  [ON DELETE <referential-action>]
                                  [ON DELETE <referential-action>]
                                  [ON UPDATE <referential-action>]

<referential-action> ::= NO ACTION
                        | RESTRICT
                        | CASCADE
                        | SET NULL
                        | SET DEFAULT

<characteristics> ::= <constraint-check-time> [[NOT] DEFERRABLE]
                    | [[NOT] DEFERRABLE] <constraint-check-time>

<constraint-check-time> ::= INITIALLY {IMMEDIATE | DEFERRED}

<set-constraint-mode> ::= SET CONSTRAINTS {ALL | <constraint-name-list>}
                         {IMMEDIATE | DEFERRED}

<table-constraint-def> ::= [CONSTRAINT <constraint-name>]
                         <table-constraint>
                         [<characteristics>]
```

117
Case Study: Implementation of Integrity Constraints in Actual Database System

<table-constraint> ::= <uniqueness-constraint>
| <referential-constraint>
| <check-constraint>

<uniqueness-constraint> ::= <uniqueness-spec> ( <attribute-list> )
| UNIQUE ( VALUE )

<referential-constraint> ::= FOREIGN KEY ( <attribute-list> )
| <references-spec>

<check-constraint> ::= CHECK ( <search-condition> )

<assertion-def> ::= CREATE ASSERTION <constraint-name>
| CHECK ( <search-condition> )
| <characteristics>

<reference-type> ::= REF ( <referenced-type> )
| SCOPE <table-name>
| REFERENCES ARE [NOT] CHECKED
| ON DELETE <referential-action>

<trigger-def> ::= CREATE TRIGGER <trigger-name>
| {BEFORE | AFTER}
| {INSERT | DELETE | UPDATE [OF <attribute-list>]} ON <table-name>
| REFERENCING <old-or-new-values-list>
| [FOR EACH {ROW | STATEMENT}]
| [WHEN ( <search-condition> )]
| <triggered-SQL-statement>

<old-or-new-values> ::= {OLD | NEW} ROW AS <correlation-name>
| {OLD | NEW} TABLE AS <table-alias>

<triggered-SQL-statement> ::= <SQL-procedure-statement>
| BEGIN ATOMIC
| <SQL-procedure-statement-list>
| END

<add-table-constraint-def> ::= ADD <table-constraint-def>

<drop-table-constraint-def> ::= DROP CONSTRAINT <constraint-name>
| {CASCADE | RESTRICT}
Appendix C

Content of CD

./CD/

- Master Thesis
- Appendices
  - Appendix A – ER Diagram
  - Appendix B – Syntax of SQL standard IC
- Implementation of Library database in Oracle
- Implementation of Library database in My SQL
- Implementation of Library database in PostgreSQL
- Implementation of Library database in MS SQL
- Implementation of Library database in Firebird
- Implementation of Library database in SQL: 99