Table of Contents

INTRODUCTION .................................................................................................................. 1

OVERVIEW ......................................................................................................................... 1
TRUSTED RUBIX™ FEATURES ....................................................................................... 2
    Advanced Security Features ...................................................................................... 2
    Traditional RDBMS Features .................................................................................... 4
TRUSTED RUBIX™ CLIENT/SERVER ARCHITECTURE .................................................. 5
STRUCTURED QUERY LANGUAGE (SQL) ......................................................................... 5
OPEN DATABASE CONNECTIVITY (ODBC) ....................................................................... 6
APPLICATION ARCHITECTURE ......................................................................................... 6

BASIC CONCEPTS ............................................................................................................. 7

CLIENTS ............................................................................................................................. 7
SERVERS ............................................................................................................................ 7
DATABASES ....................................................................................................................... 7
SCHEMAS .......................................................................................................................... 7
RELATIONS (TABLES) ...................................................................................................... 7
    Values ........................................................................................................................... 8
    Data Types .................................................................................................................. 8
    Illegal and Unknown Values ...................................................................................... 9
    Keys ............................................................................................................................. 10
    Choosing Key Columns ............................................................................................. 10
CREATING RELATIONS (TABLES) .................................................................................. 10
LOADING DATA ................................................................................................................ 11
IMPORTING AND EXPORTING FILES ............................................................................ 11

TUTORIAL DATABASE ...................................................................................................... 13

DESIGNING A CONCEPTUAL SCHEMA ......................................................................... 13
INVOKING TRUSTED RUBIX/SQL ................................................................................... 14
SETTING UP A DATABASE ................................................................................................ 15

DATA MANIPULATION .................................................................................................... 20

RETRIEVAL OPERATIONS ............................................................................................... 20
    SELECT Command and WHERE Clause ...................................................................... 20
    Multiple Search Conditions ...................................................................................... 21
    Creating an Index ....................................................................................................... 21
    Additional Comparison Operators ............................................................................ 22
    String functions ......................................................................................................... 25
    Niladic Functions (pseudo-literals) ............................................................................ 28
    Date and time functions - EXTRACT ......................................................................... 29
    Data Type Conversion - CAST .................................................................................. 29
    Arithmetic Expression ............................................................................................... 29
    ORDER BY Clause ...................................................................................................... 30
    DISTINCT keyword ................................................................................................... 31

MULTI-TABLE QUERIES .................................................................................................. 32
    Cross Join ..................................................................................................................... 32
    Join Specifications ...................................................................................................... 34
    Inner Join ..................................................................................................................... 34
    Outer Join ..................................................................................................................... 35
    Union Join .................................................................................................................... 40

TESTING FOR TRUE, FALSE, NULL .................................................................................. 40

ALIASES (TABLE AND COLUMN REFERENCES) .......................................................... 41

AGGREGATE FUNCTIONS ............................................................................................... 42
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GROUP BY AND HAVING CLAUSES</td>
<td>43</td>
</tr>
<tr>
<td>SUBQUERIES</td>
<td>44</td>
</tr>
<tr>
<td>SET OPERATIONS</td>
<td>45</td>
</tr>
<tr>
<td>UNION Operator</td>
<td>45</td>
</tr>
<tr>
<td>INTERSECT Operator</td>
<td>46</td>
</tr>
<tr>
<td>EXCEPT Operator</td>
<td>47</td>
</tr>
<tr>
<td>DATA MODIFICATION</td>
<td>48</td>
</tr>
<tr>
<td>UPDATE OPERATIONS</td>
<td>48</td>
</tr>
<tr>
<td>INSERT Command</td>
<td>48</td>
</tr>
<tr>
<td>UPDATE Command</td>
<td>50</td>
</tr>
<tr>
<td>DELETE Command</td>
<td>51</td>
</tr>
<tr>
<td>DATA DEFINITION</td>
<td>53</td>
</tr>
<tr>
<td>CREATE TABLE Command</td>
<td>53</td>
</tr>
<tr>
<td>DATA TYPES</td>
<td>53</td>
</tr>
<tr>
<td>Exact Numeric</td>
<td>53</td>
</tr>
<tr>
<td>Approximate Numeric</td>
<td>53</td>
</tr>
<tr>
<td>Character String</td>
<td>54</td>
</tr>
<tr>
<td>Character Large Object (CLOB)</td>
<td>54</td>
</tr>
<tr>
<td>Bit String</td>
<td>54</td>
</tr>
<tr>
<td>Binary Large Object (BLOB)</td>
<td>54</td>
</tr>
<tr>
<td>Date, Time and Datetime</td>
<td>54</td>
</tr>
<tr>
<td>Security Label</td>
<td>55</td>
</tr>
<tr>
<td>INTEGRITY CONSTRAINTS</td>
<td>56</td>
</tr>
<tr>
<td>ALTER TABLE Command</td>
<td>56</td>
</tr>
<tr>
<td>DROP TABLE COMMAND</td>
<td>58</td>
</tr>
<tr>
<td>VIEWS</td>
<td>59</td>
</tr>
<tr>
<td>OVERVIEW</td>
<td>59</td>
</tr>
<tr>
<td>VIEWS</td>
<td>59</td>
</tr>
<tr>
<td>VIEWS VS. RELATIONS (TABLES)</td>
<td>59</td>
</tr>
<tr>
<td>WHY VIEWS ARE USED</td>
<td>59</td>
</tr>
<tr>
<td>CREATE VIEW COMMAND</td>
<td>60</td>
</tr>
<tr>
<td>DROP VIEW COMMAND</td>
<td>61</td>
</tr>
<tr>
<td>VIEW UPDATABILITY</td>
<td>61</td>
</tr>
<tr>
<td>DATA SECURITY</td>
<td>73</td>
</tr>
<tr>
<td>DISCRETIONARY ACCESS CONTROL: GRANT AND REVOKE</td>
<td>73</td>
</tr>
<tr>
<td>MANDATORY ACCESS CONTROL</td>
<td>77</td>
</tr>
<tr>
<td>POLYINSTANTIATION</td>
<td>77</td>
</tr>
<tr>
<td>INTEGRITY CONSTRAINTS</td>
<td>79</td>
</tr>
<tr>
<td>INTEGRITY CONSTRAINTS</td>
<td>79</td>
</tr>
<tr>
<td>primary key and unique constraints</td>
<td>79</td>
</tr>
<tr>
<td>not null constraint</td>
<td>80</td>
</tr>
<tr>
<td>default Values</td>
<td>81</td>
</tr>
<tr>
<td>check constraint</td>
<td>81</td>
</tr>
<tr>
<td>FOREIGN KEY</td>
<td>82</td>
</tr>
<tr>
<td>Trusted RUBIX Referential Actions</td>
<td>83</td>
</tr>
<tr>
<td>CASCADE or RESTRICT Option</td>
<td>85</td>
</tr>
<tr>
<td>TRANSACTION MANAGEMENT/ CONCURRENCY CONTROL</td>
<td>86</td>
</tr>
</tbody>
</table>

© 2014 Infosystems Technology, Inc. 6.0 REVISION 8
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRANSACTION MANAGEMENT</td>
<td>86</td>
</tr>
<tr>
<td>Transactions and Timestamps</td>
<td>86</td>
</tr>
<tr>
<td>Access Modes</td>
<td>86</td>
</tr>
<tr>
<td>CONCURRENT TRANSACTIONS</td>
<td>87</td>
</tr>
<tr>
<td>Consistency Level</td>
<td>87</td>
</tr>
<tr>
<td>Concurrency Related Commands</td>
<td>88</td>
</tr>
<tr>
<td>Use of the Concurrency Commands</td>
<td>89</td>
</tr>
<tr>
<td>SAVEPOINTS</td>
<td>89</td>
</tr>
<tr>
<td>MULTILEVEL SECURE CONCURRENCY</td>
<td>91</td>
</tr>
<tr>
<td>SQL ODBC EXAMPLES</td>
<td>92</td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>92</td>
</tr>
<tr>
<td>COMPILIE AND LINK</td>
<td>92</td>
</tr>
<tr>
<td>CONNECT/DISCONNECT</td>
<td>93</td>
</tr>
<tr>
<td>SELECT STATEMENTS</td>
<td>95</td>
</tr>
<tr>
<td>POSITIONED UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>USING DYNAMIC ARGUMENTS</td>
<td>99</td>
</tr>
<tr>
<td>DIAGNOSTICS</td>
<td>101</td>
</tr>
</tbody>
</table>
INTRODUCTION

OVERVIEW

This tutorial is an introduction for the novice user of Trusted Rubix, a relational database management system (RDBMS). TR is a multilevel secure client-server RDBMS. A multilevel secure RDBMS can store and process information at multiple security levels and serve multiple users, some of whom may not be cleared for all the information in the machine.

The target audience for this tutorial includes users who have never used a RDBMS, and also includes users who have never used the Structured Query Language (SQL) to communicate with an RDBMS. In addition to complying with the Open Group SQL standards, TR provides extensions to the SQL language to provide sophisticated security controls and protections for user data. This tutorial deliberately does not include a security tutorial. After finishing this tutorial, the reading of the Security Features User’s Guide (SFUG) is required for users of TR.

A RDBMS stores and manipulates data in the form of relations. A relation is a two-dimensional table of related data. Each table can contain different kinds of data. For example, suppose you are managing multiple employees working on several projects in an organization. You might want to create a database that will enable you to produce the following information:

→ Which project has the highest budget?
→ What is the time to complete for each project?
→ Which projects are starting this month?
→ What is the sum of salaries of the employees in each department?
→ What is the average salary of the employees in each department?
→ In the last two years which projects have exceeded budget?
→ How many employees work on more than one project?
→ What is the salary of overhead employees?
→ Who has the maximum billing rate?
→ What is the ratio of salary to billing rate for each employee?
→ Which employees have no projects on hand?

TR functions enable you to manipulate the data contained in tables. Using RUBIX/SQL you can manipulate the data in a single table, or perform elaborate operations involving several tables in the same or different databases. A database can contain many different kinds of data, as long as the data contained in each table is in some way related to the data in the other tables in that database.

Traditionally, information systems have not allowed data to be separated into different sensitivities within a single database. Organizations have been forced to separate data physically on different machines or to provide higher security clearances than necessary for users. Many applications, particularly in the intelligence community, require integration or “fusion” of secret data with top secret/special intelligence data. Such applications require the highest level of security available in UNIX/LINUX environments. In fact, TR is the only RDBMS which can provide this level of assurance without giving up any of the traditional functions associated with the products of major RDBMS vendors.
As government and corporate organizations move to open systems, they require assurance that sensitive data is highly protected against unauthorized access, disclosure, or modification. **Trusted Rubix** is the most advanced, secure relational database management system (RDBMS) for the UNIX/LINUX environment. TR meets the EAL 4 level of assurance as specified in the Common Criteria for Information Technology Security Evaluation. **TR** is configurable at different levels of security.

Commercial and government customers will benefit from this flexibility and be able to tailor the product to their own level of security requirements. **TR** adheres to all critical industry standards in order to deliver unmatched flexibility in deployment, portability of applications and interoperability with other standards-based Open Systems. **TR** is portable across a range of UNIX/LINUX operating systems.

The RXISQL program provides an interactive SQL interface to submit ad hoc queries to the **TR** SQL Engine. The Open Database Connectivity (ODBC) module is an application programming interface (API) for database access. It implements Microsoft's Open Database Connectivity (ODBC) specification. Only “C” Language constructs are provided.

The section titled Basic Concepts of this tutorial provides a description of the basic concepts of **TR**. The section titled TUTORIAL DATABASE details the creation and loading of data into several tables and the relationships between them. In the section titled DATA MANIPULATION of this tutorial, the data retrieval statement SELECT is presented. In the section titled DATA MODIFICATION, data modification statements such as INSERT, UPDATE and DELETE are presented. In the section titled DATA DEFINITION, the data definition commands such as CREATE TABLE, CREATE VIEW, ALTER TABLE, DROP TABLE, and DROP VIEW are presented. The section titled VIEWS presents **TR** view updating capabilities. The section titled DATA SECURITY presents the details on data security, including how to control access to the data for different users. Integrity constraints are detailed in the section titled INTEGRITY CONSTRAINTS. The basic issues of concurrency control and **TR** multiversion time stamping are presented in the section titled TRANSACTION MANAGEMENT/ CONCURRENCY CONTROL. The section titled SQL ODBC EXAMPLES describes the Open Database Connectivity (ODBC) of **TR** which is an alternative invocation technique to dynamic SQL. It is ideally suited for a client/server environment, in which the target database is not known when the application program is built.

**TRUSTED RUBIX™ FEATURES**

**Trusted Rubix™** has many advanced features, including the following:

**Advanced Security Features**

→ An internal design that focuses on the modularity and layering principles which are critical in high assurance systems. **TR** has been validated at the EAL 4 Conformance Level on Trusted Solaris 8. For more information on the Common Criteria evaluation of **TR** see [http://www.commoncriteriaportal.org/files/epfiles/st_vid1015-vr.pdf](http://www.commoncriteriaportal.org/files/epfiles/st_vid1015-vr.pdf).

→ The **TR** Mandatory Access Control (MAC) policy is implemented as a minimized reference monitor within the database kernel. No query modification or SQL engine "hooks" are used.

→ The **TR** Multi-Level Security (MLS) policy restricts access to data objects based on the sensitivity of the information contained in the objects and the "clearance" of users to access such information.
→ Full integration of the trusted host operating system's OS-MAC policy with the TR MAC policy. This integration provides a unified and coherent security behavior across all database and operating system objects, simplifying security administration. Most importantly, it prevents security violations when information moves between the database (e.g., table) and the operating system (e.g., file).

→ TR MAC automatically eliminates "overt" and "covert channels by enforcing MAC on the data dictionary (which contains database objects) and still allows their use in real, internal operations.

→ Supported trusted operating systems include Trusted Solaris 8, Solaris 10 TX, Fedora SELinux, and Red Hat Enterprise Linux SELinux.

→ TR supported security policies include Multi-Level Security (MLS), Type Enforcement (TE -only in SELinux), Role Based Access Control (RBAC), Attribute Based Access Control (ABAC), and Discretionary Access Control (DAC).

→ Central to the MLS policy is the MLS label assigned to a TR subject (e.g., a database session open on behalf of a user) and object (e.g., a row in a table). TR labels its RDBMS objects with the same MLS labels used by the underlying trusted operating system. TR provides this capability to reclassify the default object label through the SQL UPDATE command.

→ Role Based Access Control (RBAC) allows highly granular administrative authorizations to be assigned to a named role. That role may then be assigned to any number of users, giving that user all of the authorizations assigned to that role. TR fully integrates its RBAC mechanism with that of the trusted host operating system.

→ Attribute Based Access Control (ABAC) security policies are enforced using the Security Policy Manager (SPM). It allows highly customized, complex, and hierarchical security policies to be created using a OASIS XACML 2.0 based XML language. The SPM policies and policy sets may be configured to further refine the underlying OS-MAC security policy or to allow highly controlled data releases across OS-MAC security domains.

→ Discretionary Access Control (DAC) specifies who can do what to the data - who can read, who can insert, who can change, etc. The controls are discretionary in the sense that a user with a certain privilege is capable of passing that privilege to other users.

→ TR's unique multi-version timestamping concurrency control technique enables the system to securely manage all changes taking place within the database, even with multiple applications running. This technique removes covert channels between transactions of different security domains as they access common database objects, thereby ensuring secure transaction processing.

→ Polyinstantiation occurs when duplicate records with different label's are inserted into a common table. If no special action is taken, data may be covertly transmitted to unauthorized users. Polyinstantiation may also be used to provide a "cover story" - where the actual data may be highly classified, but where there must be some other value visible to lower level users.

→ There are many environments where the data must be controlled during its entire life. Many commercial applications are mistakenly concerned with the security of data only until it is placed into the control of the end user. TR prevents or seriously hampers a TOP SECRET user from passing TOP SECRET data to an UNCLASSIFIED user.
What happens when data moves from the labeled security arbitrated domain of a RDBMS into another non-arbitrated domain of that RDBMS? The data ceases to be labeled and no access checks may be performed. In TR no subset of data in the database is unlabeled and all operations on that data are arbitrated by the MAC policy.

**Traditional RDBMS Features**

- A client-server architecture which allows untrusted clients to access either a single or multiple trusted servers running TR.
- An SQL interface which conforms to the ANSI SQL 92 standard and includes functions of the SQL 3 standard.
- Users can write to standard Application Programming Interfaces (API's) such as Open Database Connectivity (ODBC).
- A complete set of bit string, binary large object (BLOB), character string, character large object (CLOB), numeric, date/time, interval, and security data types gives TR total control over input/output formatting and sophisticated data operations.
- Sophisticated techniques for cost-based query optimization.
- A complete audit trail for all database operations that enables both the legitimate (but accidental) errors by users and unauthorized requests to be tracked.
- Tools for database back-up, database recovery, table import, table export, and audit management.
- Full Atomicity, Consistency, Isolation, and Durability (ACID) compliant transactional support.
- A savepoint mechanism which allows transactions to be partially rolled back (on user request). Thus, a user can undue all updates performed since the specified savepoint, while at the same time preserve updates performed prior to that point.
- Supports the ability to declare temporary tables which are used only to pass intermediate results from one portion of an application to another. Such tables are private to the application that uses them; there is no sharing of data with other applications.
- All of the features of TR are available 24 hours a day. The system need not be shut down to fix a table if the system crashes. TR enables the user to back-up and recover data, add a new table, or make other changes to the database while users continue to access the database.
TRUSTED RUBIX™ CLIENT/SERVER ARCHITECTURE

Figure 1: TRUSTED RUBIX Client Server Architecture

Figure 1 shows the client server architecture for TR. Each client is initiated by a TR user. The client process runs with the user’s credentials. When the client process connects to a specific database it creates a TR server process.

The two different user interfaces, RXISQL and ODBC, give the user flexibility in invoking the TR server. The RXISQL program provides an interactive SQL interface to submit ad hoc queries to the TR SQL Engine. The Open Database Connectivity (ODBC) module is an application programming interface (API) for database access. It is an implementation of Microsoft’s Open Database Connectivity (ODBC) specification.

Structured Query Language (SQL)

SQL is a widely accepted (by RDBMS vendors and users) non-procedural, English-like query language that is ideally suited for all types of database operations. The set of commands included in SQL are used for a variety of operations including:

→ Insert, delete and update functions
→ Create, modify, replace and drop functions
→ Integrity and consistency features
→ Access control features

The SQL syntax and semantics are used for specifying and modifying the structure and the integrity constraints of data and for declaring and invoking operations on the data in a local or remote RDBMS.

SQL also allows vendors to create extensions to the standard that enable their products to do tasks that other SQL-compliant RDBMS may not do. A prime area for such extension activity is security or trust.
Open Database Connectivity (ODBC)

The SQL Microsoft Open Database Connectivity (ODBC) is a C programming language application programming interface (API) for database access. ODBC is an alternative invocation technique to dynamic SQL that provides essentially equivalent operations. ODBC is a set of functions that application programs call directly using normal call facilities. ODBC is ideally suited for a client/server environment, in which the target database is not known when the application program is developed. The Microsoft Corporation published the ODBC specification which is vendor neutral, platform neutral, and database neutral. Thus, it is possible to use a single API for data definition, manipulation, and access throughout the enterprise. Each application uses the single API to interact with one or several data sources through DBMS-specific drivers. Writing an application to the ODBC interface ensures that it will be compatible with any ODBC compliant RDBMS, including TR.

Application Architecture

TR supports the typical 3 tier architecture associated with web database applications. The client tier is usually a browser which issues HTML requests and processes responses. The middle tier consists of the application layer - web server, web scripting language (PHP or Java), and the scripting language engine. If the request requires output from the database before responding, the web server invokes the scripting engine to perform the tasks. The scripts that are executed by the engine allow for serving dynamic content and have built in libraries to access the database tier. The bottom layer consists of Trusted RUBIX which supports simultaneous access, provides security and data integrity, and supports applications.
Basic Concepts

CLIENTS

An instance of a client consists of Trusted Rubix™ client software RXISQL and/or ODBC, residing on a host machine. The client always resides on a local host. Clients that are connected to a server may submit operations to that server. The server may reside on a local host or remote host.

SERVERS

An instance of a server consists of TR server software (rxserver) operating on a database residing on a host machine. TR clients connect to a server instance by specifying a server name. Server names are specified as DBNAME@HOSTNAME:PORT where DBNAME is the name of the database, HOSTNAME is the name of the machine where the database physically resides, and PORT is the port number on which the server is listening. If the @HOSTNAME part of the server name is omitted then the host machine is assumed to be a local host.

DATABASES

A database is a set of related relations (tables). Any number of databases may be supported on one computer; a database may contain any number of tables; and a user may own any number of databases. The organization of a TR database is illustrated in Figure 2.

SCHEMAS

A schema defines that portion of the database that is owned by a specific user. Thus, the complete database definition will usually consist of multiple schemas. Each schema contains tables which are related to each other conceptually. It also contains other objects related to query processing such as views, indexes, etc.

RELATIONS (TABLES)

Data is physically stored in TR databases in special files called relations. A relation is a table with a fixed number of columns and a variable number of rows. The creator of a table specifies each column's attributes and position within the table. A column's attributes include its name, data type, and whether or not it contains null values. Columns can also be added to a table after it is created. Table and column names must begin with an alphabetic character and must not exceed 50 characters in length. The only character which cannot be used in table and column names is the period. In TR, the creator of a table can specify additional constraints on a table or on a column. For example, you can define a default value to be given to a column when a new row is inserted and no value for that column is specified.

Each row comprises a sequence of elements, the individual pieces of data that correspond to the columns of the table. Rows can be freely inserted into or deleted from a table without altering the table's underlying structure. It is possible to have a table that contains no rows.
VALUES

A value is the smallest unit of data that can be manipulated in SQL. Tables in the database consist of values. An SQL statement can contain expressions, literals, and set functions that represent values. To represent a non-null value in an SQL statement, an application uses a literal. A character-string literal represents a character string and consists of a sequence of characters delimited at each end by the single quote character, i.e., ’Global Surveillance’. A numeric literal represents a number and consists of a character string whose characters are selected from the digits 0 through 9, the plus sign, the minus sign, the decimal point and the character E, e.g., +364.05, 0.007, 4E3 (4*10^3).

DATA TYPES

A data type is a set of representable values. A data type restricts the contents or representation of a value. Data types can be classified as either generic or named. The generic data types are character string, exact numeric or approximate numeric. A named data type is a data type, denoted by a keyword, that is based on one of the generic data types, and has additional attributes such as length and precision. TR SQL supports the named data types listed in the following table:

<table>
<thead>
<tr>
<th>Named Data Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT(n)</td>
<td>Binary string of length n bits.</td>
</tr>
<tr>
<td>BIT VARYING(n)</td>
<td>Variable length binary string with a maximum length of n bits.</td>
</tr>
<tr>
<td>BLOB</td>
<td>Variable length of bytes with a maximum length of 2,147,483,647 bytes.</td>
</tr>
<tr>
<td>CHARACTER(n)</td>
<td>Character string of length n characters.</td>
</tr>
<tr>
<td>CHARACTER VARYING(n)</td>
<td>Variable length character string with a maximum length of n characters.</td>
</tr>
</tbody>
</table>
### Named Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOB</td>
<td>Variable length character string with a maximum length of 2,147,483,647 characters.</td>
</tr>
<tr>
<td>DATE</td>
<td>Date string in YYYY-MM-DD format.</td>
</tr>
<tr>
<td>DECIMAL(p) 1 &lt;=p &lt;=50</td>
<td>Exact numeric with a precision of p digits.</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>Approximate numeric, signed, mantissa precision machine dependent.</td>
</tr>
<tr>
<td>FLOAT(p)</td>
<td>Approximate numeric, signed, mantissa precision p, maximum value of p is machine dependent.</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Exact numeric, signed, precision 10 digits value from -2,147,483,648 through 2,147,483,647.</td>
</tr>
<tr>
<td>LABEL</td>
<td>Security label string</td>
</tr>
<tr>
<td>NUMERIC(7,2) 4&lt;=p</td>
<td>Approximate numeric with precision value equivalent to float</td>
</tr>
<tr>
<td>NUMERIC(p) 1&lt;=p &lt;=50</td>
<td>Exact numeric with precision p digits.</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>Exact numeric, signed, precision 10 digits value from -2,147,483,648 through 2,147,483,647.</td>
</tr>
<tr>
<td>TIME</td>
<td>Time string as hh:mm:ss.</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>Timestamp</td>
</tr>
</tbody>
</table>

An exact numeric value has a precision and scale. The precision determines the number of significant decimal digits. The scale determines the number of decimal digits to the right of the decimal point. An approximate numeric value consists of a mantissa and an exponent. An approximate numeric value has a precision which determines the number of significant bits in the mantissa.

**TRUSTED RUBIX™** SQL allows CHAR as a synonym for CHARACTER, VARCHAR as a synonym for CHARACTER VARYING, DEC as a synonym for DECIMAL; and INT as a synonym for INTEGER.

The characteristics and limitations of the different data types are described further in the section titled DATA DEFINITION.

**Illegal and Unknown Values**

A column with a given type can only contain values from the domain of that type. When you insert a value into a column with a different domain, the value is considered Illegal. For example, a text value inserted into an integer field will be read as an Illegal value. An Illegal value is represented by an asterisk (*) when the table is displayed. If a row does not contain a value for a certain column, the value is marked as Unknown. When you print a table, TR indicates each
Unknown value with a question mark (?).

**KEYS**

Every table must have a **key**. The rows in a table are ordered according to the values found in the key. The key can consist of a single column, a group of contiguous columns (**composite key**), or it can contain all the columns in a table.

No two rows in a table may have the same key value. Any column that contains a unique value for each row may be designated the key column. The key can also consist of a group of columns which, when considered as a unit, contains a unique value for every row in the table. When a table is keyed on all its columns, the only requirement for distinct key values is that no two rows of the table may be identical. Polymorphism, described in the Security Features User’s Guide, as implemented by TR modifies the uniqueness requirement of the key column.

**CHOOSING KEY COLUMNS**

When you create a table, you must select the key columns. TR will not allow you to enter two rows with the same key value, so be careful to select a key that will allow you to enter all your data.

For example, suppose you are using Trusted Rubix to create a large phone directory. You wish to alphabetize the directory by last name, but there are several people with the same last name. You therefore should not use `last_name` as your key column singularly. Instead, a key that has the `last_name` column and other columns should be specified.

**CREATING RELATIONS (TABLES)**

To create a relation (table) follow these simple steps:

- Define the table using the `CREATE TABLE` command.
- Load data into the table using the `TR rximport` command which loads data from a text file or from standard input into a table.

During the `rximport` process, TR checks the integrity of each column value before inserting it into the table. If the value is not a valid representation for a column type (e.g., a text value in an integer column), the value is flagged as Illegal. Illegal values are represented by “*”. If there is no input for a defined column, a “?” flags the value as Unknown.

The `rximport` command can be used with one or more flags to create an error file, `<table>.e`. Rows containing unknown and/or illegal values, depending on which flag you specify, are placed in the error file during the `rximport` process. Error files can be created only if the current directory is publicly writeable.

The `INSERT` command can also be used to load or insert multiple records. This command doesn’t insert any records if one of the records violates the integrity constraints. The following syntax can be used for multiple record insertion:

```
INSERT INTO <relation> VALUES ( <row> <row> ... )
```
LOADING DATA

Data files must be properly formatted before they can be loaded into a table. Use a text editor to arrange the fields of the data file so that they correspond to the columns of the table. Each field should be in its string equivalent representation. For instance, binary data should be in hex format.

The data for each row must appear on a single line in the data file, and the fields within each row must be separated by blanks or tabs. Any field containing embedded blanks must be surrounded by double quotes or single quotes. `rximport` reads the data file into the table, row by row, interpreting any blank spaces that are not enclosed in quotes as column breaks.

For more details on bulk loading, see Chapter 6 of the Trusted Facility Manual and the `rximport` command in the Administrative Commands Reference Guide.

IMPORTING AND EXPORTING FILES

`TR` provides facilities for both reading (importing) and writing (exporting) files for exchange. File exchange is made possible by the use of a free format ASCII file structure.

In the table `projects`, each row consists of six elements with the following characteristics:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type specifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>pno</td>
<td>VARCHAR (5)</td>
</tr>
<tr>
<td>pname</td>
<td>VARCHAR (25)</td>
</tr>
<tr>
<td>dno</td>
<td>VARCHAR (3)</td>
</tr>
<tr>
<td>budget</td>
<td>NUMERIC(7,2)</td>
</tr>
<tr>
<td>startdate</td>
<td>DATE</td>
</tr>
<tr>
<td>finishdate</td>
<td>DATE</td>
</tr>
</tbody>
</table>

The rows used in this example are:

- **TMK** Tomahawk Navigation EE 500000.00 2006-11-01 2007-11-01
- **FCS** Flight Control Simulation SW 100000.00 2007-02-01 2007-12-15
- **MGS** Missile Guiding Systems EE 100000.00 2006-06-01 2007-06-01

For examples of exporting data for use by other programs, these rows of `projects` will be used as the starting point for creating an external file. For examples of importing data, an external file will create these `TR` compatible rows.

The Free Format structure provides a simple method for importing and exporting data. No headers, trailers or data type descriptors are required. The rules for file structure are minimal. A program can easily be written in any high level language to interpret or create files in this format.

In a Free Format ASCII file, data for each row (record) appears on a single line in the data file, and the elements within each row are separated by blanks or tabs. Additionally, for imported Free Format ASCII files, elements containing embedded blanks must be surrounded by double quotes or single quotes. It is not necessary for the elements of different rows to line up in column form. Each row (that is, each line) is terminated by a carriage return, or by a carriage return and line feed.
The example import data file corresponding to the first three rows of projects look like this in the Free Format ASCII structure:

- **TMK** ‘Tomahawk Navigation’ EE 500000.00 2006-11-01 2007-11-01
- **FCS** ‘Flight Control Simulation’ SW 100000.00 2007-02-01 2007-12-15
- **MGS** ‘Missile Guiding Systems’ EE 100000.00 2006-06-01 2007-06-01

This example shows a variety of line structures, but an actual file would have a structure that is consistent from line to line. All the structures shown in this example would be acceptable for importing data into **Trusted Rubix™**.

Files exported by TR in Free Format will have a uniform structure from line to line, like this:

- ‘FCS’ ‘Flight Control Simulation’ ‘SW’ ‘100000.00’ ‘2007-02-01’ ‘2007-12-15’
- ‘MGS’ ‘Missile Guiding Systems’ ‘EE’ ‘100000.00’ ‘2006-06-01’ ‘2007-06-01’
TUTORIAL DATABASE

DESIGNING A CONCEPTUAL SCHEMA

This section describes the schema used in this tutorial. The concepts of schema design are not discussed in this section. Please refer to a basic text on the subject, such as “An Introduction to Database Systems” by C.J. Date.

Let us assume an organization has several departments. Each department can have several projects and can bill their clients for the time that employees work on each project. In this situation, all the data is related and hence it can be contained in one schema.

The scenario above can be translated into a relational data model with several tables having relationships between them. The relationships are shown in Figure 3. The 1:N designation represents a one-to-many relationship, N:1 a many-to-one relationship, and M:N a many-to-many relationship.

Figure 3: project_manager Table Relationship
Each table contains several attributes. Each attribute is a unit of data which describes a specific characteristic and is referred to as a column with a name. There can be several occurrences of the set of attributes and each one is called a row. The attributes of the tables used in this tutorial are shown in **Figure 4**.

**Employees**

<table>
<thead>
<tr>
<th>ssn</th>
<th>lastname</th>
<th>firstname</th>
<th>salary</th>
<th>title</th>
<th>dno</th>
</tr>
</thead>
</table>

**Departments**

<table>
<thead>
<tr>
<th>dno</th>
<th>dname</th>
</tr>
</thead>
</table>

**Billing_Rate**

<table>
<thead>
<tr>
<th>title</th>
<th>rate</th>
</tr>
</thead>
</table>

**Projects**

<table>
<thead>
<tr>
<th>pno</th>
<th>pname</th>
<th>dno</th>
<th>budget</th>
<th>startdate</th>
<th>finishdate</th>
</tr>
</thead>
</table>

**Works**

<table>
<thead>
<tr>
<th>pno</th>
<th>essn</th>
<th>from_date</th>
<th>to_date</th>
<th>w_hours</th>
</tr>
</thead>
</table>

**Figure 4: project_manager Schema**

**INVOCKING TRUSTED RUBIX/SQL**

**Trusted Rubix** client software should be installed on your system and the **TR** server software must be installed on the server machine. To use the client software, the full path to the client executable must be specified during execution or the location of the executable must be added to the execution path environment. The location of the executables was specified during the installation of the software. For the default installation locations, please refer to the **TR** Installation Instructions sent with your CD.
**SETTING UP A DATABASE**

A database can be created by using the `CREATE DATABASE` command.

All SQL statements must be executed on the server within the context of a database. This presents a problem when it comes to database creation because no database exists from which to execute the `CREATE DATABASE` command. For this reason, there is a special “meta” database called `master` which is the context for all database `CREATE` and `DROP` statements. You can think of `master` as a database of databases which is modified by `CREATE DATABASE` and `DROP DATABASE`. To create a database you must first enter `rxsql` using the `master` database and then issue the `CREATE DATABASE` command. For example:

```bash
$ rxsql [-d SERVERNAME]
```

where `SERVERNAME = DBNAME[@HOSTNAME:[PORT]]`

For more details on how to use the interactive SQL shell, refer to the `rxsql` command in the Administrative Commands Reference Guide.

Schemata are database objects which can be created by using the `CREATE SCHEMA` statement. For this tutorial, we will use the `default_schema`, so we don’t have to create a separate schema. Before you can perform data retrieval, you must have some data in the database. Without getting into the details of how a table can be defined and how data can be inserted into the table, let us build a database using the conceptual schema discussed above. This is the basic data which the entire tutorial uses to present various capabilities of TR/SQL.
### Billing_Rate

<table>
<thead>
<tr>
<th>billing_rate.title</th>
<th>billing_rate.rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communications Engineer</td>
<td>35.00</td>
</tr>
<tr>
<td>Physicist</td>
<td>40.00</td>
</tr>
<tr>
<td>Scientist</td>
<td>65.00</td>
</tr>
<tr>
<td>Signal Engineer</td>
<td>35.00</td>
</tr>
<tr>
<td>Software Engineer</td>
<td>35.00</td>
</tr>
</tbody>
</table>

### Departments

<table>
<thead>
<tr>
<th>department. s.dno</th>
<th>departments.dname</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADM</td>
<td>Administration</td>
</tr>
<tr>
<td>EE</td>
<td>Electrical</td>
</tr>
<tr>
<td>SW</td>
<td>Software</td>
</tr>
</tbody>
</table>

### Employees

<table>
<thead>
<tr>
<th>Employees.ssn</th>
<th>employees.lastname</th>
<th>employees.firstname</th>
<th>employees.salary</th>
<th>employees.title</th>
<th>employees.dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>100254789</td>
<td>Smith</td>
<td>Bruce</td>
<td>28000.00</td>
<td>Accountant</td>
<td>ADM</td>
</tr>
<tr>
<td>111223333</td>
<td>Gomes</td>
<td>Mary</td>
<td>42000.00</td>
<td>Software Engineer</td>
<td>SW</td>
</tr>
<tr>
<td>151007777</td>
<td>Furtado</td>
<td>Judy</td>
<td>35000.00</td>
<td>Software Engineer</td>
<td>SW</td>
</tr>
<tr>
<td>151235679</td>
<td>Smith</td>
<td>Sandra</td>
<td>19000.00</td>
<td>Secretary</td>
<td>ADM</td>
</tr>
<tr>
<td>224255600</td>
<td>Brown</td>
<td>Jane</td>
<td>36000.00</td>
<td>Signal Engineer</td>
<td>EE</td>
</tr>
<tr>
<td>224552500</td>
<td>Booker</td>
<td>Tracy</td>
<td>44000.00</td>
<td>Physicist</td>
<td>EE</td>
</tr>
<tr>
<td>228992500</td>
<td>Walsh</td>
<td>James</td>
<td>55000.00</td>
<td>Software Engineer</td>
<td>SW</td>
</tr>
<tr>
<td>322330000</td>
<td>Wickham</td>
<td>Scott</td>
<td>45000.00</td>
<td>Software Engineer</td>
<td>EE</td>
</tr>
<tr>
<td>700340352</td>
<td>Xiu</td>
<td>Belinda</td>
<td>25000.00</td>
<td>Office Mgr.</td>
<td>ADM</td>
</tr>
<tr>
<td>897005240</td>
<td>Golden</td>
<td>Barbara</td>
<td>60000.00</td>
<td>Scientist</td>
<td>EE</td>
</tr>
</tbody>
</table>
Projects

<table>
<thead>
<tr>
<th>projects.pno</th>
<th>projects.pname</th>
<th>projects.dno</th>
<th>projects.budget</th>
<th>projects.startdate</th>
<th>projects.finishdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCS</td>
<td>Flight Control Simulation</td>
<td>SW</td>
<td>100000.00</td>
<td>2007-02-01</td>
<td>2007-12-15</td>
</tr>
<tr>
<td>IC</td>
<td>Inventory Control</td>
<td>SW</td>
<td>600000.00</td>
<td>2006-12-01</td>
<td>2007-09-15</td>
</tr>
<tr>
<td>MGS</td>
<td>Missile Guiding Systems</td>
<td>EE</td>
<td>100000.00</td>
<td>2006-06-01</td>
<td>2007-06-01</td>
</tr>
<tr>
<td>PCS</td>
<td>Patriot Control System</td>
<td>EE</td>
<td>600000.00</td>
<td>2007-01-15</td>
<td>2007-06-20</td>
</tr>
<tr>
<td>TMK</td>
<td>Tomahawk Navigation</td>
<td>EE</td>
<td>500000.00</td>
<td>2006-11-01</td>
<td>2007-11-01</td>
</tr>
</tbody>
</table>

Works

<table>
<thead>
<tr>
<th>works.pno</th>
<th>works.ssn</th>
<th>works.from_date</th>
<th>works.to_date</th>
<th>works.w_hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCS</td>
<td>228992500</td>
<td>2007-02-01</td>
<td>2007-02-15</td>
<td>80</td>
</tr>
<tr>
<td>IC</td>
<td>111223333</td>
<td>2007-01-01</td>
<td>2007-01-15</td>
<td>40</td>
</tr>
<tr>
<td>IC</td>
<td>151007777</td>
<td>2007-01-01</td>
<td>2007-01-15</td>
<td>40</td>
</tr>
<tr>
<td>IC</td>
<td>228992500</td>
<td>2007-01-01</td>
<td>2007-01-15</td>
<td>40</td>
</tr>
<tr>
<td>MGS</td>
<td>322333000</td>
<td>2007-01-16</td>
<td>2007-01-31</td>
<td>45</td>
</tr>
<tr>
<td>PCS</td>
<td>322333000</td>
<td>2007-02-01</td>
<td>2007-02-15</td>
<td>70</td>
</tr>
<tr>
<td>TMK</td>
<td>224552500</td>
<td>2007-01-01</td>
<td>2007-01-15</td>
<td>40</td>
</tr>
<tr>
<td>TMK</td>
<td>322333000</td>
<td>2007-01-01</td>
<td>2007-01-15</td>
<td>40</td>
</tr>
<tr>
<td>TMK</td>
<td>897005240</td>
<td>2007-02-01</td>
<td>2007-02-15</td>
<td>80</td>
</tr>
<tr>
<td>TMK</td>
<td>897005240</td>
<td>2007-01-16</td>
<td>2007-01-31</td>
<td>80</td>
</tr>
<tr>
<td>TMK</td>
<td>897005240</td>
<td>2007-01-01</td>
<td>2007-01-15</td>
<td>80</td>
</tr>
</tbody>
</table>

Typically each table has a set of columns which uniquely identifies each instance of the table, i.e., a row. These columns can be specified as a PRIMARY KEY. There can be any number of columns in the PRIMARY KEY, but each table can have only one PRIMARY KEY. In other words, if there is more than one column in the PRIMARY KEY, it is the combination of these columns which uniquely identifies the record. Note that a key is strictly a logical concept.

You can build this database using the following list of commands:

```sql
CREATE TABLE billing_rate
(
    PRIMARY KEY (title),
    title VARCHAR(25),
    rate NUMERIC(4,2)
);
COMMIT;

CREATE TABLE departments
(
    PRIMARY KEY (dno),
    dno VARCHAR(3),
    dname VARCHAR(15)
);
COMMIT;
```

© 2014 Infosystems Technology, Inc.
CREATE TABLE employees
(
    PRIMARY KEY (ssn),
    ssn INTEGER,
    lastname VARCHAR(10),
    firstname VARCHAR(10),
    salary NUMERIC(12,2),
    title VARCHAR(25),
    dno VARCHAR(3)
);
COMMIT;

CREATE TABLE projects
(
    PRIMARY KEY (pno),
    pno VARCHAR(3),
    pname VARCHAR(25),
    dno VARCHAR(3),
    budget NUMERIC(15,2),
    startdate DATE,
    finishdate DATE
);
COMMIT;

CREATE TABLE works
(
    PRIMARY KEY (pno, ssn, from_date),
    pno VARCHAR(3),
    ssn INTEGER,
    from_date DATE,
    to_date DATE,
    w_hours INTEGER
);
COMMIT;

INSERT INTO billing_rate VALUES 'Scientist',65.00;
INSERT INTO billing_rate VALUES 'Software Engineer', 35.00;
INSERT INTO billing_rate VALUES 'Signal Engineer', 35.00;
INSERT INTO billing_rate VALUES 'Physicist',40.00;
INSERT INTO billing_rate VALUES 'Communications Engineer', 35.00;
COMMIT;

INSERT INTO departments VALUES 'ADM', 'Administration';
INSERT INTO departments VALUES 'EE', 'Electrical';
INSERT INTO departments VALUES 'SW', 'Software';
COMMIT;

INSERT INTO employees VALUES 100254789,'Smith', 'Bruce', 28000,'Accountant', 'ADM';
INSERT INTO employees VALUES 151235679,'Smith', 'Sandra', 19000,'Secretary', 'ADM';
INSERT INTO employees VALUES 700340352, 'Xiu', 'Belinda', 25000, 'Office Manager', 'ADM';
INSERT INTO employees VALUES 897005240,'Golden', 'Barbara', 60000, 'Scientist', 'EE';
INSERT INTO employees VALUES 322333000, 'Wickham', 'Scott', 45000, 'Software Engineer', 'EE';
INSERT INTO employees VALUES 224255600, 'Brown', 'Jane', 36000, 'Signal
Engineer', 'EE';
INSERT INTO employees VALUES 224552500, 'Booker', 'Tracy', 44000, 'Physicist', 'EE';
INSERT INTO employees VALUES 151007777, 'Furtado', 'Judy', 35000, 'Software Engineer', 'SW';
INSERT INTO employees VALUES 228992500, 'Walsh', 'James', 55000, 'Software Engineer', 'SW';
INSERT INTO employees VALUES 111223333, 'Gomes', 'Mary', 42000, 'Software Engineer', 'SW';
COMMIT;
INSERT INTO projects VALUES 'PCS', 'Patriot Control System', 'EE', 600000, 2007-01-15, 2007-06-20;
INSERT INTO projects VALUES 'IC', 'Inventory Control', 'SW', 600000, 2006-12-01, 2007-09-15;
INSERT INTO projects VALUES 'FCS', 'Flight Control Simulation', 'SW', 100000, 2007-02-01, 2007-12-15;
COMMIT;
INSERT INTO works VALUES 'TMK', 897005240, 2007-01-01, 2007-01-15, 80;
INSERT INTO works VALUES 'TMK', 897005240, 2007-01-16, 2007-01-31, 80;
INSERT INTO works VALUES 'TMK', 897005240, 2007-02-01, 2007-02-15, 80;
INSERT INTO works VALUES 'TMK', 322333000, 2007-01-01, 2007-01-15, 40;
INSERT INTO works VALUES 'TMK', 322333000, 2007-02-01, 2007-02-15, 70;
INSERT INTO works VALUES 'MGS', 322333000, 2007-01-16, 2007-01-31, 40;
INSERT INTO works VALUES 'IC', 151007777, 2007-01-01, 2007-01-15, 45;
INSERT INTO works VALUES 'IC', 111223333, 2007-01-01, 2007-01-15, 40;
INSERT INTO works VALUES 'IC', 228992500, 2007-01-01, 2007-01-15, 40;
INSERT INTO works VALUES 'FCS', 228992500, 2007-02-01, 2007-02-15, 80;
COMMIT;

Now the database contains the data required to run the queries presented in this tutorial. You can retrieve the data you have just inserted using various forms of the SELECT statement which are presented in the next section.
DATA MANIPULATION

RETRIEVAL OPERATIONS

SELECT Command AND WHERE Clause

The most fundamental RUBIX/SQL command is SELECT. The SELECT statement, also referred to as a 'query', specifies that data is to be extracted from a table, view or combination of the two. The result of each query is displayed as a table, where each column of the table is labeled with the name of the column or the supplied alias. The order in which you list columns in the SELECT command controls the left-to-right sequencing of the columns in the display. When you specify SELECT *, all columns from the indicated table or view are projected; the left-to-right sequencing of the columns in the resulting display is determined by the order in which the columns were defined in the CREATE TABLE command.

```
SELECT  lastname, firstname, title, salary
FROM     employees;
```

<table>
<thead>
<tr>
<th>lastname</th>
<th>firstname</th>
<th>title</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>Bruce</td>
<td>Accountant</td>
<td>28000.00</td>
</tr>
<tr>
<td>Smith</td>
<td>Sandra</td>
<td>Secretary</td>
<td>19000.00</td>
</tr>
<tr>
<td>Xiu</td>
<td>Belinda</td>
<td>Office Manager</td>
<td>25000.00</td>
</tr>
<tr>
<td>Golden</td>
<td>Barbara</td>
<td>Scientist</td>
<td>60000.00</td>
</tr>
<tr>
<td>Wickham</td>
<td>Scott</td>
<td>Software Engineer</td>
<td>45000.00</td>
</tr>
<tr>
<td>Brown</td>
<td>Jane</td>
<td>Signal Engineer</td>
<td>36000.00</td>
</tr>
<tr>
<td>Booker</td>
<td>Tracy</td>
<td>Physicist</td>
<td>44000.00</td>
</tr>
<tr>
<td>Furtado</td>
<td>Judy</td>
<td>Software Engineer</td>
<td>35000.00</td>
</tr>
<tr>
<td>Walsh</td>
<td>James</td>
<td>Software Engineer</td>
<td>55000.00</td>
</tr>
<tr>
<td>Gomes</td>
<td>Mary</td>
<td>Software Engineer</td>
<td>42000.00</td>
</tr>
</tbody>
</table>

A query may be constrained by one or more Boolean criteria (i.e., value comparisons which evaluate to TRUE or FALSE). The operators which can be used in a value comparison are equal to (=), not equal to (<> or !=), less than (<), less than or equal to (<=), greater than (>), and greater than or equal to (>=). A WHERE clause causes RUBIX/SQL to search the data in the table and retrieve the indicated columns only from those rows that satisfy the search criteria. In the example below, only those rows are returned for which the salary of an employee is at least $30000.00:

```
SELECT  lastname, firstname, salary, title
FROM     employees
WHERE    salary >= 30000.00;
```

<table>
<thead>
<tr>
<th>lastname</th>
<th>firstname</th>
<th>Salary</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Golden</td>
<td>Barbara</td>
<td>60000.00</td>
<td>Scientist</td>
</tr>
<tr>
<td>Wickham</td>
<td>Scott</td>
<td>45000.00</td>
<td>Software Engineer</td>
</tr>
<tr>
<td>Brown</td>
<td>Jane</td>
<td>36000.00</td>
<td>Signal Engineer</td>
</tr>
<tr>
<td>Booker</td>
<td>Tracy</td>
<td>44000.00</td>
<td>Physicist</td>
</tr>
<tr>
<td>Furtado</td>
<td>Judy</td>
<td>35000.00</td>
<td>Software Engineer</td>
</tr>
<tr>
<td>Walsh</td>
<td>James</td>
<td>55000.00</td>
<td>Software Engineer</td>
</tr>
<tr>
<td>Gomes</td>
<td>Mary</td>
<td>42000.00</td>
<td>Software Engineer</td>
</tr>
</tbody>
</table>

© 2014 Infosystems Technology, Inc.
MULTIPLE SEARCH CONDITIONS

Boolean conditions are the chief mechanism for controlling which rows are selected from a table or view. The syntax of RUBIX/SQL provides a rich set of operators by which multiple Boolean search criteria may be combined. Basic value comparisons can be combined using the operators AND, OR, and NOT. In fact, by combining conditions using these operators, arbitrarily intricate selection criteria may be specified. For example, the query:

```
SELECT lastname, firstname, salary, title, dno
FROM employees
WHERE title = 'Software Engineer'
AND NOT (dno = 'EE')
OR salary > 50000.00;
```

produces the list of employees who are a 'Software Engineer' but are not in the Electrical Engineering department or have a salary higher than $50000.00.

<table>
<thead>
<tr>
<th>last name</th>
<th>first name</th>
<th>salary</th>
<th>title</th>
<th>dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>Golden</td>
<td>Barbara</td>
<td>60000.00</td>
<td>Scientist</td>
<td>EE</td>
</tr>
<tr>
<td>Furtado</td>
<td>Judy</td>
<td>35000.00</td>
<td>Software Engineer</td>
<td>SW</td>
</tr>
<tr>
<td>Walsh</td>
<td>James</td>
<td>55000.00</td>
<td>Software Engineer</td>
<td>SW</td>
</tr>
<tr>
<td>Gomes</td>
<td>Mary</td>
<td>42000.00</td>
<td>Software Engineer</td>
<td>SW</td>
</tr>
</tbody>
</table>

CREATING AN INDEX

It is important to note that the existence of an index or the absence of one does not change the result of any query. It acts solely to increase the speed at which certain queries are executed. An index is a quick way of accessing an arbitrary combination of columns in a base table and it is named for administrative purposes. Indexes are structures actually stored in the database. However, indexes are logically and physically independent of the data in the associated table. You can create or drop an index at any time without affecting the base tables or other indexes.

TRusted Rubix™ automatically maintains and uses indexes once they are created. TR automatically reflects changes to data, such as adding new rows, updating rows, or deleting rows, in all relevant indexes without any action by users.

For example, if you wanted to find all employees named 'Furtado', you would issue the following query:

```
SELECT * FROM employees WHERE lastname = 'Furtado';
```

Since lastname is not a keyed column and is not part of an index, TR would have no choice but to perform a sequential search for the record. The timing for such a search depends entirely on the number of records in the employees table.

The output for the above query is:

<table>
<thead>
<tr>
<th>ssn</th>
<th>last name</th>
<th>first name</th>
<th>salary</th>
<th>title</th>
<th>dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>151007777</td>
<td>Furtado</td>
<td>Judy</td>
<td>35000.00</td>
<td>Software Engineer</td>
<td>SW</td>
</tr>
</tbody>
</table>
Alternatively, creating an index on `lastname` would result in accesses directly to the record in question. To create an index for the `employees` table on the `lastname` column, the following statement may be used:

```
CREATE INDEX index_lastname
ON employees(lastname);
```

If one decides to remove an index from the database, it may be removed easily using the name with which it was created.

```
DROP INDEX index_lastname;
```

A composite index (also called a concatenated index) is an index that you can create on multiple columns in a table. Columns in a composite index can appear in any order and need not be adjacent in the table.

Composite indexes can speed retrieval of data for `SELECT` statements in which the `WHERE` clause references all or the leading portion of the columns in the composite index. Therefore, you should give some thought to the order of the columns used in the definition. Normally, the most commonly accessed or most selective columns come first.

Indexes can be unique or non-unique. Unique indexes guarantee that no two rows of a table have duplicate values in the columns that define the index. Non-unique indexes do not impose this restriction on the column values.

**Trusted Rubix™** recommends that you do not explicitly define unique indexes on tables; uniqueness is strictly a logical concept and should not be associated with the definition of a table. Alternatively, define `UNIQUE` integrity constraints on the desired columns. **TR** enforces `UNIQUE` integrity constraints by automatically defining a unique index on the unique key.

**Additional Comparison Operators**

**BETWEEN Operator**

Certain operators exist to simplify search criteria that would be quite cumbersome if expressed solely in terms of `AND`, `OR`, and `NOT`. For example, the query:

```
SELECT lastname, firstname, salary
FROM employees
WHERE salary >= 30000.00
    AND salary <= 40000.00;
```

can be reformulated more intuitively using the `BETWEEN` operator:

```
SELECT lastname, firstname, salary
FROM employees
WHERE salary BETWEEN 30000.00 AND 40000.00;
```

to produce a list of employees who earn between $30000.00 and $40000.00:

<table>
<thead>
<tr>
<th>lastname</th>
<th>firstname</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Furtado</td>
<td>Judy</td>
<td>35000.00</td>
</tr>
<tr>
<td>Brown</td>
<td>Jane</td>
<td>36000.00</td>
</tr>
</tbody>
</table>
**IN Operator**

The cumbersome query:

```sql
SELECT lastname, firstname, title
FROM employees
WHERE title = 'Software Engineer'
OR title = 'Physicist'
OR title = 'Signal Engineer';
```

can be recast using the IN operator:

```sql
SELECT lastname, firstname, title
FROM employees
WHERE title IN ('Software Engineer','Physicist','Signal Enginee');
```

to produce a list of employees who are either a software engineer, physicist, or signal engineer:

<table>
<thead>
<tr>
<th>lastname</th>
<th>firstname</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wickham</td>
<td>Scott</td>
<td>Software Engineer</td>
</tr>
<tr>
<td>Brown</td>
<td>Jane</td>
<td>Signal Engineer</td>
</tr>
<tr>
<td>Booker</td>
<td>Tracy</td>
<td>Physicist</td>
</tr>
<tr>
<td>Furtado</td>
<td>Judy</td>
<td>Software Engineer</td>
</tr>
<tr>
<td>Walsh</td>
<td>James</td>
<td>Software Engineer</td>
</tr>
<tr>
<td>Gomes</td>
<td>Mary</td>
<td>Software Engineer</td>
</tr>
</tbody>
</table>

**LIKE Operator**

String data can be compared for relationships more intricate than simple equality and inequality. Specifically, RUBIX/SQL provides mechanisms for testing whether certain patterns occur within strings. For example, employees with titles that end with the string 'Engineer' can be selected:

```sql
SELECT lastname, firstname, title
FROM employees
WHERE title LIKE '%Engineer';
```

<table>
<thead>
<tr>
<th>lastname</th>
<th>firstname</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wickham</td>
<td>Scott</td>
<td>Software Engineer</td>
</tr>
<tr>
<td>Brown</td>
<td>Jane</td>
<td>Signal Engineer</td>
</tr>
<tr>
<td>Booker</td>
<td>Tracy</td>
<td>Physicist</td>
</tr>
<tr>
<td>Furtado</td>
<td>Judy</td>
<td>Software Engineer</td>
</tr>
<tr>
<td>Walsh</td>
<td>James</td>
<td>Software Engineer</td>
</tr>
<tr>
<td>Gomes</td>
<td>Mary</td>
<td>Software Engineer</td>
</tr>
</tbody>
</table>

In a pattern match, the '_' stands for any single character and '%' for one or more characters. Thus, for a more sophisticated example, all employees whose last names contain 'mi' after the first character can be selected using this query:

```sql
SELECT lastname, firstname
FROM employees
WHERE lastname LIKE '_mi%';
```
But if the result to be retrieved contains an underscore ('_'), its pattern matching capabilities can be ignored by using the ESCAPE keyword. For example, you can retrieve all the project names that contain underscore ('_') as:

```
SELECT pno,pname,budget
FROM projects
WHERE pname LIKE '%#_%' ESCAPE '#';
```

<table>
<thead>
<tr>
<th>pno</th>
<th>pname</th>
<th>budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMK</td>
<td>Tomahawk Navigation</td>
<td>500000.00</td>
</tr>
</tbody>
</table>

**EXISTS Operator**

EXISTS can be used with subqueries similar to the IN predicate. If the subquery can retrieve at least one row then the condition is true; i.e., the title from the employees table equals the title from the billing_rate table.

```
SELECT * FROM employees
WHERE EXISTS (SELECT * FROM billing_rate
WHERE employees.title = billing_rate.title);
```

<table>
<thead>
<tr>
<th>employees</th>
<th>employees. ssn</th>
<th>employees. lastname</th>
<th>employees. firstname</th>
<th>employees. salary</th>
<th>employees. title</th>
<th>employees. dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>897005240</td>
<td>Golden</td>
<td>Barbara</td>
<td>60000.00</td>
<td>Scientist</td>
<td>EE</td>
<td></td>
</tr>
<tr>
<td>322333000</td>
<td>Wickham</td>
<td>Scott</td>
<td>45000.00</td>
<td>Software Engineer</td>
<td>EE</td>
<td></td>
</tr>
<tr>
<td>224255600</td>
<td>Brown</td>
<td>Jane</td>
<td>36000.00</td>
<td>Signal Engineer</td>
<td>EE</td>
<td></td>
</tr>
<tr>
<td>224552500</td>
<td>Booker</td>
<td>Tracy</td>
<td>44000.00</td>
<td>Physicist</td>
<td>EE</td>
<td></td>
</tr>
<tr>
<td>151007777</td>
<td>Furtado</td>
<td>Judy</td>
<td>35000.00</td>
<td>Software Engineer</td>
<td>SW</td>
<td></td>
</tr>
<tr>
<td>228992500</td>
<td>Walsh</td>
<td>James</td>
<td>55000.00</td>
<td>Software Engineer</td>
<td>SW</td>
<td></td>
</tr>
<tr>
<td>111223333</td>
<td>Gomes</td>
<td>Mary</td>
<td>42000.00</td>
<td>Software Engineer</td>
<td>SW</td>
<td></td>
</tr>
</tbody>
</table>

**ALL, SOME and ANY predicates**

The ALL predicate can be used in a nested SELECT statement, if all the rows resulting from a subquery have to be compared. You can retrieve all the employees whose salary is greater than every employee in the SW department using the following query:

```
SELECT lastname, firstname, salary
```
FROM employees
WHERE salary > ALL (SELECT salary FROM employees WHERE dno='SW');

<table>
<thead>
<tr>
<th>lastname</th>
<th>firstname</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Golden</td>
<td>Barbara</td>
<td>60000.00</td>
</tr>
</tbody>
</table>

But if you wish to list the employees whose salary is higher than one or more employees of the SW department, then you must use the SOME keyword instead of ALL. Note that ANY is synonymous to SOME. If ALL is replaced by SOME in the previous query:

```sql
SELECT lastname, firstname, salary FROM employees WHERE salary > SOME(SELECT salary FROM employees WHERE dno='SW');
```

it produces the result:

<table>
<thead>
<tr>
<th>lastname</th>
<th>firstname</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Golden</td>
<td>Wickham</td>
<td>45000.00</td>
</tr>
<tr>
<td>Brown</td>
<td>Jane</td>
<td>36000.00</td>
</tr>
<tr>
<td>Booker</td>
<td>Tracy</td>
<td>44000.00</td>
</tr>
<tr>
<td>Walsh</td>
<td>James</td>
<td>55000.00</td>
</tr>
<tr>
<td>Gomes</td>
<td>Mary</td>
<td>42000.00</td>
</tr>
</tbody>
</table>

**STRING FUNCTIONS**

There are several string functions that can be used for more flexibility in retrieving data.

**String length**

Comparisons can be performed using the string length functions:

→ CHAR_LENGTH, → OCTET_LENGTH, and → BIT_LENGTH.

For instance, you can construct the following query,

```sql
SELECT lastname, firstname, title FROM employees WHERE CHAR_LENGTH (title) = 15;
```

to produce a list of employees whose title is 15 characters long:

<table>
<thead>
<tr>
<th>lastname</th>
<th>firstname</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>Jane</td>
<td>Signal Engineer</td>
</tr>
</tbody>
</table>
The functions OCTET_LENGTH and BIT_LENGTH are useful when data is bit_packed. BIT_LENGTH is the number of bits that are used to store the field value. OCTET_LENGTH is BIT_LENGTH(string)/8. If no bit packing is provided, CHAR_LENGTH and OCTET_LENGTH are equivalent.

**Substring**

Comparisons can be accomplished with a portion of the string, if you specify the string value, its starting position and the total length for comparison.

```
SELECT lastname, firstname, title
FROM employees
WHERE SUBSTRING (title FROM 1 FOR 6) = 'Signal';
```

The example above examines the title column starting at position 1 for a length of 6 characters and compares it with the value 'Signal'. Position and length of the string can also be retrieved from a column value or another expression. For example, the following query uses a function for length.

```
SELECT lastname, firstname, title, dno
FROM employees
WHERE SUBSTRING (title FROM 1 FOR 6) = 'Signal';
```
WHERE SUBSTRING(title FROM 1 FOR CHAR_LENGTH(dno)) = 'Si';

The substring function retrieves the length of the dno column i.e., 2, and uses it to retrieve the number of characters from the title column:

<table>
<thead>
<tr>
<th>lastname</th>
<th>firstname</th>
<th>title</th>
<th>dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>Jane</td>
<td>Signal Engineer</td>
<td>EE</td>
</tr>
</tbody>
</table>

**Upper and Lower**

In a normal string comparison, the search is case sensitive. Therefore you cannot retrieve the results you need unless you know the exact case of every letter in the data. The functions upper and lower are useful to circumvent such situations. You can construct queries to produce a list of employees whose title ends with 'ENGINEER' or whose last name is 'smith' as follows:

```
SELECT lastname, firstname, title
FROM employees
WHERE UPPER (title) LIKE '%ENGINEER';
```

<table>
<thead>
<tr>
<th>lastname</th>
<th>firstname</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wickham</td>
<td>Scott</td>
<td>Software Engineer</td>
</tr>
<tr>
<td>Brown</td>
<td>Jane</td>
<td>Signal Engineer</td>
</tr>
<tr>
<td>Furtado</td>
<td>Judy</td>
<td>Software Engineer</td>
</tr>
<tr>
<td>Walsh</td>
<td>James</td>
<td>Software Engineer</td>
</tr>
<tr>
<td>Gomes</td>
<td>Mary</td>
<td>Software Engineer</td>
</tr>
</tbody>
</table>

```
SELECT lastname, firstname
FROM employees
WHERE LOWER (lastname) = 'smith';
```

<table>
<thead>
<tr>
<th>lastname</th>
<th>firstname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>Bruce</td>
</tr>
<tr>
<td>Smith</td>
<td>Sandra</td>
</tr>
</tbody>
</table>

**Position**

A column value can be compared with a specific string in a particular position. You can retrieve all the employees with a lastname containing the string 'Go' starting in position 1 as follows:

```
SELECT lastname, firstname
FROM employees
WHERE POSITION ('Go' IN lastname) = 1;
```

<table>
<thead>
<tr>
<th>lastname</th>
<th>firstname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Golden</td>
<td>Barbara</td>
</tr>
<tr>
<td>Gomes</td>
<td>Mary</td>
</tr>
</tbody>
</table>

**Cat**

Two strings can be concatenated using this function. If you want to compare the lastname and firstname together, then you could run the following query,
SELECT lastname, firstname
FROM employees
WHERE firstname || ' ' || lastname = 'Sandra Smith';

<table>
<thead>
<tr>
<th>lastname</th>
<th>firstname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>Sandra</td>
</tr>
</tbody>
</table>

Niladic Functions (pseudo-literals)

Niladic functions are functions that extract current state of the system such as an authorization identifier (e.g., CURRENT_USER) or the current date or time (e.g., CURRENT_DATE) from groups of rows. The available niladic functions are CURRENT_USER, CURRENT_USER_ID, CURRENT_GROUP, CURRENT_GROUP_ID, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, and SESSION_LABEL.

CURRENT_USER returns a string representing the current authorization ID. For example:

```sql
SELECT CURRENT_USER
FROM projects;
```

produces the result:

<table>
<thead>
<tr>
<th>CURRENT_USER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
</tr>
<tr>
<td>Alice</td>
</tr>
<tr>
<td>Alice</td>
</tr>
<tr>
<td>Alice</td>
</tr>
<tr>
<td>Alice</td>
</tr>
</tbody>
</table>

CURRENT_DATE returns a string representing the current date, i.e., “today”. For example:

```sql
SELECT CURRENT_DATE
FROM projects;
```

produces the result:

<table>
<thead>
<tr>
<th>CURRENT_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2007-11-28</td>
</tr>
<tr>
<td>2007-11-28</td>
</tr>
<tr>
<td>2007-11-28</td>
</tr>
<tr>
<td>2007-11-28</td>
</tr>
<tr>
<td>2007-11-28</td>
</tr>
</tbody>
</table>

CURRENT_TIME returns a string representing the current time in UTC, i.e., “now”. For example:

```sql
SELECT CURRENT_TIME
FROM projects;
```

produces the result:
DATE AND TIME FUNCTIONS - EXTRACT

The `EXTRACT` function can be used to extract specific values from the date, time or datetime format. For example, if you want to list the projects that started in the year 2007, you can run the following query.

```
SELECT pname, startdate
FROM projects
WHERE EXTRACT (YEAR FROM startdate) = 2007;
```

<table>
<thead>
<tr>
<th>pname</th>
<th>startdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flight Control Simulation</td>
<td>2007-02-01</td>
</tr>
</tbody>
</table>

Similarly, you can extract other parts of the date and time such as month, day, hour, minute or second.

DATA TYPE CONVERSION - CAST

The `CAST` function is used to perform explicit data type conversions of all kinds. In particular, it can be used to convert other scalar values, (e.g., character strings and numbers) to the various datetime and interval data types, and vice-versa.

```
SELECT title, CAST(rate AS INTEGER)
FROM billing_rate;
```

<table>
<thead>
<tr>
<th>Title</th>
<th>rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scientist</td>
<td>65</td>
</tr>
<tr>
<td>Software Engineer</td>
<td>35</td>
</tr>
<tr>
<td>Signal Engineer</td>
<td>35</td>
</tr>
<tr>
<td>Physicist</td>
<td>40</td>
</tr>
<tr>
<td>Communications Engineer</td>
<td>35</td>
</tr>
</tbody>
</table>

`CAST` also allows datetime values to be converted from one datetime data type to another. For example, a date or a time can be converted to a timestamp, and a timestamp can be converted to a date or a time.

```
SELECT pno, CAST(startdate AS TIMESTAMP)
FROM projects
WHERE pno = 'FCS';
```

<table>
<thead>
<tr>
<th>pno</th>
<th>Startdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCS</td>
<td>2007-02-01 00:00:00.0</td>
</tr>
</tbody>
</table>

ARITHMETIC EXPRESSION

In addition to simple column names, arithmetic expressions may appear in the `SELECT` portion of
a query. When a column is calculated rather than retrieved from a table or view, it must be given a name. For example, it may be useful to determine how much each employee's salary would be with a 5% increase. The query which performs this function is:

```sql
SELECT lastname, firstname, title, salary * 1.05
FROM employees;
```

If you would like to compare the increased salary against some amount(s), then you can use an arithmetic expression in the `WHERE` clause.

```sql
SELECT lastname, title, salary
FROM employees
WHERE title = 'Software Engineer'
AND salary * 1.05 NOT BETWEEN 40000.00 AND 55000.00;
```

Other arithmetic operators you can use are `+` for addition, `-` for subtraction and `/` for division.

**ORDER BY Clause**

You can control the order in which selected rows are displayed by adding an `ORDER BY` clause to the end of a query. For example, the rows of `employees` can be displayed in order of increasing salary:

```sql
SELECT lastname, firstname, salary
FROM employees
ORDER BY salary ASC;
```

The keyword `DESC`, for descending, may be used to display in order of decreasing salary. Note
the keyword **ASC** is implicit and thus need not be specified. You can order the result by two different columns in different order i.e., you can order by **title** in ascending order and **salary** in descending within each distinct column entry under title such as

```
SELECT lastname, firstname, salary, title
FROM employees
ORDER BY title, salary DESC;
```

<table>
<thead>
<tr>
<th>lastname</th>
<th>firstname</th>
<th>salary</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>Bruce</td>
<td>28000.00</td>
<td>Accountant</td>
</tr>
<tr>
<td>Xiu</td>
<td>Belinda</td>
<td>25000.00</td>
<td>Office Manager</td>
</tr>
<tr>
<td>Booker</td>
<td>Tracy</td>
<td>44000.00</td>
<td>Physicist</td>
</tr>
<tr>
<td>Golden</td>
<td>Barbara</td>
<td>60000.00</td>
<td>Scientist</td>
</tr>
<tr>
<td>Smith</td>
<td>Sandra</td>
<td>19000.00</td>
<td>Secretary</td>
</tr>
<tr>
<td>Brown</td>
<td>Jane</td>
<td>36000.00</td>
<td>Signal Engineer</td>
</tr>
<tr>
<td>Walsh</td>
<td>James</td>
<td>55000.00</td>
<td>Software Engineer</td>
</tr>
<tr>
<td>Wickham</td>
<td>Scott</td>
<td>45000.00</td>
<td>Software Engineer</td>
</tr>
<tr>
<td>Gomes</td>
<td>Mary</td>
<td>42000.00</td>
<td>Software Engineer</td>
</tr>
<tr>
<td>Furtado</td>
<td>Judy</td>
<td>35000.00</td>
<td>Software Engineer</td>
</tr>
</tbody>
</table>

**DISTINCT KEYWORD**

When one or more columns are projected from a table or view, duplicate values may be introduced. For example:

```
SELECT title
FROM employees;
```

<table>
<thead>
<tr>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accountant</td>
</tr>
<tr>
<td>Secretary</td>
</tr>
<tr>
<td>Office Manager</td>
</tr>
<tr>
<td>Scientist</td>
</tr>
<tr>
<td>Software Engineer</td>
</tr>
<tr>
<td>Signal Engineer</td>
</tr>
<tr>
<td>Physicist</td>
</tr>
<tr>
<td>Software Engineer</td>
</tr>
</tbody>
</table>

In order to suppress the extra occurrences of each title, the keyword **DISTINCT** may be included in the **SELECT** clause:

```
SELECT DISTINCT title
FROM employees;
```
MULTI-TABLE QUERIES

A query may entail fetching data from more than one table. For example, given the information from one row in one table, it may be necessary to retrieve related data from a row in another table and combine the two rows to produce a more complex report. This logical combination of tables is referred to as a join.

As an example, the billing_rate table tells you the hourly rate that can be billed, based on the title. However, you may wish to determine the hourly billing rate of each employee. Unfortunately, the billing_rate table doesn't contain the employee number; it must be retrieved after joining the employees table with the billing_rate table to produce the desired result:

```
SELECT firstname, lastname, title, rate
FROM employees, billing_rate
WHERE employees.title = billing_rate.title;
```

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>title</th>
<th>rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barbar</td>
<td>Golden</td>
<td>Scientist</td>
<td>65.00</td>
</tr>
<tr>
<td>Scott</td>
<td>Wickham</td>
<td>Software Engineer</td>
<td>35.00</td>
</tr>
<tr>
<td>Judy</td>
<td>Furtado</td>
<td>Software Engineer</td>
<td>35.00</td>
</tr>
<tr>
<td>James</td>
<td>Walsh</td>
<td>Software Engineer</td>
<td>35.00</td>
</tr>
<tr>
<td>Mary</td>
<td>Gomes</td>
<td>Software Engineer</td>
<td>35.00</td>
</tr>
<tr>
<td>Jane</td>
<td>Brown</td>
<td>Signal Engineer</td>
<td>35.00</td>
</tr>
<tr>
<td>Tracy</td>
<td>Booker</td>
<td>Physicist</td>
<td>40.00</td>
</tr>
</tbody>
</table>

Note that the comma appearing in the FROM clause specified that the two tables separated by the comma were to be joined to produce the result. Any number of tables may be joined in this fashion.

There are several types of join operations; cross join, natural join, inner join, outer join and union join. They are defined based on the way the tables are joined.

CROSS JOIN

A cross join is defined as the Cartesian (cross) product of the two tables, i.e., every row in the first table combines with every row in the second table. If you created a cross join on the departments and projects tables as

```
SELECT * FROM departments CROSS JOIN projects;
```

you would see the following rows:
A cross join can be specified implicitly or explicitly. In the example above, it is specified explicitly. An example of an implicit cross join, which would produce the same output as the previous table can be formed as follows:

```
SELECT * 
FROM departments, projects;
```

A condition can also be specified in this type of join such as:

```
SELECT * 
FROM projects, departments 
WHERE departments.dno = projects.dno;
```
This query retrieves rows having `dno` equal in both tables. Note that the output contains the `dno` column from both tables.

### JOIN SPECIFICATIONS

Both inner and outer joins are performed to show a relationship between one or more columns of the two tables involved. There are many possible ways that tables can be joined. One such relationship would be equality between two like-named columns in the two tables (mathematically referred to as an equi-join).

The most common of these join specifiers is the “NATURAL” join specifier. This term is used to signify that Trusted Rubix™ should select the columns which are common between the tables. The specific join operation will be performed using that assumption. Thus, only one copy of the common column(s) is displayed and the name of that column does not require a table qualifier, i.e., `dno` not `projects.dno`.

Two additional specifiers can be used in the join specification: “ON” which notes a conditional expression and “USING” which qualifies a specific column(s). If “NATURAL” is specified, neither an “ON” clause nor a “USING” clause can be specified.

### INNER JOIN

This join is the result of two or more columns from different tables being equal. The term “inner” comes from the fact all columns are non-null.

The previous example of an implicit cross join with the single equality condition specified (`departments.dno = projects.dno`) is equivalent to a natural inner join with some exceptions.

```sql
SELECT * FROM projects NATURAL JOIN departments;
```

The column header will be modified as mentioned per the natural specification. All the columns of “projects” are displayed first and include the “projects.” prefix except for the common column, "dno".
**dno.** This is followed by all the columns of “departments” all prefixed by “departments.” except for the common column, dno, which is deleted.

The output from the above query is equivalent to the following query:

```sql
SELECT projects.pno, projects.pname, projects.dno as dno, projects.budget, projects.startdate, projects.finishdate, departments.dname
FROM projects, departments
WHERE projects.dno = departments.dno;
```

<table>
<thead>
<tr>
<th>projects.</th>
<th>projects.</th>
<th>dno</th>
<th>projects.</th>
<th>projects.</th>
<th>projects.</th>
<th>depts.</th>
</tr>
</thead>
<tbody>
<tr>
<td>dno</td>
<td>pno</td>
<td>pno</td>
<td>dno</td>
<td>budget</td>
<td>startdate</td>
<td>finishdate</td>
</tr>
<tr>
<td>TMK</td>
<td>Tomahawk</td>
<td>EE</td>
<td>500000.00</td>
<td>2006-11-01</td>
<td>2007-11-01</td>
<td>Electrical</td>
</tr>
<tr>
<td>PCS</td>
<td>Patriot</td>
<td>EE</td>
<td>600000.00</td>
<td>2007-01-15</td>
<td>2007-06-20</td>
<td>Electrical</td>
</tr>
<tr>
<td>MGS</td>
<td>Missile</td>
<td>EE</td>
<td>100000.00</td>
<td>2006-06-01</td>
<td>2007-06-01</td>
<td>Electrical</td>
</tr>
<tr>
<td>IC</td>
<td>Inventory</td>
<td>SW</td>
<td>600000.00</td>
<td>2006-12-01</td>
<td>2007-09-15</td>
<td>Software</td>
</tr>
<tr>
<td>FCS</td>
<td>Flight</td>
<td>SW</td>
<td>100000.00</td>
<td>2007-02-01</td>
<td>2007-12-15</td>
<td>Software</td>
</tr>
</tbody>
</table>

**NOTE**

If the FROM clause in the query above had specified the tables in reverse order (i.e., FROM **departments** NATURAL JOIN **projects**), the resulting output would contain the same rows of data, but the columns will appear in a different order.

The natural inner join retrieves the result much more efficiently than a cross join. If you use a cross join, all the rows are joined and then the rows which don’t meet the condition are eliminated. But, in a natural join only rows which have equal department numbers are retrieved.

**Outer Join**

In outer joins, it is possible for some columns to contain NULL values. These NULL values are represented on output by "NULL". For example, when a row from one table is matched with a non-existent row from another table, the resultant column(s) will not have a value assigned to it.

There are three types of outer joins:

- \rightarrow LEF T
- \rightarrow R IGHT
- \rightarrow F UL L

If any of the three previous keywords are scanned, it is assumed that the join will be an outer join. In that case, the keyword OUTER may be omitted. As before, common fields are automatically chosen by Trusted Rubix when the keyword “NATURAL” is used.

The complete syntax for the OUTER JOIN is:

```sql
table_reference [NATURAL] outer-join-type JOIN table-reference [ON conditional-expression | USING (column-commalist)]
```

where outer-join-type is one of the following:

- left
- [outer]
If NATURAL is specified, neither an ON clause nor a USING clause can be specified; otherwise, one of the two must be specified.

All types of outer joins are similar to an inner join, except for how unmatched column values are handled. From the previous section on inner join, if a department number (dno) was retrieved from the projects table which did not exist in the departments table, the resultant joined row would not be included in the output. Thus, one would never know about its existence.

It might be useful to see even those rows which don’t have matches in the other table. Specifically, a natural left outer join would list all rows in the “left hand” table and their matching rows in the “right hand” table. When no matching row is found in the “right hand” table, all the column values for the “right hand” table will be set to NULL. The header will be the same as the header for the inner join.

```
SELECT * 
FROM   projects NATURAL LEFT JOIN departments;
```

The ON or USING clause in a join expression specifies the criteria for the join. The following two examples use the LEFT outer join.

```
table-reference LEFT JOIN table reference
    ON conditional_expression
```

If the ON clause is used, the result of the join contains every row of the Cartesian product of the two tables that satisfies the search condition; e.g.:

```
SELECT * 
FROM   projects LEFT JOIN departments 
ON     projects.dno = departments.dno 
AND    projects.budget > 500000.00;
```
Note that the expression produces a table with the common columns (projects.dno and departments.dno) appearing twice. When there are multiple join expressions and each join expression is associated with a different join condition, it is advantageous to use the ON clause.

In the USING clause variant, the user specifies the columns that are used in determining the result of the join. Each of the columns must be "unqualified", must identify a column in both tables, and have comparable data types.

```sql
SELECT * FROM projects LEFT JOIN departments USING (dno);
```

Each of the common columns in the resulting table appears only once, not twice as is produced using the ON clause.

**NOTE**

The output from the above query is equivalent to the NATURAL LEFT (OUTER) JOIN query.

If there are multiple columns (across both tables) that are comparable, the USING clause allows you to restrict the join operation to use only the specified common columns; the NATURAL JOIN clause uses all common columns for the join operation by default. To illustrate this difference, assume that some projects that were originally allocated to the Electrical Department ('EE') are now split between the Electrical ('EE') and Software Departments ('SW'). This is represented by
the table `subprojects` and is created by the following commands:

```sql
CREATE TABLE subprojects
(
    PRIMARY KEY (pno, dno),
    pno CHARACTER (3),
    dno CHARACTER (3),
    subbudget NUMERIC (8, 2)
) ;
INSERT INTO subprojects VALUES 'TMK', 'EE', 250000;
INSERT INTO subprojects VALUES 'TMK', 'SW', 250000;
INSERT INTO subprojects VALUES 'PCS', 'EE', 400000;
INSERT INTO subprojects VALUES 'PCS', 'SW', 200000;
INSERT INTO subprojects VALUES 'MGS', 'EE', 750000;
INSERT INTO subprojects VALUES 'MGS', 'SW', 250000;
COMMIT;
```

### Subprojects

<table>
<thead>
<tr>
<th>pno</th>
<th>subprojects. dno</th>
<th>subbudget</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMK</td>
<td>EE</td>
<td>250000.00</td>
</tr>
<tr>
<td>TMK</td>
<td>SW</td>
<td>250000.00</td>
</tr>
<tr>
<td>PCS</td>
<td>EE</td>
<td>400000.00</td>
</tr>
<tr>
<td>PCS</td>
<td>SW</td>
<td>200000.00</td>
</tr>
<tr>
<td>MGS</td>
<td>EE</td>
<td>750000.00</td>
</tr>
<tr>
<td>MGS</td>
<td>SW</td>
<td>250000.00</td>
</tr>
</tbody>
</table>

Note that while the total budget for individual projects remains the same (TMK original budget of $500000.00 allocated to the Electrical Department), it is now split among the Electrical and Software Departments. The following query would display all projects which had subprojects and their subbudgets.

```sql
SELECT projects.pno, projects.pname, projects.budget,
       subprojects.dno, subprojects.subbudget
FROM projects LEFT JOIN subprojects
USING (pno);
```

<table>
<thead>
<tr>
<th>pno</th>
<th>projects. pname</th>
<th>projects. budget</th>
<th>dno</th>
<th>subprojects. subbudget</th>
</tr>
</thead>
<tbody>
<tr>
<td>TML</td>
<td>Tomahawk Navigation</td>
<td>500000.00</td>
<td>EE</td>
<td>250000.00</td>
</tr>
<tr>
<td>TMK</td>
<td>Tomahawk Navigation</td>
<td>500000.00</td>
<td>SW</td>
<td>250000.00</td>
</tr>
<tr>
<td>PCS</td>
<td>Patriot Control System</td>
<td>600000.00</td>
<td>EE</td>
<td>400000.00</td>
</tr>
<tr>
<td>PCS</td>
<td>Patriot Control System</td>
<td>600000.00</td>
<td>SW</td>
<td>200000.00</td>
</tr>
<tr>
<td>MGS</td>
<td>Missile Guiding Systems</td>
<td>100000.00</td>
<td>EE</td>
<td>750000.00</td>
</tr>
<tr>
<td>MGS</td>
<td>Missile Guiding System</td>
<td>100000.00</td>
<td>SW</td>
<td>250000.00</td>
</tr>
<tr>
<td>FCS</td>
<td>Flight Control Simulation</td>
<td>100000.00</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>IC</td>
<td>Flight Control Simulation</td>
<td>100000.00</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

A natural right (outer) join would list all rows in the “right hand” table and their matching columns in the “left hand” table. The insertion of null values is handled analogously to a natural left outer join. Although the “right hand” table is used as a guide to what is displayed, the order of the header is still the same as in the left (outer) join.

```sql
SELECT * FROM employees
```
### NATURAL RIGHT OUTER JOIN billing_rate;

<table>
<thead>
<tr>
<th>title</th>
<th>ssn</th>
<th>lastname</th>
<th>firstname</th>
<th>salary</th>
<th>dno</th>
<th>rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communications Engineer</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>35.00</td>
</tr>
<tr>
<td>Physicist</td>
<td>224552500</td>
<td>Booker</td>
<td>Tracy</td>
<td>44000.00</td>
<td>EE</td>
<td>40.00</td>
</tr>
<tr>
<td>Scientist</td>
<td>897005240</td>
<td>Golden</td>
<td>Barbara</td>
<td>60000.00</td>
<td>EE</td>
<td>65.00</td>
</tr>
<tr>
<td>Signal Engineer</td>
<td>224255600</td>
<td>Brown</td>
<td>Jane</td>
<td>36000.00</td>
<td>EE</td>
<td>35.00</td>
</tr>
<tr>
<td>Software Engineer</td>
<td>111223333</td>
<td>Gomes</td>
<td>Mary</td>
<td>42000.00</td>
<td>Sw</td>
<td>35.00</td>
</tr>
<tr>
<td>Software Engineer</td>
<td>151007777</td>
<td>Furtado</td>
<td>Judy</td>
<td>35000.00</td>
<td>Sw</td>
<td>35.00</td>
</tr>
<tr>
<td>Software Engineer</td>
<td>228992500</td>
<td>Walsh</td>
<td>James</td>
<td>55000.00</td>
<td>Sw</td>
<td>35.00</td>
</tr>
<tr>
<td>Software Engineer</td>
<td>322333000</td>
<td>Wickham</td>
<td>Scott</td>
<td>45000.00</td>
<td>EE</td>
<td>35.00</td>
</tr>
<tr>
<td>Scientist</td>
<td>897005240</td>
<td>Golden</td>
<td>Barbara</td>
<td>60000.00</td>
<td>EE</td>
<td>65.00</td>
</tr>
<tr>
<td>Software Engineer</td>
<td>322333000</td>
<td>Wickham</td>
<td>Scott</td>
<td>45000.00</td>
<td>EE</td>
<td>35.00</td>
</tr>
<tr>
<td>Software Engineer</td>
<td>151007777</td>
<td>Furtado</td>
<td>Judy</td>
<td>35000.00</td>
<td>Sw</td>
<td>35.00</td>
</tr>
<tr>
<td>Software Engineer</td>
<td>228992500</td>
<td>Walsh</td>
<td>James</td>
<td>55000.00</td>
<td>Sw</td>
<td>35.00</td>
</tr>
<tr>
<td>Software Engineer</td>
<td>111223333</td>
<td>Gomes</td>
<td>Mary</td>
<td>42000.00</td>
<td>Sw</td>
<td>35.00</td>
</tr>
<tr>
<td>Signal Engineer</td>
<td>224255600</td>
<td>Brown</td>
<td>Jane</td>
<td>36000.00</td>
<td>EE</td>
<td>35.00</td>
</tr>
<tr>
<td>Physicist</td>
<td>224552500</td>
<td>Booker</td>
<td>Tracy</td>
<td>44000.00</td>
<td>EE</td>
<td>40.00</td>
</tr>
<tr>
<td>Accountant</td>
<td>100254789</td>
<td>Smith</td>
<td>Bruce</td>
<td>28000.00</td>
<td>ADM</td>
<td>NULL</td>
</tr>
<tr>
<td>Secretary</td>
<td>151235679</td>
<td>Smith</td>
<td>Sandra</td>
<td>19000.00</td>
<td>ADM</td>
<td>NULL</td>
</tr>
<tr>
<td>Office Manager</td>
<td>700340352</td>
<td>Xiu</td>
<td>Belinda</td>
<td>25000.00</td>
<td>ADM</td>
<td>NULL</td>
</tr>
<tr>
<td>Communications Engineer</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>35.00</td>
</tr>
</tbody>
</table>

A natural full (outer) join is the union of both the natural left (outer) join and the natural right (outer) join. Again, the table header is the same.

```sql
SELECT * FROM employees NATURAL FULL OUTER JOIN billing_rate;
```

<table>
<thead>
<tr>
<th>title</th>
<th>ssn</th>
<th>lastname</th>
<th>firstname</th>
<th>salary</th>
<th>dno</th>
<th>rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scientist</td>
<td>897005240</td>
<td>Golden</td>
<td>Barbara</td>
<td>60000.00</td>
<td>EE</td>
<td>65.00</td>
</tr>
<tr>
<td>Software Engineer</td>
<td>322333000</td>
<td>Wickham</td>
<td>Scott</td>
<td>45000.00</td>
<td>EE</td>
<td>35.00</td>
</tr>
<tr>
<td>Software Engineer</td>
<td>151007777</td>
<td>Furtado</td>
<td>Judy</td>
<td>35000.00</td>
<td>Sw</td>
<td>35.00</td>
</tr>
<tr>
<td>Software Engineer</td>
<td>228992500</td>
<td>Walsh</td>
<td>James</td>
<td>55000.00</td>
<td>Sw</td>
<td>35.00</td>
</tr>
<tr>
<td>Software Engineer</td>
<td>111223333</td>
<td>Gomes</td>
<td>Mary</td>
<td>42000.00</td>
<td>Sw</td>
<td>35.00</td>
</tr>
<tr>
<td>Signal Engineer</td>
<td>224255600</td>
<td>Brown</td>
<td>Jane</td>
<td>36000.00</td>
<td>EE</td>
<td>35.00</td>
</tr>
<tr>
<td>Physicist</td>
<td>224552500</td>
<td>Booker</td>
<td>Tracy</td>
<td>44000.00</td>
<td>EE</td>
<td>40.00</td>
</tr>
<tr>
<td>Accountant</td>
<td>100254789</td>
<td>Smith</td>
<td>Bruce</td>
<td>28000.00</td>
<td>ADM</td>
<td>NULL</td>
</tr>
<tr>
<td>Secretary</td>
<td>151235679</td>
<td>Smith</td>
<td>Sandra</td>
<td>19000.00</td>
<td>ADM</td>
<td>NULL</td>
</tr>
<tr>
<td>Office Manager</td>
<td>700340352</td>
<td>Xiu</td>
<td>Belinda</td>
<td>25000.00</td>
<td>ADM</td>
<td>NULL</td>
</tr>
<tr>
<td>Communications Engineer</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>35.00</td>
</tr>
</tbody>
</table>

It is now easier to understand why an inner join is so named. An inner join is the degenerative case of an outer join in which there are no nulls.
Union Join

In a union join, all the columns of the second table are filled with NULLs for each row in the first table and all columns of the first table are filled with NULLs for each row in the second table.

SELECT * FROM employees UNION JOIN billing_rate;

<table>
<thead>
<tr>
<th>ssn</th>
<th>lastname</th>
<th>firstname</th>
<th>salary</th>
<th>title</th>
<th>dno</th>
<th>billing_rate. title</th>
<th>billing_rate. rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>100254789</td>
<td>Smith</td>
<td>Bruce</td>
<td>28000.00</td>
<td>Accountant</td>
<td>ADM</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>151235679</td>
<td>Smith</td>
<td>Sandra</td>
<td>19000.00</td>
<td>Secretary</td>
<td>ADM</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>700340352</td>
<td>Xiu</td>
<td>Belinda</td>
<td>25000.00</td>
<td>Office Manager</td>
<td>ADM</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>897005240</td>
<td>Golden</td>
<td>Barbara</td>
<td>60000.00</td>
<td>Scientist</td>
<td>EE</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>322333000</td>
<td>Wickham</td>
<td>Scott</td>
<td>45000.00</td>
<td>Software Engineer</td>
<td>EE</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>224555600</td>
<td>Brown</td>
<td>Jane</td>
<td>36000.00</td>
<td>Signal Engineer</td>
<td>EE</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>224552500</td>
<td>Booker</td>
<td>Tracy</td>
<td>44000.00</td>
<td>Physicist</td>
<td>EE</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>151007777</td>
<td>Furtado</td>
<td>Judy</td>
<td>35000.00</td>
<td>Software Engineer</td>
<td>SW</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>228992500</td>
<td>Walsh</td>
<td>James</td>
<td>55000.00</td>
<td>Software Engineer</td>
<td>SW</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>111223333</td>
<td>Gomes</td>
<td>Mary</td>
<td>42000.00</td>
<td>Software Engineer</td>
<td>SW</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>Scientist</td>
<td>65.00</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>Software Engineer</td>
<td>35.00</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>Signal Engineer</td>
<td>35.00</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>Physicist</td>
<td>40.00</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>Communications Engineer</td>
<td>35.00</td>
</tr>
</tbody>
</table>

Testing for True, False, Null

A condition in the WHERE clause has to be true to retrieve results. A condition can be compared to see if it is true or false. If you want to list all employees except employees of the administration department, then you can use the following query:

SELECT * FROM employees WHERE dno = 'ADM' IS FALSE;
You can test for null values using IS NULL in the WHERE clause. There may be some employees whose titles don't have a billing rate, and that can be retrieved using the following query:

```sql
SELECT title, dno, rate
FROM employees NATURAL LEFT OUTER JOIN billing_rate
WHERE rate IS NULL;
```

<table>
<thead>
<tr>
<th>title</th>
<th>dno</th>
<th>rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accountant</td>
<td>ADM</td>
<td>NULL</td>
</tr>
<tr>
<td>Secretary</td>
<td>ADM</td>
<td>NULL</td>
</tr>
<tr>
<td>Office Manager</td>
<td>ADM</td>
<td>NULL</td>
</tr>
</tbody>
</table>

NOT can be combined with TRUE, FALSE or NULL. For example, if you wanted to list all employees who do have a billing rate, use the following query:

```sql
SELECT lastname, title, dno, rate
FROM employees NATURAL LEFT OUTER JOIN billing_rate
WHERE rate IS NOT NULL;
```

<table>
<thead>
<tr>
<th>lastname</th>
<th>title</th>
<th>dno</th>
<th>rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Golden</td>
<td>Scientist</td>
<td>EE</td>
<td>65.00</td>
</tr>
<tr>
<td>Wickham</td>
<td>Software Engineer</td>
<td>EE</td>
<td>35.00</td>
</tr>
<tr>
<td>Furtado</td>
<td>Software Engineer</td>
<td>SW</td>
<td>35.00</td>
</tr>
<tr>
<td>Walsh</td>
<td>Software Engineer</td>
<td>SW</td>
<td>35.00</td>
</tr>
<tr>
<td>Gomes</td>
<td>Software Engineer</td>
<td>SW</td>
<td>35.00</td>
</tr>
<tr>
<td>Brown</td>
<td>Signal Engineer</td>
<td>EE</td>
<td>35.00</td>
</tr>
<tr>
<td>Booker</td>
<td>Physicist</td>
<td>EE</td>
<td>40.00</td>
</tr>
</tbody>
</table>

**ALIASES (TABLE AND COLUMN REFERENCES)**

Queries can be made shorter and more readable by using aliases for table names and column names. They are useful when subqueries use the same tables to resolve ambiguity. An alias name can be used instead of the full table name to distinguish columns of a table. For example, if you would like to find out the contents of the `works` table, and have column headings listed as `w.pno` rather than `works.pno`, type the following:

```sql
SELECT *
FROM works AS w;
```
AGGREGATE FUNCTIONS

Aggregate functions are functions which extract aggregate values from groups of rows. Examples of such functions include:

→ how many employees are there in each department;
→ how much revenue is being generated by a given department; or
→ which department has more than three ongoing projects.

The available aggregate functions are COUNT, MIN, MAX, AVG, and SUM. The MIN, MAX, AVG, and SUM functions apply to a column and yield (respectively) the smallest, largest, average, and summed value of that column. The COUNT function applied to a column yields the number of all or distinct values in that column; the special COUNT(*) function counts all of the rows in the group. If you wish to find out the number of employees who earn more than $40000.00, the query,

```sql
SELECT COUNT (*)
FROM employees
WHERE salary >= 40000.00;
```

produces the result:

```
count(*)
5
```

The aggregate functions can be combined in a single statement to produce statistics on employees such as:

→ how many employees do we have,
→ what is the minimum salary of an employee,
→ what is the maximum salary of an employee,
→ what is the average salary of all employees, and
→ what is the total salaries of all employees.

```sql
SELECT COUNT (*), MIN (salary), MAX (salary), AVG (salary), SUM (salary)
FROM employees;
```
SUM and AVG can only be performed on numeric columns whereas COUNT, MIN, and MAX can be performed on any column.

GROUP BY AND HAVING CLAUSES

So far you are able to perform an aggregate function on a single group of rows. You must use the GROUP BY and HAVING clauses to perform aggregate functions on multiple groups. The group of rows to which a given function applies is specified by the GROUP BY clause.

The GROUP BY clause groups the rows based on the value of the columns in the clause and applies the aggregate function to that group.

Given this capability, you are now prepared to determine such results as the number of employees in each department and the average and total salaries for each department:

```
SELECT dno, COUNT(*), SUM(salary), AVG(salary)
FROM employees
GROUP BY dno
HAVING count(*) >= 3;
```

<table>
<thead>
<tr>
<th>dno</th>
<th>count(*)</th>
<th>sum(salary)</th>
<th>avg(salary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADM</td>
<td>3</td>
<td>72000.00</td>
<td>24000.00</td>
</tr>
<tr>
<td>EE</td>
<td>4</td>
<td>185000.00</td>
<td>46250.00</td>
</tr>
<tr>
<td>SW</td>
<td>3</td>
<td>132000.00</td>
<td>44000.00</td>
</tr>
</tbody>
</table>

The HAVING clause is analogous to the WHERE clause, except that it operates on a group of rows rather than on a particular row. Thus, if you wish to restrict the above query and list only the average and total salaries for the ADM department, you could formulate the query as follows:

```
SELECT dno, COUNT(*), SUM(salary), AVG(salary)
FROM employees
GROUP BY dno
HAVING dno = 'ADM';
```

<table>
<thead>
<tr>
<th>dno</th>
<th>count(*)</th>
<th>sum(salary)</th>
<th>avg(salary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADM</td>
<td>3</td>
<td>72000.00</td>
<td>24000.00</td>
</tr>
</tbody>
</table>

The ANSI standard for SQL specifies that GROUP BY and HAVING directives cannot be applied to a SELECT statement in a view, if the view definition itself included the GROUP BY or HAVING keywords. That is because the view definition is “substituted” online to the SELECT statement and as a result, two GROUP BY clauses appear. Fortunately, the RUBIX/SQL user is free from this restriction:

→ GROUP BY and HAVING can be freely applied to views,
→ even those which include GROUP BY and HAVING in their definitions, and
→ reasonable results will ensue.

It should be noted, that the most generalized query may contain a WHERE clause; a GROUP BY
clause with or without a corresponding HAVING clause; or any combination thereof.

**SUBQUERIES**

A query may result in printed output, or it may act as a *subquery* which provides intermediate results to be used by another query. For example, you may wish to know the name and title of employees who are working on projects whose budget is less than $200000.00. You can perform this exercise using a sequence of queries:

```sql
SELECT pno
FROM projects
WHERE budget <= 200000.00;
```

<table>
<thead>
<tr>
<th>Pno</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCS</td>
</tr>
<tr>
<td>MGS</td>
</tr>
</tbody>
</table>

```sql
SELECT DISTINCT lastname, firstname, title
FROM employees, works
WHERE works.ssn = employees.ssn
AND pno IN ( 'MGS', 'FCS');
```

<table>
<thead>
<tr>
<th>lastname</th>
<th>firstname</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>Jane</td>
<td>Signal Engineer</td>
</tr>
<tr>
<td>Walsh</td>
<td>James</td>
<td>Software Engineer</td>
</tr>
<tr>
<td>Wickham</td>
<td>Scott</td>
<td>Software Engineer</td>
</tr>
</tbody>
</table>

Notice that we have to use the results of the first query as an implicit condition in the **WHERE** clause of the second query. As an alternative, you can nest the first query within the condition part of the second query, to result in the one-step formulation:

```sql
SELECT DISTINCT lastname, firstname, title
FROM employees, works
WHERE works.ssn = employees.ssn
AND pno IN (SELECT pno
             FROM projects
             WHERE budget <= 200000.00);
```

<table>
<thead>
<tr>
<th>lastname</th>
<th>firstname</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>Jane</td>
<td>Signal Engineer</td>
</tr>
<tr>
<td>Walsh</td>
<td>James</td>
<td>Software Engineer</td>
</tr>
<tr>
<td>Wickham</td>
<td>Scott</td>
<td>Software Engineer</td>
</tr>
</tbody>
</table>

This query can be reformulated to eliminate the implicit cross join operation and form another level of nesting as:

```sql
SELECT lastname, firstname, title
FROM employees
WHERE ssn IN (SELECT ssn
               FROM projects
               WHERE budget <= 200000.00);
```

<table>
<thead>
<tr>
<th>lastname</th>
<th>firstname</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>Jane</td>
<td>Signal Engineer</td>
</tr>
<tr>
<td>Walsh</td>
<td>James</td>
<td>Software Engineer</td>
</tr>
<tr>
<td>Wickham</td>
<td>Scott</td>
<td>Software Engineer</td>
</tr>
</tbody>
</table>

© 2014 Infosystems Technology, Inc.
FROM works
WHERE pno IN (SELECT pno
FROM projects
WHERE budget <=200000.00));

to produce the same result as above:

<table>
<thead>
<tr>
<th>lastname</th>
<th>firstname</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wickham</td>
<td>Scott</td>
<td>Software Engineer</td>
</tr>
<tr>
<td>Brown</td>
<td>Jane</td>
<td>Signal Engineer</td>
</tr>
<tr>
<td>Walsh</td>
<td>James</td>
<td>Software Engineer</td>
</tr>
</tbody>
</table>

Subqueries can be part of a FROM clause. If you use subqueries in the FROM clause when a join is specified explicitly or implicitly, the performance can be greatly improved. For example, if you would like to retrieve the salaries and billing rates of all the physicists, you can formulate the query as follows:

```sql
SELECT lastname, title, salary, rate
FROM employees NATURAL JOIN (SELECT *
FROM billing_rate
WHERE title= 'Physicist'
AS T;
```

<table>
<thead>
<tr>
<th>lastname</th>
<th>title</th>
<th>salary</th>
<th>rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Booker</td>
<td>Physicist</td>
<td>44000.00</td>
<td>40.00</td>
</tr>
</tbody>
</table>

In the example above, the correlation_name T ranges over an unnamed table. Specifically, a derived unnamed table that is the result of the natural join of the employees and billing_rate tables ranging over the rows of the table. Thus, the only rows returned as output are those whose column title is indicated as “Physicist”.

A great deal of flexibility is provided by the capability to combine arbitrary Boolean conditions and subqueries within a single RUBIX/SQL query.

**SET OPERATIONS**

The set operator UNION can also be used to access information from the database. This operator may be useful in avoiding join operations in some cases. Two queries can be combined to form a single query with a UNION operator. A typical query is of the form:

```sql
<query> <set operator> <query>
```

The structure of the result of both queries should be the same as performing a UNION operation.

**UNION Operator**

The UNION operator works similar to an OR condition. A row which is part of at least one of the queries appears in the result. Duplicate rows are eliminated from the result. For example, all the present titles in use can be retrieved using both the employees and billing_rate tables:

```sql
SELECT DISTINCT title
FROM employees
```
Duplicate rows can be retained in the result by specifying **UNION ALL** as:

```sql
SELECT DISTINCT title
FROM employees
UNION ALL
SELECT DISTINCT title
FROM billing_rate;
```

**Title**

<table>
<thead>
<tr>
<th>Accountant</th>
<th>Communications Engineer</th>
<th>Office Manager</th>
<th>Physicist</th>
<th>Scientist</th>
<th>Secretary</th>
<th>Signal Engineer</th>
<th>Software Engineer</th>
</tr>
</thead>
</table>

**INTERSECT OPERATOR**

The **INTERSECT** operator returns all rows that exist in the intersection of two tables. For example, create two tables of titles of employees. Some employees are full-time while other are part-time.

**Full_time**

```sql
SELECT DISTINCT title
FROM employees
INTERSECT
SELECT DISTINCT title
FROM billing_rate;
```

**Title**

<table>
<thead>
<tr>
<th>Accountant</th>
<th>Software Engineer</th>
<th>Software Engineer</th>
<th>Administrator</th>
<th>Signal Engineer</th>
<th>Software Engineer</th>
</tr>
</thead>
</table>
Part_time

<table>
<thead>
<tr>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Office Manager</td>
</tr>
<tr>
<td>Software Engineer</td>
</tr>
<tr>
<td>Software Engineer</td>
</tr>
<tr>
<td>Administrator</td>
</tr>
<tr>
<td>Scientist</td>
</tr>
</tbody>
</table>

To illustrate the effect of the \texttt{INTERSECT} operator see the following query in which two lists of five are in the intersection. This resulting list contains only those employees who are in both tables (with duplicate elimination just like \texttt{UNION}).

\[
\text{SELECT title FROM full_time}
\text{INTERSECT}
\text{SELECT title FROM part_time;}
\]

EXCEPT OPERATOR

The \texttt{EXCEPT} operator is used to return all rows that are in the first table, except those that also appear in the second table. For example, the titles in the second table are excepted from those in the first table. The only titles listed are those in the first table (\texttt{Full_time}) and not in the second table (\texttt{Part_time}).

\[
\text{SELECT title FROM full_time}
\text{EXCEPT}
\text{SELECT title FROM part_time;}
\]

Next, the titles in the first table are excepted from those in the second table. The only titles listed are those in the second table (\texttt{Part_time}) and not in the first table (\texttt{Full_time}).

\[
\text{SELECT title FROM part_time}
\text{EXCEPT}
\text{SELECT title FROM full_time;}
\]

The \texttt{ALL} keyword functions with \texttt{INTERSECT} and \texttt{EXCEPT} in a manner similar to \texttt{UNION}. With respect to all set operators (\texttt{UNION}, \texttt{UNION ALL}, \texttt{INTERSECT}, \texttt{INTERSECT ALL}, \texttt{EXCEPT}, and \texttt{EXCEPT ALL}), Trusted RUBIX supports the \texttt{CORRESPONDING} and \texttt{CORRESPONDING BY} (column-commalist).
DATA MODIFICATION

UPDATE OPERATIONS

A single SELECT statement allowed you to retrieve sets of rows from one or more tables or views. RUBIX/SQL lets you add, delete, or modify rows with the following commands:

- **INSERT** - add new rows to a table
- **UPDATE** - modify existing rows within a table
- **DELETE** - remove existing rows from a table

INSERT COMMAND

The simplest case of the **INSERT** command adds a set of fixed values to a table, one row at a time, as follows:

```
INSERT INTO billing_rate VALUES 'Power Engineer', 45.00;
```

If the command succeeds, no message is issued and RUBIX/SQL returns to the prompt. If the insertion fails, an error message is issued. The results of the insertion into the table can be verified by using the **SELECT** statement such as

```
SELECT * FROM billing_rate;
```

<table>
<thead>
<tr>
<th>billing_rate.title</th>
<th>billing_rate.rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scientist</td>
<td>65.00</td>
</tr>
<tr>
<td>Software Engineer</td>
<td>35.00</td>
</tr>
<tr>
<td>Signal Engineer</td>
<td>35.00</td>
</tr>
<tr>
<td>Physicist</td>
<td>40.00</td>
</tr>
<tr>
<td>Communications Engineer</td>
<td>35.00</td>
</tr>
<tr>
<td>Power Engineer</td>
<td>45.00</td>
</tr>
</tbody>
</table>

You can insert multiple rows at the same time using the following command:

```
INSERT INTO billing_rate VALUES ('Network Engineer', 50.00), ('Systems Analyst', 40.00);
```

The results of the insertion into the table can be verified by using the **SELECT** statement such as

```
SELECT * FROM billing_rate;
```
It is also possible to add a set of rows to a table which are drawn from another table or group of tables. This is accomplished using the `INSERT-SELECT` command. For example, assume that there is an `emp_hourly` table containing employee's social security number, hourly salary and hourly billing rate. This table can be created using the following `CREATE TABLE` command:

```sql
CREATE TABLE emp_hourly
(
    PRIMARY KEY (ssn),
    ssn INTEGER,
    salary_rate NUMERIC(12,2),
    bill_rate NUMERIC(12,2)
);
```

The pertinent information for this table can be extracted from the `employees` and `billing_rate` tables and entered into `emp_hourly` via the command:

```sql
INSERT INTO emp_hourly
SELECT ssn, salary / 2080, rate
FROM employees NATURAL JOIN billing_rate;
```

The results of the `INSERT` can, of course, be verified by issuing a `SELECT` command:

```sql
SELECT *
FROM emp_hourly;
```

<table>
<thead>
<tr>
<th>ssn</th>
<th>salary_rate</th>
<th>bill_rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>897005240</td>
<td>28.85</td>
<td>65.00</td>
</tr>
<tr>
<td>322333000</td>
<td>21.63</td>
<td>35.00</td>
</tr>
<tr>
<td>151807777</td>
<td>16.82</td>
<td>35.00</td>
</tr>
<tr>
<td>228992500</td>
<td>26.44</td>
<td>35.00</td>
</tr>
<tr>
<td>111223333</td>
<td>20.19</td>
<td>35.00</td>
</tr>
<tr>
<td>224255600</td>
<td>17.30</td>
<td>35.00</td>
</tr>
<tr>
<td>224552500</td>
<td>21.15</td>
<td>40.00</td>
</tr>
</tbody>
</table>

One form of the `INSERT` command installs values in columns when appropriate and NULL values in columns with no data.

```sql
INSERT INTO projects
VALUES 'LMD', 'Land Mine Detection', 'EE', 4000000.00, NULL, NULL;
```
An alternative form of the INSERT command allows you to install values within only certain columns of the table while the remaining columns are set to NULL. Either a SELECT clause or a set of VALUES clauses may follow the INSERT INTO phrase. For example, the command:

```
INSERT INTO projects (pno, pname, dno, budget)
VALUES 'LMD', 'Land Mine Detection', 'EE', 4000000.00;
```

initializes only the `pno`, `pname`, `dno` and `budget` fields of the newly inserted rows, so that the resulting appearance of the `projects` table is:

```
SELECT *
FROM projects order by pno;
```

<table>
<thead>
<tr>
<th>pno</th>
<th>pname</th>
<th>dno</th>
<th>budget</th>
<th>startdate</th>
<th>finishdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCS</td>
<td>Flight Control Simulation</td>
<td>SW</td>
<td>100000.00</td>
<td>2007-02-01</td>
<td>2007-12-15</td>
</tr>
<tr>
<td>IC</td>
<td>Inventory Control</td>
<td>SW</td>
<td>600000.00</td>
<td>2006-12-01</td>
<td>2007-09-15</td>
</tr>
<tr>
<td>LMD</td>
<td>Land Mine Detection</td>
<td>EE</td>
<td>4000000.00 NULL NULL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MGS</td>
<td>Missile Guiding System</td>
<td>EE</td>
<td>1000000.00</td>
<td>2006-06-01</td>
<td>2007-06-01</td>
</tr>
<tr>
<td>PCS</td>
<td>Patriot Control System</td>
<td>EE</td>
<td>6000000.00</td>
<td>2007-01-15</td>
<td>2007-06-20</td>
</tr>
<tr>
<td>TMK</td>
<td>Tomahawk Navigation</td>
<td>EE</td>
<td>5000000.00</td>
<td>2006-11-01</td>
<td>2007-11-01</td>
</tr>
</tbody>
</table>

**UPDATE COMMAND**

The UPDATE command enables you to modify field values in rows which already exist within a table or view. Note that no rows are either added to or deleted from the table when UPDATE is specified. For example, the billing rates can be increased by 5% as follows:

```
UPDATE billing_rate
SET rate = rate * 1.05;
```

The changes can be verified by issuing a SELECT command:

```
SELECT *
FROM billing_rate;
```

<table>
<thead>
<tr>
<th>title</th>
<th>rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scientist</td>
<td>65.00</td>
</tr>
<tr>
<td>Software Engineer</td>
<td>35.00</td>
</tr>
<tr>
<td>Signal Engineer</td>
<td>35.00</td>
</tr>
<tr>
<td>Physicist</td>
<td>40.00</td>
</tr>
<tr>
<td>Communications Engineer</td>
<td>35.00</td>
</tr>
<tr>
<td>Power Engineer</td>
<td>45.00</td>
</tr>
<tr>
<td>Network Engineer</td>
<td>50.00</td>
</tr>
<tr>
<td>System Analyst</td>
<td>45.00</td>
</tr>
</tbody>
</table>

Note that the UPDATE statement is applied to all rows within the table. Such broad consequences might be desirable in certain circumstances, but, in general, it may be desirable to update only certain rows. The set of rows to be updated can be specified using the UPDATE-WHERE command. For example, if it is determined that the minimum billing rate for any employee is $20.00, then
rates can be revised upward to exactly $40.00:

```sql
UPDATE billing_rate
SET rate = 40
WHERE rate <= 40;
```

Once again, we can verify the changes by issuing a `SELECT` command:

```sql
SELECT *
FROM billing_rate
WHERE rate <= 40.00 order by title;
```

<table>
<thead>
<tr>
<th>billing_rate.title</th>
<th>billing_rate.rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communications Engineer</td>
<td>40.00</td>
</tr>
<tr>
<td>Signal Engineer</td>
<td>40.00</td>
</tr>
<tr>
<td>Software Engineer</td>
<td>40.00</td>
</tr>
</tbody>
</table>

Note that RUBIX/SQL permits the user to `UPDATE` any defined view. The semantics of the `UPDATE` will correspond exactly to the semantics of the relational model. This is in contrast to the ANSI SQL standard, where a view may only be updated if it derives from exactly one underlying base relation. See the section titled VIEW UPDATABILITY, for more details on updating `Trusted Rubix™` views.

**DELETE COMMAND**

Finally, rows can be deleted from a table by using the `DELETE` command. `DELETE` resembles the `UPDATE` command in that a `WHERE` clause may be specified. For example, you can drop employees who have left the company and view the results.

```sql
DELETE
FROM employees
WHERE lastname = 'Smith' AND firstname = 'Sandra';
```

```sql
SELECT *
FROM employees;
```

<table>
<thead>
<tr>
<th>employees.ssn</th>
<th>employees.lastname</th>
<th>employees.firstname</th>
<th>employees.salary</th>
<th>employees.title</th>
<th>employees.dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>100254789</td>
<td>Smith</td>
<td>Bruce</td>
<td>28000.00</td>
<td>Accountant</td>
<td>ADM</td>
</tr>
<tr>
<td>700340352</td>
<td>Xiu</td>
<td>Belinda</td>
<td>25000.00</td>
<td>Office Manager</td>
<td>ADM</td>
</tr>
<tr>
<td>897005240</td>
<td>Golden</td>
<td>Barbara</td>
<td>60000.00</td>
<td>Scientist</td>
<td>EE</td>
</tr>
<tr>
<td>322333000</td>
<td>Wickham</td>
<td>Scott</td>
<td>45000.00</td>
<td>Software Engineer</td>
<td>EE</td>
</tr>
<tr>
<td>224255600</td>
<td>Brown</td>
<td>Jane</td>
<td>36000.00</td>
<td>Signal Engineer</td>
<td>EE</td>
</tr>
<tr>
<td>224552500</td>
<td>Booker</td>
<td>Tracy</td>
<td>44000.00</td>
<td>Physicist</td>
<td>EE</td>
</tr>
<tr>
<td>151007777</td>
<td>Furtado</td>
<td>Judy</td>
<td>35000.00</td>
<td>Software Engineer</td>
<td>SW</td>
</tr>
<tr>
<td>228992500</td>
<td>Walsh</td>
<td>James</td>
<td>55000.00</td>
<td>Software Engineer</td>
<td>SW</td>
</tr>
<tr>
<td>111223333</td>
<td>Gomes</td>
<td>Mary</td>
<td>42000.00</td>
<td>Software Engineer</td>
<td>SW</td>
</tr>
</tbody>
</table>

Note that the `DELETE` command need not be supplied with a `WHERE` clause. The effect of such an
unconditional **DELETE** command is to delete every row from the named table. You can delete the entire *emp_hourly* table by using:

```
DELETE
FROM emp_hourly;
```

```
SELECT *
FROM emp_hourly;
```

You will notice that no records are selected, as they have all been deleted.
DATA DEFINITION

CREATE TABLE Command

Before queries can be issued to interrogate a table, the table must be created and populated with data. The Trusted Rubix/SQL user creates a table by use of the CREATE TABLE command. In the case of the employees table, you can define it as:

```
CREATE TABLE employees
(
    ssn INTEGER,
    lastname VARCHAR(10),
    firstname VARCHAR(10),
    salary NUMERIC(12,2),
    title VARCHAR(25),
    dno VARCHAR(3)
);
```

You can see that the employees table definition has six columns which are assigned names and the type of data that can be stored in them. Each column represents a specific property of the table. A column is referenced using the name assigned to it and is represented using the data type. The data type determines the data that can be stored in the column and also the operations that can be performed on the data. The amount of storage used depends on the data type used.

Data Types

The generic data types available in TR/SQL are character string, character large object, bit string, binary large object, exact numeric, approximate numeric, datetime, and security label. For a detailed description of available data types, please refer to the section on data types in the TR SQL Reference Guide. To choose a data type for a particular column, you have to analyze what type of data can possibly be stored in it. The description of what each data type can store is given below.

Exact Numeric

When the data is numeric and integral, an exact numeric such as INTEGER or SMALLINT can be used. If the data is guaranteed to be between -32,768 and 32,767, SMALLINT can be used which uses only 2 bytes for storage whereas an INTEGER type uses four bytes and can accommodate values from -2,147,483,648 and 2,147,438,647. For example, you can see that INTEGER type is used to store a social security number in the employees table. If the range of values of INTEGER is insufficient, then a DECIMAL type can be used. A DECIMAL type can be used when the data is exact, but not necessarily integral. A DECIMAL type has a precision and an optional scale associated with it. The precision specifies the maximum number of digits and the scale specifies the number of digits to the right of the decimal point. For integral numbers, scale must be defined as zero. The default value of scale is zero. The data type NUMERIC has the same characteristics as DECIMAL and it can be used alternatively.

Approximate Numeric

When the data is numeric and not exact, a floating point storage can be used. DOUBLE PRECISION, FLOAT and REAL can be used to represent approximate numeric types. These types are used mostly in scientific applications and can store a large spectrum of values. The precision of these types is implementation dependent.
CHARACTER STRING

Any data containing printable ASCII characters can be stored using the character data type. The operations such as SUM and AVG cannot be performed on character fields. The maximum length of the data that can be stored in the column has to be specified as the field length. For example, the lastname column in the employees table is specified as VARCHAR(10), which means a string of maximum length 10 can be stored in that column. The storage space used for this is fixed, irrespective of how small the string is. A maximum length of 3,145,728 can be specified for a CHARACTER type. If the column length is likely to vary significantly for each row, a variable length character string can be used. It can be specified as CHARACTER VARYING(n) where n is the maximum length that can be stored in that column. When a variable length type is specified, the data is stored in only as much space as it needs.

CHARACTER LARGE OBJECT (CLOB)

A character large object (CLOB) data type contains printable ASCII characters and may be much larger than a CHARACTER or CHARACTER VARYING type. The maximum length of a CLOB field is fixed at 2,147,483,647 characters. The data of each CLOB field is stored separately from its row. An integer identifier is stored in the row which uniquely identifies the storage location of the field’s data. During access, the integer identifier is first retrieved and then the operation (e.g., select, update) is performed. Thus CLOB access is always a two step process. This two step process is automatically performed by the RDBMS as SQL accesses are performed. Due to the large size of a CLOB field, operations are limited to simple select, insert, delete, and update. A CLOB column may not appear in a WHERE clause and may not be an argument to any function or comparison.

BIT STRING

A bit string is a sequence of bits, each having the value of 0 or 1. A bit string has a length, which is the number of bits in the string. The length is 0 or a positive integer. The data type bit string is specified by BIT. Bits in a bit string are numbered beginning with 1. If VARYING is not specified, then the length in bits of the bit string is fixed at n. If VARYING is specified, then the length in bits of the string is variable, with a minimum length of 0 and a maximum length of n. A maximum length of 25,165,824 may be specified for a BIT type. When a variable length type is specified, the data is stored in only as much space as it needs.

BINARY LARGE OBJECT (BLOB)

A binary large object (BLOB) data type contains binary data, in bytes, and may be much larger than a BIT type. The maximum length of a BLOB field is fixed at 2,147,483,647 bytes. The data of each BLOB field is stored separately from its row. An integer identifier is stored in the row which uniquely identifies the storage location of the field’s data. During access, the integer identifier is first retrieved and then the operation (e.g., select, update) is performed. Thus BLOB access is always a two step process. This two step process is automatically performed by the RDBMS as SQL accesses are performed. Due to the large size of a BLOB field, operations are limited to simple select, insert, delete, and update. A BLOB column may not appear in a WHERE clause and may not be an argument to any function or comparison.

DATE, TIME AND DATETIME

A calendar date can be represented using a DATE type. It contains a year, month and date in the format YYYY-MM-DD. For example, an employee start date can be stored in the employees table as follows:

```
CREATE TABLE employees
```
(  
    ssn          INTEGER,  
    lastname     VARCHAR(10),  
    firstname    VARCHAR(10),  
    salary       NUMERIC(12,2),  
    title        VARCHAR(25),  
    dno          VARCHAR(3),  
    startdate    DATE  
);  

Time of the day is represented using `TIME(p)` where p is the number of digits in the fractional part of the seconds. It contains hours, minutes and seconds in `HH:MM:SS` format. An exact date and time can be represented using a `TIMESTAMP` type and it contains year, month, day, hour, minute, second and its fraction in the format `YYYY-MM-DD HH:MM:SS`. To represent duration of time, an `INTERVAL` type can be used.

**SECURITY LABEL**

A trusted application may need to store security labels in the database. **Trusted Rubix™** provides the `LABEL` type for this purpose. Every security label has a full name or an alias name. For example, the `employees` table may contain a security label associated with each employee. It can be defined as:

```
CREATE TABLE employees  
(  
    ssn          INTEGER,  
    lastname     VARCHAR(10),  
    firstname    VARCHAR(10),  
    salary       NUMERIC(12,2),  
    title        VARCHAR(25),  
    dno          VARCHAR(3),  
    startdate    DATE,  
    emplabel     LABEL  
);  
```

For more information about security labels, please refer to the **Trusted Rubix™** Security Features User's Guide.
Integrity Constraints

Most database applications will have certain integrity conditions that must hold on the data. For example, the employees table is constrained to inserting integer data into an employee social security number. There can be several other constraints apart from the data type. In the employees table, every employee has a unique social security number. But using the above definition of the employees table, if someone wrongly inserts the same social security number for two different employees, the database allows it. The reason is that no constraint is specified. Each row in the employees table can be uniquely identified if ssn is used as a key column. You can specify ssn as a PRIMARY KEY and then the uniqueness constraint is maintained and also the table is indexed on the column ssn. A table can be created specifying the PRIMARY KEY constraint using the following CREATE TABLE command:

```sql
CREATE TABLE employees
(
    ssn        INTEGER PRIMARY KEY,
    lastname   VARCHAR(10),
    firstname  VARCHAR(10),
    salary     NUMERIC(12,2),
    title      VARCHAR(25),
    dno        VARCHAR(3)
);
```

Other integrity constraints such as UNIQUE, CHECK, NOT NULL and FOREIGN KEY are presented in detail in the section titled INTEGRITY CONSTRAINTS of this tutorial.

ALTER TABLE Command

The structure of a table can be dynamically changed by the addition of one or more columns. However, to add more than one column to an existing base table, you must use multiple ALTER TABLE statements. Residual columns may be freely added. The new columns are attached to the end of the row. Columns are added via the ALTER TABLE ADD command. If new columns are created, their values are initially set to NULL in each record. For example, if the following is the listing of the employees table:

```sql
SELECT * FROM employees;
```

<table>
<thead>
<tr>
<th>ssn</th>
<th>lastname</th>
<th>firstname</th>
<th>salary</th>
<th>title</th>
<th>dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>100254789</td>
<td>Smith</td>
<td>Bruce</td>
<td>28000.00</td>
<td>Accountant</td>
<td>ADM</td>
</tr>
<tr>
<td>151007777</td>
<td>Furtado</td>
<td>Judy</td>
<td>35000.00</td>
<td>Software Engineer</td>
<td>SW</td>
</tr>
<tr>
<td>700340352</td>
<td>Xiu</td>
<td>Belinda</td>
<td>25000.00</td>
<td>Office Manager</td>
<td>ADM</td>
</tr>
<tr>
<td>897005240</td>
<td>Golden</td>
<td>Barbara</td>
<td>60000.00</td>
<td>Scientist</td>
<td>EE</td>
</tr>
<tr>
<td>322333000</td>
<td>Wickham</td>
<td>Scott</td>
<td>45000.00</td>
<td>Software Engineer</td>
<td>EE</td>
</tr>
<tr>
<td>224556000</td>
<td>Brown</td>
<td>Jane</td>
<td>36000.00</td>
<td>Signal Engineer</td>
<td>EE</td>
</tr>
<tr>
<td>224552500</td>
<td>Booker</td>
<td>Tracy</td>
<td>44000.00</td>
<td>Physicist</td>
<td>EE</td>
</tr>
<tr>
<td>151235679</td>
<td>Smith</td>
<td>Sandra</td>
<td>19000.00</td>
<td>Secretary</td>
<td>ADM</td>
</tr>
<tr>
<td>228992500</td>
<td>Walsh</td>
<td>James</td>
<td>55000.00</td>
<td>Software Engineer</td>
<td>SW</td>
</tr>
<tr>
<td>111223333</td>
<td>Gomes</td>
<td>Mary</td>
<td>42000.00</td>
<td>Software Engineer</td>
<td>SW</td>
</tr>
</tbody>
</table>

You can add two new columns to the employees table using the following ALTER TABLE
The NULL values in existing records can be set to meaningful values by means of the UPDATE command. When subsequent records are added via the INSERT command, values may be specified for all columns, including the newly added ones. Integrity constraints on the columns can also be specified as part of ADD phrase.

You can specify that the employee’s department number must be ADM, SW, or EE by the following statement:

```
ALTER TABLE employees
ADD CONSTRAINT CN CHECK
    (dno in ( 'ADM', 'SW', 'EE'))
```

The new employees table shown above:

```
SELECT *
FROM employees;
```
This will ensure that, in the future, all employees added to the table will only be from one of the three departments noted.

**DROP TABLE Command**

A table can be removed from your workspace by using the *DROP TABLE* command which is the logical complement to the *CREATE TABLE* command.

Dropping a table can be problematic both from a database integrity and security perspective. If there are view objects based upon the table, and the table is dropped but not the views, then the views would be left as dangling objects, and the database’s integrity would be violated. A similar situation exists for references to a table, such as FOREIGN KEY or CHECK constraint. From a security perspective, if a user’s sensitivity label equals a given table’s sensitivity label, but is dominated by the sensitivity label of a view based on that table, allowing the user to drop the table would violate the security policy of the database.

To address this problem two keywords are provided with the *DROP TABLE* command, one of which must be chosen. The first keyword is *CASCADE*. *CASCADE* causes views that are based upon the table and FOREIGN KEY and CHECK constraints that reference the table to be dropped when the table is dropped. This is, of course, subject to the database security policy.

*RESTRICT* takes the opposite approach to preserving database integrity. The *DROP TABLE* command fails if any views are currently based upon the table, or FOREIGN KEY or CHECK constraints reference the table when the command is issued.

If the user types:

```
DROP TABLE employees RESTRICT;
```

and any views are currently based upon the table, or FOREIGN KEY or CHECK constraints reference the employee’s table when the command is issued, then the *DROP TABLE* operation fails.

To see the effects when *CASCADE* is used, please see the section titled INTEGRITY CONSTRAINTS.
VIEWS

OVERVIEW

This chapter presents the use of views, which are logical ways of accessing the information from the database.

VIEWS

A view is a logical way of looking at the data stored in a table. A table is a physical entity, stored in a database as a table, and its associated data. A view can be any way of looking at the data stored in existing tables. A view can be a simple rearrangement of data in a table, a selection of rows based on a selection condition, or an intricate composition involving the join of two or more tables.

This section discusses the use of views in Trusted Rubix™. The following topics are discussed:

→ Views vs. relations (tables)
→ Why views are used
→ Updating views

VIEWS VS. RELATIONS (TABLES)

Since looking at an entire table is one way of viewing the table, a table is also considered a view. In general, however, most views differ from tables in the following ways:

→ A view must be defined in terms of already existing tables.
→ Views generally occupy storage only while they are being used. When views are defined and stored, the view expression is stored without any accompanying data. Each time the view is used, the view expression is used as a “blueprint” for constructing the view from its underlying table(s).
→ Since a view derives its data from its underlying table(s), it reflects the content of those tables at the time the view is used.

WHY VIEWS ARE USED

Views provide you with maximum flexibility in data manipulation. These are some of the advantages of using views:

→ Views allow multiple “perspectives” of the same database without increasing storage requirements.
→ Using views, different users can work with different portions of the same table.
→ A view expression can impose restrictions on user or group access. Views can thus be used to implement security and privacy controls.
→ Data can be entered into tables through views. With selection conditions, views ensure greater accuracy in data-entry. Only data that is consistent with the conditions imposed
CREATE VIEW Command

It is possible with **Trusted Rubix**/SQL to define a view of a table. A view of a table may be a subset of columns, or subset of rows, of the table or any other form of retrieval using a combination of operations such as select, project, join and aggregate functions. For example, it may be desirable to create a version of the employees table, say, `emp_dept`, which shows the `lastname`, `firstname` and `dno` but not the `ssn`, `title` or `salary`. In this case, once `employees` has been created, the `emp_dept` view can be created using the SQL command:

```sql
CREATE VIEW emp_dept AS
    SELECT lastname, firstname, dno
    FROM employees;
```

If all the data is now selected from `emp_dept`, the result is the indicated subset of columns of `employees`:

```sql
SELECT *
FROM emp_dept;
```

<table>
<thead>
<tr>
<th>emp_dept.lastname</th>
<th>emp_dept.firstname</th>
<th>emp_dept.dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>Bruce</td>
<td>ADM</td>
</tr>
<tr>
<td>Smith</td>
<td>Sandra</td>
<td>ADM</td>
</tr>
<tr>
<td>Xiu</td>
<td>Belinda</td>
<td>ADM</td>
</tr>
<tr>
<td>Golden</td>
<td>Barbara</td>
<td>EE</td>
</tr>
<tr>
<td>Wickham</td>
<td>Scott</td>
<td>EE</td>
</tr>
<tr>
<td>Brown</td>
<td>Jane</td>
<td>EE</td>
</tr>
<tr>
<td>Booker</td>
<td>Tracy</td>
<td>EE</td>
</tr>
<tr>
<td>Furtado</td>
<td>Judy</td>
<td>SW</td>
</tr>
<tr>
<td>Walsh</td>
<td>James</td>
<td>SW</td>
</tr>
<tr>
<td>Gomes</td>
<td>Mary</td>
<td>SW</td>
</tr>
</tbody>
</table>

A view may alternately be a selection of a table. (In fact, an arbitrary combination of projections, selections and joins are permitted in the general case.) For example, the view `high_salary` can be defined as including those employees whose salary is higher than $55000.00:

```sql
CREATE VIEW high_salary AS
    SELECT *
    FROM employees
    WHERE salary > 55000.00;
```

Selection of all rows and columns from this view produces the following display:

```sql
SELECT *
FROM high_salary;
```

<table>
<thead>
<tr>
<th>ssn</th>
<th>lastname</th>
<th>firstname</th>
<th>salary</th>
<th>title</th>
<th>dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>897005240</td>
<td>Golden</td>
<td>Barbara</td>
<td>60000.00</td>
<td>Scientist</td>
<td>EE</td>
</tr>
</tbody>
</table>

Unlike a table, which corresponds to physically stored records, a view is just a convenient
mechanism which rearranges data stored in the underlying table. No physical data records are associated with the view per se; thus, data cannot be physically added to or deleted from a view, only from the underlying table. However, data can be inserted, deleted or updated through the views into the underlying tables when the view is updatable.

**NOTE**

From this point forward, we will refer to the table identified by that table reference as the (single) underlying table on which the updatable view in question is “immediately” (or “directly”) defined.

### DROP VIEW Command

A view can be removed by using the **DROP VIEW** command. This command works similarly to the **DROP TABLE** command. For example, a view can be deleted using

```
DROP VIEW emp_dept RESTRICT;
```

If the user subsequently attempts to access the `emp_dept` table by typing:

```
SELECT *
FROM  emp_dept;
```

the following error message will appear:

```
SELECT * FROM emp_dept
       named relation is inaccessible or non-existent (RR005)
```

### VIEW UPDATABILITY

In general, view update includes **INSERT**, **DELETE** and **UPDATE** from a view. A view is updatable according to SQL 92 if it satisfies the following eight conditions:

1. The table expression that defines the scope of the view is a select expression;
   that is, it does not directly contain any of the key words JOIN or UNION
2. The **SELECT** clause of that select expression does not directly contain the key word DISTINCT.
3. Every select-item in that **SELECT** clause (after any necessary expansion of “asterisk-style” select-items) consists of a possibly qualified column name (optionally accompanied by an AS clause), representing a simple reference to a column of the underlying table (see example 5 below).
4. The **FROM** clause of that select expression contains exactly one table reference.
5. That table reference identifies either a base table or an updatable view.
6. That select expression does not include a **WHERE** clause that includes a nested table expression that includes a FROM clause that includes a reference to the same table as is referenced in the FROM clause mentioned in condition 4 above.
7. That select expression does not include a **GROUP BY** clause.

---

1e.g., an expression of the form "TABLE table", or a select expression.
8. That select expression does not include a HAVING clause.

The operations INSERT, UPDATE, and DELETE can be applied to a given view only if that view is updatable as defined above. Observe too that updatability is “all or nothing,” in the sense that either all three of INSERT, UPDATE, or DELETE can be applied to a given view or none of them can; it is not possible (e.g.) DELETE to be applicable but INSERT not.\(^2\)

EXAMPLES:

1. Find employees’ names and their department name.

```
CREATE VIEW emp_dname AS
    SELECT lastname, firstname, dname
    FROM employees JOIN departments ON (employees.dno = departments.dno);
```

The user may notice that this view contains the keyword JOIN and thus is not updatable according to condition (1) as described above in the Guide to the SQL 92 Standard. If the user attempts to modify data through the view using

```
INSERT INTO emp_dname VALUES 'Roberts', 'Kathy', 'Administration';
```

```
UPDATE emp_dname SET firstname = 'Alexandra'
WHERE firstname = 'Sandra';
```

```
DELETE FROM emp_dname WHERE dname = 'Electrical';
```

the following respective error messages will appear:

```
insert into emp_dname values 'Roberts', 'Kathy', 'Administration'
view is not updatable (VU003)
```

```
update emp_dname SET firstname = 'Alexandra'
where firstname = 'Sandra'
view is not updatable (VU003)
```

```
delete from emp_dname where dname = 'Electrical'
view is not updatable (VU003)
```

```
ROLLBACK;
```

\(^2\)Definition of SQL 92 conditions are excerpted from Chapter 13, in C.J. Date and Hugh Darwen, *A Guide to the SQL 92 Standard*, 3rd ed. (Addison-Wesley, 1993).
2. Find the tasks IDs. The tasks could be either entries in the **projects** table or entries in the **works** table.

   ```sql
   CREATE VIEW projs AS
   SELECT pno
   FROM projects
   UNION
   SELECT pno
   FROM works;
   ```

   The view is not updatable because it contains the keyword `UNION`, as in condition (1) as described above in the Guide to the SQL 92 Standard. If the user attempts to modify the data by using

   ```sql
   INSERT INTO projs VALUES 'AKK';
   ```

   the following respective error messages will appear:

   ```sql
   insert into projs values 'AKK'
   view is not updatable (VU003)
   ```

   ```sql
   UPDATE projs SET pno = 'AKK'
   WHERE pno = 'PCS';
   ```

   ```sql
   update projs set pno = 'AKK' where pno = 'PCS'
   view is not updatable (VU003)
   ```

   ```sql
   ROLLBACK;
   ```

3. Find the project names and IDs. The user may decide to use the **DISTINCT** keyword (even if not necessary, in this case, because **pno** is the primary key for the **projects** table).

   ```sql
   CREATE VIEW projname AS
   SELECT DISTINCT pno, pname
   FROM projects;
   ```

   The view is not updatable because it contains keyword **DISTINCT** as in condition (2) as described above in the SQL 92 Standard. If the user attempts to modify data using

   ```sql
   INSERT INTO projname VALUES 'IW', 'Intelligent Weapons';
   ```

   the following respective error messages will appear:

   ```sql
   insert into projname values 'iw', 'Intelligent Weapons'
   view is not updatable (VU003)
   ```

   ```sql
   UPDATE projname SET pno = 'TN'
   WHERE pno = 'TMK';
   ```

   ```sql
   update projname set pno = 'TN' where pno = 'TMK'
   view is not updatable (VU003)
   ```

   ```sql
   ROLLBACK;
   ```
update projname set pno = 'tn' where pno = 'tmk'
view is not updatable (VU003)
ROLLBACK;

4. Find the cost of the most expensive project.

```sql
CREATE VIEW proj_expensive AS
SELECT MAX(budget) AS max_budget
FROM projects;
```

If you try to change the budget of the most expensive project, the operation will fail because it contains an aggregate in the SELECT clause as in condition (3) described above in the Guide to the SQL 92 Standard. If the user attempts to modify the data by using

```sql
UPDATE proj_expensive
SET max_budget = 2000000
WHERE max_budget = 600000;
```

the following error message will appear:

```sql
update proj_expensive set max_budget = 2000000
where max_budget = 600000
view is not updatable (VU003)
ROLLBACK;
```

5. Find all employees' names.

```sql
CREATE VIEW emp_names AS
SELECT lastname, firstname
FROM employees;
```

```sql
SELECT *
FROM emp_names;
```

<table>
<thead>
<tr>
<th>emp_names.lastname</th>
<th>emp_names.firstname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>Bruce</td>
</tr>
<tr>
<td>Smith</td>
<td>Sandra</td>
</tr>
<tr>
<td>Xiu</td>
<td>Belinda</td>
</tr>
<tr>
<td>Golden</td>
<td>Barbara</td>
</tr>
<tr>
<td>Wickham</td>
<td>Scott</td>
</tr>
<tr>
<td>Brown</td>
<td>Jane</td>
</tr>
<tr>
<td>Booker</td>
<td>Tracy</td>
</tr>
<tr>
<td>Furtado</td>
<td>Judy</td>
</tr>
<tr>
<td>Walsh</td>
<td>James</td>
</tr>
<tr>
<td>Gomes</td>
<td>Mary</td>
</tr>
</tbody>
</table>

Updating or deleting some names succeeds because the view is updatable (one table, e.g., condition (4) and the SELECT clause refers to columns as in condition (3) of the Guide to the SQL 92 Standard described above).

```sql
UPDATE emp_names
SET firstname = 'Alexandra'
```
WHERE firstname = 'Sandra';

DELETE FROM emp_names WHERE lastname = 'Walsh';

SELECT * FROM emp_names;

<table>
<thead>
<tr>
<th>emp_names.lastname</th>
<th>emp_names.firstname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>Bruce</td>
</tr>
<tr>
<td>Smith</td>
<td>Sandra</td>
</tr>
<tr>
<td>Xiu</td>
<td>Belinda</td>
</tr>
<tr>
<td>Golden</td>
<td>Barbara</td>
</tr>
<tr>
<td>Wickham</td>
<td>Scott</td>
</tr>
<tr>
<td>Brown</td>
<td>Jane</td>
</tr>
<tr>
<td>Booker</td>
<td>Tracy</td>
</tr>
<tr>
<td>Furtado</td>
<td>Judy</td>
</tr>
<tr>
<td>Gomes</td>
<td>Mary</td>
</tr>
</tbody>
</table>

SELECT * FROM employees;

<table>
<thead>
<tr>
<th>employees.ssn</th>
<th>employees.lastname</th>
<th>employees.firstname</th>
<th>employees.salary</th>
<th>employees.title</th>
<th>employees.dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>100254789</td>
<td>Smith</td>
<td>Bruce</td>
<td>28000.00</td>
<td>Accountant</td>
<td>ADM</td>
</tr>
<tr>
<td>151235679</td>
<td>Smith</td>
<td>Alexandra</td>
<td>19000.00</td>
<td>Secretary</td>
<td>ADM</td>
</tr>
<tr>
<td>700340352</td>
<td>Xiu</td>
<td>Belinda</td>
<td>25000.00</td>
<td>Office Manager</td>
<td>ADM</td>
</tr>
<tr>
<td>897005240</td>
<td>Golden</td>
<td>Barbara</td>
<td>60000.00</td>
<td>Scientist</td>
<td>EE</td>
</tr>
<tr>
<td>322333000</td>
<td>Wickham</td>
<td>Scott</td>
<td>45000.00</td>
<td>Software Engineer</td>
<td>EE</td>
</tr>
<tr>
<td>224255600</td>
<td>Brown</td>
<td>Jane</td>
<td>36000.00</td>
<td>Signal Engineer</td>
<td>EE</td>
</tr>
<tr>
<td>224552500</td>
<td>Booker</td>
<td>Tracy</td>
<td>44000.00</td>
<td>Physicist</td>
<td>EE</td>
</tr>
<tr>
<td>151007777</td>
<td>Furtado</td>
<td>Judy</td>
<td>35000.00</td>
<td>Software Engineer</td>
<td>SW</td>
</tr>
<tr>
<td>111223333</td>
<td>Gomes</td>
<td>Mary</td>
<td>42000.00</td>
<td>Software Engineer</td>
<td>SW</td>
</tr>
</tbody>
</table>

List only the last names using the previously defined view.

CREATE VIEW emp_lastnames AS
SELECT lastname FROM emp_names;
SELECT * FROM emp_lastnames;

emp_lastnames.lastname
Smith
Smith
Xiu
Golden
Wickham
Brown
Booker
Furtado
Gomes

Updating and deleting some names succeeds because the view is updatable (references one updatable view, e.g., condition (5) as described above in the Guide to the SQL 92 Standard).

UPDATE emp_lastnames
SET lastname = 'Alexander'
WHERE lastname = 'Gomes';

DELETE FROM emp_lastnames
WHERE lastname = 'Golden';

SELECT * FROM emp_lastnames;

emp_lastnames.lastname
Smith
Smith
Xiu
Wickham
Brown
Booker
Furtado
Alexander

SELECT * FROM emp_names;

emp_names.lastname  emp_names.firstname
Smith  Bruce
Smith  Alexandra
Xiu  Belinda
Wickham  Scott
Brown  Jane
Booker  Tracy
Furtado  Judy
Alexander  Mary
6. Find employees' names and the name of their department by rewriting an earlier view.

   CREATE VIEW emp_deptname AS
   SELECT lastname, firstname, dname
   FROM employees, departments
   WHERE employees.dno = departments.dno;

This view is not updatable because it contains more than one table, e.g., condition (4) as described above in the Guide to the SQL 92 Standard. If a user attempts to modify data by using

   INSERT INTO emp_deptname VALUES 'Roberts', 'Kathy', 'Administration';

   UPDATE emp_deptname
   SET firstname = 'Alexandra'
   WHERE firstname = 'Sandra';

   DELETE FROM emp_deptname
   WHERE dname = 'Electrical';

the following respective error messages will appear:

   insert into emp_deptname values 'Roberts', 'Kathy', 'Administration'
   view is not updatable (VU003)

   update emp_deptname set firstname = 'Alexandra'
   where firstname = 'Sandra'
   view is not updatable (VU003)

   delete from emp_deptname where dname = 'Electrical';
   view is not updatable (VU003)

   ROLLBACK;

7. Find all employees who earn more than $30000.00.
CREATE VIEW emp_sal AS
SELECT lastname, firstname, salary
FROM employees
WHERE salary > 30000.00;

Updating and deleting some employees succeeds because the view is updatable, according to condition (6) as described above in the Guide to the SQL 92 Standard.

Increase the salary of a certain employee.

UPDATE emp_sal
SET salary = 50000
WHERE lastname = 'Gomes';

Decrease the salary of one employee and delete it from the view (but not from the base table employees).

UPDATE emp_sal
SET salary = 20000
WHERE lastname = 'Furtado';

SELECT * FROM emp_sal;

<table>
<thead>
<tr>
<th>emp_sal.lastname</th>
<th>emp_sal.firstname</th>
<th>emp_sal.salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Golden</td>
<td>Barbara</td>
<td>60000.00</td>
</tr>
<tr>
<td>Wickham</td>
<td>Scott</td>
<td>45000.00</td>
</tr>
<tr>
<td>Brown</td>
<td>Jane</td>
<td>36000.00</td>
</tr>
<tr>
<td>Booker</td>
<td>Tracy</td>
<td>44000.00</td>
</tr>
<tr>
<td>Walsh</td>
<td>James</td>
<td>55000.00</td>
</tr>
<tr>
<td>Gomes</td>
<td>Mary</td>
<td>50000.00</td>
</tr>
</tbody>
</table>

SELECT * FROM employees;

<table>
<thead>
<tr>
<th>ssn</th>
<th>employees.lastname</th>
<th>employees.firstname</th>
<th>employees.salary</th>
<th>employees.title</th>
<th>employees.dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>100254789</td>
<td>Smith</td>
<td>Bruce</td>
<td>28000.00</td>
<td>Accountant</td>
<td>ADM</td>
</tr>
<tr>
<td>151235679</td>
<td>Smith</td>
<td>Sandra</td>
<td>19000.00</td>
<td>Secretary</td>
<td>ADM</td>
</tr>
<tr>
<td>700340352</td>
<td>Xiu</td>
<td>Belinda</td>
<td>25000.00</td>
<td>Office Manager</td>
<td>ADM</td>
</tr>
<tr>
<td>897005240</td>
<td>Golden</td>
<td>Barbara</td>
<td>60000.00</td>
<td>Scientist</td>
<td>EE</td>
</tr>
<tr>
<td>322333000</td>
<td>Wickham</td>
<td>Scott</td>
<td>45000.00</td>
<td>Software Engineer</td>
<td>SW</td>
</tr>
<tr>
<td>224255600</td>
<td>Brown</td>
<td>Jane</td>
<td>36000.00</td>
<td>Signal Engineer</td>
<td>EE</td>
</tr>
<tr>
<td>224552500</td>
<td>Booker</td>
<td>Tracy</td>
<td>44000.00</td>
<td>Physicist</td>
<td>EE</td>
</tr>
<tr>
<td>151007777</td>
<td>Furtado</td>
<td>Judy</td>
<td>20000.00</td>
<td>Software Engineer</td>
<td>SW</td>
</tr>
<tr>
<td>111223333</td>
<td>Gomes</td>
<td>Mary</td>
<td>50000.00</td>
<td>Software Engineer</td>
<td>SW</td>
</tr>
</tbody>
</table>
ROLLBACK;

8. Find any employees who have common last names.

    CREATE VIEW emp_samename AS
    SELECT lastname, firstname
    FROM employees
    WHERE lastname IN (SELECT lastname
                        FROM employees);

This view is not updatable because of the nested table expression, condition (6) as described above in the Guide to the SQL 92 Standard. If the user attempts to modify the data using

    UPDATE emp_samename
    SET lastname = 'Smith'
    WHERE lastname = 'Brown';

the following error message will appear:

    update emp_samename set lastname = 'Smith'
    where 'lastname' = 'Brown'
    view is not updatable (VU003)

ROLLBACK;

If a subquery is present, then it can contain any keyword including DISTINCT, JOIN, etc., any number of tables, subqueries, etc., and the view will be updatable as long as condition (6) is satisfied.

9. Find the total of employee salaries by department.

    CREATE VIEW emp_sal_dept AS
    SELECT dno, SUM(salary)
    FROM employees
    GROUP BY dno;

This view is not updatable because it includes a GROUP BY clause, condition (7) as described above in the Guide to the SQL 92 Standard. If a user attempts to modify data using

    DELETE
    FROM emp_sal_dept
    WHERE dno= 'EE';

the following error message will appear:

    delete from emp_sal_dept where dno= 'EE'
    view is not updatable (VU003)

ROLLBACK;
10. Find the total of employee salaries by department with `dno` beginning with 'E'.

```sql
CREATE VIEW emp_sal_deptE AS
SELECT  dno, SUM(salary)
FROM    employees
GROUP BY dno
HAVING  dno LIKE 'E%';
```

This view is not updatable according to condition (7) (includes a GROUP BY clause) and condition (8) (includes a HAVING clause) as described above in the Guide to the SQL 92 Standard. If the user attempts to modify the data using

```sql
DELETE
FROM    emp_sal_deptE
WHERE    dno = 'EE';
```

the following error message appears:

```sql
DELETE from emp_sal_deptE where dno = 'EE'
  view is not updatable and hence the user does not have the
DELETE privilege (RA003).
```

ROLLBACK;

WITH [LOCAL | CASCADE] CHECK OPTION

A view may be created using WITH [LOCAL | CASCADE] CHECK OPTION to ensure that the updates performed through the view are checked for violation of the view defining condition. If the conditions are violated then the operation is rejected.

For example, let us operate upon a restricted subset of the items in the `employees` table:

```sql
CREATE VIEW EE_employees AS
SELECT * 
FROM    employees
WHERE    dno = 'EE'
WITH LOCAL CHECK OPTION;
```

Deleting all the records in the `EE_employees` view deletes only those records that exist as part of the view.

```sql
DELETE
FROM    EE_employees;
```

You can observe that rows are deleted from the `employees` table consistent with the deletion performed on the view:

```sql
SELECT *
FROM    employees;
```
and there are no more rows left in EE_employees view since all rows were deleted:

```
SELECT * 
FROM EE_employees;
```

Since the EE_employees table was created using the WITH LOCAL CHECK OPTION, it is impossible to insert any items into EE_employees for which the dno is not 'EE':

```
INSERT INTO EE_employees
VALUES 333333333, 'Sawyer', 'Deborah', 43000.00, 'Software Engineer', 'SW';
```

whereas inserting rows having the dno column equal to 'EE' is possible:

```
INSERT INTO EE_employees
VALUES 333333333, 'Sawyer', 'Deborah', 43000.00, 'Software Engineer', 'EE';
```

Note that any preexisting rows in employees for which the dno is not 'EE' still reside within the table but are not visible via the EE_employees view:

```
SELECT * 
FROM EE_employees;
```

Those rows are made visible by accessing the base table employees:

```
SELECT * 
FROM employees;
```
<table>
<thead>
<tr>
<th>ssn</th>
<th>firstname</th>
<th>salary</th>
<th>title</th>
<th>dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>100254789</td>
<td>Bruce</td>
<td>28000.00</td>
<td>Accountant</td>
<td>ADM</td>
</tr>
<tr>
<td>151235679</td>
<td>Sandra</td>
<td>19000.00</td>
<td>Secretary</td>
<td>ADM</td>
</tr>
<tr>
<td>700340352</td>
<td>Belinda</td>
<td>25000.00</td>
<td>Office Manager</td>
<td>ADM</td>
</tr>
<tr>
<td>151007777</td>
<td>Judy</td>
<td>35000.00</td>
<td>Software Engineer</td>
<td>SW</td>
</tr>
<tr>
<td>228992500</td>
<td>James</td>
<td>55000.00</td>
<td>Software Engineer</td>
<td>SW</td>
</tr>
<tr>
<td>111223333</td>
<td>Mary</td>
<td>42000.00</td>
<td>Software Engineer</td>
<td>SW</td>
</tr>
<tr>
<td>333333333</td>
<td>Deborah</td>
<td>43000.00</td>
<td>Software Engineer</td>
<td>EE</td>
</tr>
</tbody>
</table>

**Trusted Rubix** does not allow a `VIEW BY` clause or any hidden columns in the updatable view. The `ROWLABEL` column is an exception and it can be inserted by a database administrator. However, it cannot be updated.

TR views are not restricted to the simple illustrated queries. Any valid TR query can be used to create a view (e.g., cross join, natural join, union).
DATA SECURITY

DISCRETIONARY ACCESS CONTROL: GRANT and REVOKE

Every table which a user creates is implicitly granted all privileges for that table. This case is true until he grants other users the privilege to access the table. Trusted Rubix™ SQL supports seven types of privileges on tables:

- SELECT,  UPDATE,  CRVIEW,  DELETE.
- REFERENCES,  INSERT,  REFVIEW

The first four privileges can be granted on a column-by-column basis, or for an entire table, whereas the others apply to the entire table and can be granted only for the entire table.

Granting privileges to other users is performed by the GRANT command:

```
GRANT SELECT, UPDATE ON employees TO bob, mary;
```

When a user is given a privilege, he is not able to pass that privilege on to other users: that authority rests solely with the creator of the table and the database administrator. However, he can give the authority to pass on privileges to other users. Implicit in the creation of a table is the assignment of grant options for each privilege. A grant option is the ability to further assign the specified privilege to another user. It is invoked by using the WITH GRANT OPTION:

```sql
GRANT SELECT, UPDATE ON employees TO bob, mary
WITH GRANT OPTION;
```

This SQL segment performs the same function as the previous SQL segment, except the users “bob” and “mary” are now allowed to grant SELECT and UPDATE privileges on employees to other users.

The appearance of the keywords SELECT and UPDATE without a column list caused SELECT, UPDATE privileges on the entire table to be passed to the appropriate users. Granting the SELECT and UPDATE privileges to “mary” and “bob” means that they can access and update all the information from the employees table. For example they can execute the following query:

```sql
SELECT * FROM employees;
```
Once the privileges for the entire table are passed, the column privileges are overridden. For example, you may add a column to the employees table by using the `ALTER TABLE` command. But “mary” and “bob” can still access and update the new column, since they have `SELECT` and `UPDATE` privileges on the entire table.

Alternatively, users may be given privileges on a subset of the columns via a command such as:

```
GRANT SELECT (lastname, firstname), UPDATE (dno)
ON employees
TO john;
```

Now, “john” cannot access all the information in the employees table, so he will not be able to run all the queries “mary” and “bob” can run. For example, “john” can run the following queries successfully:

```
SELECT lastname, firstname
FROM employees;

UPDATE employees SET dno = 'EE'
WHERE lastname = 'Smith'
AND firstname = 'Sandra';
```

However, he is unable to run the following query because of the lack of privileges on all columns:

```
SELECT *
FROM employees;
```

Similar rules apply to `INSERT` and `UPDATE` privileges on columns. However, `UPDATE` statements using columns on the right hand side of the set clause need `SELECT` privilege in addition to `UPDATE` privilege. For example, you can grant “john” `UPDATE` privileges on the `salary` column of the employees table.

```
GRANT UPDATE (salary)
ON employees
TO john;
```

But, “john” cannot increase the `salary` column based on its current value since he doesn’t have the `SELECT` privilege on the `title` column. Therefore, he cannot run the following query,

```
UPDATE employees SET salary = salary * 1.1
WHERE title LIKE '%Engineer';
```
He will be able to perform the above **UPDATE** statement, if you grant him the **SELECT** privilege on the **title** column.

The keyword **PUBLIC** can appear in place of the named recipients to grant the specified privileges to all users known to the system.

```
GRANT SELECT (lastname, firstname, dno)
ON employees
TO PUBLIC;
```

The keyword **GROUP** followed by a group name can also be used in place of recipients to grant privileges to groups of users. The group is defined as a UNIX/Linux group to which a user belongs.

```
GRANT SELECT (lastname, firstname, dno), UPDATE (dno)
ON employees
TO GROUP admin;
```

The inverse of the **GRANT** command is the **REVOKE** command.

A privilege can only be revoked by a user who has the **WITH GRANT OPTION** for the privilege on the object or the database administrator.

**Trusted Rubix** does not permit a user to be granted the same privilege from two sources. This prevents situations in which you revoke a privilege from a user (“mary”) only to discover that the privilege is still held by that user (“mary”). In the situation when a user is granted a privilege from two sources, an error is generated by the second **GRANT** command:

```
(you) GRANT SELECT, UPDATE
ON employees
TO bob, mary
WITH GRANT OPTION;
```

This command executes without error.

```
(mary)GRANT SELECT, UPDATE
ON employees
TO harry;
```

This command executes without error.

```
(you) GRANT SELECT, UPDATE
ON employees
TO harry;
```

This command will not be executed. An error is returned indicating the privileges **SELECT** and **UPDATE** are already held by the user “harry”.

At this point, “harry” has the **SELECT** and **UPDATE** privileges from only one source, “mary”. If “you” then execute:

```
REVOKE SELECT, UPDATE
ON employees
FROM harry CASCADE;
```

an error will be returned.
Since “mary” granted “harry” the SELECT and UPDATE DAC privileges, “harry” will still be able to SELECT and UPDATE from employees. The exception is with the default Database Administrator role, which can revoke any DAC privilege. On the other hand, if “you” had executed:

```
REVOKE SELECT, UPDATE
ON employees
FROM mary CASCADE;
```

then both “mary” and “harry” would lose their SELECT and UPDATE privileges on employees.

Granting the SELECT and UPDATE privileges on a table (thereby granting those privileges on every column of the table) and then revoking those privileges does not inhibit the granting of those privileges to any new columns later added to the table.

However, this analogy does not apply to granting privileges to PUBLIC. While the failure to specify a column-list in a GRANT (or REVOKE) statement is, in one way, the same as specifying a column-list that contains every column of the table, a grant to PUBLIC is not the same as granting individually to every user and every user that will ever be known to the system. The difference is that sequences like:

```
GRANT SELECT, UPDATE
ON employees
TO PUBLIC;
```

```
REVOKE SELECT, UPDATE
ON employees
FROM mary CASCADE;
```

will get an error (unless, of course, you have also explicitly granted SELECT and UPDATE on employees to “mary” in another statement). That’s because “mary” does not have the SELECT and UPDATE privileges PUBLIC does. “mary” is a member of PUBLIC, of course, and therefore can issue SELECT and UPDATE statements that select from the employees table, but “mary” does not have the privilege herself, and you cannot therefore take it away without taking it away from PUBLIC.

**TRUSTED RUBIX™** SQL provides a view mechanism that allows for imaginative combinations of privileges and data access. It may be desired to hide data from a user or allow data to be seen only through the view. It is possible to grant privileges on a view that are “passed on” to the base table. In fact, that encompasses much of the usefulness of views. They allow the creator of a view to give access to a base table without giving away access to the table itself. To create this view the creator must have the CRVIEW privilege on the base table itself.

```
CREATE VIEW ee_employees AS
SELECT lastname, firstname, title
FROM employees
WHERE dno = 'EE';
```

To use the ee_employees view effectively and securely, you can restrict access to the base table and grant SELECT privilege to the view.

```
GRANT SELECT
ON ee_employees
TO tom;
```
To allow the user ("tom") to continue using this view, the user ("tom") must have the REFVIEW privilege on the base table.

```
GRANT REFVIEW
ON employees
TO tom;
```

The REFVIEW privilege is useful because you may not want to give up control over your entire base table. The REFVIEW privilege can be revoked at any time in the future, thus causing further accesses using this special view mechanism to fail.

```
REVOKE ALL PRIVILEGES
ON employees
FROM tom
CASCADE;
```

In this circumstance, where “tom” cannot grant his DAC privileges on employees, the following command is equally effective:

```
REVOKE ALL PRIVILEGES
ON employees
FROM tom
RESTRICT;
```

The views described above may involve any normal SQL syntax. Mathematical operations (salary * 2), projections (lastname, firstname), joins, etc., are all treated similarly.

### MANDATORY ACCESS CONTROL

Mandatory access control, often denoted as MAC, is a mechanism whereby users can be prevented from accessing data for which they have no authorization or need-to-know. Information is classified into a hierarchical security level (i.e., UNCLASSIFIED, CONFIDENTIAL, SECRET and TOP SECRET) based on its sensitivity. Users are cleared up to a certain security level, i.e., they have authorization to know the information at that security level and below. A user's session label is established during login authentication. TRusted RUBIX™ uses the user's session label to access information from its database. For detailed information on MAC concepts and TRusted MAC enforcement, please refer to the Security Features User's Guide.

**NOTE**

In addition to the Multilevel Security (MLS) Mandatory Access Control (MAC) policy, TRusted RUBIX supports the Type Enforcement (TE) MAC policy of SELinux and a proprietary Attribute Based Access Control (ABAC) MAC policy of the Security Policy Manager (SPM). In general, all configured MAC policies must permit an operation for it to succeed. For more information on TE and SELinux please see the TRusted RUBIX SELinux Guide and for more information on ABAC and the SPM please see the TRusted RUBIX Security Policy Manager Reference Guide and Tutorial.

### POLYINSTANTIATION

Any leakage of information to users at lower security levels is called a covert channel. A covert channel can be created when a lower level user tries to insert a row with the same key as a row
which already exists at a higher security level; insertion fails. This reflects the fact that the row already exists and has conveyed some information to a lower level user, even though he cannot access that information. To prevent this type of covert channel, a phenomenon called polyinstantiation can be used. A polyinstantiated table can be created which would allow the above insertion, though the key is duplicated. For more details on polyinstantiation phenomenon, see the *Security Features User's Guide*. 
INTEGRITY CONSTRAINTS

INTEGRITY CONSTRAINTS

Every table may have some constraints specified at the time it is defined to maintain its data integrity. The basic inherent constraint on a table is the data type of each column. From the relational data model, you can identify additional constraints needed for an application.

NOTE

In the following examples, the CREATE TABLE statements utilize the same names as the tables created in tutorial database. Since table names cannot be duplicated, the following CREATE TABLE statements are for illustration purposes only.

PRIMARY KEY AND UNIQUE CONSTRAINTS

Typically each table has a set of columns which uniquely identifies each instance of the table i.e., a row. These columns can be specified as a PRIMARY KEY. There can be any number of columns in the PRIMARY KEY, but each table can have only one PRIMARY KEY. In other words, if there is more than one column in the PRIMARY KEY, it is the combination of these columns which uniquely identifies the record. For example, the works table can be defined using the following CREATE TABLE command:

```sql
CREATE TABLE works
(
    PRIMARY KEY (pno, ssn, from_date),
    pno VARCHAR(3),
    ssn INTEGER,
    from_date DATE,
    to_date DATE,
    hours INTEGER
);
```

The behavior of the PRIMARY KEY is also dependent on polyinstantiation for multilevel tables. Details on the polyinstantiation phenomenon are in the Security Features User's Guide.

The combination of the columns pno, ssn and from_date uniquely identify a row in the works table. You can insert the following row:

```sql
INSERT INTO works
VALUES 'MGS', 550221133, 2007-03-01, 2007-03-15, 50;
```

But you cannot insert the row with the same primary key values again:

```sql
INSERT INTO works
VALUES 'MGS', 550221133, 2007-03-01, 2007-03-31, 80;
```

It is frequently desired to assert a unique value relationship over one or more columns that are not necessarily part of the key. For example, it seems reasonable that, since department numbers must not conflict, department names must not conflict either. In order to indicate that the department name column must contain a unique value for each row, a command such as the following could
be used:

```sql
CREATE TABLE departments
(
    PRIMARY KEY (dno),
    dno CHAR(3),
    dname CHAR(15) UNIQUE
);
```

An insertion of the following record is rejected because the department with name 'Software' already exists even though `dno` with a value of 'CS' doesn't exist.

```sql
INSERT INTO departments VALUES 'CS', 'Software';
```

As was the case with the PRIMARY KEY specification, individual columns can be declared UNIQUE on the same line where they are defined, but a UNIQUE constraint governing a group of columns must be declared as a unit. For example, if both `lastname` and `firstname` are to be unique for each row in the `employees` table, the following statement must be used:

```sql
CREATE TABLE employees
(
    PRIMARY KEY (ssn),
    UNIQUE (lastname, firstname),
    ssn INTEGER,
    lastname VARCHAR(10),
    firstname VARCHAR(10)
);
```

Note carefully the distinction between two individual columns being UNIQUE and the combination of two or more columns being collectively UNIQUE. In the first case, the value under each column must be unique per row. In the second case, while the value under either column may be duplicated from one row to the next, the set of columnar values must be unique for each row. For example, the CREATE TABLE command shown above would allow either the value of last name or the first name of an employee to be duplicated across rows, but would disallow two rows from sharing both the same last name and the same first name.

**NOTE**

Although there can be only one primary key per table, there can be multiple unique keys.

UNIQUE constraint behavior is affected by multi-level operations. A UNIQUE constraint in Trusted Rubix™ is enforced to be UNIQUE within a sensitivity level. See the Security Feature’s User’s Guide.

**NOT NULL CONSTRAINT**

Some columns may not have values inserted into them when a row is inserted. Their data area may remain empty. The NOT NULL specification ensures that every row in the table must contain a non-empty value for that column. Thus, data values entered into the database conform to the data integrity assertions you have specified. If NOT NULL constraints are not specified for any column in the `employees` table, the `salary` column may remain empty causing integrity problems. The NOT NULL constraint on the `employees` table can be specified as:

```sql
```
CREATE TABLE employees
(
    PRIMARY KEY (ssn),
    ssn INTEGER,
    salary NUMERIC(7,2) NOT NULL
);

A primary key is functionally equivalent to a unique key plus NOT NULL constraints on the key columns.

Key columns by default, do not allow NULL values, therefore it is not necessary to specify NOT NULL for key columns.

DEFAULT VALUES

Instead of using a NOT NULL constraint on a column, a default value can be used for a column, which ensures that this value is inserted whenever the column is empty. Every column can be given a default value during the CREATE TABLE command. This value is useful when some columns being inserted into the table are NULL. There are situations where the columns should not contain a null value. For example, salary column in the employees table cannot have a null value. You can define a minimum salary as the default value to prevent such a situation, as:

CREATE TABLE employees
(
    PRIMARY KEY (ssn),
    ssn INTEGER,
    salary NUMERIC(7,2) DEFAULT 10000.00
);

To specify a default value on PRIMARY KEY columns or UNIQUE columns may not be very useful because the default value is unique only once and the rest are rejected. However, it may be of some use if a subset of the PRIMARY KEY or UNIQUE columns are inserted.

CHECK CONSTRAINT

The CREATE TABLE statement can be extended to include any number of CHECK clauses. The purpose of such a clause is to specify a condition or integrity constraint that must be satisfied by all rows in the table. For example:

CREATE TABLE billing_rate
(
    rate NUMERIC(7,2)
    CHECK (rate >= 25.00)
);

The CHECK clause associated with rate states that the rate charged for any employee must be greater than or equal to $25.00. Any attempt to insert a row that violates either a PRIMARY KEY, a UNIQUE constraint, or the CHECK clause is rejected. Hidden columns (e.g., ROW LABEL) can be used in CHECK constraints.
FOREIGN KEY

A foreign key is a column (or combination of columns) in one table whose values are required to match values of columns in some other table. For example, in the tutorial database, the column `ssn` of the table `employees` is a foreign key matching the column `ssn` of the table `works`; every employee social security value appearing in the column `works.ssn` must also appear in the column `employees.ssn` to maintain database consistency. In a sense, a child (`works.ssn`) can only exist if it is tied to a parent (`employees.ssn`). This type of constraint is referred to as a referential integrity constraint. Referential integrity involves ensuring that every foreign key value does in fact match a value of the corresponding primary key. A composite foreign key must reference a composite primary or unique key with the same number of columns and data types. A unique or primary key for the foreign table being referenced must exist prior to creating the foreign key constraints.

In order to maintain referential integrity with respect to `employees` and `works`, RUBIX/SQL must reject any operation that would violate it. For example, suppose the constraint existed which stated that an employee could work on the project only if he/she is on the payroll. The SQL formulation of this constraint is as follows:

```
CREATE TABLE works

(pno, ssn, from_date),
FOREIGN KEY (ssn) REFERENCES employees(ssn),
  pno VARCHAR(3),
  ssn INTEGER,
  from_date DATE,
  to_date DATE,
  hours INTEGER
);
```

The operations that can violate the constraint are an insertion into or update of `works` (the child table), or a deletion from or update of `employees` (the parent table). In the first case, the operation is valid if, and only if, the corresponding key `ssn` occurs in `employees`.

If the contents of `employees` are:

<table>
<thead>
<tr>
<th>ssn</th>
<th>lastname</th>
<th>firstname</th>
<th>salary</th>
<th>title</th>
<th>dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>100254789</td>
<td>Smith</td>
<td>Bruce</td>
<td>28000.00</td>
<td>Accountant</td>
<td>ADM</td>
</tr>
<tr>
<td>151235679</td>
<td>Smith</td>
<td>Sandra</td>
<td>19000.00</td>
<td>Secretary</td>
<td>ADM</td>
</tr>
<tr>
<td>700340352</td>
<td>Xiu</td>
<td>Belinda</td>
<td>25000.00</td>
<td>Office Manager</td>
<td>ADM</td>
</tr>
<tr>
<td>897005240</td>
<td>Golden</td>
<td>Barbara</td>
<td>60000.00</td>
<td>Scientist</td>
<td>EE</td>
</tr>
<tr>
<td>322333000</td>
<td>Wickham</td>
<td>Scott</td>
<td>45000.00</td>
<td>Software Engineer</td>
<td>EE</td>
</tr>
<tr>
<td>224255600</td>
<td>Brown</td>
<td>Jane</td>
<td>36000.00</td>
<td>Signal Engineer</td>
<td>EE</td>
</tr>
<tr>
<td>224552500</td>
<td>Booker</td>
<td>Tracy</td>
<td>44000.00</td>
<td>Physicist</td>
<td>EE</td>
</tr>
<tr>
<td>151007777</td>
<td>Furtado</td>
<td>Judy</td>
<td>35000.00</td>
<td>Software Engineer</td>
<td>SW</td>
</tr>
<tr>
<td>228992500</td>
<td>Walsh</td>
<td>James</td>
<td>55000.00</td>
<td>Software Engineer</td>
<td>SW</td>
</tr>
<tr>
<td>111223333</td>
<td>Gomes</td>
<td>Mary</td>
<td>42000.00</td>
<td>Software Engineer</td>
<td>SW</td>
</tr>
</tbody>
</table>

the following insertion into the `works` table succeeds:
INSERT INTO works
VALUES 'MGS', 100254789, 2007-04-01, 2007-05-01, 1;

whereas the following insertion fails because the ssn of this record doesn't exist in the employees table.

INSERT INTO works
VALUES 'MGS', 222222222, 2007-04-01, 2007-05-01, 1;

If a row is deleted from the employees table, the corresponding rows in the works table should be deleted to preserve integrity. The actions taken to preserve integrity are called referential action or trigger and can be specified during table definition. The intent of a trigger is to preserve integrity by specifying an action associated with an update or delete operation in the parent table.

The CREATE TABLE command used to define the works table above does not specify the trigger. In this case, the default is NO ACTION i.e., deleting the records from the employees table fails if there is a corresponding row in the works table.

* NO ACTION Deletion of parent row is disallowed if there is a corresponding child row.
* CASCADE Deletion of parent record triggers the deletion of the child record. Note that this operation does not trigger update of a child record on an update operation.
* SET NULL The column in the child record is set to NULL if the parent record is deleted or updated.
* SET DEFAULT The column in the child record is set to DEFAULT if the parent record is deleted or updated.

NOTE
Referential integrity is valid only over the same security MAC level.

TRUSTED RUBIX REFERENTIAL ACTIONS

The following matrix describes how Trusted Rubix™ will handle the specified referential actions:

<table>
<thead>
<tr>
<th></th>
<th>NO ACTION</th>
<th>CASCADE</th>
<th>SET NULL</th>
<th>SET DEFAULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE</td>
<td>Disallow</td>
<td>Delete</td>
<td>Set Null</td>
<td>Set Default</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Referencing</td>
<td>Column</td>
<td>Referencing Column</td>
</tr>
<tr>
<td>UPDATE</td>
<td>Disallow</td>
<td>N/A</td>
<td>Set Null</td>
<td>Set Default</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Referencing</td>
<td>Column</td>
<td>Referencing Column</td>
</tr>
</tbody>
</table>

An example of a referential integrity relationship which will cause records from all child tables to be deleted when the parent table records are deleted follows:
CREATE TABLE works
(
    PRIMARY KEY (pno, ssn, from_date),
    FOREIGN KEY (ssn) REFERENCES employees(ssn)
    ON DELETE CASCADE,
    .
    .
);

Now, if you delete a row in the employees table, the corresponding rows in the works table are deleted. For example, if you delete an employee with social security number 111223333:

```
DELETE
FROM employees
WHERE ssn = 111223333;
```

you can see that the row associated with that ssn is no longer part of the employees table.

```
SELECT *
FROM employees;
```

You can observe that the works table does not contain the record(s) any longer:

```
SELECT *
FROM works;
```
The application must specify the keywords CASCADE or RESTRICT to modify the drop operation. If RESTRICT is specified, no views or constraints may currently be referencing the table to be dropped. If CASCADE is specified, all views based on the table, or constraints referencing the table, will also be dropped. It can be specified as follows:

```
DROP TABLE employees CASCADE;
```

If a user attempts to access employees or any other table or view that referenced the employees table, the appropriate error message will appear. For example, if a user types the following:

```
SELECT * FROM employees;
```

the error message is:

```
select * from employees
named relation is inaccessible or non-existent (RR005).
```

If a user attempts to access the high_salary view which references the employees table by typing:

```
SELECT * FROM high_salary;
```

the error message is:

```
select * from high_salary
named relation is inaccessible or non-existent (RR005).
```
TRANSACTION MANAGEMENT/CONCURRENCY CONTROL

TRANSACTION MANAGEMENT

TRANSACTIONS AND TIMESTAMPS

Transactions are conceptual objects that provide the basis for accessing all Trusted Rubix™ objects. The transaction is identified by a timestamp (transaction-id) which contains the starting time for the transaction. A transaction must be allocated and assigned to a process before that process can perform any work under TR (this includes reading data from the RDBMS). All work performed by the process is tagged with a transaction-id so that you can identify what and when work was performed. Each transaction terminates by being either committed, which makes the modifications to the relation permanent, or omitted (aborted), which nullifies the changes.

TR uses the transaction-id to stamp records and tables within the database. A record has three timestamps associated with it to control whether or not the record is visible to a particular process and whether an update is allowed. These timestamps are part of the underlying storage structure of the record, although they are not part of its logical structure.

One might ask how TR handles modifications to a record. The answer is that it has no need for such a mechanism. No modifications can be made to a record in the database. Instead, TR copies the old version of the record and makes the changes in the copied data. The old (original) record is then marked as deleted.

ACCESS MODES

Access modes provide the mechanism for a process to indicate to TR the type of access it intends to utilize and what effects it will have on other processes (and vice-versa).

The transaction access modes are specified when the transaction is started. There are two transaction access modes; READ ONLY and READ WRITE.

READ ONLY

READ ONLY access mode (as the name indicates) provides for READ ONLY access to TR. A transaction is marked as READ ONLY by the following command:

```
SET TRANSACTION READ ONLY;
```

Once set, it cannot be used to perform any updates. When a READ ONLY transaction accesses a record, the record's timestamp is not updated because it is impossible for a READ ONLY transaction to conflict with another transaction.

This mode is used for obtaining consistent views of the database without having to worry about conflicting with currently executing processes. READ ONLY mode views the database as it existed as of the latest consistent moment (LCM). The LCM is a moment that is guaranteed to have a consistent (non-changing) view of the database because there are no active transactions with a transaction id that is prior to the LCM. READ ONLY access provides significant performance gains and should be used whenever practical.
**READ WRITE**

READ WRITE access mode is the default access mode and provides for update access to the database. READ WRITE access mode can conflict with other update access mode processes and therefore may be aborted by the system at the time of the conflict.

**CONCURRENT TRANSACTIONS**

In a single-user system, a user might operate upon a relation by seizing it exclusively, manipulating its contents, and releasing it. The user would be required to complete all of these steps before a second user could be permitted to access the relation. In a multi-user system, however, it is often the case that several users may wish to read from and write into the same relation at the same time. Mechanisms must be provided, then, to ensure that user programs which access the database simultaneously do not accidentally corrupt one another's results. These mechanisms are referred to as concurrency control mechanisms.

**Trusted Rubix™** provides a mechanism whereby multiple processes can concurrently access and update the same relation at the same time. The outcome of these concurrent actions must be the same as if they were executed one at a time in order of their timestamps. This is referred to as **serializability**. In addition, if a transaction is reading a portion of the relation, it must be able to ensure that the same reads return the same data (i.e., the data is “consistent” and thus does not change during the life of the transaction). These requirements cause significant overhead and adversely impact database performance. **TR** has been designed to provide user control over which consistency related problems they will tolerate in order to improve performance.

Consistency problems can be caused by the following phenomena:

→ **Dirty Read**
  A dirty read is a read operation by a transaction on a record that has been added or deleted by another transaction which has not yet committed its update. If the latter transaction were to omit, the former would be processing data that does not really exist in the database.

→ **Non-repeatable Read**
  A non-repeatable read would occur if a transaction reads the data from a record that is subsequently updated by another transaction.

→ **Phantom Rows**
  A phantom-row would occur if a transaction was allowed to add a record to a relation after a different transaction with a later timestamp had read the relation.

**Consistency Level**

A process can use the **consistency level** to control which phenomena it will allow to occur. As the consistency level gets more restrictive, the performance penalty increases. This should persuade processes to allow the phenomena to occur wherever it does not present problems with the operations of the process.

The consistency level of a transaction defines which of the three phenomena are allowed to occur while processing. The following table shows the four consistency levels and defines which phenomena are allowed to occur at each level.
The consistency level can only be set when a transaction is started. This can be done with the `SET TRANSACTION` command, see the Trusted Rubix™ SQL Reference Guide for details. The default consistency level is set as `SERIALIZABLE`. TR uses the record in and out stamps in conjunction with the transaction table to make a determination as to whether or not the insertion or deletion is uncommitted.

**Concurrency Related Commands**

TR SQL includes several commands which provide the full control over allocating and completing transactions. `SET TRANSACTION` command is used to start a new transaction at a specified mode. If work is attempted prior to running `SET TRANSACTION`, TR will automatically start an update transaction for the work (as if the user had entered `SET TRANSACTION READ WRITE, ISOLATION LEVEL SERIALIZABLE`).

`SET TRANSACTION` can be used with several options. Transaction mode can be either `READ ONLY` or `READ WRITE`. The consistency level can be specified with `ISOLATION LEVEL` keywords with any one of the following options: `SERIALIZABLE`, `REPEATABLE READ`, `READ COMMITTED`, `READ UNCOMMITTED`. The behavior at each consistency level is explained in the previous section.

A **moment** can be specified with the `MOMENT` keyword in the `SET TRANSACTION` command. The moment of a transaction is a timestamp that determines what state of the database it will see. This option is used when a user wants a view of the database as it existed at some time in the past.

If no transaction is currently active and a transaction-initiating SQL statement is executed, TR automatically starts a transaction.

The following SQL statements are transaction-initiating statements; i.e., if there is no current transaction, and a statement of this class is executed, a transaction is initiated:

→ **SQL Data Definition Language Statements**
  - `CREATE/DROP SCHEMA` statement
  - `ALTER TABLE` statement
  - `CREATE/DROP INDEX` statement
  - `CREATE/DROP TABLE` statement
  - `CREATE/DROP VIEW` statement
  - `GRANT/REVOKE` statement

→ **SQL Data Manipulation Language Statements**
  - `OPEN` statement
  - `CLOSE` statement
  - `FETCH` statement
  - `SELECT` statement: single row
↑ INSERT statement
↑ DELETE statement: searched
↑ DELETE statement: positioned
↑ UPDATE statement: searched
↑ UPDATE statement: positioned

→ SET TRANSACTION statement³

The following SQL statement types are not transaction initiating statements:

→ COMMIT and ROLLBACK statements
→ all SQL CONNECTION statements
→ all SQL SESSION statements
→ all SQL DIAGNOSTICS statements; and
→ the DECLARE CURSOR statement.

**NOTE**

The Open Group specification requires that SQL Data Definition Language (DDL) Statements automatically initiate transactions. Some DBMS vendors provide facilities to automatically commit any SQL DDL statement. **Trusted Rubix™** requires an explicit COMMIT statement to complete the transaction.

The COMMIT/ROLLBACK commands are used to dispose of the current transaction by committing or omitting it, respectively. If there is not a transaction in progress, the command will fail with an error message. For details, see the SQL Reference Guide.

**USE OF THE CONCURRENCY COMMANDS**

The **TR** concurrency commands are straightforward to use. When you want to set the characteristics of the next transaction, you run SET TRANSACTION. Transactions are initiated when the user executes a transaction initiating SQL Statement. When the transaction is complete, you call ROLLBACK or COMMIT to omit or commit the changes, respectively.

**SAVEPOINTS**

Savepoints are used within an application program to establish points at which the execution flow could be restarted if problems occur. When a particular savepoint within a transaction is reached, the completion of the actions up to that point creates a new atomic action, which then carries on with the processing. Only the final atomic action within a sequence can initiate a COMMIT WORK of the transaction; that COMMIT WORK is then passed back to all of the previous atomic actions until the transaction is committed. Note, each savepoint does not initiate an irrevocable COMMIT WORK of the work done up to that point within the transaction.

Rollbacks may be initiated from any atomic action (i.e., savepoint), not just the last one. At any

---

¹ This is in compliance with the ANSI ISO SQL specification which requires this statement to initiate a transaction. The Open Group specification treats this statement as non-transaction initiating.
given point in time, only the most recently created savepoint can initiate a rollback of the entire transaction or part of a transaction (i.e., to a previous savepoint).

Savepoint names must be distinct within a given transaction. If a second savepoint has been created with the same identifier as an earlier savepoint, the earlier savepoint is erased. After a savepoint has been created, SQL processing may continue, be committed, rolled back to a prior savepoint or the transaction can be rolled back in its entirety.

Savepoints are declared with the `SAVEPOINT` command and released with the `RELEASE SAVEPOINT` command. Database actions, since the establishment of a savepoint, are rolled back with the `ROLLBACK TO SAVEPOINT` command. The syntax of these SQL commands appears in the SQL Reference Guide.

The use of the savepoint mechanism is illustrated in the example below using the same schema definition that was used in earlier examples. In this example, we wish to update the salaries of two employees, Gomes and Golden, and use the `SAVEPOINT` command to create intermediate points in the transaction, from which we can continue processing in the event of errors.

First, update the salary field of the `employees` table as follows:

```
UPDATE employees
SET salary = 44000
WHERE employees.lastname = 'Gomes';
```

next, declare a savepoint so we can come back to this point if needed:

```
SAVEPOINT gomes_sal;
```

then, update the next record of the `employees` table:

```
UPDATE employees
SET salary = 62000
WHERE employees.lastname = 'Golden';
```

Declare another savepoint:

```
SAVEPOINT golden_sal;
```

If it is discovered that the last update was an error, rollback the transaction to the previously declared savepoint (`gomes_sal`) using the following command:

```
ROLLBACK TO SAVEPOINT gomes_sal;
```

and continue processing. Then, update this record with the corrected salary:

```
UPDATE employees
SET salary = 63000
WHERE employees.lastname = 'Golden';
```

Commit all the updates:

```
COMMIT;
```
MULTILEVEL SECURE CONCURRENCY

In a MLS environment, the concurrency control mechanisms of the DBMS must not violate security requirements or introduce covert channels. Read locks can result in an inference channel when a higher level user queries a lower level object. Because users at the lower level are prevented from accessing the data due to the read lock, these users can infer information about activity by the higher level user. In addition, if a lower level user is reading (locking) data, a higher level user cannot retrieve the lower level information, causing denial of service. The concurrency control algorithm used in Trusted Rubix™ is a form of Multi Version Concurrency Control (MVCC) called Secure Multi-Version Timestamp Ordering (MVTO) which avoids this problem.

TR includes several mechanisms in its implementation of MVTO which capture attempts of multiple processes to update the same record at the same time and attempts to read partial updates that have not been completed. TR also provides for some mechanisms to bypass and/or enhance the way MVTO is implemented on a transaction by transaction basis. These mechanisms provide for several access modes which enable processes to:

→ relax the serializability constraints at run-time, thereby enabling a process to make the decision between performance vs. serializability (higher serializability causes lower performance because of the requirement for additional overhead of ensuring that serializability is maintained).

→ relax the constraints on the visibility of work performed by other transactions that have not yet completed.

→ not automatically abort a transaction when an exception is raised. Instead, TR refuses to perform the operation and gives the user the choice of continuing on with other work, or aborting the transaction. This is much more suitable because only the process can know whether to abort because of the exception.
SQL ODBC EXAMPLES

INTRODUCTION

This section consists of a series of sample ODBC programs using C as the host language. These programs assume the existence of the database defined in the section titled TUTORIAL DATABASE. Also, the examples assume the Trusted Rubix ODBC driver and data source (named MyDSN) have been properly initialized. Please see the ODBC Guide for more information. For the entire Microsoft ODBC specification, please see http://msdn.microsoft.com/en-us/library/ms710252(VS.85).aspx.

The tutorial will progress in six steps:

1. the first section will cover how to compile and link an ODBC source file;
2. the next section will illustrate a program that opens multiple connections;
3. the next section will be an example program doing a simple select;
4. the next section illustrates the use of SQL update with ODBC;
5. the next section in the tutorial illustrates the use of dynamic input arguments with ODBC;
   and,
6. the last section illustrates how to obtain and use ODBC diagnostic information.

Note that in the following examples error checking has been left out for succinctness. Error handling should be performed for all ODBC API calls using the SQL_SUCCEEDED macro as the following code snippet demonstrates.

```c
SQLRETURN ret;

ret = SQLAllocHandle(SQL_HANDLE_DBC, ehdle, &chdle);
if (!SQL_SUCCEEDED(ret))
{
    // Error handling here
}
```

COMPILE AND LINK

The ODBC program has to be compiled and linked before the resulting program can be executed. The following illustration assumes we are compiling and linking the source code stored in the file “sample.c”. The command for compiling and linking is:

```
gcc sample.c -o sample -lodbc
```

This command will compile the source program and link it with the ODBC libraries to produce the executable program sample. It is more convenient to use the UNIX/LINUX make utility to create the executable in a single step. The Make file to build the standalone version of sample.c should have the following content:

```make
LIBS := -lodbc
CFLAGS :=

sample: 
    cc $(CFLAGS) sample.c -o sample $(LIBS)
```

© 2014 Infosystems Technology, Inc.
After compiling and linking as described above, the executable program sample is ready to run.

**CONNECT/DISCONNECT**

The core of the ODBC design is represented by a series of handles representing environments, connections, statements, and descriptors. There can be a multiple number of each handle type. Environment handles are the parent handles for connection handles. Each environment can have many connections to many databases, although they all share certain attributes. An environment defines whether the connections for it are opened in local or distributed transaction mode. Before any connections are opened in the environment, this attribute can be altered from its default of local transaction mode.

Each connection handle has associated with it statement and descriptor handles. These handles operate only on their parent handle, and cannot be shared among the different connections. The statement handles are used for executing statements and data retrieval from the connection to which they belong.

A ODBC application program can communicate with different servers using separate connections with each of the servers. Connections may be established using the `SQLConnect` call and are closed using the `SQLDisconnect` statement. An application program does not have to specifically switch between the different open connections. The use of the different statement handle ensures that the correct connection is used.

The following sample program shows the use of these calls. The name of the connection and the corresponding server can be retrieved using the `SQLGetDiagField` call.

```c
#include <stdlib.h>
#include <stdio.h>
#include <sql.h>
#include <sqlext.h>

main( )
{
    SQLHENV  ehdle;
    SQLHDBC  chdle;
    SQLHDBC  chdle2;
    SQLCHAR  server[256];
    SQLCHAR  connection[256];
    SQLCHAR  * dsnname;
    SQLCHAR  * username;
    SQLCHAR  * pass;
    SQLSMALLINT msg_len;
    // Get an environment handle and save it in ehdle
    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &ehdle);
    // Set the ODBC version to version 3
    SQLSetEnvAttr(ehdle, SQL_ATTR_ODBC_VERSION, 
                 (void *) SQL_OV_ODBC3, 0);
    //Get a connection handle and store it in chdle
    SQLAllocHandle(SQL_HANDLE_DBC,ehdle, &chdle);

    // Set our DSN, user name, and password
    dsnname = "MyDSN";
    username = "MyUsername";
    pass = "MyPassword";
    //Now connect, the DSN contains all other connection info
```
SQLConnect(chdle, dsnname, SQL_NTS, username, SQL_NTS, pass, SQL_NTS);

//Get a second connection handle
SQLAllocHandle(SQL_HANDLE_DBC,ehdle, &chdle2);
// The DSN, username, and password for the second connection
// can be the same or different
dsnname = "MyDSN";
username = "MyUsername";
pass = "MyPassword";
//Connect to the second DSN
SQLConnect(chdle, dsnname, SQL_NTS, username, SQL_NTS, pass, SQL_NTS);

printf("\nConnection One:\n") ;
//Get connection name
SQLGetDiagField(SQL_HANDLE_DBC, chdle, 0,
    SQL_DIAG_CONNECTION_NAME,
    (SQLPOINTER)connection, sizeof(connection),
    &msg_len);
//Get server name
SQLGetDiagField(SQL_HANDLE_DBC, chdle, 0,
    SQL_DIAG_SERVER_NAME,
    (SQLPOINTER)server, sizeof(server),
    &msg_len);
printf("CONNECTION_NAME : %s\n",connection);
printf("DSN_NAME : %s\n",server);

printf("\nConnection Two:\n") ;
SQLGetDiagField(SQL_HANDLE_DBC,chdle2,0,
    SQL_DIAG_CONNECTION_NAME,
    (SQLPOINTER)connection, sizeof(connection),
    &msg_len);
SQLGetDiagField(SQL_HANDLE_DBC,chdle2, 0,
    SQL_DIAG_SERVER_NAME,
    (SQLPOINTER)server, sizeof(server),
    &msg_len);
printf("CONNECTION_NAME : %s\n",connection);
printf("DSN_NAME : %s\n",server);

//Disconnect and free handles
SQLDisconnect(chdle);
SQLFreeHandle(SQL_HANDLE_DBC, chdle);
SQLDisconnect(chdle2);
SQLFreeHandle(SQL_HANDLE_DBC, chdle2);
SQLFreeHandle(SQL_HANDLE_ENV, ehdl);
exit(0) ;
}

Executing the program above, produces the results shown below:

Connection One:
CONNECTION_NAME : CONN_AAA
DSN_NAME : MyDSN

Connection Two:
CONNECTION_NAME : CONN_AAB
DSN_NAME : MyDSN

SELECT STATEMENTS

ODBC translates all SELECT statements into a cursor-specification. Executing a cursor-specification defines a result set. The application can fetch individual rows of the result set by calling SQLFetch or SQLFetchScroll. The application can retrieve data from a fetched row by binding an application variable to the column. The application can call SQLBindCol or SQLSetDescRec to make a complete specification for a column of the result set, or can use SQLSetDescField to set individual descriptor fields. The ODBC driver performs type conversion on affected columns when it fetches the row. This program is equivalent to the following rxisql command.

```
SELECT lastname, firstname, title
FROM employees;
```

The following ODBC program will implement the SQL statement above.

```c
#include <stdlib.h>
#include <stdio.h>
#include <sql.h>
#include <sqlext.h>

main( )
{
    SQLHENV ehdle;
    SQLHDBC chdle;
    SQLHSTMT shdle;
    SQLCHAR * dsnname;
    SQLCHAR * username;
    SQLCHAR * pass;
    SQLCHAR * stmt;
    SQLRETURN ret;
    SQLCHAR lastname[11];
    SQLLEN lastname_len;
    SQLCHAR firstname[11];
    SQLLEN firstname_len;
    SQLCHAR title[26];
    SQLLEN title_len;

    // Get an environment handle and save it in ehdle
    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &ehdle);
    // Set the ODBC version to version 3
    SQLSetEnvAttr(ehdle, SQL_ATTR_ODBC_VERSION,
                   (void *) SQL_OV_ODBC3, 0);
    // Get a connection handle and store it in chdle
    SQLAllocHandle(SQL_HANDLE_DBC,ehdle, &chdle);

    // Set our DSN, user name, and password
    dsnname = "MyDSN";
    username = "MyUsername";
    pass = "MyPassword";
```
Now connect, the DSN contains all other connection info
SQLConnect(chdle, dsname, SQL_NTS, username, SQL_NTS,
    pass, SQL_NTS);

// Allocate a statement handle (shdle) on the connection
SQLAllocHandle(SQL_HANDLE_STMT, chdle, &shdle);
// Initialize the SQL statement we will execute
stmt = "SELECT lastname, firstname, title FROM employees";
// Execute the query
SQLExecDirect(shdle, stmt, SQL_NTS);

// Bind column 1 to the variable lastname
// Request the data be returned as default, which is character
// The length of the result will be returned in lastname_len
SQLBindCol(shdle, 1, SQL_DEFAULT, &lastname,
    sizeof(lastname), &lastname_len);
// Bind column 2 to variable firstname
SQLBindCol(shdle, 2, SQL_DEFAULT, &firstname,
    sizeof(firstname), &firstname_len);
// Bind column 3 to variable title
SQLBindCol(shdle, 3, SQL_DEFAULT, &title,
    sizeof(title), &title_len);

// Retrieve and display the data while the fetch
// operation is successful
printf("Lastname: \t Firstname: \t Title: \n");
for (ret = SQLFetch(shdle);
    SQL_SUCCEEDED(ret);
    ret = SQLFetch(shdle))
{
    printf("%11s %11s %21s \n", lastname, firstname, title);
}

// Disconnect and free handles
SQLDisconnect(chdle);
// Note that freeing chdle implicitly frees all statement handles
// It also closes any open cursors
SQLFreeHandle(SQL_HANDLE_DBC, chdle);
SQLFreeHandle(SQL_HANDLE_ENV, ehdle);
exit(0);

Executing the program above, produces the results below:

<table>
<thead>
<tr>
<th>Lastname</th>
<th>Firstname</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>Bruce</td>
<td>Accountant</td>
</tr>
<tr>
<td>Smith</td>
<td>Sandra</td>
<td>Secretary</td>
</tr>
<tr>
<td>Xiu</td>
<td>Belinda</td>
<td>Office Manager</td>
</tr>
<tr>
<td>Wickham</td>
<td>Scott</td>
<td>Software Engineer</td>
</tr>
<tr>
<td>Brown</td>
<td>Jane</td>
<td>Signal Engineer</td>
</tr>
<tr>
<td>Golden</td>
<td>Barbara</td>
<td>Scientist</td>
</tr>
</tbody>
</table>

POSITIONED UPDATE

A positioned update is an update of a record in a base or derived table pointed to by an open
cursor. An update is done only to the record at the current cursor position. In the following example program, all the software engineers are given a 10% increase in salary. Note that only the record pointed to by the cursor is updated.

```c
#include <stdlib.h>
#include <stdio.h>
#include <sql.h>
#include <sqlext.h>

main()
{
  SQLHENV ehdle;
  SQLHDBC chdle;
  SQLHSTMT shdle;
  SQLHSTMT upd_shdle;
  SQLCHAR * dsnname;
  SQLCHAR * username;
  SQLCHAR * pass;
  SQLCHAR * sel_stmt;
  SQLCHAR * base_upd_stmt;
  SQLCHAR updstmt[256];
  SQLCHAR curs[256];
  SQLRETURN ret;
  SQLCHAR lastname[11];
  SQLLEN lastname_len;
  SQLCHAR title[26];
  SQLLEN title_len;
  SQLCHAR salary[11];
  SQLLEN salary_len;

  // Get an environment handle and save it in ehdle
  SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &ehdle);
  // Set the ODBC version to version 3
  SQLSetEnvAttr(ehdle, SQL_ATTR_ODBC_VERSION,
                 (void *) SQL_OV_ODBC3, 0);
  // Get a connection handle and store it in chdle
  SQLAllocHandle(SQL_HANDLE_DBC,ehdle, &chdle);

  // Set our DSN, user name, and password
dsnname = "MyDSN";
  username = "MyUsername";
  pass = "MyPassword";
  // Now connect, the DSN contains all other connection info
  SQLConnect(chdle, dsnname, SQL_NTS, username, SQL_NTS, pass, SQL_NTS);

  // Allocate a statement handle for the select
  SQLAllocHandle(SQL_HANDLE_STMT, chdle, &shdle);
  // Allocate a statement handle for the positioned update
  SQLAllocHandle(SQL_HANDLE_STMT, chdle, &upd_shdle);
  // Initialize the SQL statement we will execute
  // Note we indicate it is updatable
  sel_stmt = "SELECT lastname, title, salary "
             "FROM employees "
             "WHERE title = 'Software Engineer' "
             "FOR UPDATE";
```
// Execute the query
SQLExecDirect(shdle, sel_stmt, SQL_NTS);

// Bind column 1 to the variable lastname
// Request the data be returned as default, which is character
// The length of the result will be returned in lastname_len
SQLBindCol(shdle, 1, SQL_DEFAULT, &lastname,
    sizeof(lastname), &lastname_len);
// Bind column 2 to variable title
SQLBindCol(shdle, 2, SQL_DEFAULT, &title,
    sizeof(title), &title_len);
// Bind column 3 to variable salary
SQLBindCol(shdle, 3, SQL_DEFAULT, &salary,
    sizeof(salary), &salary_len);

// The positioned update statement requires the cursor name
// Let's extract it from the statement handle
SQLGetCursorName (shdle, curs, sizeof(curs), NULL);
// Construct the update statement
base_upd_stmt = "UPDATE employees 
    "SET   salary = salary*1.1 
    "WHERE CURRENT OF ";
// Append the cursor name to the above update statement
sprintf(updstmt, "%s%s", base_upd_stmt, curs);

printf("\nLastname: \t Firstname: \t Salary: \n")
for ( ret = SQLFetch(shdle);
    SQL_SUCCEEDED(ret);
    ret = SQLFetch(shdle))
{
    printf("%11s %25s %21s 
", lastname, title, salary);
   // For each fetched statement, execute the update request
   // Note that each update statement
   // will automatically be committed, this is ODBC default
   SQLExecDirect(upd_shdle, updstmt, SQL_NTS);
}

// Now let's select the new salaries
// Note we reuse shdle, no need to re-bind the columns
// but we must close the cursor
SQLCloseCursor(shdle);
printf("Salary after a 10\% raise.\n");
sel_stmt = "SELECT lastname, title, salary "
    "FROM employees "
    "WHERE title = 'Software Engineer';"
SQLExecDirect(shdle, sel_stmt, SQL_NTS);
printf("\nLastname: \t Firstname: \t Salary:\n");
for ( ret = SQLFetch(shdle);
    SQL_SUCCEEDED(ret);
    ret = SQLFetch(shdle))
{
    printf("%11s %25s %21s 
", lastname, title, salary);
}

//Disconnect and free handles
SQLDisconnect(chdle);
Executing the program above, produces the results below:

<table>
<thead>
<tr>
<th>Lastname</th>
<th>Title</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wickham</td>
<td>Software Engineer</td>
<td>$45000.00</td>
</tr>
</tbody>
</table>

Salary after 10% increase

<table>
<thead>
<tr>
<th>Lastname</th>
<th>Title</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wickham</td>
<td>Software Engineer</td>
<td>$49500.00</td>
</tr>
</tbody>
</table>

USING DYNAMIC ARGUMENTS

An SQL statement executed under ODBC may contain question mark (?) characters. Each of these is a “dynamic parameter marker”. A dynamic parameter marker represents a position in the SQL statement (a “dynamic parameter”) where the application is to provide a value (a “dynamic argument”).

The application must provide a dynamic argument for each dynamic parameter in the SQL statement before it executes the statement. The easiest way for an application to provide a parameter specification is through the use of the SQLBindParam call. The following example uses a dynamic parameter to specify a query qualifier. A user can also use dynamic arguments in the SQL INSERT and UPDATE statements.

```c
#include <stdlib.h>
#include <stdio.h>
#include <sql.h>
#include <sqlext.h>

main( )
{
    SQLHENV ehdle;
    SQLHDBC chdle;
    SQLHSTMT shdle;
    SQLCHAR * dsnname;
    SQLCHAR * username;
    SQLCHAR * pass;
    SQLCHAR * stmt;
    SQLRETURN ret;
    SQLCHAR  lastname[11];
    SQLLEN   lastname_len;
    SQLCHAR  title[26];
    SQLLEN   title_len;
    SQLCHAR  salary[11];
    SQLLEN   salary_len;
    SQLCHAR  title_arg[26];

    // Get an environment handle and save it in ehdle
    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &ehdle);
```
// Set the ODBC version to version 3
SQLSetEnvAttr(ehdl, SQL_ATTR_ODBC_VERSION,
    (void *) SQL_OV_ODBC3, 0);

// Get a connection handle and store it in chdle
SQLAllocHandle(SQL_HANDLE_DBC,ehdle, &chdle);

// Set our DSN, user name, and password
dsnname = "MyDSN";
username = "MyUsername";
pass = "MyPassword";

// Now connect, the DSN contains all other connection info
SQLConnect(chdle, dsnname, SQL_NTS, username, SQL_NTS,
    pass, SQL_NTS);

// Allocate a statement handle for the select
SQLAllocHandle(SQL_HANDLE_STMT, chdle, &shdle);

// Build a query that uses a dynamic argument
stmt = "SELECT lastname, title, salary "
     "FROM employees "
     "WHERE title = ? ";

// Prepare the statement for execution
SQLPrepare(shdle, stmt, SQL_NTS);

// Bind the first parameter to our local variable, title_arg
SQLBindParam(shdle, 1, SQL_CHAR, SQL_CHAR,
    sizeof(title_arg), 0, &title_arg, NULL);

// Place the correct value in the dynamic argument
strcpy(title_arg, "Software Engineer");

// Execute the select dynamic statement
SQLExecute(shdle);

// Bind our columns so they can displayed
// Bind column 1 to the variable lastname
// Request the data be returned as default, which is character
// The length of the result will be returned in lastname_len
SQLBindCol(shdle, 1, SQL_DEFAULT, &lastname,
    sizeof(lastname), &lastname_len);

// Bind column 2 to variable title
SQLBindCol(shdle, 2, SQL_DEFAULT, &title,
    sizeof(title), &title_len);

// Bind column 3 to variable salary
SQLBindCol(shdle, 3, SQL_DEFAULT, &salary,
    sizeof(salary), &salary_len);

// Fetch and display the result set
printf("\nLastname: \t Firstname: \t Salary: \n");
for( ret = SQLFetch(shdle);
    SQL_SUCCEEDED(ret);
    ret = SQLFetch(shdle))
{
    printf("%11s %25s %21s \n", lastname, title, salary);
}

// Now let's select using a non-dynamic statement
// The results should be the same
// Note we reuse shdle, no need to re-bind the columns
// but we must close the cursor
SQLCloseCursor(shdle);
stmt = "SELECT lastname, title, salary 
       "FROM employees 
       "WHERE title = 'Software Engineer' ";
SQLExecDirect(shdle, stmt, SQL_NTS);
printf("%s %s %s
", lastname, title, salary); 
for ( ret = SQLFetch(shdle); 
     SQL_SUCCEEDED(ret); 
     ret = SQLFetch(shdle)) 
{
    printf("%ls %25s %21s
", lastname, title, salary) ;
}

// Disconnect and free handles
SQLDisconnect(chdle);
// Note that freeing chdle implicitly frees all statement handles
// It also closes any open cursors
SQLFreeHandle(SQL_HANDLE_DBC, chdle);
SQLFreeHandle(SQL_HANDLE_ENV, ehdle);
exit(0);
}

Executing the queries above, produces the results below:

<table>
<thead>
<tr>
<th>Lastname</th>
<th>Title</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wickham</td>
<td>Software Engineer</td>
<td>$45000.00</td>
</tr>
<tr>
<td>Wickham</td>
<td>Software Engineer</td>
<td>$45000.00</td>
</tr>
</tbody>
</table>

**DIAGNOSTICS**

The information about the last executed SQL statement is stored in its diagnostics area. This information can be retrieved by the `SQLGetDiagField` or the `SQLGetDiagRec` calls. The `SQLGetDiagRec` retrieves a fixed set of fields from the diagnostic area, whereas `SQLGetDiagField` can return a larger number of diagnostic fields. The following sample program displays the status of the execution of the Searched Update statement which tries to give all software engineers a 10% raise in salary.

```c
#include <stdlib.h>
#include <stdio.h>
#include <sql.h>
#include <sqlext.h>

main( )
{
    SQLHENV   ehdle;
    SQLHDBC   chdle;
    SQLHSTMT  shdle;
    SQLCHAR * dsnname;
    SQLCHAR * username;
    SQLCHAR * pass;
    SQLCHAR * stmt;
    SQLRETURN ret;
```
// Get an environment handle and save it in ehdle
SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &ehdle);

// Set the ODBC version to version 3
SQLSetEnvAttr(ehdle, SQL_ATTR_ODBC_VERSION,
(void *) SQL_OV_ODBC3, 0);

// Get a connection handle and store it in chdle
SQLAllocHandle(SQL_HANDLE_DBC, ehdle, &chdle);

// Set our DSN, user name, and password
dsnname = "MyDSN";
username = "MyUsername";
pass = "MyPassword";
// Now connect, the DSN contains all other connection info
SQLConnect(chdle, dsnname, SQL_NTS, username, SQL_NTS, pass, SQL_NTS);

// Allocate a statement handle for the select
SQLAllocHandle(SQL_HANDLE_STMT, chdle, &shdle);

// Construct an update statement,
// note the misspelled UPDATE
stmt = "UP
ATE employees"
"SET salary = salary*1.1";
// Attempt to execute the bogus update statement
ret = SQLExecDirect(shdle, stmt, SQL_NTS);
if (!SQL_SUCCEEDED(ret))
{
    // Get the first sql error code
    SQLGetDiagField(SQL_HANDLE_STMT, shdle, 1,
    SQL_DIAG_SQLSTATE, (SQLPOINTER)sqlstat,
    sizeof(sqlstate), (SQLSMALLINT*)NULL);
    // Get the associated text message
    SQLGetDiagField(SQL_HANDLE_STMT, shdle, 1,
    SQL_DIAG_MESSAGE_TEXT, (SQLPOINTER)msgtxt,
    sizeof(msgtxt), (SQLSMALLINT*)NULL);
    printf("\nDiagnostics:
RETURNED_SQLSTATE : %s
MESS
AGE_TEXT      : %s\n", sqlstate, msgtxt);
}
else
{
    // Get the number of rows updated
    SQLGetDiagField(SQL_HANDLE_STMT, shdle, 0,
    SQL_DIAG_ROW_COUNT, (SQLPOINTER)&rowcount,
    NULL, (SQLSMALLINT*)NULL);
    printf("\nNumber of rows updated: %d\n", rowcount);
}

// Disconnect and free handles
SQLDisconnect(chdle);
// Note that freeing chdle implicitly frees all statement handles
// It also closes any open cursors
SQLFreeHandle(SQL_HANDLE_DBC, chdle);
```c
SQLFreeHandle(SQL_HANDLE_ENV, ehdle);
exit(0);
}
```

Executing the program above, produces the results shown below:

Diagnostics :
RETURNED_SQLSTATE : 2A000
MESSAGE_TEXT : Syntax error or access rule violation in direct SQL statement.